

Сиддхартха Рао

**ВОСЬМОЕ
ИЗДАНИЕ**

**ОПИСАН
C++14 и C++17**

Освой самостоятельно

C++

по одному часу в день

 **SAMS**

Освой самостоятельно

C++

по одному часу в день

ВОСЬМОЕ ИЗДАНИЕ

Sams Teach Yourself

C++

in One Hour a Day

EIGHTH EDITION

Siddhartha Rao

SAMS

800 East 96th Street, Indianapolis, Indiana 46240

Освой самостоятельно

C++

по одному часу в день

ВОСЬМОЕ ИЗДАНИЕ

Сиддхартха Рао



Москва · Санкт-Петербург · Киев
2017

ББК 32.973.26-018.2.75

P22

УДК 681.3.07

Компьютерное издательство “Диалектика”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:

info@dialektika.com, <http://www.dialektika.com>

Рао, Сиддхартха.

P22 Освой самостоятельно C++ по одному часу в день, 8-е изд. : Пер. с англ. — СПб. : ООО “Альфа-книга”, 2017. — 752 с. : ил. — Парал. тит. англ.

ISBN 978-5-9909445-6-5 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Sams Publishing.

Authorized translation from the English language edition published by Sams Publishing, Copyright © 2017 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the Publisher, except for the inclusion of brief quotations in a review.

Russian language edition published by Dialektika Computer Books Publishing according to the Agreement with R&I Enterprises International, Copyright © 2017.

Научно-популярное издание

Сиддхартха Рао

Освой самостоятельно C++ по одному часу в день 8-е издание

Литературный редактор	<i>Л.Н. Красножон</i>
Верстка	<i>Л.В. Чернокозинская</i>
Художественный редактор	<i>Е.П. Дынник</i>
Корректор	<i>Л.А. Гордиенко</i>

Подписано в печать 28.08.2017. Формат 70х100/16.

Гарнитура Times.

Усл. печ. л. 47,0. Уч.-изд. л. 34,3.

Тираж 400 экз. Заказ № 5941

Отпечатано в АО «Первая Образцовая типография»

Филиал «Чеховский Печатный Двор»

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

ООО “Альфа-книга”, 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9909445-6-5 (рус.)

© Компьютерное издательство “Диалектика”, 2017
перевод, оформление, макетирование

ISBN 978-0-7897-5774-6 (англ.)

© by Pearson Education, Inc., 2017

Оглавление

ВВЕДЕНИЕ	25
ЧАСТЬ I. Основы C++	29
ЗАНЯТИЕ 1. Первые шаги	31
ЗАНЯТИЕ 2. Структура программы на C++	41
ЗАНЯТИЕ 3. Использование переменных и констант	55
ЗАНЯТИЕ 4. Массивы и строки	85
ЗАНЯТИЕ 5. Выражения, инструкции и операторы	105
ЗАНЯТИЕ 6. Управление потоком выполнения программы	129
ЗАНЯТИЕ 7. Организация кода с помощью функций	165
ЗАНЯТИЕ 8. Указатели и ссылки	191
ЧАСТЬ II. Объектно-ориентированное программирование на C++	227
ЗАНЯТИЕ 9. Классы и объекты	229
ЗАНЯТИЕ 10. Реализация наследования	283
ЗАНЯТИЕ 11. Полиморфизм	315
ЗАНЯТИЕ 12. Типы операторов и их перегрузка	343
ЗАНЯТИЕ 13. Операторы приведения	381
ЗАНЯТИЕ 14. Введение в макросы и шаблоны	395
ЧАСТЬ III. Стандартная библиотека шаблонов	425
ЗАНЯТИЕ 15. Введение в стандартную библиотеку шаблонов	427
ЗАНЯТИЕ 16. Класс строки библиотеки STL	439
ЗАНЯТИЕ 17. Классы динамических массивов библиотеки STL	457
ЗАНЯТИЕ 18. Классы <code>list</code> и <code>forward_list</code>	475
ЗАНЯТИЕ 19. Классы множеств STL	495
ЗАНЯТИЕ 20. Классы отображений библиотеки STL	513

часть IV. Углубляемся в STL	535
ЗАНЯТИЕ 21. Понятие о функциональных объектах	537
ЗАНЯТИЕ 22. Лямбда-выражения языка C++11	553
ЗАНЯТИЕ 23. Алгоритмы библиотеки STL	567
ЗАНЯТИЕ 24. Адаптивные контейнеры: стек и очередь	599
ЗАНЯТИЕ 25. Работа с битовыми флагами при использовании библиотеки STL	615
часть V. Сложные концепции C++	625
ЗАНЯТИЕ 26. Понятие интеллектуальных указателей	627
ЗАНЯТИЕ 27. Применение потоков для ввода и вывода	641
ЗАНЯТИЕ 28. Обработка исключений	663
ЗАНЯТИЕ 29. Что дальше	677
часть VI. Приложения	691
ПРИЛОЖЕНИЕ А. Двоичные и шестнадцатеричные числа	693
ПРИЛОЖЕНИЕ Б. Ключевые слова языка C++	699
ПРИЛОЖЕНИЕ В. Приоритет операторов	701
ПРИЛОЖЕНИЕ Г. Коды ASCII	703
ПРИЛОЖЕНИЕ Д. Ответы	707
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ	747

Содержание

Благодарности	23
Об авторе	23
Поддержка читателя	24
Ждем ваших отзывов!	24
ВВЕДЕНИЕ	25
Для кого написана эта книга	25
Структура книги	25
Соглашения, принятые в книге	26
Примеры кода	27
ЧАСТЬ I. Основы C++	29
ЗАНЯТИЕ 1. Первые шаги	31
Краткий экскурс в историю языка C++	32
Связь с языком C	32
Преимущества языка C++	32
Развитие стандарта C++	33
Кто использует программы, написанные на C++	33
Создание приложения C++	33
Этапы создания исполнимого файла	34
Анализ и устранение ошибок	34
Интегрированные среды разработки	34
Создание первого приложения на C++	35
Построение и запуск вашего первого приложения C++	36
Понятие ошибок компиляции	38
Что нового в C++	38
Резюме	39
Вопросы и ответы	39
Коллоквиум	40
Контрольные вопросы	40
Упражнения	40
ЗАНЯТИЕ 2. Структура программы на C++	41
Части программы Hello World	42
Директива препроцессора #include	42
Тело программы — функция main()	43
Возврат значения	44
Концепция пространств имен	44
Комментарии в коде C++	46
Функции в C++	47
Ввод-вывод с использованием потоков std::cin и std::cout	50
Резюме	52
Вопросы и ответы	52
Коллоквиум	52
Контрольные вопросы	53
Упражнения	53

ЗАНЯТИЕ 3. Использование переменных и констант	55
Что такое переменная	56
Коротко о памяти и адресации	56
Объявление переменных для получения доступа и использования памяти	56
Объявление и инициализация нескольких переменных одного типа	58
Понятие области видимости переменной	59
Глобальные переменные	61
Соглашения об именовании	62
Распространенные типы переменных, поддерживаемые компилятором C++	63
Использование типа <code>bool</code> для хранения логических значений	64
Использование типа <code>char</code> для хранения символьных значений	64
Концепция знаковых и беззнаковых целых чисел	65
Знаковые целочисленные типы <code>short</code> , <code>int</code> , <code>long</code> и <code>long long</code>	66
Беззнаковые целочисленные типы <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code> и <code>unsigned long long</code>	66
Избегайте переполнения, выбирая подходящие типы	67
Типы с плавающей точкой <code>float</code> и <code>double</code>	69
Определение размера переменной с использованием оператора <code>sizeof</code>	69
Запрет сужающего преобразования при использовании инициализации списком	71
Автоматический вывод типа с использованием <code>auto</code>	72
Использование ключевого слова <code>typedef</code> для замены типа	73
Что такое константа	74
Литеральные константы	74
Объявление переменных как констант с использованием ключевого слова <code>const</code>	75
Объявление констант с использованием ключевого слова <code>constexpr</code>	76
Перечисления	78
Определение констант с использованием директивы <code>#define</code>	80
Ключевые слова, недопустимые для использования в качестве имен переменных и констант	80
Резюме	81
Вопросы и ответы	82
Коллоквиум	84
Контрольные вопросы	84
Упражнения	84
ЗАНЯТИЕ 4. Массивы и строки	85
Что такое массив	86
Необходимость в массивах	86
Объявление и инициализация статических массивов	87
Как данные хранятся в массиве	88
Доступ к данным, хранимым в массиве	89
Изменение данных в массиве	90
Многомерные массивы	93
Объявление и инициализация многомерных массивов	93
Доступ к элементам многомерного массива	94
Динамические массивы	95
Строки символов в стиле C	97

Строки C++: использование <code>std::string</code>	100
Резюме	101
Вопросы и ответы	102
Коллоквиум	103
Контрольные вопросы	103
Упражнения	103
ЗАНЯТИЕ 5. Выражения, инструкции и операторы	105
Выражения	106
Составные инструкции, или блоки	107
Использование операторов	107
Оператор присваивания (=)	107
Понятие l- и r-значений	107
Операторы сложения (+), вычитания (-), умножения (*), деления (/) и деления по модулю (%)	108
Операторы инкремента (++) и декремента (--)	109
Что значит “постфиксный” и “префиксный”	109
Операторы равенства (==) и неравенства (!=)	111
Операторы сравнения	111
Логические операции НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ	114
Использование логических операторов C++ !, && и	115
Побитовые операторы ~, &, и ^	119
Побитовые операторы сдвига вправо (>>) и влево (<<)	121
Составные операторы присваивания	122
Использование оператора <code>sizeof</code> для определения объема памяти, занимаемого переменной	124
Приоритет операторов	125
Резюме	127
Вопросы и ответы	127
Коллоквиум	128
Контрольные вопросы	128
Упражнения	128
ЗАНЯТИЕ 6. Управление потоком выполнения программы	129
Условное программирование с использованием конструкции <code>if...else</code>	131
Условное выполнение нескольких инструкций	133
Вложенные инструкции <code>if</code>	134
Условная обработка с использованием конструкции <code>switch-case</code>	138
Тернарный условный оператор (?:)	141
Выполнение кода в циклах	142
Рудиментарный цикл с использованием инструкции <code>goto</code>	143
Цикл <code>while</code>	145
Цикл <code>do...while</code>	146
Цикл <code>for</code>	148
Цикл <code>for</code> для диапазона	151
Изменение поведения цикла с использованием операторов <code>continue</code> и <code>break</code>	153
Бесконечные циклы, которые никогда не заканчиваются	154
Управление бесконечными циклами	154

Программирование вложенных циклов	157
Использование вложенных циклов для перебора многомерного массива	158
Использование вложенных циклов для вычисления чисел Фибоначчи	160
Резюме	161
Вопросы и ответы	162
Коллоквиум	162
Контрольные вопросы	163
Упражнения	163
ЗАНЯТИЕ 7. Организация кода с помощью функций	165
Потребность в функциях	166
Что такое прототип функции	167
Что такое определение функции	168
Что такое вызов функции и аргументы	168
Создание функций с несколькими параметрами	169
Создание функций без параметров и возвращаемых значений	170
Параметры функций со значениями по умолчанию	171
Рекурсия — функция, вызывающая сама себя	173
Функции с несколькими операторами <code>return</code>	175
Использование функций для работы с данными различных видов	176
Перегрузка функций	177
Передача в функцию массива значений	178
Передача аргументов по ссылке	180
Как процессор обрабатывает вызовы функций	182
Встраиваемые функции	183
Автоматический вывод возвращаемого типа	184
Лямбда-функции	186
Резюме	187
Вопросы и ответы	188
Коллоквиум	188
Контрольные вопросы	188
Упражнения	189
ЗАНЯТИЕ 8. Указатели и ссылки	191
Что такое указатель	192
Объявление указателя	192
Определение адреса переменной с использованием оператора получения адреса <code>&</code>	193
Использование указателей для хранения адресов	194
Доступ к данным с использованием оператора разыменования <code>*</code>	197
Значение <code>sizeof()</code> для указателя	199
Динамическое распределение памяти	201
Использование <code>new</code> и <code>delete</code> для выделения и освобождения памяти	201
Указатели и операции инкремента и декремента	204
Использование ключевого слова <code>const</code> с указателями	207
Передача указателей в функции	208
Сходство между массивами и указателями	209
Наиболее распространенные ошибки при использовании указателей	212
Утечки памяти	212

Когда указатели указывают на недопустимые области памяти	213
Висячие (беспризорные, дикие) указатели	214
Проверка успешности запроса с использованием оператора <code>new</code>	215
Полезные советы по применению указателей	218
Что такое ссылка	218
Зачем нужны ссылки	220
Использование ключевого слова <code>const</code> со ссылками	221
Передача аргументов в функции по ссылке	221
Резюме	223
Вопросы и ответы	223
Коллоквиум	225
Контрольные вопросы	225
Упражнения	225
часть II. Объектно-ориентированное программирование на C++	227
ЗАНЯТИЕ 9. Классы и объекты	229
Концепция классов и объектов	230
Объявление класса	230
Объект как экземпляр класса	231
Доступ к членам класса с использованием оператора точки (.)	232
Обращение к членам класса с использованием оператора указателя (->)	232
Ключевые слова <code>public</code> и <code>private</code>	234
Абстракция данных с помощью ключевого слова <code>private</code>	236
Конструкторы	237
Объявление и реализация конструктора	237
Когда и как использовать конструкторы	238
Перегрузка конструкторов	240
Класс без конструктора по умолчанию	242
Параметры конструктора со значениями по умолчанию	243
Конструкторы со списками инициализации	244
Деструктор	246
Объявление и реализация деструктора	246
Когда и как использовать деструкторы	247
Копирующий конструктор	250
Поверхностное копирование и связанные с ним проблемы	250
Глубокое копирование с использованием копирующего конструктора	252
Перемещающий конструктор улучшает производительность	257
Способы использования конструкторов и деструктора	258
Класс, который не разрешает себя копировать	258
Класс-синглтон, обеспечивающий наличие только одного экземпляра	259
Класс, запрещающий создание экземпляра в стеке	262
Применение конструкторов для преобразования типов	264
Указатель <code>this</code>	266
Размер класса	267
Чем структура отличается от класса	269
Объявление друзей класса	270

Специальный механизм хранения данных — <code>union</code>	272
Объявление объединения	272
Где используется объединение	273
Агрегатная инициализация классов и структур	275
<code>constexpr</code> с классами и объектами	278
Резюме	279
Вопросы и ответы	279
Коллоквиум	280
Контрольные вопросы	280
Упражнения	281
ЗАНЯТИЕ 10. Реализация наследования	283
Основы наследования	284
Наследование и порождение	284
Синтаксис наследования C++	286
Модификатор доступа <code>protected</code>	288
Инициализация базового класса — передача параметров базовому классу	290
Перекрытие методов базового класса в производном	293
Вызов перекрытых методов базового класса	295
Вызов методов базового класса в производном классе	296
Производный класс, скрывающий методы базового класса	298
Порядок конструирования	300
Порядок деструкции	300
Закрытое наследование	303
Защищенное наследование	305
Проблема срезки	308
Множественное наследование	309
Запрет наследования с помощью ключевого слова <code>final</code>	311
Резюме	313
Вопросы и ответы	313
Коллоквиум	313
Контрольные вопросы	314
Упражнения	314
ЗАНЯТИЕ 11. Полиморфизм	315
Основы полиморфизма	316
Потребность в полиморфном поведении	316
Полиморфное поведение, реализованное с помощью виртуальных функций	318
Необходимость виртуальных деструкторов	320
Как работают виртуальные функции. Понятие таблицы виртуальных функций	324
Абстрактные классы и чисто виртуальные функции	328
Использование виртуального наследования для решения проблемы ромба	330
Ключевое слово <code>override</code> для указания преднамеренного перекрытия	335
Использование ключевого слова <code>final</code> для предотвращения перекрытия функции	336
Виртуальные копирующие конструкторы?	336
Резюме	340
Вопросы и ответы	340

Коллоквиум	341
Контрольные вопросы	341
Упражнения	342
ЗАНЯТИЕ 12. Типы операторов и их перегрузка	343
Что такое операторы C++	344
Унарные операторы	345
Типы унарных операторов	345
Программирование унарного оператора инкремента или декремента	345
Создание операторов преобразования	348
Создание оператора разыменования (*) и оператора выбора члена (->)	351
Бинарные операторы	353
Типы бинарных операторов	353
Создание бинарных операторов сложения (a+b) и вычитания (a-b)	354
Реализация операторов сложения с присваиванием (+=)	
и вычитания с присваиванием (-=)	357
Перегрузка операторов равенства (==) и неравенства (!=)	359
Перегрузка операторов <, >, <= и >=	361
Перегрузка оператора копирующего присваивания (=)	363
Оператор индексации ([])	366
Оператор функции ()	369
Перемещающий конструктор и оператор перемещающего присваивания	370
Проблема излишнего копирования	370
Объявление перемещающих конструктора и оператора присваивания	371
Пользовательские литералы	376
Операторы, которые не могут быть перегружены	378
Резюме	379
Вопросы и ответы	379
Коллоквиум	380
Контрольные вопросы	380
Упражнения	380
ЗАНЯТИЕ 13. Операторы приведения	381
Потребность в приведении типов	382
Почему приведения в стиле C не нравятся некоторым программистам C++	383
Операторы приведения C++	383
Использование оператора static_cast	384
Использование оператора dynamic_cast и идентификация типа	
времени выполнения	385
Использование оператора reinterpret_cast	388
Использование оператора const_cast	389
Проблемы с операторами приведения C++	390
Резюме	392
Вопросы и ответы	392
Коллоквиум	392
Контрольные вопросы	393
Упражнения	393

ЗАНЯТИЕ 14. Введение в макросы и шаблоны	395
Препроцессор и компилятор	396
Использование <code>#define</code> для определения констант	396
Использование макроса для защиты от множественного включения	399
Использование директивы <code>#define</code> для написания макрофункции	400
Зачем все эти скобки?	402
Использование макроса <code>assert</code> для проверки выражений	402
Преимущества и недостатки использования макрофункций	404
Введение в шаблоны	405
Синтаксис объявления шаблона	406
Типы объявлений шаблонов	406
Шаблонные функции	407
Шаблоны и безопасность типов	409
Шаблонные классы	409
Объявление шаблонов с несколькими параметрами	410
Объявление шаблонов параметрами по умолчанию	411
Простой шаблон класса <code>HoldsPair</code>	411
Инстанцирование и специализация шаблона	413
Шаблонные классы и статические члены	415
Шаблоны с переменным количеством параметров (вариадические шаблоны)	416
Использование <code>static_assert</code> для выполнения проверок времени компиляции	420
Использование шаблонов в практическом программировании на C++	421
Резюме	422
Вопросы и ответы	422
Коллоквиум	423
Контрольные вопросы	423
Упражнения	423
 ЧАСТЬ III. Стандартная библиотека шаблонов	 425
ЗАНЯТИЕ 15. Введение в стандартную библиотеку шаблонов	427
Контейнеры STL	428
Последовательные контейнеры	428
Ассоциативные контейнеры	429
Адаптеры контейнеров	431
Итераторы STL	431
Алгоритмы STL	432
Взаимодействие контейнеров и алгоритмов с использованием итераторов	432
Использование ключевого слова <code>auto</code> для определения типа	434
Выбор правильного контейнера	435
Классы строк библиотеки STL	437
Резюме	437
Вопросы и ответы	437
Коллоквиум	438
Контрольные вопросы	438

ЗАНЯТИЕ 16. Класс строки библиотеки STL	439
Потребность в классах обработки строк	440
Работа с классом строки STL	441
Создание экземпляров и копий строк STL	441
Доступ к символу в строке <code>std::string</code>	443
Конкатенация строк	445
Поиск символа или подстроки в строке	446
Усечение строк STL	448
Обращение строки	450
Смена регистра символов	451
Реализация строки на базе шаблона STL	453
Оператор <code>"s</code> в <code>std::string</code> в C++14	453
Резюме	454
Вопросы и ответы	455
Коллоквиум	455
Контрольные вопросы	455
Упражнения	455
ЗАНЯТИЕ 17. Классы динамических массивов библиотеки STL	457
Характеристики класса <code>std::vector</code>	458
Типичные операции с вектором	458
Создание экземпляра вектора	458
Вставка элементов в конец вектора с помощью <code>push_back()</code>	460
Инициализация списком	461
Вставка элементов в определенную позицию с помощью <code>insert()</code>	461
Доступ к элементам вектора с использованием семантики массива	464
Доступ к элементам вектора с использованием семантики указателя	465
Удаление элементов из вектора	466
Концепции размера и емкости	468
Класс <code>deque</code> библиотеки STL	470
Резюме	473
Вопросы и ответы	473
Коллоквиум	474
Контрольные вопросы	474
Упражнения	474
ЗАНЯТИЕ 18. Классы <code>list</code> и <code>forward_list</code>	475
Характеристики класса <code>std::list</code>	476
Основные операции со списком	476
Инстанцирование класса <code>std::list</code>	476
Вставка элементов в начало и в конец списка	478
Вставка в середину списка	479
Удаление элементов из списка	482
Обращение списка и сортировка его элементов	483
Обращение элементов списка с помощью <code>list::reverse()</code>	484
Сортировка элементов	485
Сортировка и удаление элементов из списка, который содержит объекты класса	487
Шаблон класса <code>std::forward_list</code>	490

Резюме	492
Вопросы и ответы	492
Коллоквиум	493
Контрольные вопросы	493
Упражнения	493
ЗАНЯТИЕ 19. Классы множеств STL	495
Введение в классы множеств STL	496
Фундаментальные операции с классами <code>set</code> и <code>multiset</code>	496
Инстанцирование объекта <code>std::set</code>	497
Вставка элементов в множество и мультимножество	499
Поиск элементов в множестве и мультимножестве	500
Удаление элементов из множества и мультимножества	502
Преимущества и недостатки использования множеств и мультимножеств	507
Реализация хеш-множеств <code>std::unordered_set</code> и <code>std::unordered_multiset</code>	507
Резюме	511
Вопросы и ответы	511
Коллоквиум	511
Контрольные вопросы	512
Упражнения	512
ЗАНЯТИЕ 20. Классы отображений библиотеки STL	513
Введение в классы отображений библиотеки STL	514
Фундаментальные операции с классами <code>std::map</code> и <code>std::multimap</code>	515
Инстанцирование классов <code>std::map</code> и <code>std::multimap</code>	515
Вставка элементов в <code>map</code> и <code>multimap</code>	517
Поиск элементов в отображении	519
Поиск элементов в мультиотображении STL	522
Удаление элементов из <code>map</code> и <code>multimap</code>	522
Применение пользовательского предиката	524
Контейнер для пар “ключ–значение” на базе хеш-таблиц	528
Как работают хеш-таблицы	528
Использование <code>unordered_map</code> и <code>unordered_multimap</code>	529
Резюме	533
Вопросы и ответы	533
Коллоквиум	534
Контрольные вопросы	534
Упражнения	534
часть IV. Углубляемся в STL	535
ЗАНЯТИЕ 21. Понятие о функциональных объектах	537
Концепция функциональных объектов и предикатов	538
Типичные приложения функциональных объектов	538
Унарные функции	538
Унарный предикат	543
Бинарные функции	545
Бинарный предикат	547

Резюме	550
Вопросы и ответы	550
Коллоквиум	550
Контрольные вопросы	550
Упражнения	551
ЗАНЯТИЕ 22. Лямбда-выражения языка C++11	553
Что такое лямбда-выражение	554
Как определить лямбда-выражение	555
Лямбда-выражение для унарной функции	555
Лямбда-выражение для унарного предиката	557
Лямбда-выражения с состоянием и списки захвата [. . .]	558
Обобщенный синтаксис лямбда-выражений	560
Лямбда-выражение для бинарной функции	561
Лямбда-выражение для бинарного предиката	563
Резюме	565
Вопросы и ответы	565
Коллоквиум	566
Контрольные вопросы	566
Упражнения	566
ЗАНЯТИЕ 23. Алгоритмы библиотеки STL	567
Что такое алгоритмы STL	568
Классификация алгоритмов STL	568
Не изменяющие алгоритмы	568
Изменяющие алгоритмы	569
Использование алгоритмов STL	571
Поиск элементов по заданному значению или условию	571
Подсчет элементов с использованием значения или условия	573
Поиск элемента или диапазона в коллекции	575
Инициализация элементов в контейнере заданным значением	577
Использование алгоритма <code>std::generate()</code>	
для инициализации значениями, генерируемыми во время выполнения	579
Обработка элементов диапазона с использованием алгоритма <code>for_each()</code>	581
Выполнение преобразований с помощью алгоритма <code>std::transform()</code>	583
Операции копирования и удаления	585
Замена значений и элементов с использованием условия	588
Сортировка, поиск в отсортированной коллекции и удаление дубликатов	590
Разделение диапазона	592
Вставка элементов в отсортированную коллекцию	594
Резюме	597
Вопросы и ответы	597
Коллоквиум	598
Контрольные вопросы	598
Упражнения	598
ЗАНЯТИЕ 24. Адаптивные контейнеры: стек и очередь	599
Поведенческие характеристики стеков и очередей	600
Стеки	600
Очереди	600

Использование класса STL <code>stack</code>	601
Создание экземпляра стека	601
Функции-члены класса <code>stack</code>	602
Вставка и извлечение с помощью методов <code>push()</code> и <code>pop()</code>	603
Использование класса STL <code>queue</code>	605
Создание экземпляра очереди	605
Функции-члены класса <code>queue</code>	606
Вставка в конец и извлечение из начала очереди с использованием методов <code>push()</code> и <code>pop()</code>	607
Использование класса STL <code>priority_queue</code>	608
Создание экземпляра очереди с приоритетами	608
Функции-члены класса <code>priority_queue</code>	610
Вставка в конец и извлечение из начала очереди с приоритетами с использованием методов <code>push()</code> и <code>pop()</code>	611
Резюме	613
Вопросы и ответы	613
Коллоквиум	613
Контрольные вопросы	614
Упражнения	614
ЗАНЯТИЕ 25. Работа с битовыми флагами при использовании библиотеки STL	615
Класс <code>bitset</code>	616
Инстанцирование класса <code>std::bitset</code>	616
Использование класса <code>std::bitset</code> и его членов	617
Полезные операторы, предоставляемые классом <code>std::bitset</code>	618
Методы класса <code>std::bitset</code>	618
Класс <code>vector<bool></code>	621
Создание экземпляра класса <code>vector<bool></code>	621
Функции и операторы класса <code>vector<bool></code>	622
Резюме	623
Вопросы и ответы	623
Коллоквиум	624
Контрольные вопросы	624
Упражнения	624
ЧАСТЬ V. Сложные концепции C++	625
ЗАНЯТИЕ 26. Понятие интеллектуальных указателей	627
Что такое интеллектуальный указатель	628
Проблемы обычных указателей	628
Чем могут помочь интеллектуальные указатели	628
Как реализованы интеллектуальные указатели	629
Типы интеллектуальных указателей	630
Глубокое копирование	631
Механизм копирования при записи	633
Интеллектуальные указатели со счетчиком ссылок	633
Интеллектуальный указатель со списком ссылок	634

Деструктивное копирование	634
Использование интеллектуального указателя <code>std::unique_ptr</code>	637
Популярные библиотеки интеллектуальных указателей	639
Резюме	639
Вопросы и ответы	639
Коллоквиум	640
Контрольные вопросы	640
Упражнения	640
ЗАНЯТИЕ 27. Применение потоков для ввода и вывода	641
Концепция потоков	642
Важнейшие классы и объекты потоков C++	643
Использование <code>std::cout</code> для вывода форматированных данных на консоль	644
Изменение формата представления чисел	645
Выравнивание текста и установка ширины поля	647
Использование <code>std::cin</code> для ввода	648
Использование <code>std::cin</code> для ввода простых старых типов данных	648
Использование метода <code>std::cin::get()</code> для ввода в буфер <code>char*</code>	649
Использование <code>std::cin</code> для ввода в переменную типа <code>std::string</code>	650
Использование потока <code>std::fstream</code> для работы с файлом	652
Открытие и закрытие файла с помощью методов <code>open()</code> и <code>close()</code>	652
Создание и запись текстового файла с использованием метода <code>open()</code> и оператора <code><<</code>	653
Чтение текстового файла с использованием метода <code>open()</code> и оператора <code>>></code>	655
Запись и чтение из бинарного файла	656
Использование <code>std::stringstream</code> для преобразования строк	658
Резюме	660
Вопросы и ответы	660
Коллоквиум	660
Контрольные вопросы	660
Упражнения	661
ЗАНЯТИЕ 28. Обработка исключений	663
Что такое исключение	664
Что вызывает исключения	664
Реализация безопасности в отношении исключений с помощью блоков <code>try</code> и <code>catch</code>	665
Использование блока <code>catch(...)</code> для обработки всех исключений	665
Обработка исключения конкретного типа	666
Генерация исключения с помощью оператора <code>throw</code>	668
Как работает обработка исключений	669
Класс <code>std::exception</code>	671
Пользовательский класс исключения, производный от <code>std::exception</code>	672
Резюме	674
Вопросы и ответы	675
Коллоквиум	675
Контрольные вопросы	676
Упражнения	676

ЗАНЯТИЕ 29. Что дальше	677
Чем отличаются современные процессоры	678
Как лучше использовать несколько ядер	679
Что такое поток	679
Зачем создавать многопоточные приложения	680
Как потоки осуществляют транзакцию данных	681
Использование мьютексов и семафоров для синхронизации потоков	682
Проблемы, вызываемые многопоточностью	682
Как писать отличный код C++	683
C++17: что новенького	684
Инициализация в if и switch	684
Гарантия устранения копирования	685
Устранение накладных расходов выделения памяти с помощью <code>std::string_view</code>	686
<code>std::variant</code> как безопасная с точки зрения типов альтернатива объединению	686
Условная компиляция с использованием <code>if constexpr</code>	687
Усовершенствованные лямбда-выражения	688
Автоматический вывод типа для конструкторов <code>template<auto></code>	688
Изучение C++ на этом не заканчивается	688
Документация в вебе	688
Сетевые сообщества и помощь	689
Резюме	689
Вопросы и ответы	689
Коллоквиум	690
Контрольные вопросы	690
 ЧАСТЬ VI. Приложения	 691
ПРИЛОЖЕНИЕ А. Двоичные и шестнадцатеричные числа	693
Десятичная система счисления	694
Двоичная система счисления	694
Почему компьютеры используют двоичные числа	695
Что такое биты и байты	695
Сколько байтов в килобайте	695
Шестнадцатеричная система счисления	696
Зачем нужна шестнадцатеричная система	696
Преобразование в различные системы счисления	697
Обобщенный процесс преобразования	697
Преобразование десятичного числа в двоичное	697
Преобразование десятичного числа в шестнадцатеричное	698
ПРИЛОЖЕНИЕ Б. Ключевые слова языка C++	699
ПРИЛОЖЕНИЕ В. Приоритет операторов	701
ПРИЛОЖЕНИЕ Г. Коды ASCII	703
Таблица ASCII отображаемых символов	704

ПРИЛОЖЕНИЕ Д. Ответы	707
Ответы к занятию 1	707
Контрольные вопросы	707
Упражнения	707
Ответы к занятию 2	708
Контрольные вопросы	708
Упражнения	708
Ответы к занятию 3	709
Контрольные вопросы	709
Упражнения	709
Ответы к занятию 4	710
Контрольные вопросы	710
Упражнения	711
Ответы к занятию 5	711
Контрольные вопросы	711
Упражнения	712
Ответы к занятию 6	712
Контрольные вопросы	712
Упражнения	713
Ответы к занятию 7	716
Контрольные вопросы	716
Упражнения	716
Ответы к занятию 8	717
Контрольные вопросы	717
Упражнения	717
Ответы к занятию 9	717
Контрольные вопросы	717
Упражнения	718
Ответы к занятию 10	719
Контрольные вопросы	719
Упражнения	719
Ответы к занятию 11	720
Контрольные вопросы	720
Упражнения	720
Ответы к занятию 12	722
Контрольные вопросы	722
Упражнения	722
Ответы к занятию 13	723
Контрольные вопросы	723
Упражнения	723
Ответы к занятию 14	724
Контрольные вопросы	724
Упражнения	724
Ответы к занятию 15	725
Контрольные вопросы	725
Ответы к занятию 16	726
Контрольные вопросы	726
Упражнения	726

Ответы к занятию 17	729
Контрольные вопросы	729
Упражнения	729
Ответы к занятию 18	732
Контрольные вопросы	732
Упражнения	733
Ответы к занятию 19	734
Контрольные вопросы	734
Упражнения	734
Ответы к занятию 20	737
Контрольные вопросы	737
Упражнения	738
Ответы к занятию 21	738
Контрольные вопросы	738
Упражнения	738
Ответы к занятию 22	740
Контрольные вопросы	740
Упражнения	740
Ответы к занятию 23	741
Контрольные вопросы	741
Упражнения	742
Ответы к занятию 24	743
Контрольные вопросы	743
Упражнения	743
Ответы к занятию 25	743
Контрольные вопросы	743
Упражнения	744
Ответы к занятию 26	744
Контрольные вопросы	744
Упражнения	744
Ответы к занятию 27	745
Контрольные вопросы	745
Упражнения	746
Ответы к занятию 28	746
Контрольные вопросы	746
Упражнения	746
Ответы к занятию 29	746
Контрольные вопросы	746

Памяти моего отца, который остается для меня источником вдохновения.

Благодарности

Я благодарен за огромную поддержку моей семье, и особенно жене Кларе, а также всем сотрудникам редакции за активное участие в судьбе этой книги.

Об авторе

Сиддхартха Рао — вице-президент по вопросам безопасности в компании SAP AG, ведущем мировом поставщике корпоративного программного обеспечения. Постоянная эволюция языка C++ постоянно убеждает Рао в том, что приложения на C++ можно создавать быстрее, проще и эффективнее.

Сиддхартха любит путешествовать и является страстным поклонником горного велосипеда. Он с нетерпением ждет ваших отзывов о своей работе!

Поддержка читателя

Для доступа к исходному коду, файлам примеров, обновлениям и исправлениям, когда они появятся, зарегистрируйте свою книгу на informit.com/register.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@dialektika.com

WWW: <http://www.dialektika.com>

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116

в Украине: 03150, Киев, а/я 152

Введение

2011 и 2014 годы были особенно важными для языка C++. В то время как новый стандарт C++11 внес в язык программирования кардинальные изменения, новые ключевые слова и конструкции, повышающие эффективность программирования, C++14 скорее добавил завершающие штрихи к возможностям, внесенным в язык стандартом C++11.

Эта книга поможет вам изучить язык C++11 маленькими шагами. Она специально разделена на отдельные занятия, на которых основные принципы этого языка объектно-ориентированного программирования излагаются с практической точки зрения. Вы сможете овладеть языком C++11, уделяя каждому занятию всего один час.

Наилучший способ изучения языка программирования — его практическое применение, поэтому в книге очень много разнообразных примеров кода, анализируя которые, вы улучшите свои знания языка программирования C++. Эти фрагменты кода протестированы с использованием последних версий компиляторов, имеющихся на момент написания книги, а именно — компиляторов Microsoft Visual C++ и GNU C++, которые охватывают большинство возможностей C++14.

Для кого написана эта книга

Книга начинается с основ языка C++. Необходимы лишь желание изучить этот язык и сообразительность, чтобы понять, как он работает. Наличие навыков программирования на языке C++ может быть преимуществом, но не является обязательным. Кроме того, к этой книге имеет смысл обратиться, если вы уже знаете язык C++, но хотите изучить дополнения, которые были внесены в него последними стандартами. Если вы профессиональный программист, то часть III, “Стандартная библиотека шаблонов”, книги поможет узнать, как создавать более эффективные приложения C++.

ПРИМЕЧАНИЕ

Для доступа к исходному коду, файлам примеров, обновлениям и исправлениям, когда они появятся, зарегистрируйте свою книгу на informit.com/register.

Структура книги

В зависимости от уровня своей квалификации вы можете начать изучение с любого раздела. Концепции C++11 и C++14 не выносятся в отдельные главы, а разбросаны по всей книге. Книга состоит из пяти частей.

- Часть I, “Основы C++”, позволяет приступить к написанию простых приложений C++. Одновременно она знакомит с ключевыми словами, которые вы чаще всего видите в коде C++, а также с переменными, но не затрагивает безопасность типов.
- Часть II, “Объектно-ориентированное программирование на C++”, знакомит с концепцией классов. Вы узнаете, как язык C++ поддерживает важнейшие принципы объектно-ориентированного программирования, включая инкапсуляцию, абстракцию, наследование и полиморфизм. Занятие 9, “Классы и объекты”, представляет такую концепцию C++, как перемещающий конструктор, а занятие 12, “Типы операторов и их перегрузка”, — оператор перемещающего присваивания. Эти эффективные средства помогают сократить ненужные и нежелательные этапы копирования, увеличивая производительность приложения. Занятие 14, “Введение в макросы и шаблоны”, является краеугольным камнем для написания мощного обобщенного кода на C++.
- Часть III, “Стандартная библиотека шаблонов”, поможет писать эффективный код C++, использующий класс STL `std::string` и контейнеры. Вы узнаете, как класс `std::string` упрощает операции конкатенации строк и позволяет избежать использования символьных строк в стиле C. Вы сможете использовать динамические массивы и связанные списки библиотеки STL, а не создавать их самостоятельно.
- Часть IV, “Углубляемся в STL”, посвящена алгоритмам. Вы узнаете, как, используя итераторы, применить сортировку в таких контейнерах, как вектор. Здесь также изложено, как ключевое слово C++11 `auto` позволяет существенно сократить длину объявлений итератора. Занятие 22, “Лямбда-выражения языка C++11”, представляет мощное новое средство, позволяющее существенно сократить размеры кода при использовании алгоритмов библиотеки STL.
- Часть V, “Сложные концепции C++”, объясняет такие средства языка, как интеллектуальные указатели и обработка исключений, которые не являются необходимостью в приложении C++, но вносят существенный вклад в увеличение его стабильности и качества. Эта часть завершается полезными советами по написанию приложений C++11 и новыми возможностями, которые должны появиться в новейшем стандарте C++17.

Соглашения, принятые в книге

На занятиях приводятся следующие элементы с дополнительной информацией.

ПРИМЕЧАНИЕ

Здесь приводится дополнительная информация, связанная с материалом занятия.

ВНИМАНИЕ!

Эта врезка привлекает внимание к проблемам или побочным эффектам, которые могут проявиться в тех или иных ситуациях.

СОВЕТ

В этой врезке приводятся практические советы по написанию программ на C++.

РЕКОМЕНДУЕТСЯ

Используйте эти рекомендации для поиска краткого резюме фундаментальных концепций, представленных на занятии.

НЕ РЕКОМЕНДУЕТСЯ

Не пропускайте важные замечания и предупреждения, показанные в этом столбце.

В книге используются различные шрифты для того, чтобы подчеркнуть те или иные моменты, с применением соглашений, общепринятых в компьютерной литературе.

- Новые термины в тексте выделяются курсивом. Чтобы привлечь внимание читателя на отдельные фрагменты текста, также применяется курсив.
- Текст программ, функций, переменных, URL веб-страниц и другой код представлены моноширинным шрифтом.
- Все, что придется вводить с клавиатуры, выделено полужирным шрифтом.
- Знакоместо в описаниях синтаксиса выделено курсивом. Это указывает на необходимость заменить знакоместо фактическим именем переменной, параметром или другим элементом, который должен находиться на этом месте: `class Производный: Модификатор _ Доступа Базовый.`
- Пункты меню и названия диалоговых окон представлены следующим образом: Пункт меню.
- В листингах каждая строка имеет номер. Это сделано исключительно для удобства описания. В реальном коде нумерация отсутствует.

Примеры кода

Примеры кода, приведенные в этой книге, доступны на веб-сайте издательства <http://www.williamspublishing.com/Books/978-5-9909445-6-5.html>.

ЧАСТЬ I

Основы C++

В ЭТОЙ ЧАСТИ...

ЗАНЯТИЕ 1. Первые шаги

ЗАНЯТИЕ 2. Структура программы на C++

ЗАНЯТИЕ 3. Использование переменных и констант

ЗАНЯТИЕ 4. Массивы и строки

ЗАНЯТИЕ 5. Выражения, инструкции и операторы

ЗАНЯТИЕ 6. Управление потоком выполнения программы

ЗАНЯТИЕ 7. Организация кода с помощью функций

ЗАНЯТИЕ 8. Указатели и ссылки

ЗАНЯТИЕ 1

Первые шаги

Добро пожаловать на страницы книги *Освой самостоятельно C++ по одному часу в день!* Сегодня начинается долгий путь, который позволит вам достичь профессионального уровня в программировании на языке C++.

На этом занятии...

- Почему язык C++ стал стандартом в области разработки программного продукта
- Как набрать, откомпилировать и скомпоновать первую рабочую программу C++
- Что нового в C++

Краткий экскурс в историю языка C++

Задача языка программирования — упростить использование вычислительных ресурсов. Язык C++ отнюдь не нов, но весьма популярен и продолжает совершенствоваться. На момент написания этой книги его последняя версия, принятая Международным комитетом по стандартам (ISO) и опубликованная в декабре 2014 года, называется среди программистов “C++14”.

Связь с языком C

Первоначально разработанный Бьярне Страуструпом в 1979 году, язык C++ был задуман как преемник языка C. В противоположность языку программирования C язык C++ был спроектирован как объектно-ориентированный язык, который реализует такие концепции, как наследование, абстракция, полиморфизм и инкапсуляция. Классы языка C++ используют для работы с данными данные-члены и методы-члены. Эти методы работают с данными, хранящимися в данных-членах. В результате такой организации программист моделирует данные и действия, которые планирует выполнить над ними. Многие популярные компиляторы C++ также традиционно продолжают поддерживать программы на языке C.

ПРИМЕЧАНИЕ

При изучении C++ знания и опыт работы с языком C не нужны. Если ваша конечная цель — изучить объектно-ориентированный язык программирования, такой как C++, нет необходимости начинать с изучения процедурного языка программирования наподобие C.

Преимущества языка C++

C++ считается языком программирования среднего уровня. Это означает, что он позволяет создавать как высокоуровневые приложения, так и низкоуровневые библиотеки, работающие с аппаратными средствами. Для многих программистов язык C++ представляет собой оптимальную комбинацию: являясь языком высокого уровня, он позволяет любому создавать сложные приложения, тем самым сохраняя для разработчика возможность достичь максимальной производительности за счет строгого контроля над использованием ресурсов и их доступностью.

Несмотря на наличие более новых языков программирования, таких как Java, и языков на платформе .NET, язык C++ остается популярным и продолжает развиваться. Более новые языки предоставляют дополнительные средства, такие как управление памятью за счет сбора “мусора”, реализованное в компоненте исполняющей среды, которые нравятся некоторым программистам. Однако там, где нужны высокая производительность создаваемого приложения и уменьшенное потребление ресурсов компьютера, программисты все же выбирают язык C++. Многоуровневая архитектура, когда веб-сервер создается на языке C++, а пользовательская часть приложения — на HTML, Java или .NET, является в настоящее время достаточно распространенной.

Развитие стандарта C++

В силу популярности языка годы развития сделали язык C++ доступным на многих разных платформах, большинство из которых имеет собственные компиляторы C++. Развитие языка привело к наличию определенных отклонений от стандарта в разных компиляторах и, соответственно, к большому количеству проблем совместимости и переносимости кода. В результате появилась насущная потребность в стандартизации данного языка программирования.

В 1998 году первая стандартная версия языка C++ была ратифицирована Международной организацией по стандартизации ISO Committee в виде стандарта ISO/IEC 14882:1998. С тех пор стандарт претерпел множество изменений, которые повысили удобство использования языка и расширили поддержку стандартной библиотеки. На момент написания этой книги текущая ратифицированная версия стандарта — ISO/IEC 14882:2014, неофициально именуемая C++14.

ПРИМЕЧАНИЕ

Зачастую текущий стандарт поддерживается популярными компиляторами не сразу или не в полном объеме. Таким образом, хотя с академической точки зрения знание новейших дополнений к стандарту оправданно и безусловно верно, надо помнить, что эти дополнения не являются обязательным условием для написания хороших многофункциональных приложений на C++.

Кто использует программы, написанные на C++

Список приложений, операционных систем, драйверов устройств, офисных приложений, веб-сервисов, баз данных и прочего программного обеспечения, созданного с использованием C++, очень длинный. Независимо от того, кто вы или что вы делаете на компьютере, очень высоки шансы, что вы постоянно используете программное обеспечение, написанное на C++. Этот язык программирования используют не только разработчики программ; он часто выбирается в качестве рабочего языка программирования для исследовательской работы физиками и математиками.

Создание приложения C++

Запуская Блокнот в Windows или терминал Linux, вы фактически указываете процессору запустить выполнимый файл этой программы. *Выполнимый файл* (executable) — это готовый продукт, который может быть выполнен на компьютере и должен сделать то, чего намеревался достичь программист.

Этапы создания исполнимого файла

Написание программы C++ является первым этапом создания исполнимого файла, который в конечном счете может быть выполнен в вашей операционной системе. Основные этапы создания приложений C++ приведены ниже.

1. Написание (программирование) кода C++ с использованием текстового редактора.
2. Компиляция кода с помощью компилятора C++, который преобразовывает исходный текст в команды машинного языка и записывает их в *объектный файл* (object file).
3. Компоновка результатов работы компилятора с помощью компоновщика и получение окончательного исполнимого файла (.exe в Windows, например).

Компиляция (compilation) представляет собой этап, на котором код C++, содержащийся обычно в текстовых файлах с расширением .cpp, преобразуется в бинарный код, который может быть выполнен процессором. *Компилятор* (compiler) преобразует по одному файлу кода за раз, создавая объектный файл с расширением .o или .obj и игнорируя связи, которые код в этом файле может иметь с кодом в другом файле. Распознавание этих связей и объединение кода в одно целое является задачей *компоновщика* (linker). Кроме объединения различных объектных файлов, он разрешает имеющиеся связи и в случае успешной компоновки создает исполнимый файл, который можно выполнять и в конечном счете распространять среди пользователей. Весь процесс в целом называется построением исполнимого файла.

Анализ и устранение ошибок

Большинство приложений редко компилируются и начинают хорошо работать сразу же. Большое или сложное приложение, написанное на любом языке (включая C++), зачастую требует множества запусков для выполнения тестирования, анализа проблем и обнаружения ошибок. Затем ошибки исправляются, программа перекомпилируется и процесс тестирования продолжается. Таким образом, в дополнение к перечисленным выше трем этапам (программирование, компиляция и компоновка) разработка зачастую подразумевает этап *отладки* (debugging), на котором программист анализирует ошибки в приложении и исправляет их. Хорошая интегрированная среда разработки обеспечивает программиста инструментальными средствами отладки, облегчающими указанный процесс.

Интегрированные среды разработки

Большинство программистов предпочитают использовать *интегрированную среду разработки* (Integrated Development Environment — IDE), объединяющую этапы программирования, компиляции и компоновки в пределах единого пользовательского интерфейса, предоставляющего также средства отладки, облегчающие обнаружение ошибок и устранение проблем.

СОВЕТ

Самым быстрым способом приступить к написанию, компиляции и выполнению программ на C++ может быть использование удаленной интегрированной среды разработки, работающей через браузер. Один из таких инструментов доступен по адресу http://www.tutorialspoint.com/compile_cpp_online.php.

Кроме того, вы можете установить на свой компьютер множество бесплатных интегрированных сред разработки и компиляторов C++. Наиболее популярные из них — Microsoft Visual Studio Express для Windows и GNU C++ Compiler (называемый также g++) для Linux. Если вы программируете на Linux, то можете установить бесплатную интегрированную среду разработки Eclipse для разработки приложений C++ с использованием компилятора g++.

РЕКОМЕНДУЕТСЯ

Сохраняйте свои файлы исходного кода в файлах с расширением `.cpp`.

Используйте для создания исходного кода простой текстовый редактор либо интегрированную среду разработки.

НЕ РЕКОМЕНДУЕТСЯ

Не используйте расширение `.c` для файлов с исходными текстами, поскольку большинство компиляторов рассматривают такие файлы как содержащие код на языке C, а не C++.

Не используйте сложные редакторы текста, поскольку они зачастую добавляют собственную разметку в текст кода.

Создание первого приложения на C++

Теперь, когда вы знаете о том, какие есть инструментальные средства и как создаются программы, пришло время создать первое приложение C++, которое по традиции выводит на экран текст Hello World! (“Привет, мир!”).

Если вы программируете в Linux, воспользуйтесь простым текстовым редактором (я в Ubuntu использую gedit) для создания `.cpp`-файла с содержимым, показанным в листинге 1.1.

Если вы работаете под управлением операционной системы Windows и используете IDE Microsoft Visual Studio, то можете следовать описанным ниже шагам.

1. Создайте проект, используя команду `File⇒Create⇒Project` (Файл⇒Создать⇒Проект).
2. Выберите тип проекта Win32 Console Application (Консольное приложение Win32) и назовите свой проект Hello. Щелкните на кнопке OK.
3. В окне настроек проекта снимите флажок Precompiled Headers (Предварительно скомпилированный заголовок). Щелкните на кнопке Finish (Готово).
4. Замените автоматически созданное содержимое в файле `Hello.cpp` фрагментом кода, представленным в листинге 1.1.

ЛИСТИНГ 1.1. Программа Hello World (файл Hello.cpp)

```
1: #include <iostream>
2:
3: int main()
4: {
5:     std::cout << "Hello World!" << std::endl;
6:     return 0;
7: }
```

Это простое приложение всего лишь выводит на экран строку, используя оператор `std::cout`. Оператор `std::endl` указывает объекту потока `cout` закончить вывод строки переходом на новую строку, а оператор `return 0` обеспечивает завершение работы приложения и возврат операционной системе кода 0.

ВНИМАНИЕ!

Помните, дьявол — в деталях, а значит, код необходимо вводить абсолютно точно так же, как он представлен в листинге. Компиляторы известны своей придирчивостью. Если вы по ошибке поместите в конце оператора двоеточие там, где ожидается точка с запятой, то получите сообщение о неудавшейся компиляции с длинным пояснение, почему!

Построение и запуск вашего первого приложения C++

Работая под управлением Linux, откройте терминал и перейдите в каталог, содержащий файл `Hello.cpp`. Вызовите компилятор `g++` и компоновщик, используя следующую командную строку:

```
g++ -o hello Hello.cpp
```

Эта команда приказывает компилятору `g++` создать выполнимый файл `hello` путем компиляции исходного файла C++ `Hello.cpp`.

Если вы используете Microsoft Visual Studio в Windows, нажмите для запуска программы непосредственно в интегрированной среде разработки комбинацию клавиш `<Ctrl+F5>`. В результате программа будет откомпилирована, скомпонована и запущена на выполнение. Эти же этапы можно пройти индивидуально.

1. Щелкните правой кнопкой мыши на проекте и в появившемся контекстном меню выберите пункт `Build`, чтобы создать выполнимый файл.
2. Используя приглашение ко вводу команд, перейдите по пути выполнимого файла (обычно это каталог `Debug` папки проекта).
3. Запустите приложение, введя имя его выполнимого файла.

В среде разработки Microsoft Visual Studio ваша программа будет выглядеть примерно так, как показано на рис. 1.1.

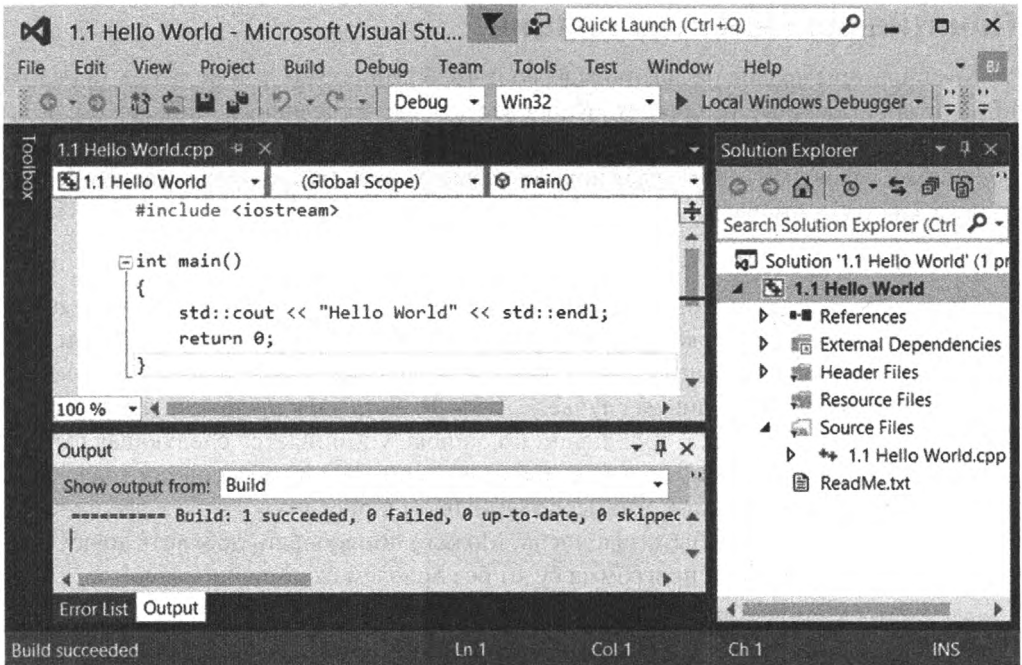


Рис. 1.1. Пример программы C++ “Hello World” в среде разработки Microsoft Visual Studio

Запуск файла `./hello` в Linux или `Hello.exe` на Windows даст следующий вывод:
Hello World!

Поздравляю! Вы начали свой путь к изучению одного из самых популярных и мощных языков программирования!

Значение стандарта C++ ISO

Как можно заметить, соответствие стандарту позволяет компилировать и выполнять фрагмент кода из листинга 1.1 на нескольких платформах или операционных системах — это преимущество соответствующих стандарту компиляторов C++. Таким образом, если необходимо создать продукт, который способен выполняться в операционной системе как Windows, так и Linux, например, то совместимые со стандартом практики программирования (которые не подразумевают использование семантики или компилятора, специфического для конкретной платформы) предоставят недорогой способ завоевать более широкую аудиторию пользователей без необходимости создавать специальную версию программы для каждой среды. Это, безусловно, прекрасно подходит для приложений, которые не нуждаются в частом взаимодействии на уровне операционной системы.

Понятие ошибок компиляции

Компиляторы крайне педантичны в своих требованиях, но, тем не менее, предпринимают определенные усилия, чтобы оповестить вас о сделанных ошибках. Если вы столкнулись с проблемой при компиляции приложения в листинге 1.1, то сообщение об ошибке, вероятнее всего, будет похоже на следующее (автор преднамеренно убрал точку с запятой в строке 5):

```
hello.cpp(6): error C2143: syntax error : missing ';' before 'return'
```

Это сообщение об ошибке от компилятора Visual C++ весьма описательно: в нем указываются имя файла, в котором содержится ошибка, номер строки (в данном случае — 6), в которой пропущена точка с запятой, и описание самой ошибки, предваряемое номером ошибки (в данном случае — C2143). Хотя знак препинания был удален из строки 5 кода примера, в сообщении об ошибке упоминается следующая строка, поскольку для компилятора ошибка стала очевидной, только когда он проанализировал оператор `return` и понял, что перед переходом к оператору `return` предыдущая инструкция должна была быть закончена. Можете попробовать добавить точку с запятой в начале строки 6, и программа будет без проблем откомпилирована!

ПРИМЕЧАНИЕ

В C++ конец строки не считается автоматически концом инструкции, как в некоторых других языках, таких как VBScript.

В C++ инструкция может распространяться на несколько строк кода. Можно также разместить несколько инструкций в одной строке, заканчивая каждую из них точкой с запятой.

Что нового в C++

Если вы опытный программист C++, то, вероятно, уже обратили внимание на то, что в примере программы C++ из листинга 1.1 за последние, пожалуй, десятилетия не изменился ни один бит. Но хотя язык C++ остается полностью обратно совместимым с предыдущими версиями языка, на самом деле не так давно было проделано очень много работы, чтобы упростить его использование.

Последние крупные обновления языка были выпущены как часть стандарта ISO, ратифицированного в 2011 году, который программисты называют “C++11”. C++14, выпущенный в 2014 году, содержит в основном не столь значительные улучшения и исправления C++11.

Такое средство, как ключевое слово `auto`, введенное в C++11, позволяет определить переменную, тип которой компилятор выводит автоматически, позволяя компактно записать многословные объявления (например, итераторов) и не нарушая безопасность типов. C++14 добавляет эту возможность для возвращаемых значений функций. *Лямбда-функции* — это функции без имени. Они позволяют писать компактные объекты функций без длинных определений класса, значительно сокращая строки кода. Стандарт C++ обещал программистам возможность писать переносимые, многопоточные и соответствующие стандарту приложения C++. Эти приложения при правильном

построении поддерживают парадигму параллельного выполнения и хорошо позиционируются как масштабируемые по производительности, когда пользователь наращивает возможности своих аппаратных средств, увеличивая количество ядер процессора. Это лишь некоторые из многих преимуществ языка C++, обсуждаемых в этой книге.

Новые возможности языка, ожидаемые в будущем стандарте, именуемом “C++17”, рассматриваются на занятии 29, “Что дальше”.

Резюме

На этом занятии вы узнали, как написать, откомпилировать, скомпоновать и выполнить свою первую программу C++. Здесь приведен также краткий обзор развития языка C++ и продемонстрирована эффективность стандарта на примере того, как одна и та же программа может быть откомпилирована с использованием разных компиляторов на разных операционных системах.

Вопросы и ответы

■ Могу ли я игнорировать предупреждающие сообщения компилятора?

В некоторых случаях компиляторы выдают предупреждающие сообщения. Предупреждения отличаются от ошибок тем, что рассматриваемая строка синтаксически правильна и вполне компилируема. Но, возможно, есть лучший способ написать данный код, и хорошие компиляторы выдают предупреждение об этом с рекомендацией по исправлению.

Предложенное исправление может означать более безопасный способ программирования или способ, позволяющий вашему приложению работать с символами и буквами не латинских языков. Вы должны учесть эти предупреждения и соответствующим образом улучшить свою программу. Не игнорируйте предупреждающие сообщения, если не уверены абсолютно, что они ошибочны.

■ Чем интерпретируемый язык отличается от компилируемого?

Интерпретируемыми являются такие языки, как Windows Script. У них нет этапа компиляции. Интерпретируемый язык использует интерпретатор, который читает текстовый файл сценария (код) и выполняет желаемые действия. Поэтому на машине, на которой должен быть выполнен сценарий, необходимо установить интерпретатор; следовательно, страдает производительность, поскольку интерпретатор работает как транслятор времени выполнения, расположенный между написанным кодом и микропроцессором.

■ Что такое ошибки времени выполнения и чем они отличаются от ошибок времени компиляции?

Ошибки, которые появляются при выполнении приложения, называются *ошибками времени выполнения* (runtime error). Возможно, вам встречалось сообщение “Access Violation” в старых версиях Windows, являющееся оповещением об ошибке времени выполнения программы. Сообщения об ошибках компиляции не доходят до конечного пользователя и являются свидетельством синтаксических проблем; они не позволяют программисту создать выполнимый файл.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. В чем разница между интерпретатором и компилятором?
2. Что делает компоновщик?
3. Каковы этапы обычного цикла разработки?

Упражнения

1. Рассмотрите следующую программу и попытайтесь предположить, что она делает, не запуская ее:

```
1: #include <iostream>
2: int main()
3: {
4:     int x = 8;
5:     int y = 6;
6:     std::cout << std::endl;
7:     std::cout << x-y << " " << x*y << " " << x+y;
8:     std::cout << std::endl;
9:     return 0;
10: }
```

2. Введите программу из упражнения 1, а затем откомпилируйте и скомпонуйте ее. Что она делает? Она делает то, что вы предполагали?
3. Где ошибка в следующей программе?

```
1: include <iostream>
2: int main()
3: {
4:     std::cout << "Hello Buggy World \n";
5:     return 0;
6: }
```

4. Исправьте ошибку в программе из упражнения 3, откомпилируйте, скомпонуйте и запустите ее. Что она делает?

ЗАНЯТИЕ 2

Структура программы на C++

Программы C++ состоят из классов (которые включают функции-члены и данные-члены), функций, переменных и других элементов. Большая часть данной книги посвящена подробному описанию этих элементов и их взаимодействию в программе, чтобы продемонстрировать, как программа работает в целом.

На этом занятии...

- Части программы C++
- Взаимодействие частей
- Что такое функция и что она делает
- Простые операции ввода и вывода

Части программы Hello World

Ваша первая программа C++ (занятие 1, “Первые шаги”) всего лишь выводила на экран простое приветствие Hello World. Тем не менее в ней содержатся некоторые из наиболее важных фундаментальных составляющих программы C++. Давайте воспользуемся листингом 2.1 как отправной точкой для анализа компонентов, содержащихся во всех программах C++.

ЛИСТИНГ 2.1. Файл HelloWorldAnalysis.cpp: анализ простой программы C++

```
1: // Директива препроцессора, подключающая заголовочный файл iostream
2: #include <iostream>
3:
4: // Начало программы: блок функции main()
5: int main()
6: {
7:     /* Вывод на экран */
8:     std::cout << "Hello World" << std::endl;
9:
10:    // Возврат значения операционной системе
11:    return 0;
12: }
```

Эту программу C++ можно грубо разделить на две части: директивы препроцессора, которые начинаются с символа #, и основную часть, которая начинается с `int main()`.

ПРИМЕЧАНИЕ

Строки 1, 4, 7 и 10, начинающиеся с символов `//` или `/*`, являются комментариями и игнорируются компилятором. Комментарии предназначены для чтения людьми, а не компилятором.

Более подробная информация о комментариях приведена в следующем разделе.

Директива препроцессора #include

Как и предполагает его название, *препроцессор* (preprocessor) — это инструмент, запускающийся перед фактическим началом компиляции. *Директивы препроцессора* (preprocessor directive) — это команды препроцессору, которые всегда начинаются со знака “диз” (#). В строке 2 листинга 2.1 директива `#include <имя_файла>` требует от препроцессора взять содержимое файла (в данном случае — `iostream`) и включить его вместо строки, в которой расположена директива. `iostream` — это стандартный заголовочный файл, который включается потому, что он содержит определение объекта потока `cout`, используемого в строке 8 для вывода на экран слов Hello World. Другими словами, компилятор смог откомпилировать строку 8, содержащую выражение `std::cout`, только потому, что мы заставили препроцессор включать определение объекта потока `cout` в строке 2.

ПРИМЕЧАНИЕ

В профессиональных приложениях C++ включаются не только стандартные заголовочные файлы, но и разработанные программистом. Сложные приложения, как правило, состоят из нескольких исходных файлов, причем одни из них должны включать другие. Так, если некоторый объект, объявленный в файле FileA, должен использоваться в файле FileB, то первый файл необходимо включить в последний. Обычно для этого в файл FileB помещают директиву `#include`:

```
#include "...путь к файлу FileA\FileA"
```

При включении самодельного заголовочного файла мы используем кавычки, а не угловые скобки. Угловые скобки (`<>`) обычно используются при включении стандартных заголовочных файлов.

Тело программы — функция `main()`

После директив препроцессора следует тело программы, расположенное в функции `main()`. Выполнение программ на языке C++ всегда начинается с функции `main()`. Согласно стандарту перед функцией `main()` указывается тип `int`. Тип `int` в данном случае — это тип возвращаемого значения функции `main()`.

ПРИМЕЧАНИЕ

Во многих приложениях C++ можно найти вариант функции `main()`, выглядящий следующим образом:

```
int main(int argc, char* argv[])
```

Это объявление совместимо со стандартом и вполне приемлемо, поскольку функция `main()` возвращает тип `int`, а содержимое круглых скобок — это *аргументы* (*argument*), передаваемые программе. Такая программа позволяет пользователю запускать ее с аргументами командной строки, например как

```
program.exe /DoSomethingSpecific
```

`/DoSomethingSpecific` — это аргумент данной программы, передаваемый операционной системой в качестве параметра для обработки в функции `main()`.

Рассмотрим строку 8, фактически выполняющую задачу этой программы.

```
std::cout << "Hello World" << std::endl;
```

`cout` (“console-out” (вывод на консоль); произносится как see-out (си-аут)), является инструкцией, фактически выводящей на экран строку `Hello World`. `cout` — это *поток*, определенный в *пространстве имен* `std` (поэтому и `std::cout`), а то, что мы делаем, — это помещение текстовой строки `Hello World` в данный поток с использованием оператора вывода (или вставки) в поток `<<`. Выражение `std::endl` используется для завершения строки, а его вывод в поток эквивалентен вставке символа возврата каретки. Обратите внимание: *оператор вывода в поток* (*stream insertion operator*) используется каждый раз, когда в поток нужно вывести новый элемент.

Преимущество потоков C++ заключается в одинаковой семантике, используемой потоками разного типа. В результате различные операции, осуществляемые с одним и тем же текстом, например вывод в файл, а не на консоль, выглядят одинаково и используют один и тот же оператор `<<`, только для `std::fstream` вместо `std::cout`. Таким образом, работа с потоками становится интуитивно понятной и, когда вы прибываете к одному потоку (такому, как `cout`, выводящему текст на консоль), то без проблем можете работать с другими (такими, как имеющие тип `fstream`, и записывающие текстовые файлы на диск).

Более подробная информация о потоках рассматривается на занятии 27, “Применение потоков для ввода и вывода”.

ПРИМЕЧАНИЕ

Фактический текст, заключенный в кавычки ("Hello World"), называется *строковым литералом* (string literal).

Возврат значения

Функции в языке C++ должны возвращать значение, если иное не указано явным образом. `main()` — это функция, всегда и обязательно возвращающая целое число. Это целочисленное значение возвращается операционной системе и, в зависимости от характера вашего приложения, может быть очень полезным, поскольку большинство операционных систем предусматривает для других приложений возможность обратиться к возвращенному значению. Не так уж и редко одно приложение запускает другое, и родительскому приложению (запустившему дочернее) желательно знать, закончило ли дочернее приложение свою задачу успешно. Программист может использовать возвращаемое значение функции `main()` для передачи родительскому приложению сообщения об успехе или неудаче.

ПРИМЕЧАНИЕ

Традиционно программисты возвращают значение 0 в случае успеха и -1 в случае ошибки. Однако тип `int` (целое число) возвращаемого значения обеспечивает разработчику, в пределах диапазона доступных значений, достаточную гибкость для передачи множества различных состояний успеха или неудачи.

ВНИМАНИЕ!

Язык C++ чувствителен к регистру. Поэтому готовьтесь к неудаче компиляции, если напишете `Int` вместо `int`, `Void` вместо `void` или `Std::Cout` вместо `std::cout`.

Концепция пространств имен

Причина использования в программе синтаксиса `std::cout`, а не просто `cout`, в том, что используемый элемент (`cout`) находится в стандартном пространстве имен (`std`).

Так что же такое *пространство имен* (namespace)?

Предположим, вы не использовали спецификатор пространства имен и обратились к объекту `cout`, который объявлен в двух известных компилятору местах. Какой из них компилятор должен использовать? Безусловно, это приведет к конфликту и неудаче компиляции. Вот где оказываются полезными пространства имен. Пространства имен — это имена, присвоенные частям кода, помогающие снизить вероятность конфликтов имен. При вызове `std::cout` вы указываете компилятору использовать именно тот объект `cout`, который доступен в пространстве имен `std`.

ПРИМЕЧАНИЕ

Пространство имен `std` (произносится как “standard” (стандарт)) используется для вызова функций, потоков и утилит, которые были утверждены ISO Standards Committee.

Многие программисты находят утомительным регулярный ввод при наборе исходного текста спецификатора `std` при использовании имени `cout` и других подобных средств, содержащихся в том же пространстве имен. Объявление `using namespace`, представленное в листинге 2.2, позволит избежать этого повторения.

ЛИСТИНГ 2.2. Объявление `using namespace`

```
1: // Директива препроцессора
2: #include <iostream>
3:
4: // Начало программы
5: int main()
6: {
7:     // Указать компилятору пространство имен для поиска
8:     using namespace std;
9:
10:    /* Вывод на экран с использованием std::cout */
11:    cout << "Hello World" << endl;
12:
13:    // Возврат значения операционной системе
14:    return 0;
15: }
```

Анализ

Обратите внимание на строку 8. Сообщив компилятору, что предполагается использовать пространство имен `std`, можно не указывать пространство имен в строке 11 явно при использовании выражений `std::cout` и `std::endl`.

Листинг 2.3 содержит более ограничительный вариант кода листинга 2.2. Здесь подключается не все пространство имен полностью, а только те его элементы, которые предстоит использовать.

ЛИСТИНГ 2.3. Другая демонстрация ключевого слова `using`

```
1: // Директива препроцессора
2: #include <iostream>
3:
4: // Начало программы
5: int main()
6: {
7:     using std::cout;
8:     using std::endl;
9:
10:    /* Вывод на экран с использованием cout */
11:    cout << "Hello World" << endl;
12:
13:    // Возврат значения операционной системе
14:    return 0;
15: }
```

Анализ

В листинге 2.3 строка 8 листинга 2.2 была заменена строками 7 и 8. Различие между инструкциями `using namespace std` и `using std::cout` заключается в том, что первая позволяет использовать все элементы пространства имен `std` без явного указания квалификатора `std::`. Удобство последней в том, что без необходимости устранять неоднозначность пространств имен явно можно использовать только выражения `std::cout` и `std::endl`.

Комментарии в коде C++

Строки 1, 4, 10 и 13 листинга 2.3 содержат текст на русском языке, но программа все равно компилируется. Они никак не влияют и на вывод программы. Такие строки называются *комментариями* (comment). Комментарии игнорируются компилятором и обычно используются программистами для пояснений в коде. Следовательно, они пишутся на человеческом языке (или профессиональном жаргоне).

- Символ `//` означает, что следующая далее строка — комментарий. Например:
`// Это комментарий`
- Текст, содержащийся между символами `/*` и `*/`, также является комментарием, даже если он занимает несколько строк:
`/* Это комментарий,
 занимающий две строки */`

ПРИМЕЧАНИЕ

Может показаться странным, зачем программисту объяснять собственный код. Однако большие программы создаются большим количеством программистов, каждый из которых работает над определенной частью кода, который должен быть понятен другим разработчикам. Хорошо написанные комментарии позволяют объяснить, что и почему делается именно так. Они выступают в качестве документации кода.

Заметим также, что программист уже через месяц-другой может быть не в состоянии вспомнить, что именно и зачем писал в том или ином месте исходного текста. Так что не экономьте на комментариях! Они нужны прежде всего вам самому.

РЕКОМЕНДУЕТСЯ

Добавляйте комментарии, объясняющие работу сложных алгоритмов и частей вашей программы.

Оформляйте комментарии в стиле, принятом вашим коллективом программистов.

НЕ РЕКОМЕНДУЕТСЯ

Не используйте комментарии для повторения или объяснения очевидного.

Не забывайте, что добавление комментариев не сделает запутанный код более понятным.

Не забывайте изменять комментарии при изменении кода.

Функции в C++

Функции (function) — это элементы, позволяющие разделить содержимое вашего приложения на функциональные модули, которые могут быть вызваны по вашему выбору. При вызове функция обычно возвращает значение вызывающей функции. Самая известная функция, конечно, — `int main()`. Она распознается компилятором как отправная точка приложения C++ и должна возвращать значение типа `int` (т.е. целое число).

У программиста всегда есть возможность, а как правило, и необходимость, создавать собственные функции. В листинге 2.4 приведено простое приложение, которое использует функцию для отображения текста на экране, используя `std::cout` с различными параметрами.

ЛИСТИНГ 2.4. Объявление, определение и вызов функции, демонстрирующей возможности `std::cout`

```
1: #include <iostream>
2: using namespace std;
3:
4: // Объявление функции
5: int DemoConsoleOutput();
6:
7: int main()
8: {
```

```
9:      // Вызов функции
10:     DemoConsoleOutput();
11:
12:     return 0;
13: }
14:
15: // Определение, т.е. реализация объявленной ранее функции
16: int DemoConsoleOutput()
17: {
18:     cout << "Простой строковый литерал" << endl;
19:     cout << "Запись числа пять: " << 5 << endl;
20:     cout << "Выполнение деления 10/5 = " << 10/5 << endl;
21:     cout << "Пи примерно равно 22/7 = " << 22/7 << endl;
22:     cout << "Более точно Пи равно 22/7 = " << 22.0/7 << endl;
23:
24:     return 0;
25: }
```

Результат

Простой строковый литерал
Запись числа пять: 5
Выполнение деления 10/5 = 2
Пи примерно равно 22/7 = 3
Более точно Пи равно 22/7 = 3.14286

Анализ

Интерес представляют строки 5, 10 и 16–25. В строке 5 находится *объявление функции* (function declaration), которое в основном указывает компилятору, что вы хотите создать функцию по имени `DemoConsoleOutput()`, возвращающую значение типа `int` (целое число). Именно из-за этого *объявления* компилятор соглашается откомпилировать строку 10, в которой функция `DemoConsoleOutput()` вызывается в функции `main()`. Компилятор считает, что где-то далее он встретит *определение функции* (function definition), и действительно встречается его позже, в строках 16–25.

Фактически эта функция демонстрирует возможности потока `cout`. Обратите внимание: она выводит не только текст, как в предыдущих примерах, но и результаты простых арифметических вычислений. Две строки, 21 и 22, отображают результат вычисления числа “пи” как $22/7$, но последний результат немного точнее просто потому, что при делении 22.0 на 7 вы указываете компилятору вычислить результат как вещественное число (тип `double` в терминах C++), а не как целое значение.

Обратите внимание, что функция должна вернуть целое число, и она возвращает значение `0`. Точно так же функция `main()` тоже возвращает значение `0`. Поскольку функция `main()` делегирует все свои действия функции `DemoConsoleOutput()`, имело бы смысл использовать возвращаемое ею значение для возврата значения из функции `main()`, как это сделано в листинге 2.5.

ЛИСТИНГ 2.5. Использование возвращаемого значения функции

```
1: #include <iostream>
2: using namespace std;
3:
4: // Объявление и определение функции
5: int DemoConsoleOutput()
6: {
7:     cout << "Простой строковый литерал" << endl;
8:     cout << "Запись числа пять: " << 5 << endl;
9:     cout << "Выполнение деления 10/5 = " << 10/5 << endl;
10:    cout << "Пи примерно равно 22/7 = " << 22/7 << endl;
11:    cout << "Более точно Пи равно 22/7 = " << 22.0/7 << endl;
12:
13:    return 0;
14: }
15:
16: int main()
17: {
18:     // Вызов функции с возвратом результата при выходе
19:     return DemoConsoleOutput();
20: }
```

Анализ

Вывод этого приложения такой же, как предыдущего. Небольшие изменения есть только в способе его получения. Поскольку функция определена (т.е. реализована) перед функцией `main()` в строке 5, ее дополнительное объявление уже не нужно. Современные компиляторы C++ понимают это как одновременное объявление и определение функции. Функция `main()` также немного короче. В строке 19 осуществляется вызов функции `DemoConsoleOutput()` и одновременно возврат ее возвращаемого значения при выходе из приложения.

ПРИМЕЧАНИЕ

В таких случаях, как здесь, когда функция не обязана принимать решение или возвращать сообщение об успехе или отказе, можно объявить функцию с типом возвращаемого значения `void`:

```
void DemoConsoleOutput()
```

Такая функция не может возвращать значение, и ее нельзя использовать для принятия решения.

Функции могут получать параметры, могут быть рекурсивными, содержать несколько операторов выхода, могут быть перегруженными, встраиваемыми и т.д. Эти концепции вводятся далее, на занятии 7, “Организация кода с помощью функций”.

Ввод-вывод с использованием потоков `std::cin` и `std::cout`

Ваш компьютер позволяет взаимодействовать с выполняющимися на нем приложениями разными способами, а также позволяет этим приложениям взаимодействовать с вами разными способами. Вы можете взаимодействовать с приложениями, используя клавиатуру или мышь. Информация может быть отображена на экране как текст или в виде сложной графики, может быть напечатана с помощью принтера на бумаге или просто сохранена в файловой системе для последующего использования. В этом разделе рассматривается простейший ввод и вывод информации в языке C++ — использование консоли для отображения и ввода информации.

Для работы с консолью используются потоки `std::cout` (для вывода простой текстовой информации) и `std::cin` (для чтения информации с консоли; как правило, с клавиатуры). Вы уже встречались с потоком `cout` при выводе слов `Hello World` на экран в листинге 2.1:

```
8:      std::cout << "Hello World" << std::endl;
```

В этой инструкции после имени потока `cout` идет оператор вывода `<<` (позволяющий вставить данные в поток вывода) с последующим подлежащим выводу строковым литералом `"Hello World"` и символом новой строки в виде выражения `std::endl`.

Применение потока `cin` также очень простое; он работает в паре с переменной, в которую следует поместить вводимые данные:

```
std::cin >> Переменная;
```

Таким образом, за потоком `cin` следуют *оператор извлечения значения* `>>` (данные извлекаются из входного потока) и переменная, в которую следует поместить считываемые данные. Если вводимые данные разделены пробелом, и их следует сохранить в двух разных переменных, можно использовать цепочку операторов:

```
std::cin >> Переменная1 >> Переменная2;
```

Обратите внимание на то, что поток `cin` применяется для ввода как текстовых, так и числовых данных, как показано в листинге 2.6.

ЛИСТИНГ 2.6. Использование потоков `cin` и `cout` для отображения числовых и текстовых данных, вводимых пользователем

```
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: int main()
6: {
7:     // Объявление переменной для хранения целого числа
8:     int inputNumber;
9:
```

```
10:     cout << "Введите целое число: ";
11:
12:     // Сохранить введенное пользователем целое число
13:     cin >> inputNumber;
14:
15:     // Аналогично текстовым данным
16:     cout << "Введите ваше имя: ";
17:     string inputName;
18:     cin >> inputName;
19:
20:     cout << inputName << " ввел " << inputNumber << endl;
21:
22:     return 0;
23: }
```

Результат

```
Введите целое число: 2017
Введите ваше имя: Siddhartha
Siddhartha ввел 2017
```

Анализ

В строке 8 переменная `inputNumber` объявляется как способная хранить данные типа `int`. В строке 10 пользователя просят ввести число, используя поток `cout`, а введенное значение сохраняется в целочисленной переменной с использованием потока `cin` в строке 13. То же самое повторяется при сохранении имени пользователя, которое, конечно, не может содержаться в целочисленной переменной. Для этого используется другой тип — `string`, как видно из строк 17 и 18. Именно поэтому, чтобы можно было использовать тип `string` в функции `main()`, в строке 2 была включена директива `#include <string>`. И наконец в строке 20 поток `cout` используется для отображения введенных имени и числа с промежуточным текстом, чтобы получить вывод `Siddhartha ввел 2017`.

Это очень простой пример ввода и вывода в C++. Не волнуйтесь, если концепция переменных пока что вам непонятна: подробно мы рассмотрим ее на следующем занятии.

ПРИМЕЧАНИЕ

Если я введу пару слов в качестве имени (например, `Siddhartha Rao`) при выполнении листинга 2.6, то поток `cin` все равно сохранит в строке только первое слово — `Siddhartha`. Чтобы вводить и сохранять строки целиком, следует использовать функцию `getline()` (которая будет рассмотрена на занятии 4, “Массивы и строки”, в листинге 4.7).

Резюме

Это занятие знакомит с основными частями простых программ C++. Здесь продемонстрировано, что такое функция `main()`, изложено введение в пространства имен и основы ввода и вывода на консоль. Вы будете использовать многие из них в каждой программе, которую пишете.

Вопросы и ответы

■ Что делает директива `#include`?

Это директива препроцессора, которая выполняется при вызове компилятора. Данная конкретная директива требует включить содержимое файла, имя которого указано в угловых скобках `<>` после нее, вместо текущей строки, как если бы оно было введено в этом месте исходного текста.

■ В чем разница между комментариями `//` и `/*`?

Комментарий после двойной косой черты (`//`) завершается в конце строки. Комментарий после косой черты со звездочкой (`/*`) продолжается до тех пор, пока не встретится завершающий знак комментария (`*/`). Комментарии двойной косой чертой называют также *однострочными комментариями*, а косой чертой со звездочкой — *многострочными комментариями*. Помните, что даже конец функции не завершает многострочный комментарий; его необходимо закрыть явно, в противном случае произойдет ошибка при компиляции.

■ Зачем программе нужны аргументы командной строки?

Чтобы дать пользователю возможность изменять поведение программы. Например, команда `ls` в Linux или `dir` в Windows позволяет просматривать содержимое текущего каталога или папки. Чтобы просмотреть файлы в другом каталоге, вы можете указать путь к ним, используя аргументы командной строки, как, например, в вызове `ls /` или `dir \`.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Что неправильно в объявлении `Int main()`?
2. Могут ли комментарии быть длиннее одной строки?

Упражнения

1. **Отладка.** Введите исходный текст программы и откомпилируйте ее. Почему она не компилируется? Как можно ее исправить?

```
1: #include <iostream>
2: void main()
3: {
4:     std::Cout << "Is there a bug here?";
5: }
```

2. Исправьте ошибки в программе из упражнения 1 и, перекомпилировав, запустите ее снова.
3. Измените листинг 2.4 так, чтобы продемонстрировать вычитание (используя оператор `-`) и умножение (используя оператор `*`).

ЗАНЯТИЕ 3

Использование переменных и констант

Переменные (variable) представляют собой средство, позволяющее программисту временно хранить данные в течение некоторого конечного времени. *Константа (constant)* — это средство, позволяющее программисту определить элемент, который не должен изменяться в процессе выполнения программы.

На этом занятии...

- Как объявить и определить переменные и константы
- Как присвоить значения переменным и манипулировать ими
- Как вывести значение переменной на экран
- Как использовать ключевые слова `auto` и `constexpr`

Что такое переменная

Прежде чем перейти к рассмотрению потребности в использовании переменных в языке программирования, сделаем небольшое отступление и рассмотрим, как компьютер хранит и обрабатывает данные.

Коротко о памяти и адресации

Все компьютеры, смартфоны и другие программируемые устройства имеют микропроцессор и определенный объем памяти для временного хранения, называемый *оперативной памятью* (Random Access Memory — RAM). Кроме того, многие устройства позволяют сохранять данные на долгосрочном запоминающем устройстве, таком как жесткий диск. Микропроцессор выполняет ваше приложение и использует при этом оперативную память для загрузки его бинарного кода, а также связанных с ним данных, включая те, которые отображаются на экране и вводятся пользователем.

Саму оперативную память, являющуюся областью хранения, можно сравнить с рядом шкафчиков в общежитии, каждый из которых имеет свой номер, т.е. адрес. Чтобы получить доступ к области памяти, скажем, к ее ячейке 578, процессор нужно с помощью специальной инструкции попросить выбрать оттуда значение или записать в нее значение.

Объявление переменных для получения доступа и использования памяти

Приведенные ниже примеры помогут понять, что такое переменные. Предположим, вы пишете программу для умножения двух чисел, предоставляемых пользователем. Пользователя просят ввести два значения — множитель и множимое, и каждое из этих значений необходимо хранить до момента умножения. В зависимости от того, что вы хотите делать с результатом умножения, вам может понадобиться хранить эти значения для более позднего использования в программе. Было бы слишком медленно (и программисты часто ошибались бы), если бы для хранения чисел нужно было помнить и записывать явные адреса областей памяти (такой, как номер ячейки 578), поскольку при этом приходилось бы постоянно помнить о том, где какие данные находятся, и заботиться о том, чтобы случайно не перезаписать уже хранящиеся в ячейках памяти данные другими.

При программировании на таких языках, как C++, для хранения значений определяют переменные. Определить переменную очень просто по такому шаблону:

```
Тип_переменной Имя_переменной;
```

или

```
Тип_переменной Имя_переменной = Начальное_значение;
```

Атрибут типа переменной указывает компилятору характер данных, которые могут храниться в этой переменной, и то, какое количество памяти компилятор должен резервировать для этого. Выбранное программистом имя переменной является более

осмысленной заменой адреса области в памяти, где хранится значение переменной. Если *Начальное_значение* не указано, вы не можете быть уверены в содержимом этой области памяти, что может быть плохо для программы. Поэтому, будучи необязательной, инициализация является хорошей практикой программирования. Листинг 3.1 демонстрирует объявление переменных, их инициализацию и использование в программе, которая умножает два числа, предоставленные пользователем.

ЛИСТИНГ 3.1. Использование переменных для хранения чисел и результата их умножения

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Программа для умножения двух чисел" << endl;
7:
8:     cout << "Введите первое число: ";
9:     int firstNumber = 0;
10:    cin >> firstNumber;
11:
12:    cout << "Введите второе число: ";
13:    int secondNumber = 0;
14:    cin >> secondNumber;
15:
16:    // Умножение двух чисел, сохранение результата в переменной
17:    int multiplicationResult = firstNumber*secondNumber;
18:
19:    // Вывод результата
20:    cout << firstNumber << " x " << secondNumber;
21:    cout << " = " << multiplicationResult << endl;
22:
23:    return 0;
24: }
```

Результат

```
Программа для умножения двух чисел
Введите первое число: 51
Введите второе число: 24
51 x 24 = 1224
```

Анализ

Это приложение просит пользователя ввести два числа, результат умножения которых затем выводит. Чтобы использовать введенные пользователем числа, следует сохранить их в памяти. Переменные `firstNumber` и `secondNumber`, объявленные в строках 9 и 13, решают задачу временного хранения введенных пользователем

целочисленных значений. Поток `std::cin` в строках 10 и 14 используется для получения введенных пользователем значений и их сохранения в двух целочисленных переменных. Поток `cout` в строке 21 используется для вывода результата на консоль.

Давайте проанализируем объявление переменной подробнее:

```
9:     int firstNumber = 0;
```

Эта строка объявляет переменную типа `int`, который означает целое число, с именем `firstNumber`. В качестве начального значения переменной присваивается нулевое значение.

Компилятор выполняет задачу по отображению имени этой (и прочих объявленных в программе) переменной на область памяти и вместо вас заботится о сохранении соответствующей информации об адресах памяти. Таким образом, программист работает с понятными человеку именами, предоставляя компилятору работу с непосредственными адресами памяти и создание команд для работы микропроцессора с оперативной памятью.

ВНИМАНИЕ!

Хорошие имена переменных важны для написания хорошего, понятного и удобного в сопровождении кода.

Имена переменных в C++ могут состоять из букв и цифр, но не могут начинаться с цифр, а также содержать пробелы и арифметические операторы (такие, как `+`, `-` и т.п.).

Именами переменных не могут быть зарезервированные ключевые слова. Например, переменная по имени `return` приведет к ошибке при компиляции.

В именах переменных можно использовать символ подчеркивания, который позволяет создавать более понятные, самодокументируемые имена переменных.

Объявление и инициализация нескольких переменных одного типа

Переменные `firstNumber`, `secondNumber` и `multiplicationResult` в листинге 3.1 имеют одинаковый тип (целое число), но объявляются в трех отдельных строках. При желании можно было бы уплотнить объявление этих трех переменных до одной строки кода, которая выглядела бы следующим образом:

```
int firstNumber = 0, secondNumber = 0, multiplicationResult = 0;
```

ПРИМЕЧАНИЕ

Как видите, язык C++ позволяет объявлять сразу несколько переменных одного типа, а также объявлять переменные в начале функции. Но все же объявление переменной непосредственно перед ее первым применением зачастую оказывается более удобным, поскольку делает код более удобочитаемым и вам не требуется долго искать тип переменной — он указан возле места ее первого применения.

ВНИМАНИЕ!

Данные, хранимые в переменных, находятся в оперативной памяти. Они теряются при отключении компьютера или завершении работы приложения, если программист не сохраняет их специально на постоянном носителе данных наподобие жесткого диска.

Более подробно о сохранении данных в файле на диске вы узнаете на занятии 27, "Применение потоков для ввода и вывода".

Понятие области видимости переменной

У обычных переменных, подобных рассмотренным выше, есть точно определенная *область видимости* (scope), в пределах которой к ним можно обращаться и использовать хранящиеся в них данные. При использовании вне области видимости имя переменной не будет распознано компилятором, и ваша программа не будет скомпилирована. Вне своей области видимости переменная представляет собой неопознанный объект, о котором компилятор ничего не знает.

Чтобы лучше понять концепцию области видимости переменной, реорганизуем программу в листинге 3.1 в функцию `MultiplyNumbers()`, которая умножает два числа и возвращает результат (листинг 3.2).

ЛИСТИНГ 3.2. Демонстрация области видимости переменных

```
1: #include <iostream>
2: using namespace std;
3:
4: void MultiplyNumbers()
5: {
6:     cout << "Введите первое число: ";
7:     int firstNumber = 0;
8:     cin >> firstNumber;
9:
10:    cout << "Введите второе число: ";
11:    int secondNumber = 0;
12:    cin >> secondNumber;
13:
14:    // Умножение двух чисел, сохранение результата в переменной
15:    int multiplicationResult = firstNumber * secondNumber;
16:
17:    // Вывод результата
18:    cout << firstNumber << " x " << secondNumber;
19:    cout << " = " << multiplicationResult << endl;
20: }
21: int main()
22: {
23:     cout << "Программа для умножения двух чисел" << endl;
24:
25:     // Вызов функции, выполняющей всю работу
26:     MultiplyNumbers();
```

```
27:
28:     // cout << firstNumber << " x " << secondNumber;
29:     // cout << " = " << multiplicationResult << endl;
30:
31:     return 0;
32: }
```

Результат

Программа для умножения двух чисел

Введите первое число: **51**

Введите второе число: **24**

51 x 24 = 1224

Анализ

Код из листинга 3.2 выполняет те же действия, что и код из листинга 3.1, генерируя тот же вывод. Единственное различие состоит в том, что все действия перенесены в функцию `MultiplyNumbers()`, вызываемую функцией `main()`. Обратите внимание на то, что переменные `firstNumber` и `secondNumber` не могут использоваться за пределами функции `MultiplyNumbers()`. Если убрать комментарий из строки 28 или 29 в функции `main()`, то компиляция потерпит неудачу с наиболее вероятной причиной `undeclared identifier` (необъявленный идентификатор).

Дело в том, что переменные `firstNumber` и `secondNumber` имеют локальную область видимости, а значит, она ограничивается той функцией, в которой они объявлены, в данном случае — функцией `MultiplyNumbers()`. *Локальная переменная* (local variable) может использоваться в функции от места объявления переменной до конца функции. Фигурная скобка `{}`, означающая конец функции, означает также конец области видимости объявленных в ней переменных. Когда функция заканчивается, все ее локальные переменные уничтожаются, а занимаемая ими память освобождается.

При компиляции объявленные в пределах функции `MultiplyNumbers()` переменные уничтожаются по завершении функции и, если они используются в функции `main()`, происходит ошибка, поскольку эти переменные в ней не были объявлены.

ВНИМАНИЕ!

Если в функции `main()` объявить другой набор переменных с теми же именами, то не надейтесь, что они будут содержать те же значения, которые, возможно, были присвоены им в функции `MultiplyNumbers()`. Компилятор рассматривает переменные в функции `main()` как независимые сущности, даже если их имена совпадают с именами переменных, объявленных в другой функции, поскольку существование этих переменных ограничено их областями видимости.

Глобальные переменные

Если бы переменные, используемые в функции `MultiplyNumbers()` листинга 3.2, были объявлены не в ней, а за ее пределами, то они были бы пригодны для использования и в функции `main()`, и в функции `MultiplyNumbers()`. Листинг 3.3 демонстрирует *глобальные переменные* (global variable), имеющие самую широкую область видимости в программе.

ЛИСТИНГ 3.3. Использование глобальных переменных

```
1: #include <iostream>
2: using namespace std;
3:
4: // Три глобальные целочисленные переменные
5: int firstNumber = 0;
6: int secondNumber = 0;
7: int multiplicationResult = 0;
8:
9: void MultiplyNumbers()
10: {
11:     cout << "Введите первое число: ";
12:     cin >> firstNumber;
13:
14:     cout << "Введите второе число: ";
15:     cin >> secondNumber;
16:
17:     // Умножение двух чисел, сохранение результата в переменной
18:     multiplicationResult = firstNumber * secondNumber;
19:
20:     // Вывод результата
21:     cout << "Вывод из MultiplyNumbers(): ";
22:     cout << firstNumber << " x " << secondNumber;
23:     cout << " = " << multiplicationResult << endl;
24: }
25: int main()
26: {
27:     cout << "Программа для умножения двух чисел" << endl;
28:
29:     // Вызов функции, выполняющей всю работу
30:     MultiplyNumbers();
31:
32:     cout << "Вывод из main(): ";
33:
34:     // Теперь эта строка компилируется и работает!
35:     cout << firstNumber << " x " << secondNumber;
36:     cout << " = " << multiplicationResult << endl;
37:
38:     return 0;
39: }
```

Результат

```
Программа для умножения двух чисел
Введите первое число: 51
Введите второе число: 19
Вывод из MultiplyNumbers(): 51 x 19 = 969
Вывод из main(): 51 x 19 = 969
```

Анализ

Листинг 3.3 выводит результат умножения в двух функциях, причем переменные `firstNumber`, `secondNumber` и `multiplicationResult` объявлены за их пределами. Эти переменные *глобальны* (`global`), поскольку были объявлены в строках 5–7, вне области видимости всех функций. Обратите внимание на строки 22 и 35, которые используют эти переменные и отображают их значения. Обратите особое внимание на то, что переменная `multiplicationResult` сначала получает значение в функции `MultiplyNumbers()`, которое потом повторно используется в функции `main()`.

ВНИМАНИЕ!

Безосновательное использование глобальных переменных обычно считается плохой практикой программирования. Это связано с тем, что значение глобальной переменной может быть присвоено в любой функции, и это значение может оказаться непредсказуемым, в особенности если разные функциональные модули разрабатываются разными программистами группы или выполняются в разных потоках. Элегантный способ получения результата умножения в листинге 3.3 функцией `main()` без применения глобальных переменных — использовать возврат результата умножения функцией `MultiplyNumbers()`.

Соглашения об именовании

Если вы еще не обратили на это внимания, заметьте, что мы дали функции имя `MultiplyNumbers()`, в котором каждое слово начинается с прописной буквы (стиль языка программирования Pascal), в то время как переменные `firstNumber`, `secondNumber` и `multiplicationResult` имеют имена, первое слово которых начинается с буквы в нижнем регистре (так называемый “верблюжий стиль”). Именно этому соглашению мы следуем дальше в этой книге — в именах переменных используется “верблюжий стиль”, в то время как другие объекты, такие как имена функций, следуют стилю Pascal.

Вы можете встретить код C++, в котором имя переменной предваряется символами, описывающими тип переменной. Это соглашение называется *венгерской нотацией* и часто используется в программировании в Windows. Так, переменная `firstNumber` в венгерской нотации имела бы имя `iFirstNumber`, где префикс `i` означает тип `int`. Глобальная переменная имела бы имя `g_iFirstNumber`. Венгерская нотация в последние годы теряет популярность, частично из-за улучшения интегрированных сред разработки, при необходимости отображающих тип переменной, например при наведении на них указателя мыши.

Ниже приводятся часто встречающиеся примеры плохих имен переменных:

```
int i = 0;
bool b = false;
```

Имя переменной должно указывать ее предназначение, так что эти переменные было бы лучше объявить следующим образом:

```
int totalCash = 0;
bool isLampOn = false;
```

ВНИМАНИЕ!

Соглашения об именовании используются для повышения удобочитаемости кода для программистов, а не для компиляторов. Поэтому с умом выбирайте соглашение, которое лучше всего подходит вам, и последовательно его используйте.

Работая в команде, имеет смысл определиться с соглашением об именовании еще до начала работы над новым проектом. Работая над существующим проектом, принимайте и следуйте используемому в команде соглашению, чтобы новый код по-прежнему оставался понятным для других программистов.

Распространенные типы переменных, поддерживаемые компилятором C++

В большинстве примеров до сих пор определялись переменные типа `int`, т.е. целые числа. Однако в распоряжении программистов C++ есть множество других фундаментальных типов переменных, непосредственно поддерживаемых компилятором. Выбор правильного типа переменной так же важен, как и выбор правильных инструментов для работы! Крестообразная отвертка не подойдет для работы с шурупом под плоскую, точно так же целое число без знака не может использоваться для хранения отрицательного значения! Типы переменных и характер данных, которые они могут содержать, приведены в табл. 3.1. Эта информация очень важна при написании эффективных и надежных программ C++.

ТАБЛИЦА 3.1. Типы переменных

Тип	Типичный диапазон значений
bool	true (истина) или false (ложь)
char	256 символьных значений
unsigned short int	От 0 до 65 535
short int	От -32 768 до 32 767
unsigned long int	От 0 до 4 294 967 295
long int	От -2 147 483 648 до 2 147 483 647
unsigned long long	От 0 до 18 446 744 073 709 551 615
long long	От -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
int (16 бит)	От -32 768 до 32 767

Тип	Типичный диапазон значений
int (32 бита)	От -2 147 483 648 до 2 147 483 647
unsigned int (16 бит)	От 0 до 65 535
unsigned int (32 бита)	От 0 до 4 294 967 295
float	От 1.2e-38 до 3.4e38
double	От 2.2e-308 до 1.8e308

Более подробная информация о важнейших типах приведена в следующих разделах.

Использование типа `bool` для хранения логических значений

Язык C++ предоставляет тип, специально созданный для хранения логических значений `true` или `false` (оба являются зарезервированными ключевыми словами C++). Этот тип особенно полезен при хранении настроек и флагов, которые могут быть установлены или сброшены, существовать или отсутствовать, быть доступными или недоступными.

Типичное объявление инициализированной логической переменной имеет следующий вид:

```
bool alwaysOnTop = false;
```

Выражение, вычисляющее значение логического типа, может иметь следующий вид:

```
bool deleteFile = (userSelection == "yes");  
// Истинно, если переменная userSelection содержит "yes",  
// в противном случае – ложно
```

Условные выражения рассматриваются на занятии 5, “Выражения, инструкции и операторы”.

Использование типа `char` для хранения символьных значений

Тип `char` используется для хранения одного символа. Типичное объявление показано ниже.

```
char userInput = 'Y'; // Инициализация символом 'Y'
```

Обратите внимание, что память состоит из битов и байтов. Биты могут содержать значения 0 и 1, а байты могут хранить числовые представления, использующие эти биты. Таким образом, работая или присваивая символьные данные, как показано в примере, компилятор преобразует символы в числовое представление, которое может быть помещено в память. Числовое представление латинских символов A–Z, a–z,

чисел 0–9, некоторых специальных клавиш (например,) и специальных символов (таких, как “забой”), было стандартизовано в американском коде обмена информацией (American Standard Code for Information Interchange), именуемом также ASCII.

Вы можете открыть таблицу в приложении Г, “Коды ASCII”, и увидеть, что символ 'Y', присвоенный переменной `userInput`, имеет согласно стандарту ASCII десятичное значение 89. Таким образом, компилятор просто сохраняет значение 89 в области памяти, зарезервированной для переменной `userInput`.

Концепция знаковых и беззнаковых целых чисел

Знак (sign) делает число положительным или отрицательным. Все числа, с которыми работает компьютер, хранятся в памяти просто как биты и байты. Область памяти размером 1 байт содержит 8 битов. Каждый бит может содержать значение 0 или 1 (т.е. хранить только одно из этих двух значений). Таким образом, область памяти размером 1 байт может хранить одно из 2 в степени 8, т.е. 256 разных значений. Аналогично область памяти размером 16 битов может хранить одно из 2 в степени 16 разных значений, т.е. одно из 65 536 уникальных значений.

Если эти значения должны быть беззнаковыми, т.е. память может содержать только положительные значения, то один байт мог бы содержать целочисленные значения в пределах от 0 до 255, а два байта будут содержать значения в пределах от 0 до 65 535 соответственно. Загляните в табл. 3.1 и обратите внимание на то, что тип `unsigned short int`, который поддерживает этот диапазон, занимает в памяти 16 бит. Таким образом, положительные значения в битах и байтах очень просто представить схематически (рис. 3.1).

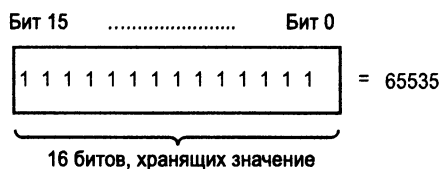


РИС. 3.1. Организация битов в 16-разрядном коротком беззнаковом целом числе

Но как же представить в этой же области отрицательные числа? Один из способов — “пожертвовать” одним из разрядов для хранения знака, который указывал бы, положительное или отрицательное значение содержится в других битах (рис. 3.2). Такой знаковый разряд имеет смысл делать *самым старшим битом* (Most-Significant-Bit — MSB) для согласованности знаковых и беззнаковых чисел, например чтобы у обоих типов *самый младший бит* (Least-Significant-Bit — LSB) указывал нечетность числа. Если старший бит содержит информацию о знаке, предполагается, что значение 0 означает положительное число, а значение 1 — отрицательное. Другие биты содержат абсолютное значение числа.

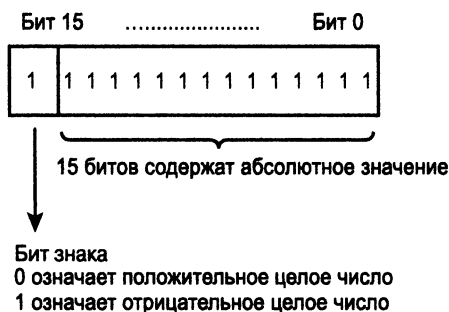


РИС. 3.2. Организация битов в 16-битовом коротком знаковом целом числе

Таким образом, занимающее 8 битов знаковое число может содержать значения в пределах от -128 до 127 , а занимающее 16 битов — значения в пределах от $-32\,768$ до $32\,767$. Еще раз посмотрите на табл. 3.1 и обратите внимание на то, что тип `short int` (знаковый) поддерживает положительные и отрицательные целочисленные значения в 16-разрядном пространстве¹.

Знаковые целочисленные типы `short`, `int`, `long` и `long long`

Эти типы различаются своими размерами, а следовательно, и диапазоном значений, которые могут содержать. Тип `int` (самый популярный целочисленный тип) у большинства современных компиляторов имеет размер 32 бита. Используйте подходящий тип в зависимости от максимального значения, которое предположительно будет содержать определенная переменная.

Объявление переменной знакового типа очень простое:

```
short int smallNumber      = -100;
int      largerNumber      = -70000;
long     possiblyLargerThanInt = -70000;
long long largerThanInt    = -700000000000;
```

Беззнаковые целочисленные типы `unsigned short`, `unsigned int`, `unsigned long` и `unsigned long long`

В отличие от знаковых аналогов, беззнаковые целочисленные типы не могут содержать информацию о знаке, зато могут содержать вдвое большие положительные значения.

¹ Здесь автор изложил не более чем наброски использования знакового бита. На самом деле ситуация гораздо сложнее, и существует несколько вариантов представления отрицательных чисел в компьютерах. Заинтересованному читателю предлагается ознакомиться с этой темой в Интернете или соответствующей литературе; язык C++ на битовом уровне со знаковыми числами фактически не работает. — *Примеч. ред.*

Объявление переменной беззнакового типа тоже очень простое:

```
unsigned short int smallNumber      = 255;
unsigned int      largerNumber      = 70000;
unsigned long     possiblyLargerThanInt = 70000;
unsigned long long largerThanInt     = 70000000000;
```

ПРИМЕЧАНИЕ

Переменные беззнакового типа используются тогда, когда ожидаются только неотрицательные значения. Так, если вы подсчитываете количество яблок, не используйте тип `int`; воспользуйтесь типом `unsigned int`. Последний может содержать вдвое больше положительных значений, чем первый.

ВНИМАНИЕ!

Беззнаковый тип нельзя использовать в банковском приложении для переменной, хранящей остаток на счете, поскольку банки обычно допускают отрицательные значения на счету, предоставляя кредит своим клиентам. Пример, демонстрирующий разницу между знаковыми и беззнаковыми типами, вы найдете в листинге 5.3 занятия 5, “Выражения, инструкции и операторы”.

Избегайте переполнения, выбирая подходящие типы

Типы данных, такие как `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long` и другие, имеют ограниченную емкость и могут содержать числа, не превышающие некоторые пределы. Превысив предел для выбранного типа, вы получаете переполнение.

Возьмем в качестве примера `unsigned short`. Этот тип данных обычно содержит 16 бит, а потому может содержать значения от 0 до 65 535. Если вы прибавите 1 к 65 535 в переменной типа `unsigned short`, циклический переход приведет к значению 0. Это напоминает счетчик пробега автомобиля, у которого происходит механическое переполнение, если он может показывать только пять цифр, а автомобиль проехал 99 999 км.

В этом случае тип `unsigned short` явно не является правильным выбором для такого счетчика. Программист должен выбирать тип `unsigned int`, если ему необходимо работать с числами, большими, чем 65 535.

В случае типа `signed short`, диапазон которого — от $-32\,768$ до $32\,767$, прибавление 1 к 32 767 может привести к наибольшему по модулю представимому отрицательному значению. В случае знаковых чисел стандарт не определяет конкретное поведение, позволяя принять решение конкретному компилятору (и его разработчикам).

В листинге 3.4 продемонстрированы циклический переход и ошибка переполнения, которые вы можете непреднамеренно получить при выполнении арифметических операций.

ЛИСТИНГ 3.4. Демонстрация переполнения знаковых и беззнаковых целых чисел

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     unsigned short uShortValue = 65535;
6:     cout << "Увеличение unsigned short " << uShortValue << ": ";
7:     cout << ++uShortValue << endl;
8:
9:     short signedShort = 32767;
10:    cout << "Увеличение signed short " << signedShort << ": ";
11:    cout << ++signedShort << endl;
12:
13:    return 0;
14: }
```

Результат

Увеличение unsigned short 65535: 0
Увеличение signed short 32767: -32768

Анализ

Выходные данные показывают, что непреднамеренное переполнение приводит к непредсказуемому и непонятному на интуитивном уровне поведению приложения. Строки 7 и 11 увеличивают переменные типа `unsigned short` и `signed short`, которые были инициализированы их максимальными представимыми значениями — 65535 и 32767 соответственно. Вывод программы показывает значения, которые они получают после операции увеличения, а именно — циклический переход 65535 до нуля в случае `unsigned short` и переполнение 32767 до -32768 в случае `signed short`. Вряд ли вы ожидаете, что в результате операции увеличения рассматриваемое значение уменьшается, но именно это и происходит при переполнении целочисленного типа. Если вы используете такие значения для выделения памяти, то можете запросить у операционной системы нуль байт, в то время как на самом деле вам требуется 64 Кбайта — 65 536 байт.

ПРИМЕЧАНИЕ

Операции `++uShortValue` и `++signedShort`, показанные в листинге 3.4 в строках 7 и 11, являются операциями префиксного инкремента. Они подробно рассматриваются на занятии 5, “Выражения, инструкции и операторы”.

Типы с плавающей точкой float и double

Числа с плавающей точкой вы, вероятно, изучали в школе как вещественные числа. Они могут быть положительными и отрицательными, а также содержать десятичные значения. Так, если в переменной C++ необходимо сохранить значение числа π (3,141592... или, приближенно, 22/7), можете использовать для нее тип с плавающей точкой.

Объявление переменных этих типов следует тому же шаблону, что и тип `int` в листинге 3.1. Так, переменная типа `float`, позволяющая хранить десятичные значения, могла бы быть объявлена следующим образом:

```
float pi = 3.1415926;
```

Переменная вещественного типа с двойной точностью (типа `double`) определяется аналогично:

```
double morePrecisePi = 3.1415926;
```

СОВЕТ

Стандарт C++14 добавляет возможность использовать при записи числа разделитель разрядов в виде одинарной кавычки, что повышает удобочитаемость кода:

```
int moneyInBank           = -70'000;           // -70000
long populationChange     = -85'000;           // -85000
long long countryGDPChange = -70'000'000'000;   // -70 млрд.
double pi                  = 3.141'592'653'59;  // 3.14159265359
```

ПРИМЕЧАНИЕ

Рассматривавшиеся до этого момента типы данных зачастую называют *простыми старыми данными* (Plain Old Data — POD). К этой категории относятся также объединения этих типов, такие как структуры, перечисления или объединения.

Определение размера переменной с использованием оператора sizeof

Размер представляет собой объем памяти, резервируемый компилятором при объявлении программистом переменной для хранения присваиваемых ей данных. Размер переменной зависит от ее типа, и в языке C++ есть очень удобный оператор `sizeof`, который возвращает размер переменной или типа в байтах.

Применение оператора `sizeof` очень простое. Чтобы определить размер целого числа, вызовите оператор `sizeof` с параметром в виде типа `int`, как показано в листинге 3.5.

```
cout << "Размер int: " << sizeof(int);
```

ЛИСТИНГ 3.5. Поиск размера стандартных типов языка C++

```
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:     cout << "Размеры некоторых встроенных типов C++" << endl;
7:
8:     cout<<"bool           : "<<sizeof(bool)           <<endl;
9:     cout<<"char           : "<<sizeof(char)           <<endl;
10:    cout<<"unsigned short int: "<<sizeof(unsigned short) <<endl;
11:    cout<<"short int       : "<<sizeof(short)          <<endl;
12:    cout<<"unsigned long int : "<<sizeof(unsigned long)  <<endl;
13:    cout<<"long           : "<<sizeof(long)            <<endl;
14:    cout<<"int            : "<<sizeof(int)             <<endl;
15:    cout<<"unsigned long long: "<<sizeof(unsigned long long)<<endl;
16:    cout<<"long long       : "<<sizeof(long long)       <<endl;
17:    cout<<"unsigned int     : "<<sizeof(unsigned int)    <<endl;
18:    cout<<"float           : "<<sizeof(float)           <<endl;
19:    cout<<"double          : "<<sizeof(double)          <<endl;
20:
21:    cout << "Вывод зависит от компилятора, компьютера и ОС" <<endl;
22:
23:    return 0;
24: }
```

Результат

Размеры некоторых встроенных типов C++

```
bool           : 1
char           : 1
unsigned short int: 2
short int      : 2
unsigned long int : 4
long           : 4
int            : 4
unsigned long long: 8
long long      : 8
unsigned int    : 4
float          : 4
double         : 8
```

Вывод зависит от компилятора, компьютера и ОС

Анализ

Вывод листинга 3.5 демонстрирует размеры различных типов в байтах (которые зависят от конкретной платформы: компилятора, операционной системы и аппаратных средств). Данный конкретный вывод — это результат выполнения программы в

32-битовом режиме (32-битовый компилятор) в 64-битовой операционной системе. Обратите внимание, что 64-битовый компилятор, возможно, даст другие результаты, а 32-битовый компилятор автор выбрал потому, что должен был иметь возможность запускать приложение как на 32-, так и на 64-битовых системах. Вывод оператора `sizeof` свидетельствует о том, что размер переменной знакового и беззнакового типов одинаков; единственным различием этих двух типов является знаковый старший бит.

ПРИМЕЧАНИЕ

Все размеры в выводе приведены в байтах. Размер типа — важный параметр при выделении памяти для переменной, особенно для типов, используемых для хранения чисел. Тип `short int` может хранить числа из меньшего диапазона, чем `long long`. Так что вы не можете использовать тип `short int` для хранения, например, численности населения страны.

СОВЕТ

Стандарт C++11 вводит целочисленные типы фиксированного размера, позволяющие выбрать тип с точным количеством битов. Это `int8_t` и `uint8_t` для 8-битовых знаковых и беззнаковых целых. Имеются также целочисленные типы размером 16 бит (`int16_t` и `uint16_t`), размером 32 бита (`int32_t` и `uint32_t`) и 64 бита (`int64_t` и `uint64_t`). Для их использования требуется включить в программу заголовочный файл `<cstdint>`.

Запрет сужающего преобразования при использовании инициализации списком

При инициализации переменной меньшего целочисленного типа (скажем, `short`) значением переменной большего типа (скажем, `int`) вы рискуете получить ошибку сужающего преобразования, при которой компилятор должен преобразовать значение, хранящееся в типе, который потенциально может содержать гораздо большие числа, в тип, который имеет меньшие размеры, например:

```
int largeNum    = 5000000;
short smallNum = largeNum; // Компилируется, но возможна ошибка
```

Сужение не ограничивается преобразованиями только между целочисленными типами. С этой ошибкой можно столкнуться при инициализации `float` с помощью `double`, инициализации `float` (или `double`) с использованием `int` или `int` с помощью `float`. Некоторые компиляторы могут предупреждать о возможной ошибке, но это предупреждение не является критичной ошибкой, которая приведет к прекращению компиляции. В таких случаях вы можете столкнуться с ошибками, которые трудно выловить, так как это ошибки времени выполнения, которые могут происходить достаточно редко.

Чтобы избежать этой проблемы, C++11 рекомендует *инициализацию списком*, которая предотвращает сужение. Для использования этой возможности поместите значения или переменные инициализации в фигурные скобки `{}`. Синтаксис инициализации списком выглядит следующим образом:

```
int largeNum = 5000000;  
short anotherNum{ largeNum }; // Ошибка сужения!  
int anotherNum{ largeNum };   // ОК!  
float someFloat{ largeNum };  // Ошибка! Возможно сужение  
float someFloat{ 5000000 };   // ОК! 5000000 помещается в float
```

Возможно, это не очевидно на первый взгляд, но эта возможность потенциально в состоянии избавить вас от ошибок, которые происходят, когда данные, хранящиеся в типе, подвергаются сужающему преобразованию во время выполнения.

Автоматический вывод типа с использованием `auto`

В ряде случаев тип переменной очевиден — по присваиваемому при инициализации значению. Например, если переменная инициализируется значением `true`, следует ожидать, что, скорее всего, типом переменной будет `bool`. Компиляторы с поддержкой C++11 и выше дают возможность определять тип неявно, с использованием вместо типа переменной его ключевого слова `auto`:

```
auto coinFlippedHeads = true;
```

Здесь задача определения конкретного типа переменной `coinFlippedHeads` оставлена компилятору. Компилятор просто проверяет природу значения, которым инициализируется переменная, а затем выбирает тип, наилучшим образом подходящий для этой переменной. В данном случае для инициализирующего значения `true` лучше всего подходит тип `bool`. Таким образом, компилятор определяет тип `bool` как наилучший для переменной `coinFlippedHeads` и внутренне рассматривает ее как имеющую тип `bool`, что и демонстрирует листинг 3.6.

ЛИСТИНГ 3.6. Использование ключевого слова `auto`
для вывода типов компилятором

```
1: #include <iostream>  
2: using namespace std;  
3:  
4: int main()  
5: {  
6:     auto coinFlippedHeads = true;  
7:     auto largeNumber = 25000000000000;  
8:  
9:     cout << "coinFlippedHeads = " << coinFlippedHeads;  
10:    cout << " , sizeof(coinFlippedHeads) = "  
11:        << sizeof(coinFlippedHeads) << endl;  
12:    cout << "largeNumber = " << largeNumber;  
13:    cout << " , sizeof(largeNumber) = "  
14:        << sizeof(largeNumber) << endl;  
15:  
16:    return 0;  
17: }
```

Результат

```
coinFlippedHeads = 1 , sizeof(coinFlippedHeads) = 1  
largeNumber = 25000000000000 , sizeof(largeNumber) = 8
```

Анализ

Как можно заметить, вместо явного указания типа `bool` для переменной `coinFlippedHeads` и типа `long long` для переменной `largeNumber` в строках 6 и 7, где они объявляются, было использовано ключевое слово `auto`. Это ключевое слово делегирует принятие решения о типе переменных компилятору, который использует для этого инициализирующее значение. Чтобы проверить, создал ли компилятор фактически предполагаемые типы, используется оператор `sizeof`, позволяющий убедиться, что это действительно так.

ПРИМЕЧАНИЕ

Использование ключевого слова `auto` требует инициализации переменной, поскольку компилятор нуждается в инициализирующем значении, чтобы принять решение о наилучшем типе для переменной.

Если вы не инициализируете переменную, то применение ключевого слова `auto` приведет к ошибке при компиляции.

Хотя, на первый взгляд, ключевое слово `auto` кажется не особенно полезным, оно существенно упрощает программирование в тех случаях, когда тип переменной сложен. Роль ключевого слова `auto` в написании более простых, но безопасных с точки зрения использования типов программ рассматривается на занятиях с 15, “Введение в стандартную библиотеку шаблонов”, и далее.

Использование ключевого слова typedef для замены типа

Язык C++ позволяет переименовывать типы переменных так, как вам кажется более удобным. Для этого используется ключевое слово `typedef`. Например, программист хочет назначить типу `unsigned int` более описательное имя `STRICTLY_POSITIVE_INTEGER`.

```
typedef unsigned int STRICTLY_POSITIVE_INTEGER;  
STRICTLY_POSITIVE_INTEGER numEggsInBasket = 4532;
```

При компиляции первая строка указывает компилятору, что `STRICTLY_POSITIVE_INTEGER` — это не что иное, как тип `unsigned int`. Впоследствии, когда компилятор встречает уже определенный тип `STRICTLY_POSITIVE_INTEGER`, он заменяет его типом `unsigned int` и продолжает компиляцию.

ПРИМЕЧАНИЕ

`typedef` или подстановка типа особенно удобна при работе со сложными типами, у которых может быть громоздкий синтаксис, например при использовании шаблонов. Шаблоны рассматриваются на занятии 14, "Введение в макросы и шаблоны".

Что такое константа

Предположим, вы пишете программу для вычисления площади и периметра круга. Формулы таковы:

Площадь = π * Радиус * Радиус;

Периметр = $2 * \pi$ * Радиус

В данных формулах π — это константа со значением 3,141592... Вы хотите избежать случайного изменения значения π где-нибудь в вашей программе. Вы также не хотите случайно присвоить π неправильное значение, скажем, при небрежном копировании и вставке или при контекстном поиске и замене. Язык C++ позволяет определить π как константу, которая не может быть изменена после объявления. Другими словами, после того как значение константы определено, оно не может быть изменено. Попытки присваивания значения константе в языке C++ приводят к ошибке при компиляции.

Таким образом, в C++ константы похожи на переменные, за исключением того, что они не могут быть изменены. Подобно переменной, константа также занимает пространство в памяти и имеет имя для идентификации адреса выделенной для нее области. Однако содержимое этой области не может быть перезаписано. В языке C++ возможны следующие константы.

- Литеральные константы.
- Константы, объявленные с использованием ключевого слова `const`.
- Константные выражения, использующие ключевое слово `constexpr` (нововведение C++11).
- Константы перечислений, использующие ключевое слово `enum`.
- Константы, определенные с помощью макроопределений, использование которых не рекомендуется и осуждается.

Литеральные константы

Литеральные константы могут быть многих типов — целочисленные, строки и т.д. В нашей первой программе в листинге 1.1 строка "Hello World" выводится с помощью следующей инструкции:

```
std::cout << "Hello World" << std::endl;
```

Здесь "Hello World" — это константа *строкового литерала* (string literal). Когда вы объявляете целое число наподобие

```
int someNumber = 10;
```

целочисленной переменной `someNumber` присваивается начальное значение, равное 10. Здесь 10 — это часть кода, компилируемая в приложение, которая является неизменной и тоже является *литеральной константой* (literal constant). Вы можете инициализировать целочисленную переменную в восьмеричной записи:

```
int someNumber = 012; // Восьмеричное 12 равно десятичному 10
```

Начиная с C++14 можно использовать бинарные литералы:

```
int someNumber = 0b1010; // Двоичное 1010 равно десятичному 10
```

СОВЕТ

C++ позволяет определять собственные литералы, например температуру `0.0_C`, расстояние `10_km` и т.д.

Эти суффиксы (наподобие `_C`, `_km`) называются пользовательскими литералами. О них речь пойдет на занятии 12, “Типы операторов и их перегрузка”.

Объявление переменных как констант с использованием ключевого слова `const`

Самый важный тип констант C++ с практической и программной точек зрения объявляется с помощью ключевого слова `const`, расположенного перед типом переменной. В общем виде объявление выглядит следующим образом:

```
const имя_типа имя_константы = значение;
```

Давайте рассмотрим простое приложение, которое отображает значение константы по имени `pi` (листинг 3.7).

ЛИСТИНГ 3.7. Объявление константы `pi`

```
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     const double pi = 3.1415926;
8:     cout << "Значение pi равно: " << pi << endl;
9:
10:    // Удаление следующего комментария ниже ведет к ошибке:
11:    // pi = 345;
12:
13:    return 0;
14: }
```

Результат

Значение `pi` равно: 3.1415926

Анализ

Обратите внимание на объявление константы `pi` в строке 7. Ключевое слово `const` позволяет указать компилятору, что `pi` — это константа типа `double`. Если убрать комментарий со строки 11, в которой предпринимается попытка присвоить значение переменной, которую вы определили как константу, произойдет ошибка при компиляции примерно с таким сообщением: *You cannot assign to a variable that is const* (Вы не можете присвоить значение переменной, которая является константой). Таким образом, константы — это прекрасное средство гарантировать неизменность определенных данных.

ПРИМЕЧАНИЕ

Хорошей практикой программирования является определение переменных, значения которых предполагаются неизменными, как констант. Применение ключевого слова `const` указывает, что программист позаботился об обеспечении неизменности данных и защищает свое приложение от непреднамеренных изменений этой константы.

Это особенно полезно, когда над проектом работает несколько программистов.

Константы полезны при объявлении массивов постоянной длины, которые неизменны во время компиляции. В листинге 4.2 из занятия 4, “Массивы и строки”, содержится пример использования синтаксиса `const int` при определении длины массива.

Объявление констант с использованием ключевого слова `constexpr`

Ключевое слово `constexpr` позволяет объявлять константы подобно функциям:

```
constexpr double GetPi() {return 3.1415926;}
```

Одно `constexpr`-выражение может использовать другое:

```
constexpr double TwicePi() {return 2 * GetPi();}
```

`constexpr` может выглядеть как функция, однако обеспечивает возможности оптимизации с точки зрения компилятора и приложения. До тех пор, пока компилятор в состоянии вычислять константное выражение как конкретное значение, оно может использоваться в инструкциях и выражениях везде, где ожидается константа. В предыдущем примере `TwicePi()` является выражением `constexpr`, использующим константное выражение `GetPi()`. Скорее всего, это приведет к оптимизации во время компиляции, при которой компилятор каждый вызов `TwicePi()` просто заменяет числом 6.2831852, а не кодом, который будет вызывать `GetPi()` и умножать возвращенное им значение на 2.

В листинге 3.8 показан пример использования `constexpr`.

ЛИСТИНГ 3.8. Использование constexpr для вычисления pi

```
1: #include <iostream>
2: constexpr double GetPi() { return 3.141593; }
3: constexpr double TwicePi() { return 2 * GetPi(); }
4:
5: int main()
6: {
7:     using namespace std;
8:     const double pi = 3.141593;
9:
10:    cout << "Константа pi равна " << pi << endl;
11:    cout << "constexpr GetPi() возвращает " << GetPi() << endl;
12:    cout << "constexpr TwicePi() возвращает " << TwicePi() << endl;
13:    return 0;
14: }
```

Результат

```
Константа pi равна 3.141593
constexpr GetPi() возвращает 3.141593
constexpr TwicePi() возвращает 6.283186
```

Анализ

Программа демонстрирует два способа получения значения числа π : как константной переменной `pi`, объявленной в строке 8, и как константного выражения `GetPi()`, объявленного в строке 2. `GetPi()` и `TwicePi()` могут казаться функциями, но это не совсем функции. Дело в том, что функции вызываются во время выполнения программы. Эти же константные выражения компилятор заменяет числом 3.141593 при каждом использовании `GetPi()` и числом 6.283186 при использовании `TwicePi()`. Такое разрешение `TwicePi()` в константу увеличивает скорость выполнения программы по сравнению выполнением вычисления, содержащегося в функции.

СОВЕТ

Константные выражения должны содержать простые вычисления, возвращающие простые типы, такие как `int`, `double` и т.п. C++14 позволяет constexpr-выражениям содержать инструкции принятия решения, такие как `if` и `switch`. Подробно эти условные инструкции рассматриваются на занятии 6, "Управление потоком выполнения программы".

Использование `constexpr` не гарантирует оптимизацию времени компиляции, например если вы используете выражение `constexpr` для удвоения числа, предоставляемого пользователем. Компилятор не в состоянии получить результат вычисления такого выражения во время компиляции, так что он может игнорировать модификатор `constexpr` и компилировать выражение, как обычную функцию.

Как константное выражение используется там, где компилятор ожидает константу, показано в листинге 4.2 занятия 4, "Массивы и строки".

СОВЕТ

В предыдущих примерах мы определяли собственную константу `pi` просто как пример синтаксиса объявления констант и использования ключевого слова `constexpr`. Но большинство популярных компиляторов C++ содержат более точное значение числа π в константе `M_PI`, которую можно использовать в своих программах после включения заголовочного файла `<cmath>`.

Перечисления

Иногда некая переменная должна принимать значения только из определенного набора. Например, вы не хотите, чтобы среди цветов радуги случайно оказался бирюзовый или среди направлений компаса оказалось направление влево. В обоих этих случаях необходим тип переменной, значения которой ограничиваются определенным вами набором. *Перечисления* (enumerations) — это именно то, что необходимо в данной ситуации. Перечисления объявляются с помощью ключевого слова `enum`.

Вот пример перечисления, которое определяет цвета радуги:

```
enum RainbowColors
{
    Violet = 0,
    Indigo,
    Blue,
    Green,
    Yellow,
    Orange,
    Red
};
```

А вот другой пример — направления компаса:

```
enum CardinalDirections
{
    North,
    South,
    East,
    West
};
```

Перечисления используются как пользовательские. Переменные этого типа могут принимать значения, ограниченные объявленными ранее значениями перечисления. Так, при определении переменной, которая содержит цвет радуги, вы объявляете ее следующим образом:

```
RainbowColors MyWorldsColor = Blue; // Начальное значение
```

В приведенной выше строке кода объявляется переменная `MyWorldsColor`, имеющая тип перечисления `RainbowColors`. Эта переменная может содержать только один из семи цветов радуги и не может хранить никакие другие значения.

ПРИМЕЧАНИЕ

При объявлении перечисления компилятор преобразует его константы, такие как `Violet` и другие, в целые числа. Каждое последующее значение перечисления на единицу больше предыдущего. Начальное значение вы можете задать сами, но если вы этого не сделаете, компилятор начнет счет с 0. Так, значению `North` соответствует числовое значение 0.

По желанию можно также явно определить числовое значение напротив каждой из перечисляемых констант при их инициализации.

Листинг 3.9 демонстрирует использование перечисления для хранения четырех направлений с инициализацией первого значения.

ЛИСТИНГ 3.9. Использование перечислений для указания направлений ветра

```
1: #include <iostream>
2: using namespace std;
3:
4: enum CardinalDirections
5: {
6:     North = 25,
7:     South,
8:     East,
9:     West
10: };
11:
12: int main()
13: {
14:     cout << "Направления и их значения" << endl;
15:     cout << "North: " << North << endl;
16:     cout << "South: " << South << endl;
17:     cout << "East: " << East << endl;
18:     cout << "West: " << West << endl;
19:
20:     CardinalDirections windDirection = South;
21:     cout << "windDirection = " << windDirection << endl;
22:
23:     return 0;
24: }
```

Результат

```
Направления и их значения
North: 25
South: 26
East: 27
West: 28
windDirection = 26
```

Анализ

Обратите внимание на то, что у нас определены четыре константы перечисления, но первая константа `North` получила значение 25 (см. строку 6). Это автоматически гарантирует, что следующим константам будут соответствовать значения 26, 27 и 28, что и видно из вывода программы. В строке 20 создается переменная типа `Cardinal Directions`, которой присваивается начальное значение `South`. При выводе на экран в строке 21 компилятор отображает целочисленное значение, назначенное константе `South`, которое равно 26.

СОВЕТ

Имеет смысл обратиться к листингам 6.4 и 6.5 занятия 6, “Управление потоком выполнения программы”. В них перечисление используется для дней недели, а условное выражение позволяет указать выбранный пользователем день.

Определение констант с использованием директивы `#define`

Первое и главное: не используйте этот способ при написании новых программ. Единственная причина упоминания определения констант с использованием директивы `#define` в этой книге — помочь вам понять некоторые устаревшие программы, в которых для определения числа π мог бы использоваться такой синтаксис:

```
#define pi 3.141593
```

Это макрокоманда препроцессора, предписывающая компилятору заменять все упоминания `pi` значением 3.141593. Обратите внимание: это *текстовая* (читай: неинтеллектуальная) замена, осуществляемая препроцессором. Компилятор не знает фактический тип рассматриваемой константы и не заботится о нем.

ВНИМАНИЕ!

Определение констант с использованием директивы препроцессора `#define` считается устаревшим и не рекомендуется.

Ключевые слова, недопустимые для использования в качестве имен переменных и констант

Некоторые слова зарезервированы языком C++, и их нельзя использовать в качестве имен переменных. У этих ключевых слов есть специальное значение с точки зрения компилятора C++. К ключевым относятся такие слова, как `if`, `while`, `for` и `main`. Список ключевых слов языка C++ приведен в табл. 3.2, а также в приложении Б, “Ключевые слова языка C++”. У вашего компилятора могут быть дополнительные зарезервированные слова, поэтому для полноты списка проверьте его документацию.

ТАБЛИЦА 3.2. Ключевые слова языка C++

asm	dynamic_cast	namespace	template
auto	else	new	this
bool	enum	operator	throw
break	explicit	private	true
case	export	protected	try
catch	extern	public	typedef
char	false	register	typeid
class	float	reinterpret_cast	typename
const	for	return	union
constexpr	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while

Кроме того, зарезервированы следующие слова:

and	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
<p>Присваивайте переменным осмысленные имена, даже если они становятся длинными.</p> <p>Инициализируйте переменные и используйте инициализацию списком, чтобы избежать ошибок сужающего преобразования.</p> <p>Удостоверьтесь, что имя переменной объясняет ее назначение.</p> <p>Поставьте себя на место того, кто еще не видел ваш код, и подумайте, ясен ли смысл применяемых в нем имен.</p> <p>Используйте в своей группе соглашение об именованиях и строго придерживайтесь его.</p>	<p>Не присваивайте переменным имена, которые слишком коротки или содержат только один символ.</p> <p>Не присваивайте переменным имена, которые используют экзотические сокращения, понятные только вам.</p> <p>Не присваивайте переменным имена, совпадающие с зарезервированными ключевыми словами языка C++, поскольку такой код не будет компилироваться.</p>

Резюме

На этом занятии речь шла об использовании памяти для временного хранения значений в переменных и константах. Вы узнали, что тип переменных определяет их размер и что оператор `sizeof` позволяет выяснить этот размер. Вы также узнали о существовании различных типов переменных, таких как `bool`, `int` и другие, и о том, что

они должны использоваться для содержания данных различных типов. Правильный выбор типа переменной важен для эффективности программы; если для переменной выбран слишком маленький тип, это может закончиться циклическим переходом или переполнением. Вы познакомились с ключевым словом `auto`, которое позволяет компилятору самостоятельно вывести тип данных на основе значения, инициализирующего переменную.

Кроме того, мы рассмотрели различные типы констант и применение самых важных из них с использованием ключевых слов `const` и `enum`.

Вопросы и ответы

■ Зачем вообще определять константы, если вместо них можно использовать обычные переменные?

Константы, особенно те, в объявлении которых используется ключевое слово `const`, являются средством для указания компилятору того, что значение определенной переменной должно быть постоянным (не должно изменяться во время выполнения программы). Таким образом, компилятор гарантирует, что переменной, объявленной константной, никогда не будет присвоено другое значение, даже если другой программист, исправляя вашу работу, по неосторожности попытается перезаписать значение этой переменной. Если вы знаете, что значение переменной в программе не должно изменяться, ее следует объявить как константу, что увеличивает качество и надежность вашего приложения.

■ Зачем инициализировать значение переменной?

Не инициализировав переменную, вы не можете знать, какое значение она содержит изначально. Начальное значение — это просто содержимое области памяти, выделенной для переменной. Инициализация переменной, такая как

```
int myFavoriteNumber = 0;
```

записывает в область памяти, выделенной для переменной `myFavoriteNumber`, исходное значение по вашему выбору, в данном случае — 0. Распространены ситуации, когда некоторые действия осуществляется по-разному в зависимости от значения переменной (как правило, выполняется проверка на отличие ее значения от нуля). Без инициализации такая логика работает ненадежно, поскольку вновь выделенная область памяти содержит то, что в ней было раньше, т.е. случайное значение, которое невозможно предсказать².

■ Почему язык C++ позволяет использовать для целых чисел разные типы: `short int`, `int` и `long int`? Почему бы не использовать всегда только тот тип, который позволяет хранить наибольшие значения?

Язык программирования C++ используется для разработки множества приложений, некоторые из которых выполняются на устройствах с небольшими вычислительными возможностями и ресурсами памяти. (Простой старый сотовый телефон — один из примеров таких устройств.) В таком случае программист может сэкономить

² Если только это не глобальная переменная. — *Примеч. ред.*

память и ускорить выполнение, выбрав правильный тип переменной, если он не нуждается в больших значениях. Если программа предназначена для рабочего стола или высокопроизводительного смартфона, экономия памяти и увеличение производительности за счет выбора типа одного целого числа будет незначительной, а в некоторых случаях — просто отсутствовать.

- **Почему не следует широко использовать глобальные переменные? Раз они пригодны для использования повсюду в приложении, не могу ли я сэкономить время на передаче значений в функции и из них?**

Значения глобальных переменных можно читать и присваивать глобально. Последнее и является проблемой, поскольку они могут быть изменены в любом месте программы. Предположим, вы работаете над проектом вместе с несколькими программистами. Вы объявили свои целочисленные и другие переменные глобальными. Если программист вашей группы изменяет значение целого числа в своем коде, который может даже находиться не в том файле `.cpp`, который используете вы, это влияет и на ваш код. Поэтому экономия нескольких секунд или минут не должна быть критерием и, чтобы гарантировать стабильность своего кода, вы не должны использовать глобальные переменные без разбора.

- **Язык C++ позволяет объявлять беззнаковые целочисленные переменные, которые, как предполагается, способны содержать только положительные целочисленные значения и нуль. Что случится при уменьшении нулевого значения переменной типа `unsigned int`?**

Произойдет *циклический переход* (*wrapping*). Уменьшение значения 0 беззнаковой целочисленной переменной на 1 превратит его в самое большое значение, которое она способна содержать! Просмотрите табл. 3.1 и убедитесь, что переменная типа `unsigned short` способна содержать значения от 0 до 65 535. Итак, объявим переменную типа `unsigned short`, осуществим декремент и увидим нечто неожиданное:

```
unsigned short myShortInt = 0;           // Исходное значение
myShortInt = myShortInt - 1;             // Уменьшение на 1
std::cout << myShortInt << std::endl;    // Вывод: 65535!
```

Обратите внимание: это проблема не типа `unsigned short`, а способа ее применения. Целочисленный тип без знака (не важно, короткий или длинный) не должен использоваться там, где ожидается наличие отрицательных значений. Если содержимое переменной `myShortInt` должно использоваться для количества динамически резервируемых байтов, то небольшая ошибка, допустившая декремент нулевого значения, привела бы к выделению 64 Кбайт! Хуже того, если бы переменная `myShortInt` использовалась как индекс при доступе к памяти, ваше приложение, вероятнее всего, обратилось бы за пределы выделенной области памяти и аварийно завершилось бы!

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. В чем разница между знаковым и беззнаковым целыми числами?
2. Почему не стоит использовать директиву `#define` при объявлении константы?
3. Зачем инициализировать переменную?
4. Рассмотрите перечисление ниже. Каково значение константы `QUEEN`?
`enum YOURCARDS {ACE, JACK, QUEEN, KING};`
5. Что не так с именем этой переменной?
`int Integer = 0;`

Упражнения

1. Измените перечисление `YOURCARDS` контрольного вопроса 4 так, чтобы значением константы `QUEEN` стало 45.
2. Напишите программу, демонстрирующую, что размер беззнакового целого числа и обычного целого числа одинаков и что размер их обоих не превышает размер длинного целого числа.
3. Напишите программу для вычисления площади и периметра круга, радиус которого вводится пользователем.
4. Что будет, если в коде упражнения 3 площадь и периметр хранить в целочисленных переменных, а результат — в переменной любого другого типа?
5. **Отладка.** Что неверно в следующей инициализации?
`auto Integer;`

ЗАНЯТИЕ 4

Массивы и строки

На предыдущих занятиях мы объявляли переменные для хранения одиночного значения типа `int`, `char` или `string`, даже если использовалось несколько их экземпляров. Однако можно объявить коллекцию объектов, например 20 целых чисел или строку символов для хранения имени.

На этом занятии...

- Что такое массивы, как их объявлять и использовать
- Что такое строки и как использовать для их создания символьные массивы
- Краткое введение в тип `std::string`

Что такое массив

Определение слова *массив* (array) в словаре довольно близко к тому, что мы хотим понять. Согласно словарю Вебстера *массив* — это “группа элементов, формирующих единое целое, например массив солнечных панелей”.

Ниже приведены характеристики массива.

- Массив — это коллекция элементов.
- Все содержащиеся в массиве элементы — одного типа.
- Такая коллекция формирует полный набор.

В языке C++ массивы позволяют сохранить в памяти элементы данных некоторого типа в последовательном упорядоченном виде.

Необходимость в массивах

Предположим, вы пишете программу, в которой пользователь может ввести пять целых чисел и отобразить их на экране. Один из способов обработать эту ситуацию состоит в объявлении в программе пяти отдельных целочисленных переменных и сохранение в них отображаемых значений. Такое объявление может выглядеть следующим образом:

```
int firstNumber = 0;
int secondNumber = 0;
int thirdNumber = 0;
int fourthNumber = 0;
int fifthNumber = 0;
```

Но если бы пользователю понадобилось хранить и впоследствии отображать 500 и более целых чисел, то пришлось бы объявить 500 таких целочисленных переменных, используя приведенную выше систему. Требуются огромная работа и терпение для ее выполнения. А что делать, если пользователь попросит обеспечить 500 000 целых чисел вместо 5?

Правильнее объявить массив из пяти целых чисел, каждое из которых инициализировалось бы нулевым значением:

```
int myNumbers[5] = {0};
```

Таким образом, если бы вас попросили обеспечить 500 000 целых чисел, то ваш массив без проблем можно было бы увеличить:

```
int manyNumbers[500000] = {0};
```

Массив из пяти символов можно определить следующим образом:

```
char myCharacters[5];
```

Такие массивы называются *статическими* (static array), поскольку количество содержащихся в них элементов, а также размер выделенной для них области памяти остаются неизменными во время компиляции.

Объявление и инициализация статических массивов

В приведенных выше строках кода мы объявили массив `myNumbers`, который содержит пять элементов типа `int` (т.е. целых чисел), инициализированных значением 0. Таким образом, для объявления массива в языке C++ используется следующий синтаксис:

```
Тип_элемента Имя_массива [Количество_элементов] =  
    {Необязательные исходные значения};
```

Можно даже объявить массив и инициализировать содержимое всех его элементов. Так, целочисленный массив из пяти целых чисел можно инициализировать пятью разными целочисленными значениями:

```
int myNumbers[5] = {34, 56, -21, 5002, 365};
```

Все элементы массива можно инициализировать нулем (значение по умолчанию, предоставляемое компилятором):

```
int myNumbers[5] = {0}; // Инициализировать все элементы нулем
```

Вы можете также инициализировать только часть элементов массива:

```
int myNumbers[5] = {34, 56}; // инициализировать первые два  
// элемента значениями 34 и 56, прочие элементы равны нулю
```

Вы можете определить длину массива (т.е. указать количество элементов в нем) как константу и использовать ее при определении массива:

```
const int ARRAY_LENGTH = 5;  
int myNumbers[ARRAY_LENGTH] = {34, 56, -21, 5002, 365};
```

Это особенно полезно, когда необходимо иметь доступ и использовать длину массива в нескольких местах, например при переборе всех элементов массива. В таком случае при изменении длины массива достаточно будет исправить лишь одно значение, объявленное как `const int`.

Если исходное количество элементов в массиве неизвестно, его можно не указывать:

```
int myNumbers[] = {2017, 2052, -525};
```

Приведенный выше код создает массив из трех целых чисел со значениями 2017, 2052 и -525.

ПРИМЕЧАНИЕ

Массивы, которые мы объявляли до сих пор, называются *статическими*, поскольку их длина фиксирована во время компиляции. Такой массив не может принять больше данных, чем указано программистом. Он не может использовать и меньшее количество памяти, если она используется только наполовину или вообще не используется. Массивы, размер которых определяется во время выполнения, называются *динамическими*. Динамические массивы бегло затрагиваются на данном занятии и подробно обсуждаются на занятии 17, "Классы динамических массивов библиотеки STL".

Как данные хранятся в массиве

Рассмотрим книги, стоящие рядом на полке. Это пример одномерного массива, поскольку он распространяется только в одной размерности, представленной количеством книг на полке. Каждая книга представляет собой элемент массива, а полка напоминает область памяти, выделенную для хранения этой коллекции книг (рис. 4.1).

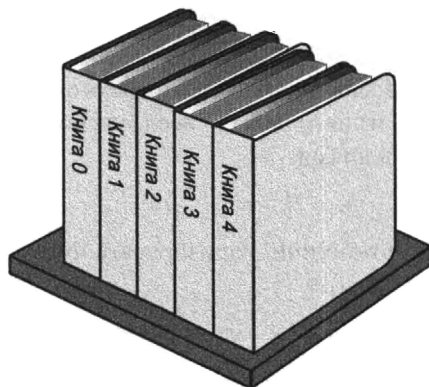


РИС. 4.1. Книги на полке: одномерный массив

Здесь нет ошибки, мы действительно начинаем нумеровать книги с нуля — именно так, как вы узнаете позже, нумеруются индексы в языке C++. Массив `myNumbers`, содержащий пять целых чисел и показанный на рис. 4.2, выглядит очень похожим на пять книг на полке.

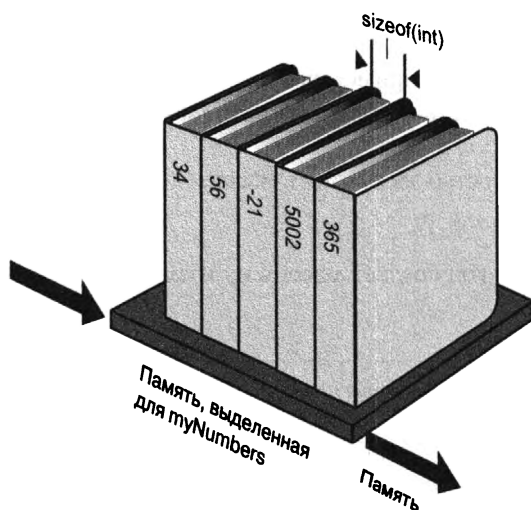


РИС. 4.2. Организация массива `myNumbers` из пяти целых чисел

Обратите внимание, что занятая массивом область памяти состоит из пяти блоков равного размера, определяемого типом хранимых массивом данных, в данном

случае — типом `int`. Если вы помните, мы рассматривали размер целочисленных типов на занятии 3, “Использование переменных и констант”. Таким образом, объем памяти, выделенной компилятором для массива `myNumbers`, равен `sizeof(int) * 5` байт. В общем виде объем памяти в байтах, резервируемой компилятором для массива, составляет

$$\text{Байты_массива} = \text{sizeof(Тип_элемента)} * \text{Количество_элементов}$$

Доступ к данным, хранимым в массиве

Для обращения к элементам массива можно использовать *индексы* (`index`), или номер элемента в массиве. Первый элемент массива имеет индекс 0. Так, первое целочисленное значение, хранимое в массиве `myNumbers`, — это `myNumbers[0]`, второе — `myNumbers[1]` и т.д. Пятый элемент массива — `myNumbers[4]`. Другими словами, индекс последнего элемента в массиве всегда на единицу меньше его длины.

Когда запрашивается доступ к элементу с индексом `N`, компилятор использует адрес первого элемента (позиция элемента с нулевым индексом) в качестве отправной точки, а затем пропускает `N` элементов, добавляя к этому адресу смещение, вычисляемое как `N * sizeof(тип_элемента)`, чтобы получить адрес `N+1`-го элемента. Компилятор C++ не проверяет, находится ли индекс в пределах фактически определенных границ массива. Вы можете попытаться выбрать элемент с индексом 1001 в массиве, содержащем только 10 элементов, поставив тем самым под угрозу безопасность и стабильность своей программы. Ответственность за предотвращение обращений к элементам за пределами массива лежит исключительно на программисте.

ВНИМАНИЕ!

Результат доступа к массиву за его пределами непредсказуем. Как правило, такое обращение ведет к аварийному завершению программы¹. Этого нужно избегать любой ценой.

В листинге 4.1 демонстрируются объявление массива целых чисел, инициализация его элементов целочисленными значениями и обращение к ним для отображения на экране.

ЛИСТИНГ 4.1. Объявление массива целых чисел и доступ к его элементам

```
0: #include <iostream>
1:
2: using namespace std;
3:
4: int main()
5: {
6:     int myNumbers[5] = {34, 56, -21, 5002, 365};
7:
```

¹ И это — наилучший вариант, поскольку в противном случае программа, продолжая работать, будет давать неверные результаты. — *Примеч. ред.*


```
8:     cout << "Элемент с индексом 0: " << myNumbers[0] << endl;
9:     cout << "Элемент с индексом 1: " << myNumbers[1] << endl;
10:    cout << "Элемент с индексом 2: " << myNumbers[2] << endl;
11:    cout << "Элемент с индексом 3: " << myNumbers[3] << endl;
12:    cout << "Элемент с индексом 4: " << myNumbers[4] << endl;
13:
14:    return 0;
15: }
```

Результат

```
Элемент с индексом 0: 34
Элемент с индексом 1: 56
Элемент с индексом 2: -21
Элемент с индексом 3: 5002
Элемент с индексом 4: 365
```

Анализ

В строке 6 объявлен массив из пяти целых чисел с определенными для каждого элемента исходными значениями. Последующие строки просто отображают целые числа, используя поток `cout` и переменную типа массива `myNumbers` с соответствующим индексом.

ПРИМЕЧАНИЕ

Чтобы вы лучше ознакомились с концепцией отсчитываемых от нуля индексов, используемых для доступа к элементам массива, начиная с листинга 4.1 строки кода нумеруются с нуля, а не с единицы.

Изменение данных в массиве

В коде предыдущего листинга пользовательские данные в массив не вводились. Синтаксис присваивания целого числа элементу в этом массиве очень похож на синтаксис присваивания значения целочисленной переменной.

Например, присваивание значения 2017 целочисленной переменной выглядит так:

```
int thisYear;
thisYear = 2017;
```

Присваивание значения 2017 четвертому элементу в рассматриваемом массиве выглядит аналогично:

```
myNumbers[3] = 2017; // Присваивание 2017 четвертому элементу
```

В листинге 4.2 демонстрируются использование констант в объявлении длины массива, а также присваивание значений отдельным элементам массива во время выполнения программы.

ЛИСТИНГ 4.2. Присваивание значений элементам массива

```
0: #include <iostream>
1: using namespace std;
2: constexpr int Square(int number) { return number*number; }
3:
4: int main()
5: {
6:     const int ARRAY_LENGTH = 5;
7:
8:     // Инициализированный массив из 5 целых чисел
9:     int myNumbers[ARRAY_LENGTH] = {5, 10, 0, -101, 20};
10:
11:     // Использование constexpr для массива из 25 целых чисел
12:     int moreNumbers[Square(ARRAY_LENGTH)];
13:
14:     cout << "Введите индекс изменяемого элемента: ";
15:     int elementIndex = 0;
16:     cin >> elementIndex;
17:
18:     cout << "Введите новое значение: ";
19:     int newValue = 0;
20:     cin >> newValue;
21:
22:     myNumbers[elementIndex] = newValue;
23:     moreNumbers[elementIndex] = newValue;
24:
25:     cout << "Элемент " << elementIndex << " myNumbers равен: ";
26:     cout << myNumbers[elementIndex] << endl;
27:
28:     cout << "Элемент " << elementIndex << " moreNumbers равен: ";
29:     cout << moreNumbers[elementIndex] << endl;
30:
31:     return 0;
32: }
```

Результат

Введите индекс изменяемого элемента: **3**

Введите новое значение: **101**

Элемент 3 myNumbers равен: 101

Элемент 3 moreNumbers равен: 101

Анализ

Длина массива должна быть константным целочисленным значением. Это значение может быть задано как константа `ARRAY_LENGTH` (строка 9) или как константное выражение `Square()` в строке 12. Таким образом, массив `myNumbers` объявлен как имеющий 5 элементов, в то время как массив `moreNumbers` содержит 25 элементов. В строках 14–20

пользователю предлагается ввести индекс элемента массива, который он хочет изменить, и новое значение, которое будет храниться в элементе с этим индексом. В строках 22 и 23 показано, как изменить конкретный элемент массива с использованием индекса. Обращение к элементам массива с помощью индекса показано в строках 26–29. Обратите внимание, что на изменение элемента с индексом 3 изменяет четвертый элемент массива, так как индексы отсчитываются от нуля. Вы должны привыкнуть к этому.

ПРИМЕЧАНИЕ

Многие новички в программировании на языке C++ присваивают значение пятому элементу, используя индекс 5 в массиве из пяти целых чисел. Обратите внимание: этот элемент находится уже за пределами массива, и откомпилированный код на самом деле пытается обратиться к шестому элементу массива из пяти элементов.

Этот вид ошибки иногда называется *ошибкой поста охраны* (fence-post error). Это название связано с тем фактом, что количество постов охраны на один больше количества охраняемых участков.

ВНИМАНИЕ!

В листинге 4.2 отсутствует кое-что очень важное: проверка введенного пользователем индекса на соответствие границам массива. Предыдущая программа должна на самом деле проверять, находится ли значение переменной `elementIndex` в пределах от 0 до 4 для массива `myNumbers` и от 0 до 24 для массива `moreNumbers`, и отбрасывать все остальные значения. Отсутствие такой проверки позволяет пользователю присвоить значение за границами массива. Потенциально это может привести к аварийному завершению работы приложения, а в наихудшем случае — и к сбою работы операционной системы.

Более подробно проверки рассматриваются на занятии 6, “Управление потоком выполнения программы”.

Использование циклов для доступа к элементам массива

При последовательной работе с элементами массива для обращения к ним (их перебора) используют циклы. Чтобы быстро научиться эффективно работать с элементами массива, используя цикл `for`, обратитесь к листингу 6.10 занятия 6, “Управление потоком выполнения программы”.

РЕКОМЕНДУЕТСЯ

Всегда инициализируйте массивы, иначе они будут содержать произвольные, непредсказуемые значения.

Всегда проверяйте, не выходят ли используемые индексы за границы массива.

НЕ РЕКОМЕНДУЕТСЯ

Никогда не обращайтесь к элементу с номером `N`, используя индекс `N`, в массиве из `N` элементов. Для обращения к последнему элементу используйте индекс `N-1`.

Не забывайте, что к первому элементу в массиве обращаются с помощью индекса 0.

Многомерные массивы

Массивы, которые мы рассматривали до сих пор, напоминали книги, стоящие на полке. На более длинной полке может быть больше книг, на более короткой — меньше. Таким образом, длина полки — единственная размерность, определяющая ее емкость, т.е. полка *одномерна*. Но что если нам нужно использовать массив для моделирования солнечных панелей, показанных на рис. 4.3? Солнечные панели, в отличие от книжных полок, распространяются в двух размерностях: по длине и по ширине.



РИС. 4.3. Массив солнечных панелей на крыше

Как можно заметить на рис. 4.3, шесть солнечных панелей располагаются в двумерном порядке: два ряда (строки) по три столбца. Но можно рассматривать такое расположение и как массив из двух элементов, каждый из которых сам является массивом из трех панелей; другими словами, как массив массивов. В языке C++ вы можете создавать двумерные массивы, но вы не ограничены только двумя размерностями. В зависимости от необходимости и характера приложения вы можете создавать в памяти многомерные массивы.

Объявление и инициализация многомерных массивов

Язык C++ позволяет объявлять многомерные массивы, указывая количество элементов, которое необходимо выделить в каждой размерности. Таким образом, двумерный массив целых чисел, представляющий солнечные панели на рис. 4.3, можно объявить так:

```
int solarPanelIDs[2][3];
```

Обратите внимание, что на рис. 4.3 каждой из шести панелей присвоен также идентификатор в диапазоне от 0 до 5. Если мы инициализируем целочисленный массив в том же порядке, то эта инициализация будет иметь следующий вид:

```
int solarPanelIDs[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Как видите, синтаксис инициализации подобен синтаксису, используемому при инициализации двух одномерных массивов. Если бы массив состоял из трех строк и трех столбцов, его объявления и инициализация выглядели бы следующим образом:

```
int threeRowsThreeColumns[3][3] =  
    {{-501, 206, 2017}, {989, 101, 206}, {303, 456, 596}};
```

ПРИМЕЧАНИЕ

Несмотря на то что язык C++ позволяет использовать модель многомерных массивов, в памяти такие массивы все равно хранятся как одномерные. Компилятор отображает многомерный массив на область памяти, которая расширяется только в одном направлении.

Если хотите, можете инициализировать тот же массив `solarPanelIDs` следующим образом — результат при этом оказывается тем же:

```
int solarPanelIDs[2][3] = {0, 1, 2, 3, 4, 5};
```

Однако предыдущий способ нагляднее, поскольку так проще представить и понять, что многомерный массив — это массив массивов.

Доступ к элементам многомерного массива

Рассматривайте многомерный массив как массив массивов. Поскольку рассмотренный ранее двумерный массив включал три строки и три столбца, содержащих целые числа, вы можете рассматривать его как массив, состоящий из трех элементов, каждый из которых является массивом, состоящим из трех целых чисел.

Поэтому, когда необходимо получить доступ к целому числу в этом массиве, следует использовать первый индекс для указания номера массива, хранящего целые числа, а второй индекс — для указания номера целого числа в этом массиве. Рассмотрим следующий массив:

```
int threeRowsThreeColumns[3][3] =  
    {{-501, 206, 2017}, {989, 101, 206}, {303, 456, 596}};
```

Он инициализирован так, что его можно рассматривать как три массива, каждый из которых содержит три целых числа. Здесь целочисленный элемент со значением 206 находится в позиции `[0][1]`, а элемент со значением 456 — в позиции `[2][1]`. В листинге 4.3 демонстрируется, как можно обращаться к целочисленным элементам этого массива.

ЛИСТИНГ 4.3. Доступ к элементам многомерного массива

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: int main()  
4: {  
5:     int threeRowsThreeColumns[3][3] =  
6:     {{-501, 206, 2016}, {989, 101, 206}, {303, 456, 596}};  
7:  
8:     cout << "Row 0: " << threeRowsThreeColumns[0][0] << " "  
9:             << threeRowsThreeColumns[0][1] << " "  
10:            << threeRowsThreeColumns[0][2] << endl;  
11:  
12:  
13:     cout << "Row 1: " << threeRowsThreeColumns[1][0] << " "  
14:            << threeRowsThreeColumns[1][1] << " "
```

```
15:                 << threeRowsThreeColumns[1][2] << endl;  
16:  
17:     cout << "Row 2: " << threeRowsThreeColumns[2][0] << " "  
18:                 << threeRowsThreeColumns[2][1] << " "  
19:                 << threeRowsThreeColumns[2][2] << endl;  
20:  
21:     return 0;  
22: }
```

Результат

```
Row 0: -501 206 2016  
Row 1: 989 101 206  
Row 2: 303 456 596
```

Анализ

Обратите внимание на метод построчного обращения к элементам массива, начиная с массива `Row 0` (первая строка, с индексом 0) и заканчивая массивом `Row 2` (третья строка, с индексом 2). Поскольку каждая из строк — это массив, синтаксис обращения к третьему элементу (индекс 2) в первой строке (индекс 0) такой, как показано в строке 10.

ПРИМЕЧАНИЕ

Длина кода в листинге 4.3 существенно увеличивается при увеличении количества элементов в массиве или его размерностей. При профессиональной разработке такой код неприемлем.

Более эффективный способ обращения к элементам многомерного массива показан в листинге 6.15 занятия 6, "Управление потоком выполнения программы". Там для доступа ко всем элементам подобного массива используется вложенный цикл `for`. Код с применением цикла `for` существенно короче и меньше склонен к ошибкам, а кроме того, на его длину не влияет изменение количества элементов в массиве.

Динамические массивы

Рассмотрим приложение, которое хранит медицинские записи больницы. Программист никак не может заранее знать, сколько записей должно хранить и обрабатывать его приложение. Он может сделать предположение о разумном пределе количества записей для маленькой больницы и превысить его, допустив ошибку в безопасном направлении. Но в этом случае он бессмысленно резервирует огромные объемы памяти и уменьшает производительность системы.

В таком случае нужно использовать не статические массивы, которые мы рассмотрели только что, а динамические, которые оптимизируют использование памяти и при необходимости увеличивают размер занимаемых ими ресурсов и памяти во время выполнения. Язык C++ предоставляет очень удобный в работе динамический массив в форме типа `std::vector`, как показано в листинге 4.4.

ЛИСТИНГ 4.4. Создание динамического массива целых чисел и его заполнение значениями

```
0: #include <iostream>
1: #include <vector>
2:
3: using namespace std;
4:
5: int main()
6: {
7:     vector<int> DynArrNums(3); // Динамический массив int'ов
8:
9:     dynArrNums[0] = 365;
10:    dynArrNums[1] = -421;
11:    dynArrNums[2] = 789;
12:
13:    cout << "Чисел в массиве: " << DynArrNums.size() << endl;
14:
15:    cout << "Введите новое число для вставки в массив: ";
16:    int anotherNum = 0;
17:    cin >> AnotherNum;
18:    dynArrNums.push_back(anotherNum);
19:
20:    cout << "Чисел в массиве: " << dynArrNums.size() << endl;
21:    cout << "Последний элемент массива: ";
22:    cout << dynArrNums[dynArrNums.size() - 1] << endl;
23:
24:    return 0;
25: }
```

Результат

```
Чисел в массиве: 3
Введите новое число для вставки в массив: 2017
Чисел в массиве: 4
Последний элемент массива: 2017
```

Анализ

Не волнуйтесь о синтаксисе с векторами и шаблонами в листинге 4.4, они пока еще не были объяснены. Попробуйте просто просмотреть вывод и соотнести его с кодом. Согласно выводу начальный размер массива составляет три элемента, что согласуется с объявлением вектора в строке 7. Зная это, вы все же можете в строке 15 попросить пользователя ввести четвертое число и, что интереснее всего, в строке 18 добавить его в конец массива, используя метод `push_back()`. `vector` динамически изменит свои размеры так, чтобы приспособиться к хранению большего объема данных. Это заметно по последующему увеличению размера массива до 4. Обратите внимание на использование знакомого по статическим массивам синтаксиса доступа к данным

в векторе. В строке 22 осуществляется доступ к последнему элементу (каким бы он ни был по счету, поскольку его позиция вычисляется во время выполнения) с помощью индекса, который для последнего элемента имеет значение `size() - 1`. Функция `size()` возвращает общее количество элементов, содержащихся в векторе.

ПРИМЕЧАНИЕ

Для использования класса динамического массива `std::vector` в код необходимо включить заголовочный файл `vector`, как это сделано в строке 1 листинга 4.4.

```
#include <vector>
```

Более подробно о векторах пойдет речь на занятии 17, “Классы динамических массивов библиотеки STL”.

Строки символов в стиле C

Строки в стиле C (C-style string) — это частный случай массива символов. Вы уже видели несколько примеров таких строк в виде строковых литералов, когда писали код:

```
std::cout << "Hello World";
```

Это эквивалентно такому объявлению массива:

```
char sayHello[] = {'H','e','l','l','o',' ','W','o','r','l','d','\0'};  
std::cout << sayHello << std::endl;
```

Обратите внимание: последний символ в массиве — нулевой символ `'\0'`. Он также называется *завершающим нулевым символом* (string-terminating character), поскольку указывает компилятору, что строка на этом заканчивается. Такие строки в стиле C — это частный случай символьных массивов, последним символом которых всегда является нулевой символ `'\0'`. Когда вы используете в коде строковый литерал, компилятор сам добавляет после него символ `'\0'`.

Если вставить символ `'\0'` в середину массива, то это не изменит его размер; однако обработка строки, хранящейся в данном массиве, остановится на этой точке. Это демонстрируется в листинге 4.5.

ПРИМЕЧАНИЕ

Символ `'\0'` может показаться двумя символами (в самом деле, для его ввода следует нажать на клавиатуре две клавиши). Однако обратная косая черта — это просто специальный управляющий код, который понимает компилятор, и потому воспринимает последовательность `\0` как ноль, т.е. это способ указать компилятору на необходимость вставить нулевой символ (символ со значением ноль).

Вы не можете записать нулевой символ непосредственно, поскольку литерал `'0'` будет воспринят, как символ нуля с кодом ASCII 48, а не 0.

Чтобы увидеть этот и другие значения кодов ASCII, обратитесь к таблице в приложении Г, “Коды ASCII”.

ЛИСТИНГ 4.5. Анализ строки в стиле C с завершающим нулевым символом

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     char sayHello[] = {'H','e','l','l','o',' ','
6:                       'W','o','r','l','d','\0'};
7:     cout << sayHello << endl;
8:     cout << "Размер массива: " << sizeof(sayHello) << endl;
9:     cout << "Замена пробела нулем" << endl;
10:    sayHello[5] = '\0';
12:    cout << sayHello << endl;
12:    cout << "Размер массива: " << sizeof(sayHello) << endl;
13:
14:    return 0;
15: }
```

Результат

```
Hello World
Размер массива: 12
Замена пробела нулем
Hello
Размер массива: 12
```

Анализ

Код в строке 10 заменяет пробел в строке "Hello World" нулевым символом. Теперь у массива есть два нулевых завершающих символа, но при выводе используется первый, что и создает получаемый результат. Когда пробел заменяется нулевым символом, отображаемая строка усекается до части "Hello". Метод `sizeof()` в строках 8 и 12 указывает, что размер массива не изменился, несмотря на изменение отображаемых данных.

ВНИМАНИЕ!

Если при объявлении и инициализации символьного массива в строках 5 и 6 листинга 4.5 вы забудете добавить символ `'\0'`, то после вывода "Hello World" на консоль будет выведен случайный набор символов. Дело в том, что оператор `std::cout` не остановится по окончании массива и будет продолжать вывод, пока не встретит нулевой символ, даже если для этого придется перейти границы массива. Эта ошибка может привести вашу программу к аварийному останову, а в некоторых случаях поставить под угрозу стабильность системы.

Строки в стиле C чреваты опасностями. В листинге 4.6 демонстрируются риски, связанные с их применением.

ЛИСТИНГ 4.6. Риски использования строк в стиле C и пользовательского ввода

```
0: #include <iostream>
1: #include <string.h>
2: using namespace std;
3: int main()
4: {
5:     cout << "Введите слово не длиннее 20 символов: ";
6:
7:     char userInput[21] = {'\0'};
8:     cin >> userInput;
9:
10:    cout << "Длина ввода: " << strlen(userInput) << endl;
11:
12:    return 0;
13: }
```

Результат

Введите слово не длиннее 20 символов: **Don'tUseThisProgram**
Длина ввода: 19

Анализ

Опасность видна в выводе. Программа просит пользователя не вводить больше двадцати символов. Дело в том, что объявленный в строке 7 символьный буфер, предназначенный для хранения пользовательского ввода, имеет фиксированную статически длину 21 символ. Поскольку последний символ в строке должен быть нулевым, '\0', максимальная длина текста, хранимого буфером, ограничивается двадцатью символами. Обратите внимание на применение функции `strlen` в строке 10 для вычисления длины строки. Она проходит по символьному буферу и подсчитывает количество символов, пока не достигает нулевого, который означает конец строки. Этот символ был вставлен в конец введенных пользователем данных потоком `cin`. Подобное поведение опасно, поскольку так можно легко пересечь границы символьного массива при вводе пользователем текста длиннее упомянутого предела. Чтобы узнать, как реализовать проверку, гарантирующую, что запись в массив не выйдет за его пределы, обратитесь к листингу 6.2 занятия 6, “Управление потоком выполнения программы”.

ВНИМАНИЕ!

Приложения, написанные на языке C (или на языке C++ программистами с большим опытом в языке C), зачастую используют в своем коде функции копирования строк, такие как `strcpy`, функции конкатенации, такие как `strcat`, и определения длины строк, такие как `strlen`.

Эти функции используют строки в стиле C и потому опасны, так как ищут завершающий нулевой символ и могут легко выйти за границы символьного массива, если программист не гарантировал наличие завершающего нулевого символа.

Строки C++: использование `std::string`

Стандартная строка C++ — самое эффективное средство работы и с текстовым вводом, и при работе со строками, например, при их конкатенации.

Язык C++ предоставляет мощное и в то же время безопасное средство работы со строками — класс `std::string`. Класс `std::string` не является статическим массивом элементов типа `char` неизменного размера, как строки в стиле C, и допускает увеличение размера, когда в нем необходимо сохранить больше данных. Применение `std::string` для работы со строковыми данными показано в листинге 4.7

ЛИСТИНГ 4.7. Использование `std::string` для инициализации и хранения пользовательского ввода, а также для копирования, конкатенации и определения длины строки

```
0: #include <iostream>
1: #include <string>
2:
3: using namespace std;
4:
5: int main()
6: {
7:     string greetString("Hello std::string!");
8:     cout << greetString << endl;
9:
10:    cout << "Введите текстовую строку: " << endl;
11:    string firstLine;
12:    getline(cin, firstLine);
13:
14:    cout << "Введите другую строку: " << endl;
15:    string secondLine;
16:    getline(cin, secondLine);
17:
18:    cout << "Результат конкатенации: " << endl;
19:    string concatString = firstLine + " " + secondLine;
20:    cout << concatString << endl;
21:
22:    cout << "Копия полученной строки: " << endl;
23:    string aCopy;
24:    aCopy = concatString;
25:    cout << aCopy << endl;
26:
27:    cout << "Длина строки: " << concatString.length() << endl;
28:
29:    return 0;
30: }
```

Результат

```

Hello std::string!
Введите текстовую строку:
I love
Введите другую строку:
C++ strings
Результат конкатенации:
I love C++ strings
Копия полученной строки:
I love C++ strings
Длина строки: 18

```

Анализ

Постарайтесь понять вывод и связать его с соответствующими элементами в коде. Не беспокойтесь пока что о новых синтаксических возможностях. Программа начинается с отображения инициализированной в строке 7 строки "Hello std::string!" Затем, в строках 12 и 16, она считывает введенные пользователем строки текста, которые сохраняются в переменных `firstLine` и `secondLine`. Фактически конкатенация очень проста и выглядит в строке 19 как арифметическая сумма, в которой к первой строке к тому же добавлен пробел. Копирование выглядит как простое присваивание в строке 24. Определение длины строки осуществляется с помощью вызова функции `length()` в строке 27.

ПРИМЕЧАНИЕ

Для использования строк C++ в код необходимо включить заголовочный файл `string`:

```
#include <string>
```

Это было сделано в строке 1 листинга 4.7.

Чтобы подробнее изучить различные функции класса `std::string`, обратитесь к занятию 16, "Класс строки библиотеки STL". Поскольку вы еще не изучали классы и шаблоны, игнорируйте пока соответствующие разделы и уделите внимание сути примеров.

Резюме

На этом занятии вы познакомились с азами массивов и способами их применения. Вы научились объявлять и инициализировать их элементы, получать доступ к значениям элементов массива и записывать их. Вы узнали, как важно не выходить за границы массива. Это явление называется *переполнением буфера* (buffer overflow), и проверка ввода перед его использованием для индексации элементов позволяет гарантировать отсутствие пересечения границ массива.

Динамические массивы позволяют программисту не беспокоиться об установке максимальной длины массива во время компиляции, а также обеспечивают лучшее управление памятью в случае, если размер массива меньше ожидаемого максимума.

Вы также узнали, что строки в стиле C — это частный случай символьного массива, в котором конец строки отмечается завершающим нулевым символом `'\0'`. Кроме того, вы узнали, что язык C++ обеспечивает намного лучшую возможность — класс `std::string`, — предоставляющую удобные вспомогательные функции и позволяющую определять длину строк, объединять их и выполнять иные действия с ними.

Вопросы и ответы

■ Зачем заботиться об инициализации элементов статического массива?

Если не инициализировать массив, он будет содержать случайные и непредсказуемые значения, поскольку область занимаемой им памяти останется неизменной после последних операций. Инициализация массивов гарантирует, что находящаяся в нем информация будет иметь определенное и предсказуемое начальное состояние.

■ Следует ли инициализировать элементы динамического массива по причинам, упомянутым в первом вопросе?

Вообще-то, нет. Динамический массив весьма интеллектуален. Нет необходимости инициализировать элементы динамического массива значениями по умолчанию, если для этого нет причин, связанных с приложением, которому нужно иметь в массиве определенные исходные значения.

■ Когда имеет смысл использовать строки в стиле C с завершающим нулевым символом?

Только когда кто-то приставил пистолет к вашей голове. Язык C++ предоставляет намного более безопасное средство — класс `std::string`, позволяющий любому программисту избежать использования строк в стиле C.

■ Включает ли длина строки завершающий нулевой символ?

Нет, не включает. Длина строки "Hello World" составляет 11 символов, включая пробел, но исключая завершающий нулевой символ.

■ Хорошо, но я все же хочу использовать строки в стиле C в символьных массивах, определенных мною. Каким должен быть размер используемого массива?

Здесь вы столкнетесь с одной из сложностей использования строк в стиле C. Размер массива должен быть на единицу больше размера наибольшей строки, которую он будет когда-либо содержать. Это место необходимо для завершающего нулевого символа в конце самой длинной строки. Если бы строка "Hello World" была наибольшей, которую предстоит содержать символьному массиву, то размер массива должен был бы быть 11+1 символ, т.е. всего 12 символов.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Посмотрите на `myNumbers` в листинге 4.1. Каковы индексы первого и последнего его элементов?
2. Если необходимо дать возможность пользователю вводить строки, использовали бы вы строки в стиле C?
3. Сколько символов в строке `"\0"` насчитает компилятор?
4. Вы забываете завершить строку в стиле C нулевым символом. Что может случиться при ее использовании?
5. Просмотрите объявление вектора в листинге 4.4 и попробуйте создать динамический массив, содержащий элементы типа `char`.

Упражнения

1. Объявите массив, представляющий клетки на шахматной доске; типом массива должно быть перечисление, определяющее фигуры на доске.
Подсказка: перечисление должно содержать константы для различных фигур на доске, но вы не должны забывать, что клетка доски может быть и пустой.

2. **Отладка.** Что не так в приведенном фрагменте кода?

```
int myNumbers[5] = {0};
myNumbers[5] = 450; // Присваивание значения 450 пятому элементу
```

3. **Отладка.** Что не так в приведенном фрагменте кода?

```
int myNumbers[5];
cout << myNumbers[3];
```


ЗАНЯТИЕ 5

Выражения, инструкции и операторы

Основой программ является набор последовательно выполняемых команд. Эти команды формируются в выражения и инструкции и используют операторы для выполнения определенных вычислений или действий.

На этом занятии...

- Что такое выражения
- Что такое блоки, или составные выражения
- Что такое операторы
- Как выполнять простые арифметические и логические операции

Выражения

Языки программирования состоят из *инструкций* (statement), которые следуют одна за другой. Давайте проанализируем первую инструкцию, которую вы изучили:

```
cout << "Hello World" << endl;
```

Эта инструкция использует поток `cout` для вывода текста на консоль (т.е. на экран). Все инструкции в языке C++ заканчиваются точкой с запятой (;), определяющей границу инструкции. Эта точка с запятой подобна точке, которую вы добавляете в конце предложения разговорного языка. Следующая инструкция может начинаться непосредственно после точки с запятой, но для удобства и удобочитаемости программисты, как правило, записывают инструкции с новой строки. Вот, например, две инструкции в одной строке:

```
cout << "Hello World" << endl; cout << "Another hello" << endl;  
// Одна строка, две инструкции
```

ПРИМЕЧАНИЕ

Пробельные символы обычно не воспринимаются компилятором. К ним относятся пробелы, символы табуляции, символы новой строки и т.д. Тем не менее пробельные символы в составе строковых литералов отображаются при выводе.

Поэтому следующий код недопустим:

```
cout << "Hello  
World" << endl;  
// Символ новой строки в строковом литерале недопустим
```

Такой код обычно заканчивается сообщением об ошибке, указывающим, что компилятор не обнаружил в первой строке закрывающую кавычку (") и завершающую инструкцию точку с запятой (;). Если по каким-то причинам необходимо распространить инструкцию на несколько строк, достаточно добавить последним символом символ обратной косой черты (\):

```
cout << "Hello \  
World" << endl; // Разделение строки на две вполне допустимо
```

Еще один способ разместить приведенную выше инструкцию в двух строках — это использовать два строковых литерала вместо одного:

```
cout << "Hello "  
"World" << endl; // Два строковых литерала подряд вполне допустимы
```

Встретив такой код, компилятор обратит внимание на два соседних строковых литерала и сам объединит их.

ПРИМЕЧАНИЕ

Разделение инструкций на несколько строк может быть полезным, если у вас есть длинные текстовые элементы или сложные инструкции, состоящие из множества переменных, которые делают инструкцию намного длиннее, чем может вместить большинство экранов.

Составные инструкции, или блоки

Сгруппировав инструкции в фигурных скобках `{ ... }`, вы создаете *составную инструкцию* (compound statement), или *блок* (block).

```
{  
    int Number = 365;  
    cout << "Этот блок содержит две инструкции" << endl;  
}
```

Как правило, блок объединяет несколько связанных инструкций. Блоки особенно полезны при применении условной инструкции `if` и циклов, которые рассматриваются на занятии 6, “Управление потоком выполнения программы”.

Использование операторов

Операторы (operator) в C++ представляют собой инструменты, предоставляемые языком для работы с данными, их преобразования, обработки и принятия решений на их основе.

Оператор присваивания (=)

Оператор присваивания (assignment operator) мы уже использовали в этой книге. Он вполне интуитивно понятен:

```
int daysInYear = 365;
```

Приведенное выше выражение использует оператор присваивания для инициализации целочисленной переменной значением 365. Оператор присваивания заменяет значение, содержащееся в операнде слева от оператора присваивания (называемого *l-значением* (l-value)), значением операнда справа (называемого *r-значением* (r-value)).

Понятие l- и r-значений

Зачастую l-значения называют областями памяти. Такая переменная, как `daysInYear`, из приведенного выше примера фактически является дескриптором области памяти и, соответственно, l-значением. С другой стороны, r-значения могут быть самим содержимым области памяти.

Все l-значения могут быть r-значениями, но не все r-значения могут быть l-значениями. Чтобы понять это лучше, рассмотрим следующий пример, который не имеет никакого смысла, а потому не будет компилироваться:

```
365 = daysInYear;
```

Операторы сложения (+), вычитания (-), умножения (*), деления (/) и деления по модулю (%)

Вы можете выполнять арифметические операции между двумя операндами, используя оператор + для сложения, оператор - для вычитания, оператор * для умножения, оператор / для деления и оператор % для деления по модулю:

```
int num1 = 22;
int num2 = 5;
int addNums    = num1 + num2; // 27
int subtractNums = num1 - num2; // 17
int multiplyNums = num1 * num2; // 110
int divideNums  = num1 / num2; // 4
int moduloNums  = num1 % num2; // 2
```

Оператор деления (/) возвращает результат деления двух операндов. Однако в случае целых чисел результат не содержит дробной части, поскольку целые числа по определению не могут ее содержать. Оператор деления по модулю (%) возвращает остаток от деления и применим только к целочисленным значениям. В листинге 5.1 содержится простая программа, демонстрирующая выполнение арифметических действий с двумя введенными пользователем числами.

ЛИСТИНГ 5.1. Демонстрация арифметических операторов с введенными пользователем целыми числами

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите два целых числа: ";
6:     int num1 = 0, num2 = 0;
7:     cin >> num1;
8:     cin >> num2;
9:
10:    cout << num1 << " + " << num2 << " = " << num1+num2 << endl;
11:    cout << num1 << " - " << num2 << " = " << num1-num2 << endl;
12:    cout << num1 << " * " << num2 << " = " << num1*num2 << endl;
13:    cout << num1 << " / " << num2 << " = " << num1/num2 << endl;
14:    cout << num1 << " % " << num2 << " = " << num1%num2 << endl;
15:
16:    return 0;
17: }
```

Результат

```
Введите два целых числа: 365 25
365 + 25 = 390
365 - 25 = 340
365 * 25 = 9125
365 / 25 = 14
365 % 25 = 15
```

Анализ

Большая часть программы говорит сама за себя. Интереснее всего, вероятно, строка, использующая оператор деления по модулю %. Она возвращает остаток деления значения переменной `num1` (365) на значение переменной `num2` (25).

Операторы инкремента (++) и декремента (--)

Иногда в программе необходим *инкремент* (increment), т.е. простое увеличение значения переменной на единицу. Это особенно важно для переменных, контролирующих циклы, в которых значение переменной должно увеличиваться или уменьшаться на единицу при каждом выполнении цикла.

Для сокращения записей наподобие `num=num+1` или `num=num-1` язык C++ предоставляет операторы ++ (инкремента) и -- (декремента).

Синтаксис их использования следующий:

```
int num1 = 101;
int num2 = num1++; // Постфиксный оператор инкремента
int num2 = ++num1; // Префиксный оператор инкремента
int num2 = num1--; // Постфиксный оператор декремента
int num2 = --num1; // Префиксный оператор декремента
```

Пример кода демонстрирует два разных способа применения операторов инкремента и декремента: до и после операнда. Операторы, которые располагаются перед операндом, называются *префиксными* (prefix) операторами инкремента или декремента, а те, которые располагаются после, — *постфиксными* (postfix).

Что значит “постфиксный” и “префиксный”

Сначала следует понять различие между префиксными и постфиксными операторами, а затем использовать тот, который нужен вам в каждом конкретном случае. Результат выполнения постфиксных операторов заключается в том, что сначала *l*-значению присваивается *r*-значение, а потом *r*-значение увеличивается или уменьшается. Это значит, что во всех случаях использования постфиксного оператора значением переменной `num2` будет прежнее значение переменной `num1` (т.е. то значение, которое она имела до операции инкремента или декремента).

Действие префиксных операторов прямо противоположно: сначала изменяется *r*-значение, а затем оно присваивается *l*-значению. В этих случаях переменные `num2` и `num1` имеют одинаковые значения. Листинг 5.2 демонстрирует результат выполнения префиксных и постфиксных операторов инкремента и декремента для определенного целого числа.

ЛИСТИНГ 5.2. Различия между постфиксными и префиксными операторами

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int startValue = 101;
6:     cout << "Начальное значение: " << startValue << endl;
7:
8:     int postfixIncrement = startValue++;
9:     cout << "Постфиксный ++ = " << postfixIncrement << endl;
10:    cout << "После постфиксного ++ startValue = "
11:        << startValue << endl;
12:    startValue = 101;                // Сброс
13:    int prefixIncrement = ++startValue;
14:    cout << "Префиксный ++ = " << prefixIncrement << endl;
15:    cout << "После префиксного ++ startValue = "
16:        << startValue << endl;
17:    startValue = 101;                // Сброс
18:    int postfixDecrement = startValue--;
19:    cout << "Постфиксный -- = " << postfixDecrement << endl;
20:    cout << "После постфиксного -- startValue = "
21:        << startValue << endl;
22:    startValue = 101;                // Сброс
23:    int prefixDecrement = --startValue;
24:    cout << "Префиксный -- = " << prefixDecrement << endl;
25:    cout << "После префиксного -- startValue = "
26:        << startValue << endl;
27:    return 0;
28: }
```

Результат

```
Начальное значение: 101
Префиксный ++ = 101
После постфиксного ++ startValue = 102
Постфиксный ++ = 102
После префиксного ++ startValue = 102
Постфиксный -- = 101
После постфиксного -- startValue = 100
Префиксный -- = 100
После префиксного -- startValue = 100
```

Анализ

Результаты показывают, чем постфиксные операторы отличаются от префиксных. При использовании постфиксных операторов в строках 8 и 18 значения содержат

исходные значения целого числа, — какими они были до операций инкремента или декремента. Использование префиксных операторов в строках 13 и 23, напротив, присваивает результат инкремента или декремента. Это самое важное различие, о котором следует помнить, выбирая правильный тип оператора.

В следующих выражениях префиксные или постфиксные операторы никак не влияют на результат:

```
startValue++;    // То же, что и...
++startValue;
```

Дело в том, что здесь нет присваивания исходного значения и конечный результат в обоих случаях — увеличенное на единицу значение переменной `startValue`.

ПРИМЕЧАНИЕ

Нередко приходится слышать о ситуациях, когда префиксные операторы инкремента или декремента являются более предпочтительными с точки зрения производительности, т.е. `++startValue` предпочтительнее, чем `startValue++`.

Это правда, по крайней мере теоретически, поскольку при постфиксных операторах компилятор должен временно хранить исходное значение на случай его присваивания. Влияние на производительность в случае целых чисел незначительно, но в случае некоторых классов этот аргумент мог бы иметь смысл. Интеллектуальные компиляторы могут полностью устранить различия, оптимизируя код.

Операторы равенства (==) и неравенства (!=)

Зачастую необходимо проверить выполнение или не выполнение определенного условия прежде, чем предпринять некое действие. Операторы равенства `==` (операнды равны) и неравенства `!=` (операнды не равны) позволяют сделать именно это.

Результат проверки равенства имеет логический тип `bool`, т.е. `true` (истина) или `false` (ложь).

```
int personAge = 20;
bool checkEquality      = (personAge == 20); // true
bool checkInequality    = (personAge != 100); // true
bool checkEqualityAgain = (personAge == 200); // false
bool checkInequalityAgain = (personAge != 20); // false
```

Операторы сравнения

Кроме проверки на равенство и неравенство, может возникнуть необходимость в сравнении, значение какой переменной больше, а какой меньше. Для этого язык C++ предоставляет операторы сравнения, приведенные в табл. 5.1.

ТАБЛИЦА 5.1. Операторы сравнения

Оператор	Назначение
Меньше (<)	Возвращает значение true, если один операнд меньше другого ($Op1 < Op2$), в противном случае возвращает значение false
Больше (>)	Возвращает значение true, если один операнд больше другого ($Op1 > Op2$), в противном случае возвращает значение false
Меньше или равно (<=)	Возвращает значение true, если один операнд меньше или равен другому, в противном случае возвращает значение false
Больше или равно (>=)	Возвращает значение true, если один операнд больше или равен другому, в противном случае возвращает значение false

Как свидетельствует табл. 5.1, результатом операции сравнения всегда является значение true или false, другими словами, значение типа bool. Следующий пример кода демонстрирует применение операторов сравнения, приведенных в табл. 5.1:

```
int personAge = 20;
bool checkLessThan          = (personAge < 100); // true
bool checkGreaterThan       = (personAge > 100); // false
bool checkLessThanEqualTo   = (personAge <= 20); // true
bool checkGreaterThanEqualTo = (personAge >= 20); // true
bool checkGreaterThanEqualToAgain = (personAge >= 100); // false
```

Код листинга 5.3 демонстрирует использование этих операторов при отображении результата на экране.

ЛИСТИНГ 5.3. Операторы равенства и сравнения

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите два целых числа:" << endl;
6:     int num1 = 0, num2 = 0;
7:     cin >> num1;
8:     cin >> num2;
9:
10:    bool Equality = (num1 == num2);
11:    cout << "Проверка равенства: " << Equality << endl;
12:
13:    bool Inequality = (num1 != num2);
14:    cout << "Проверка неравенства: " << Inequality << endl;
15:
16:    bool GreaterThan = (num1 > num2);
17:    cout << "Результат сравнения " << num1 << " > " << num2;
18:    cout << ": " << GreaterThan << endl;
19:
20:    bool LessThan = (num1 < num2);
```

```
21:     cout << "Результат сравнения " << num1 << " < " << num2;
22:     cout << ": " << LessThan << endl;
23:
24:     bool GreaterThanEquals = (num1 >= num2);
25:     cout << "Результат сравнения " << num1 << " >= " << num2;
26:     cout << ": " << GreaterThanEquals << endl;
27:
28:     bool LessThanEquals = (num1 <= num2);
29:     cout << "Результат сравнения " << num1 << " <= " << num2;
30:     cout << ": " << LessThanEquals << endl;
31:
32:     return 0;
33: }
```

Результат

Введите два целых числа:

365

-24

Проверка равенства: 0

Проверка неравенства: 1

Результат сравнения 365 > -24: 1

Результат сравнения 365 < -24: 0

Результат сравнения 365 >= -24: 1

Результат сравнения 365 <= -24: 0

Следующий запуск:

Введите два целых числа:

101

101

Проверка равенства: 1

Проверка неравенства: 0

Результат сравнения 101 > 101: 0

Результат сравнения 101 < 101: 0

Результат сравнения 101 >= 101: 1

Результат сравнения 101 <= 101: 1

Анализ

Программа отображает результат различных операций в двоичном виде. Интересно отметить в выводе случаи, когда сравниваются два одинаковых целых числа. Операторы ==, >= и <= дают идентичные результаты.

Тот факт, что результат операторов равенства и сравнения является логическим значением, делает их отлично подходящими для использования в операторах принятия решения и в выражениях условий циклов, гарантирующих выполнение цикла, только пока условие истинно. Более подробная информация об условном выполнении и циклах приведена на занятии 6, “Управление потоком выполнения программы”.

ПРИМЕЧАНИЕ

В листинге 5.3 булево значение `false` выводится как 0. Значение `true` выводится как 1. С точки зрения компилятора выражение равно `false`, если его вычисляемое значение нулевое. Проверка на равенство `false` представляет собой проверку того, что данное значение нулевое. Выражение с ненулевым значением рассматривается как логическое значение `true`.

Логические операции НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ

Логическая операция НЕ выполняется с помощью оператора `!` и выполняется над одним операндом. Таблица истинности логической операции НЕ, которая просто инвертирует значение логического флага, приведена в табл. 5.2.

ТАБЛИЦА 5.2. Таблица истинности логической операции НЕ

Операнд	Результат операции “НЕ Операнд”
false	true
true	false

Для других операций, таких как И, ИЛИ или ИСКЛЮЧАЮЩЕЕ ИЛИ, необходимы два операнда. Логическая операция И возвращает значение `true` только тогда, когда каждый операнд содержит значение `true`. Таблица истинности логической операции И приведена в табл. 5.3.

ТАБЛИЦА 5.3. Таблица истинности логической операции И

Операнд 1	Операнд 2	Результат операции “Операнд 1 И Операнд 2”
false	false	false
true	false	false
false	true	false
true	true	true

Логическая операция И выполняется с помощью оператора `&&`.
Логическая операция ИЛИ возвращает значение `true` тогда, когда по крайней мере один из операндов содержит значение `true`. Таблица истинности логической операции ИЛИ приведена в табл. 5.4.

ТАБЛИЦА 5.4. Таблица истинности логической операции ИЛИ

Операнд 1	Операнд 2	Результат операции “Операнд 1 ИЛИ Операнд 2”
false	false	false
true	false	true
false	true	true
true	true	true

Логическая операция ИЛИ выполняется с помощью оператора `||`.

Логическая операция ИСКЛЮЧАЮЩЕГО ИЛИ (XOR) немного отличается от логической операции ИЛИ и возвращает значение `true` тогда, когда любой из операндов содержит значение `true`, но не оба одновременно (т.е. когда логические значения операндов не равны). Таблица истинности логической операции ИСКЛЮЧАЮЩЕГО ИЛИ приведена в табл. 5.5.

ТАБЛИЦА 5.5. Таблица истинности логической операции ИСКЛЮЧАЮЩЕГО ИЛИ

Операнд 1	Операнд 2	Результат операции "Операнд 1 XOR Операнд 2"
false	false	false
true	false	true
false	true	true
true	true	false

Логическая операция ИСКЛЮЧАЮЩЕГО ИЛИ выполняется с помощью оператора `^`. Результат получается путем выполнения операции ИСКЛЮЧАЮЩЕГО ИЛИ над битами операндов.

Использование логических операторов C++ `!`, `&&` и `||`

Рассмотрим следующие утверждения.

- "Если идет дождь И если нет автобуса, то я не смогу попасть на работу".
- "Если есть большая скидка ИЛИ если я получу премию, то смогу купить этот автомобиль".

В программировании часто необходима некоторая логическая конструкция, когда результат двух операций используется в логическом контексте для принятия решения о выполнении последующего потока программы. Язык C++ предоставляет логические операторы `И` и `ИЛИ`, которые можно использовать в условных инструкциях, а следовательно, в зависимости от условий изменять поток выполнения программы.

В листинге 5.4 демонстрируется работа логических операторов `И` и `ИЛИ`.

ЛИСТИНГ 5.4. Анализ логических операторов C++ `&&` и `||`

```

0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите true(1) или false(0) "
6:         "для двух операндов:";
7:     bool Op1 = false, Op2 = false;
8:     cin >> Op1;
9:     cin >> Op2;
10:    cout << Op1 << " И " << Op2 << " = " << (Op1&&Op2) << endl;

```

```

11:     cout << Op1 << " ИЛИ " << Op2 << " = " << (Op1||Op2) << endl;
12:
13:     return 0;
14: }

```

Результат

Введите true(1) или false(0) для двух операндов: 1 0
 1 И 0 = 0
 1 ИЛИ 0 = 1

Следующий запуск:

Введите true(1) или false(0) для двух операндов: 1 1
 1 И 1 = 1
 1 ИЛИ 1 = 1

Анализ

Программа демонстрирует, что позволяют делать логические операции И и ИЛИ. Однако она не показывает, как их использовать для принятия решений.

В листинге 5.5 представлена программа, которая, используя условные и логические операторы, выполняет разные строки кода в зависимости от значений, содержащихся в переменных.

ЛИСТИНГ 5.5. Использование логических операторов ! и && в условных инструкциях для изменения потока выполнения

```

0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите 0 или 1 для ответа на вопрос" << endl;
6:     cout << "Идет дождь? ";
7:     bool isRaining = false;
8:     cin >> isRaining;
9:
10:    cout << "На улице есть автобус? ";
11:    bool busesPly = false;
12:    cin >> busesPly;
13:
14:    // Условный оператор использует операторы && и !
15:    if (isRaining && !busesPly)
16:        cout << "Вы не попадете на работу" << endl;
17:    else
18:        cout << "Вы попадете на работу" << endl;
19:
20:    if (isRaining && busesPly)

```

```

21:         cout << "Возьмите зонтик" << endl;
22:
23:     if ((!isRaining) && busesPly)
24:         cout << "Приятного дня и хорошей погоды!" << endl;
25:
26:     return 0;
27: }

```

Результат

Введите 0 или 1 для ответа на вопрос
Идет дождь? **1**
На улице есть автобус? **1**
Вы попадете на работу
Возьмите зонтик

Следующий запуск:

Введите 0 или 1 для ответа на вопрос
Идет дождь? **1**
На улице есть автобус? **0**
Вы не попадете на работу

Последний запуск:

Введите 0 или 1 для ответа на вопрос
Идет дождь? **0**
На улице есть автобус? **1**
Вы попадете на работу
Приятного дня и хорошей погоды!

Анализ

Код в листинге 5.5 использует условную инструкцию `if`, которая пока еще не рассматривалась. Но вы все же попробуйте понять поведение этой инструкции, сопоставив ее с выводом на консоль. Строка 15 содержит логическое выражение `(isRaining && !busesPly)`, которое можно прочитать как “идет дождь И НЕТ автобуса”. Логический оператор И здесь использован для объединения отсутствия автобусов (обозначенного логическим оператором НЕ перед наличием автобусов) и присутствия дождя.

ПРИМЕЧАНИЕ

Более подробная информация об инструкции `if` будет приведена на занятии 6, “Управление потоком выполнения программы”.

Код листинга 5.6 использует логические операторы `!` и `||` для демонстрации условной обработки.

ЛИСТИНГ 5.6. Использование логических операторов !и || для решения о возможности купить автомобиль

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите 0 или 1 для ответа на вопрос" << endl;
6:     cout << "Есть ли скидка на автомобиль? ";
7:     bool onDiscount = false;
8:     cin >> onDiscount;
9:
10:    cout << "Вы получили премию? ";
11:    bool fantasticBonus = false;
12:    cin >> fantasticBonus;
13:
14:    if (onDiscount || fantasticBonus)
15:        cout << "Вы можете купить автомобиль!" << endl;
16:    else
17:        cout << "Покупку придется отложить..." << endl;
18:
19:    if (!onDiscount)
20:        cout << "Скидки на автомобиль нет" << endl;
21:
22:    return 0;
23: }
```

Результат

Введите 0 или 1 для ответа на вопрос
Есть ли скидка на автомобиль? **0**
Вы получили премию? **1**
Вы можете купить автомобиль!
Скидки на автомобиль нет

Следующий запуск:

Введите 0 или 1 для ответа на вопрос
Есть ли скидка на автомобиль? **0**
Вы получили премию? **0**
Покупку придется отложить...
Скидки на автомобиль нет

Последний запуск:

Введите 0 или 1 для ответа на вопрос
Вы получили премию? **1**
Есть ли скидка на автомобиль? **1**
Вы можете купить автомобиль!

Анализ

Программа сообщает о возможности купить автомобиль, если на него есть скидка или если вы получили премию. Инструкция в строке 19, кроме того, сообщает, когда скидки на автомобиль нет. В строке 14 инструкция `if` используется вместе с конструкцией `else` в строке 16. Часть `if` выполняет инструкцию в строке 15, если условие `(onDiscount || fantasticBonus)` истинно (имеет значение `true`). Это выражение содержит логический оператор **ИЛИ** и возвращает значение `true`, если есть скидка на ваш любимый автомобиль или если вы получили фантастическую премию. Если рассматриваемое выражение возвращает значение `false`, выполняется инструкция в строке 17, идущая после конструкции `else`.

Побитовые операторы `~`, `&`, `|` и `^`

Различие между логическими и побитовыми операторами в том, что они возвращают не логический результат, а значение, отдельные биты которого получены в результате выполнения оператора над соответствующими битами операндов. Язык C++ позволяет выполнять такие операции, как **НЕ**, **ИЛИ**, **И** и **ИСКЛЮЧАЮЩЕЕ ИЛИ** (**XOR**) в побитовом режиме, позволяя работать с отдельными битами, инвертируя их с помощью оператора `~`, применяя операцию **ИЛИ** с помощью оператора `|`, операцию **И** с помощью оператора `&` и операцию **XOR** с помощью оператора `^`. Последние три операции обычно выполняются с некоторой специально подготовленной битовой маской.

Некоторые битовые операции полезны в тех случаях, когда, например, каждый из битов, содержащихся в целом числе, определяет состояние некоего флага. Так, целое число размером 32 бита можно использовать для хранения 32-х логических флагов. Использование побитовых операторов продемонстрировано в листинге 5.7.

ЛИСТИНГ 5.7. Использование побитовых операторов для работы с отдельными битами целого числа

```

0: #include <iostream>
1: #include <bitset>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Введите число (0-255): ";
7:     unsigned short inputNum = 0;
8:     cin >> inputNum;
9:
10:    bitset<8> inputBits(inputNum);
11:    cout << inputNum << " в бинарном виде равно "
12:         << inputBits << endl;
13:    bitset<8> BitwiseNOT = (~inputNum);
14:    cout << "Побитовое НЕ ~" << endl;
15:    cout << "~" << inputBits << " = "
```

```
16:         << BitwiseNOT << endl;
17:     cout << "Логическое И (&) с 00001111" << endl;
18:     bitset<8> BitwiseAND = (0x0F&inputNum); // 0x0F == 0001111
19:     cout << "0001111 & " << inputBits << " = "
20:         << BitwiseAND << endl;
21:     cout << "Логическое ИЛИ (|) с 00001111" << endl;
22:     bitset<8> BitwiseOR = (0x0F | inputNum);
23:     cout << "00001111 | " << inputBits << " = "
24:         << BitwiseOR << endl;
25:     cout << "Логическое XOR (^) с 00001111" << endl;
26:     bitset<8> BitwiseXOR = (0x0F ^ inputNum);
27:     cout << "00001111 ^ " << inputBits << " = "
28:         << BitwiseXOR << endl;
29:     return 0;
30: }
```

Результат

```
Введите число (0-255): 181
181 в бинарном виде равно 10110101
Побитовое НЕ ~
~10110101 = 01001010
Логическое И (&) с 00001111
0001111 & 10110101 = 00000101
Логическое ИЛИ (|) с 00001111
00001111 | 10110101 = 10111111
Логическое XOR (^) с 00001111
00001111 ^ 10110101 = 10111010
```

Анализ

Эта программа использует *битовое множество* (bitset) — тип, который нами еще не рассматривался, — для облегчения отображения двоичных данных. Роль класса `std::bitset` здесь исключительно вспомогательная — он помогает с отображением данных и не более того. В строках 10, 13, 18 и 22 вы фактически присваиваете целое число объекту битового множества, используемому для отображения этого целочисленного значения в двоичном виде. Все операции выполняются с целыми числами. Сначала обратите внимание на вывод введенного пользователем исходного числа 181 в двоичном виде, а затем переходите к результату выполнения различных побитовых операторов — `~`, `&`, `|` и `^` — с этим целым числом. Как можно заметить, побитовое НЕ, использованное в строке 13, просто инвертирует все биты числа. Программа демонстрирует также работу операторов `&`, `|` и `^`, создающих результат путем выполнения вычислений с каждым битом двух операндов. Сопоставьте представленные результаты с приведенными ранее таблицами истинности, и работа побитовых операторов станет вам понятнее.

ПРИМЕЧАНИЕ

Если вы хотите узнать больше о работе с битовыми флагами в языке C++, обратитесь к занятию 25, "Работа с битовыми флагами при использовании библиотеки STL", на котором класс `std::bitset` обсуждается подробнее.

Побитовые операторы сдвига вправо (>>) и влево (<<)

Операторы сдвига перемещают всю последовательность битов вправо или влево, позволяя осуществить умножение или деление на степень двойки, а также имеют многие другие применения.

Вот типичный пример применения оператора сдвига для умножения на два:

```
int doubledValue = Num << 1; // Для удвоения значения биты
                          // сдвигаются на одну позицию влево
```

Вот типичный пример применения оператора сдвига для деления на два:

```
int halvedValue = Num >> 2; // Для деления значения на два биты
                          // сдвигаются на одну позицию вправо
```

Использование операторов сдвига для быстрого умножения и деления целочисленных значений продемонстрировано в листинге 5.8.

ЛИСТИНГ 5.8. Использование побитового оператора сдвига вправо >> для получения четверти и половины значения, а оператора сдвига влево << для умножения значения на два и четыре

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите число: ";
6:     int inputNum = 0;
7:     cin >> inputNum;
8:
9:     int halfNum      = inputNum >> 1;
10:    int quarterNum    = inputNum >> 2;
11:    int doubleNum     = inputNum << 1;
12:    int quadrupleNum  = inputNum << 2;
13:
14:    cout << "Четверть: "      << quarterNum << endl;
15:    cout << "Половина: "     << halfNum    << endl;
16:    cout << "Удвоенное: "    << doubleNum  << endl;
17:    cout << "Учетверенное: " << quadrupleNum << endl;
18:
19:    return 0;
20: }
```

Результат

Введите число: 16
Четверть: 4
Половина: 8
Удвоенное: 32
Учетверенное: 64

Анализ

Пользователь вводит число 16, которое в двоичном представлении имеет вид 1000. В строке 9 осуществляется его сдвиг вправо на один бит, и получается 0100, что в десятичном виде составляет 4 — фактически половина исходного значения. В строке 10 осуществляется сдвиг вправо на два бита, 1000 превращается в 0010, что составляет 2. Результат операторов сдвига влево в строках 11 и 12 прямо противоположен. Сдвиг на один бит влево дает значение 10000 или 32, а на два бита — соответственно 100000 или 64, фактически удваивая и учетверяя исходное значение.

ПРИМЕЧАНИЕ

Побитовые операторы сдвига не выполняют циклический сдвиг. Кроме того, результат сдвига знаковых чисел зависит от конкретного компилятора.

Составные операторы присваивания

Составные операторы присваивания (compound assignment operator) — это операторы присваивания, в которых результат операции присваивается левому операнду.

Рассмотрим следующий код:

```
int num1 = 22;  
int num2 = 5;  
num1 += num2; // После операции num1 содержит значение 27
```

Этот код эквивалентен следующей строке кода:

```
num1 = num1 + num2;
```

Таким образом, результат оператора += — это сумма этих двух операндов, присвоенная затем левому операнду (num1). Краткий перечень составных операторов присваивания с объяснением их работы приведен в табл. 5.6.

ТАБЛИЦА 5.6. Составные операторы присваивания

Оператор	Применение	Эквивалент
Присваивание с добавлением	num1 += num2;	num1 = num1 + num2;
Присваивание с вычитанием	num1 -= num2;	num1 = num1 - num2;
Присваивание с умножением	num1 *= num2;	num1 = num1 * num2;
Присваивание с делением	num1 /= num2;	num1 = num1 / num2;
Присваивание с делением по модулю	num1 %= num2;	num1 = num1 % num2;
Присваивание с побитовым сдвигом влево	num1 <<= num2;	num1 = num1 << num2;

Оператор	Применение	Эквивалент
Присваивание с побитовым сдвигом вправо	num1 >>= num2;	num1 = num1 >> num2;
Присваивание с побитовым И	num1 &= num2;	num1 = num1 & num2;
Присваивание с побитовым ИЛИ	num1 = num2;	num1 = num1 num2;
Присваивание с побитовым XOR	num1 ^= num2;	num1 = num1 ^ num2;

Применение этих операторов продемонстрировано в листинге 5.9.

ЛИСТИНГ 5.9. Использование составных операторов присваивания

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите число: ";
6:     int value = 0;
7:     cin >> value;
8:
9:     value += 8;
10:    cout << "После += 8, value = " << value << endl;
11:    value -= 2;
12:    cout << "После -= 2, value = " << value << endl;
13:    value /= 4;
14:    cout << "После /= 4, value = " << value << endl;
15:    value *= 4;
16:    cout << "После *= 4, value = " << value << endl;
17:    value %= 1000;
18:    cout << "После %= 1000, value = " << value << endl;
19:
20:    // Примечание: далее присваивание происходит в пределах cout
21:    cout << "После <= 1, value = " << (value <= 1) << endl;
22:    cout << "После >= 2, value = " << (value >= 2) << endl;
23:
24:    cout << "После |= 0x55, value = " << (value |= 0x55) << endl;
25:    cout << "После ^= 0x55, value = " << (value ^= 0x55) << endl;
26:    cout << "После &= 0x0F, value = " << (value &= 0x0F) << endl;
27:
28:    return 0;
29:}
```

Результат

Введите число: **440**
После += 8, value = 448
После -= 2, value = 446

```
После /= 4, value = 111
После *= 4, value = 444
После %= 1000, value = 444
После <= 1, value = 888
После >= 2, value = 222
После |= 0x55, value = 223
После ^= 0x55, value = 138
После &= 0x0F, value = 10
```

Анализ

Обратите внимание, как последовательно изменяется значение переменной `value` по мере применения в программе различных составных операторов присваивания. Каждая операция осуществляется с использованием переменной `value`, и ее результат снова присваивается переменной `value`. Так, в строке 9 ко введенному пользователем значению 440 прибавляется 8, а результат, 448, снова присваивается переменной `value`. При следующей операции в строке 11 из 448 вычитается 2, что дает значение 446, которое снова присваивается переменной `value`, и т.д.

Использование оператора `sizeof` для определения объема памяти, занимаемого переменной

Этот оператор возвращает объем памяти в байтах, используемой определенным типом или переменной. Оператор `sizeof` имеет следующий синтаксис:

```
sizeof(Переменная);
или
sizeof(Тип);
```

ПРИМЕЧАНИЕ

Оператор `sizeof(...)` выглядит как вызов функции, но это не функция, а оператор. Данный оператор не может быть определен программистом, а следовательно, не может быть перегружен.

Более подробная информация об определении собственных операторов рассматривается на занятии 12, "Типы операторов и их перегрузка".

В листинге 5.10 демонстрируется применение оператора `sizeof` для определения объема памяти, занятого массивом. Кроме того, можно вернуться к листингу 3.5 и проанализировать применение оператора `sizeof` для определения объема памяти, занятого переменными наиболее распространенных типов.

ЛИСТИНГ 5.10. Использование оператора `sizeof` для определения количества байтов, занятых массивом из 100 целых чисел и каждым его элементом

```
0: #include <iostream>
1: using namespace std;
2:
```

```
3: int main()
4: {
5:     cout << "Использование sizeof для массива" << endl;
6:     int myNumbers[100] = {0};
7:
8:     cout << "Байт для типа int: " << sizeof(int) << endl;
9:     cout << "Байт для массива myNumbers: "
10:         << sizeof(myNumbers) << endl;
11:     cout << "Байт для элемента массива: "
12:         << sizeof(myNumbers[0]) << endl;
13:     return 0;
14: }
```

Результат

```
Использование sizeof для массива
Байт для типа int: 4
Байт для массива myNumbers: 400
Байт для элемента массива: 4
```

Анализ

Программа демонстрирует, как оператор `sizeof` возвращает размер в байтах массива из 100 целых чисел, составляющий 400 байтов, а также что размер каждого его элемента составляет 4 байта.

Оператор `sizeof` может быть весьма полезен, когда необходимо динамически разместить в памяти N объектов, особенно если их тип создан вами самостоятельно. Вы можете использовать результат выполнения оператора `sizeof` для определения объема памяти, занимаемого каждым объектом, а затем динамически выделить память, используя оператор `new`.

Более подробная информация о динамическом распределении памяти рассматривается на занятии 8, “Указатели и ссылки”.

Приоритет операторов

Возможно, вы помните из школы, что арифметические операции в сложном арифметическом выражении выполняются в определенном порядке.

В языке C++ также используются сложные выражения, например:

```
int myNumber = 10 * 30 + 20 - 5 * 5 << 2;
```

Вопрос: какое значение будет содержать переменная `myNumber`? Здесь нет места догадкам. Порядок выполнения различных операторов строго определен стандартом C++. Этот порядок определяется *приоритетом операторов*, приведенным в табл. 5.7.

ТАБЛИЦА 5.7. Приоритет операторов

Приоритет	Название	Оператор
1	Область видимости	::
2	Прямое и косвенное обращение к члену класса, вызов функции	. -> ()
3	Инкремент и декремент, логические операторы отрицания и побитового дополнения, унарные “минус” и “плюс”, получение адреса и разыменование, а также операторы new, new[], delete, delete[], sizeof и операторы приведения типов	++ -- ~ ! - + & *
4	Обращение к элементу по указателю	.* ->*
5	Умножение, деление, деление по модулю	* / %
6	Сложение, вычитание	+ -
7	Сдвиг влево, сдвиг вправо	<< >>
8	Меньше, меньше или равно, больше, больше или равно	< <= > >=
9	Равно, не равно	== !=
10	Побитовое И	&
11	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ	^
12	Побитовое ИЛИ	
13	Логическое И	&&
14	Логическое ИЛИ	
15	Тернарный условный оператор	?:
16	Операторы присваивания	= *= /= %= += -= <<= >>= &= = ^=
17	Оператор “запятая”	,

Давайте теперь еще раз рассмотрим сложное выражение, приведенное ранее в качестве примера:

```
int myNumber = 10 * 30 + 20 - 5 * 5 << 2;
```

При вычислении результата этого выражения необходимо использовать правила приоритета операторов, приведенные в табл. 5.7, чтобы понять, как их выполняет компилятор. Так, умножение и деление имеют более высокий приоритет, чем сложение и вычитание, приоритет которых, в свою очередь, выше приоритета оператора сдвига. В результате после первого упрощения мы получаем:

```
int myNumber = 300 + 20 - 25 << 2;
```

Поскольку сложение и вычитание имеют более высокий приоритет по сравнению со сдвигом, это выражение упрощается до

```
int myNumber = 295 << 2;
```

И наконец выполняется операция сдвига. Зная, что сдвиг влево на один бит удваивает число, а сдвиг влево на два бита умножает его на 4, можно утверждать, что выражение сводится к $295 \cdot 4$, а результат равен 1180.

ВНИМАНИЕ!

Чтобы код был более понятным, используйте круглые скобки. Приведенное выше выражение просто написано плохо. Компилятору не составляет труда в нем разобраться, но написанный код должен быть понятен, в первую очередь, людям.

То же выражение будет намного понятнее, если записать его следующим образом:

```
int myNumber = ((10*30)-(5*5)+20)<<2; // 1180
```

РЕКОМЕНДУЕТСЯ

Используйте круглые скобки, чтобы сделать свой код более понятным.

Используйте правильные типы переменных и убедитесь, что они никогда не приведут к переполнению.

Помните, что все l-значения (например, переменные) могут быть r-значениями, но не все r-значения (например, "Hello World") могут быть l-значениями.

НЕ РЕКОМЕНДУЕТСЯ

Не создавайте сложные выражения, полагающиеся на таблицу приоритета операторов; ваш код должен быть понятен, в первую очередь, людям.

Не забывайте, что выражения ++*Переменная* и *Переменная*++ отличаются одно от другого при использовании в присваивании.

Резюме

На этом занятии вы узнали, что такое инструкции, выражения и операторы языка C++. Вы научились выполнять простые арифметические операции, такие как сложение, вычитание, умножение и деление. Был также приведен краткий обзор таких логических операторов, как НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ. Мы рассмотрели логические операторы !, && и ||, используемые в условных выражениях, и побитовые операторы, такие как ~, &, | и ^, которые позволяют работать с отдельными битами.

Вы узнали о приоритете операторов, а также о том, почему так важно использовать круглые скобки при написании кода, который должен быть понятен, в первую очередь, поддерживающим его программистам. Было дано общее представление о переполнении целочисленных переменных и о способах его избегания.

Вопросы и ответы

- Почему некоторые программы используют тип `unsigned int`, если тип `unsigned short` занимает меньше памяти и код вполне компилируется?

Тип `unsigned short` обычно имеет предел 65535, а при его превышении происходит переполнение, дающее неверное значение. Чтобы избежать этого, если нет абсолютной уверенности, что рабочие значения всегда останутся ниже указанного предела, следует выбирать более емкий тип, например `unsigned int`.

- Я должен удвоить результат деления на три. Нет ли в моем коде каких-либо проблем: `int result = Number / 3 << 1;`?

Есть. Почему бы вам не использовать круглые скобки, чтобы сделать эту строку проще для понимания другими программистами? Комментарий тоже не повредил бы.

- Мое приложение делит два целочисленных значения — 5 и 2:

```
int num1 = 5, num2 = 2;  
int result = num1 / num2;
```

- При выполнении `result` содержит значение 2. Разве это не ошибка?

Нет. Целые числа не предназначены для хранения вещественных чисел. Поэтому результат этой операции — 2, а не 2,5. Если вам нужен результат 2,5, то измените все типы данных на `float` или `double`, так как именно они предназначены для операций с плавающей точкой.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Я пишу приложение для деления чисел. Какой тип данных подойдет мне лучше: `int` или `float`?
2. Каков результат выражения `32/7`?
3. Каков результат выражения `32.0/7`?
4. Является ли `sizeof(...)` функцией?
5. Я должен вычислить удвоенное число, добавить к нему 5, а затем снова удвоить результат. Все ли я сделал правильно?

```
int Result = number << 1 + 5 << 1;
```
6. Каков результат операции ИСКЛЮЧАЮЩЕЕ ИЛИ, если оба операнда содержат значение `true`?

Упражнения

1. Исправьте код в контрольном вопросе 5, используя круглые скобки для устранения неоднозначности.
2. Каким будет значение переменной `result` в этом выражении?

```
int result = number << 1 + 5 << 1;
```
3. Напишите программу, которая запрашивает у пользователя два логических значения и демонстрирует результаты различных побитовых операций над ними.

ЗАНЯТИЕ 6

Управление потоком выполнения программы

Большинство приложений ведут себя по-разному в зависимости от ситуации или введенных пользователем данных. Чтобы позволить приложению реагировать на обстоятельства по-разному, необходимы условные инструкции, выполняющие разные части кода в разных ситуациях.

На этом занятии...

- Как заставить программу вести себя по-разному в определенных условиях
- Как многократно выполнить фрагмент кода в цикле
- Как лучше управлять потоком выполнения в цикле

Условное выполнение с использованием конструкции `if...else`

Программы, которые вы видели и создавали до сих пор, имели последовательный порядок выполнения — сверху вниз. Все строки выполнялись, и ни одна не игнорировалась. Однако последовательное выполнение всех строк кода редко используется в приложениях.

Предположим, программа должна умножать два числа, если пользователь ввел 'м', или суммировать их в противном случае.

Как можно видеть из рис. 6.1, при каждом запуске проходятся не все пути выполнения. Если пользователь вводит 'м', выполняется код, умножающий два числа. Если он вводит нечто другое, выполняется код суммирования. Ни при какой ситуации не выполняются оба варианта кода.

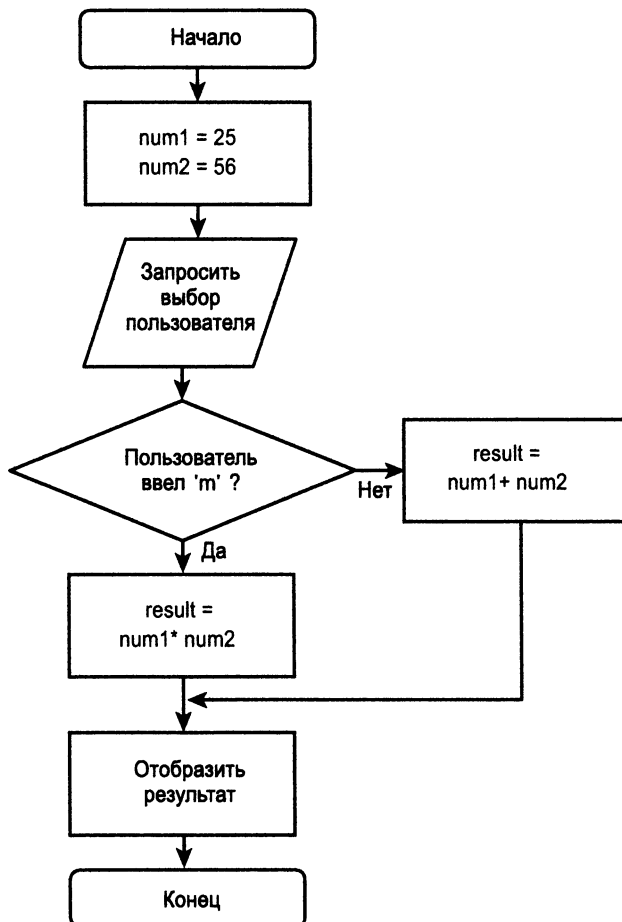


РИС. 6.1. Пример условного выполнения на основе пользовательского ввода

Условное программирование с использованием конструкции if...else

Условное выполнение кода реализовано в языке C++ на основе конструкции if...else, синтаксис которой имеет следующий вид:

```
if (Условное_выражение)
    Сделать нечто, когда условное_выражение равно true;
else // Необязательно
    Сделать нечто иное, когда условное_выражение равно false;
```

Таким образом, конструкция if...else, позволяющая программе перемножать числа, если пользователь вводит 'm', и суммировать их в противном случае, выглядит следующим образом:

```
if (userSelection == 'm')
    result = num1 * num2; // Произведение
else
    result = num1 + num2; // Сумма
```

ПРИМЕЧАНИЕ

В языке C++ результат true выражения по существу означает, что оно возвратило не значение false, которое является нулем. Таким образом, условное выражение, возвращающее любое ненулевое число, как положительное, так и отрицательное, по существу рассматривается как возвращающее значение true.

Проанализируем конструкцию в листинге 6.1, обрабатывающую условное выражение и позволяющую пользователю решить, следует ли умножить или просуммировать два числа.

ЛИСТИНГ 6.1. Умножение или сложение двух целых чисел на основе пользовательского ввода

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите два целых числа: " << endl;
6:     int num1 = 0, num2 = 0;
7:     cin >> num1;
8:     cin >> num2;
9:
10:    cout << "'m' для умножения, любое иное для сложения: ";
11:    char userSelection = '\0';
12:    cin >> userSelection;
13:
14:    int result = 0;
15:    if (userSelection == 'm')
16:        result = num1 * num2;
```

```
17:     else
18:         result = num1 + num2;
19:
20:     cout << "Результат: " << result << endl;
21:
22:     return 0;
23: }
```

Результат

Введите два целых числа:

25

56

'm' для умножения, любое иное для сложения: **m**

Результат: 1400

Следующий запуск:

Введите два целых числа:

25

56

'm' для умножения, любое иное для сложения: **a**

Результат: 81

Анализ

Обратите внимание на использование `if` в строке 15 и `else` в строке 17. В строке 15 мы требуем от компилятора выполнить умножение, если следующее за `if` выражение (`userSelection == 'm'`) истинно (возвращает значение `true`), или выполнить сложение, если выражение ложно (возвращает значение `false`). Выражение (`userSelection == 'm'`) возвращает значение `true`, когда пользователь вводит символ 'm' (с учетом регистра), и значение `false` в любом другом случае. Таким образом, эта простая программа моделирует блок-схему на рис. 6.1 и демонстрирует, как ваше приложение может вести себя по-разному в разных ситуациях.

ПРИМЕЧАНИЕ

Часть `else` конструкции `if...else` является необязательной и может не использоваться в тех ситуациях, когда при ложности условия не нужно делать ничего.

ВНИМАНИЕ!

Если бы строка 15 в листинге 6.1 выглядела как

```
15:     if (userSelection == 'm');
```

то конструкция `if` была бы бессмысленной, поскольку она завершилась бы в той же строке пустой инструкцией (точкой с запятой). Будьте внимательны и избегайте такой ситуации, поскольку вы не получите от компилятора сообщение об ошибке, если за `if` не следует часть `else`. Некоторые хорошие компиляторы в такой ситуации предупреждают об этом сообщением наподобие “empty control statement” (пустая управляющая инструкция).

Условное выполнение нескольких инструкций

Если вы хотите выполнить не одну, а несколько инструкций в зависимости от условий, заключите их в блок. По существу, это фигурные скобки ({...}), включающие несколько инструкций, которые будут выполняться как единая инструкция. Рассмотрим пример:

```
if (Условие)
{
    // Блок при истинности условия
    Инструкция 1;
    Инструкция 2;
}
else
{
    // Блок при ложности условия
    Инструкция 3;
    Инструкция 4;
}
```

Такие блоки именуются также *составными инструкциями* (compound statement).

Листинг 6.2 представляет собой более безопасную версию листинга 4.6 из занятия 4, “Массивы и строки”. В нем используется составная инструкция, которая копирует пользовательский ввод в статический символьный массив, если длина пользовательского ввода это позволяет.

ЛИСТИНГ 6.2. Проверка емкости перед копированием строки в символьный массив

```
0: #include <iostream>
1: #include <string>
2: #include <string.h>
3: using namespace std;
4: int main()
5: {
6:     cout << "Введите строку текста: " << endl;
7:     string userInput;
8:     getline(cin, userInput);
9:
10:    char copyInput[20] = { '\0' };
11:    if (userInput.length() < 20) // Проверка границ
12:    {
13:        strcpy(copyInput, userInput.c_str());
14:        cout << "copyInput содержит: " << copyInput << endl;
15:    }
16:    else
17:        cout << "Превышение размера строки!" << endl;
18:
19:    return 0;
20: }
```

Результат

Введите строку текста:

Короткая строка

соруInput содержит: Короткая строка

Следующий запуск:

Введите строку текста:

Длинная строка, не помещающаяся в массив

Превышение размера строки!

Анализ

Обратите внимание, как в строке 11 сравниваются длина строки и размер буфера. Интересным в этой проверке является также присутствие блока инструкций в строках 12–15 (составной инструкции).

Вложенные инструкции if

Нередки ситуации, когда необходимо проверить несколько разных условий, некоторые из которых зависят от результата проверки предыдущего условия. Язык C++ допускает вложенные инструкции if для выполнения таких действий.

Вложенные инструкции if имеют следующий синтаксис:

```
if (Условие1)
{
    Сделать_Нечто1;
    if(Условие2)
        Сделать_Нечто2;
    else
        Сделать_Нечто_Иное2;
}
else
    Сделать_Нечто_Иное1;
```

Рассмотрим приложение, подобное представленному в листинге 6.1, в котором пользователь, вводя 'd' или 'm', может указать приложению, следует ли разделить или умножить значения. Деление при этом должно быть разрешено, только если делитель отличается от нуля. Поэтому в дополнение к проверке вводимой пользователем команды в случае деления следует проверить, не является ли делитель нулем. Для этого в листинге 6.3 используется вложенная конструкция if.

ЛИСТИНГ 6.3. Использование вложенных if в приложении умножения или деления чисел

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите два числа: " << endl;
```

```
6:   float num1 = 0, num2 = 0;
7:   cin >> num1;
8:   cin >> num2;
9:
10:  cout << "Введите 'd' для деления, иное для умножения: ";
11:  char userSelection = '\0';
12:  cin >> userSelection;
13:
14:  if (userSelection == 'd')
15:  {
16:      cout << "Вы запросили деление." << endl;
17:      if (num2 != 0)
18:          cout << num1<<"/"<<num2<<" = "<< num1/num2 << endl;
19:      else
20:          cout << "Деление на 0 запрещено." << endl;
21:  }
22:  else
23:  {
24:      cout << "Вы запросили умножение." << endl;
25:      cout << num1<<"x"<<num2<<" = "<< num1*num2 << endl;
26:  }
27:
28:  return 0;
29: }
```

Результат

Введите два числа:

45

9

Введите 'd' для деления, иное для умножения: **m**

Вы запросили умножение.

45x9 = 405

Следующий запуск:

Введите два числа:

22

7

Введите 'd' для деления, иное для умножения: **d**

Вы запросили деление.

22/7 = 3.14286

Последний запуск:

Введите два числа:

365

0

Введите 'd' для деления, иное для умножения: **d**

Вы запросили деление.

Деление на 0 запрещено.

Анализ

Результаты содержат вывод трех запусков программы с тремя разными наборами входных данных. Как видите, программа использовала различные пути выполнения кода для каждого из этих трех запусков. По сравнению с листингом 6.1 эта программа имеет довольно много изменений.

- Числа хранятся в переменных типа `float`, способных хранить десятичные числа, что очень важно при делении.
- Условие инструкции `if` отличается от использованного в листинге 6.1. Вы больше не проверяете, ввел ли пользователь 'm'; выражение `(userSelection=='d')` в строке 14 возвращает значение `true`, если пользователь ввел 'd'. Если это так, то пользователь запросил деление.
- С учетом того, что эта программа делит два числа, и делитель вводится пользователем, нужно удостовериться, что делитель не равен нулю. Это выполняется в строке 17 с помощью вложенной инструкции `if`.

Таким образом, программа демонстрирует, как вложенные конструкции `if` могут оказаться очень полезными при решении различных задач, связанных с вычислением с несколькими параметрами.

СОВЕТ

Отступы, сделанные в коде, необязательны, но существенно повышают удобочитаемость вложенных конструкций `if`. Многие современные интегрированные среды разработки вставляют такие отступы автоматически.

Обратите внимание: конструкции `if...else` можно группировать. Программа в листинге 6.4 запрашивает у пользователя день недели, а затем, используя групповую конструкцию `if...else`, сообщает, в честь чего назван этот день.

ЛИСТИНГ 6.4. Узнайте, в честь чего назван день недели

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     enum DaysOfWeek
6:     {
7:         Sunday = 0,
8:         Monday,
9:         Tuesday,
10:        Wednesday,
11:        Thursday,
12:        Friday,
13:        Saturday
14:    };
15:
```

```
16: cout << "Узнайте, в честь чего назван день недели!" << endl;
17: cout << "Введите номер дня недели (воскресенье = 0): ";
18:
19: int dayInput = Sunday; // Инициализация воскресеньем
20: cin >> dayInput;
21:
22: if (dayInput == Sunday)
23:     cout << "Воскресенье названо в честь Солнца" << endl;
24: else if (dayInput == Monday)
25:     cout << "Понедельник назван в честь Луны" << endl;
26: else if (dayInput == Tuesday)
27:     cout << "Вторник назван в честь Марса" << endl;
28: else if (dayInput == Wednesday)
29:     cout << "Среда названа в честь Меркурия" << endl;
30: else if (dayInput == Thursday)
31:     cout << "Четверг назван в честь Юпитера" << endl;
32: else if (dayInput == Friday)
33:     cout << "Пятница названа в честь Венеры" << endl;
34: else if (dayInput == Saturday)
35:     cout << "Суббота названа в честь Сатурна" << endl;
36: else
37:     cout << "Неверный ввод" << endl;
38:
39: return 0;
40: }
```

Результат

```
Узнайте, в честь чего назван день недели!
Введите номер дня недели (воскресенье = 0): 5
Пятница названа в честь Венеры
```

Следующий запуск:

```
Узнайте, в честь чего назван день недели!
Введите номер дня недели (воскресенье = 0): 9
Неверный ввод
```

Анализ

Конструкция `if...else if`, используемая в строках 22–37, проверяет ввод пользователя и генерирует соответствующий вывод. Вывод во втором запуске свидетельствует, что программа в состоянии указать пользователю, что он ввел номер вне диапазона 0–6, а следовательно, не соответствующий никакому дню недели. Преимущество этой конструкции в том, что она отлично подходит для проверки множества взаимоисключающих условий, т.е. понедельник не может быть вторником, а недопустимый ввод не может быть никаким днем недели. Другим интересным моментом, на который стоит обратить внимание в этой программе, является использование перечисления `Days OfWeek`, объявленного в строке 5 и используемого повсюду в инструкции `if`. Можно

было бы просто сравнивать пользовательский ввод с целочисленными значениями; так, например, 0 соответствовал бы воскресенью и т.д. Однако использование перечисляемой константы наподобие `Sunday` делает код более наглядным.

Условная обработка с использованием конструкции `switch-case`

Задача конструкции `switch-case` в том, чтобы сравнить результат некоего выражения с набором заранее определенных возможных константных значений и выполнить разные действия, соответствующие каждой из этих констант. Новые ключевые слова C++, которые используются в такой конструкции, — это `switch`, `case`, `default` и `break`.

Конструкция `switch-case` имеет следующий синтаксис:

```
switch (Выражение)
{
    case Метка_A:
        Сделай_Нечто;
        break;
    case Метка_B:
        Сделай_Нечто_Другое;
        break;
    // И так далее...
    default:
        Сделай_Нечто_Если_Выражение_Не_Равно_Ничему_Выше;
}
```

Код вычисляет результат *Выражения* и сравнивает его на равенство с каждой из меток частей `case` ниже (каждая *Метка* должна быть целочисленной константой), а затем выполняет код после этой метки. Если результат выражения не равен метке *Метка_A*, он сравнивается с меткой *Метка_B*. Если значения совпадают, выполняется часть *Сделай_Нечто_Другое*. Выполнение продолжается до тех пор, пока не встретится оператор `break`. Сейчас вы впервые встретились с оператором `break`. Он означает выход из блока кода. Операторы `break` в рассматриваемой конструкции не обязательны, однако без них выполнение продолжится в коде следующей метки и так далее до конца всей конструкции (или до оператора `break`). Такого явления, как правило, желательно избегать. Часть конструкции `default` также является необязательной; она выполняется тогда, когда результат выражения не соответствует ни одной из меток в конструкции `switch-case`.

СОВЕТ

Отступы, сделанные в коде, необязательны, но существенно повышают удобочитаемость вложенных конструкций `if`. Многие современные интегрированные среды разработки вставляют такие отступы автоматически.

Код листинга 6.5 является эквивалентом программы для дней недели из листинга 6.4, но с использованием конструкции `switch-case`.

ЛИСТИНГ 6.5. Узнайте, в честь чего назван день недели, с помощью конструкции switch-case

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     enum DaysOfWeek
6:     {
7:         Sunday = 0,
8:         Monday,
9:         Tuesday,
10:        Wednesday,
11:        Thursday,
12:        Friday,
13:        Saturday
14:    };
15:
16:    cout << "Узнайте, в честь чего назван день недели!" << endl;
17:    cout << "Введите номер дня недели (воскресенье = 0): ";
18:
19:    int dayInput = Sunday; // Инициализация воскресеньем
20:    cin >> dayInput;
21:
22:    switch(dayInput)
23:    {
24:        case Sunday:
25:            cout << "Воскресенье названо в честь Солнца" << endl;
26:            break;
27:
28:        case Monday:
29:            cout << "Понедельник назван в честь Луны" << endl;
30:            break;
31:
32:        case Tuesday:
33:            cout << "Вторник назван в честь Марса" << endl;
34:            break;
35:
36:        case Wednesday:
37:            cout << "Среда названа в честь Меркурия" << endl;
38:            break;
39:
40:        case Thursday:
41:            cout << "Четверг назван в честь Юпитера" << endl;
42:            break;
43:
44:        case Friday:
45:            cout << "Пятница названа в честь Венеры" << endl;
```

```
46:         break;
47:
48:     case Saturday:
49:         cout << "Суббота названа в честь Сатурна"    << endl;
50:         break;
51:
52:     default:
53:         cout << "Неверный ввод" << endl;
54:         break;
55:     }
56:
57:     return 0;
58: }
```

Результат

Узнайте, в честь чего назван день недели!
Введите номер дня недели (воскресенье = 0): 5
Пятница названа в честь Венеры

Следующий запуск:

Узнайте, в честь чего назван день недели!
Введите номер дня недели (воскресенье = 0): 9
Неверный ввод

Анализ

Строки 22–55 содержат конструкцию switch-case, осуществляющую различный вывод в зависимости от того, какое целое число введено пользователем и сохранено в переменной dayInput. Когда пользователь вводит число 5, приложение вычисляет значение выражения dayInput в инструкции switch, которое оказывается равным 5, и сравнивает его с метками, являющимися константами перечисления — от Sunday (значение 0) до Thursday (значение 4), пропуская код после каждой из них, поскольку ни одна из этих констант не равна 5. По достижении метки Friday, значение 5 которой равно выражению dayInput инструкции switch, выполняется код, находящийся ниже нее, до тех пор, пока не встречается оператор break, предписывающий завершить выполнение и выйти из конструкции switch. При следующем запуске, когда пользователем вводится недопустимое значение, выполнение достигает части default, и выполнение кода ниже этой метки выводит сообщение о некорректных данных.

В этой программе конструкция switch-case используется для получения того же вывода, который создан конструкцией if...else...if в листинге 6.4. Однако версия с использованием конструкции switch-case выглядит более структурированной и, пожалуй, лучше подходящей к ситуациям, когда необходимо сделать большее, чем просто вывести строку на экран (в этом случае код ниже меток следовало бы заключать в фигурные скобки, создавая блоки инструкций).

Тернарный условный оператор (? :)

Язык C++ предоставляет интересный и мощный оператор, называемый *тернарным условным оператором* (conditional operator), который подобен сжатой конструкции if...else.

Тернарный условный оператор, называемый также просто *условным оператором*, использует три операнда:

```
(Логическое_выражение) ? Инструкция_для_true : Инструкция_для_false;
```

Такой оператор применим, например, при компактном сравнении двух чисел, как показано ниже.

```
int max = (num1 > num2) ? num1 : num2; // Максимум из num1 и num2
```

В листинге 6.6 показано применение оператора ? :.

ЛИСТИНГ 6.6. Использование оператора ? : для получения большего из двух чисел

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Введите два числа:" << endl;
6:     int num1 = 0, num2 = 0;
7:     cin >> num1;
8:     cin >> num2;
9:
10:    int max = (num1 > num2)? num1 : num2;
11:    cout << "Большее из " << num1 << " и " \
12:        << num2 << " равно: " << Max << endl;
13:
14:    return 0;
15: }
```

Результат

Введите два числа:

365

-1

Большее из 365 и -1 равно: 365

Анализ

Интерес представляет код строки 10. Он содержит очень компактное выражение, принимающее решение о том, какое из двух введенных чисел больше. Используя конструкцию if...else, эту строку можно было бы переписать следующим образом:

```
int max = 0;
if (num1 > num2)
    max = num1;
else
    max = num2;
```

Таким образом, тернарный условный оператор сэкономил нам несколько строк кода! Однако экономия строк кода не должна быть главным приоритетом программиста. Одни программисты предпочитают троичные условные операторы, другие нет. Главное, чтобы пути выполнения кода были легко понятны читающему программу.

РЕКОМЕНДУЕТСЯ

Используйте перечисления и в выражениях switch, чтобы сделать код более удобочитаемым.

Помните о разделе default инструкции switch, не забывайте записывать его (кроме тех случаев, когда он гарантированно не нужен).

Проверяйте, не забыли ли вы включить оператор break в конец каждой инструкции case.

НЕ РЕКОМЕНДУЕТСЯ

Не добавляйте две инструкции case с одинаковой меткой — это не имеет смысла и не будет компилироваться.

Не усложняйте свои инструкции case, отказываясь от оператора break и разрешая последовательное выполнение. Такое решение может привести к неработоспособности кода позже, при перемещении инструкции case.

Не используйте в операторах ?: сложные условия или выражения.

СОВЕТ

Ожидается, что в стандарте C++17 будет введена возможность условной компиляции с помощью конструкции `if constexpr`, а также инициализаторы в конструкциях `if` и `switch`. Подробнее об этом рассказывается на занятии 29, “Что дальше”.

Выполнение кода в циклах

К этому времени вы уже узнали, как заставить программу вести себя по-разному, когда переменные содержат разные значения. Например, код из листинга 6.2 осуществлял умножение, когда пользователь вводил символ 'm', а в противном случае выполнял сложение. Но что если пользователь не хочет, чтобы программа на этом закончила выполнение? Что если он хочет выполнить еще одну операцию суммирования или умножения или, возможно, еще пять? То есть если программисту нужно многократное выполнение существующего кода?

В этом случае в программе необходим цикл.

Рудиментарный цикл с использованием инструкции goto

Как подразумевает название инструкции `goto`, она приказывает процессору продолжать выполнение программы с определенной точки в коде. Вы можете использовать ее для перехода назад и повторного выполнения определенных инструкций. Синтаксис инструкции `goto` таков:

```
SomeFunction()
{
    Start:    // Эта инструкция называется меткой

    Многократно_выполняемый_код;

    goto Start;
}
```

Вы объявляете метку `Start` и используете инструкцию `goto` для повторного выполнения кода с этого места, как показано в листинге 6.7. Если вы не выполните условие, которое приведет к пропуску выполнения инструкции `goto`, и если в многократно повторяемом коде не содержится оператор `return`, выполняемый при определенных условиях, то часть кода между командой `goto` и меткой будет повторяться бесконечно и программа никогда не завершится.

ЛИСТИНГ 6.7. Запрос повторения вычислений с использованием инструкции `goto`

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     Start:
6:         int num1 = 0, num2 = 0;
7:
8:         cout << "Введите два целых числа: " << endl;
9:         cin >> num1;
10:        cin >> num2;
11:
12:        cout << num1 << "x" << num2 << " = " << num1*num2 << endl;
13:        cout << num1 << "+" << num2 << " = " << num1+num2 << endl;
14:
15:        cout << "Еще раз (y/n)?" << endl;
16:        char Repeat = 'y';
17:        cin >> Repeat;
18:
19:        if (Repeat == 'y')
20:            goto Start;
21:
22:        cout << "До свидания!" << endl;
```

```
23:
24:     return 0;
25: }
```

Результат

Введите два целых числа:

56

25

$56 \times 25 = 1400$

$56 + 25 = 81$

Еще раз (y/n)?

y

Введите два целых числа:

95

-47

$95 \times -47 = -4465$

$95 + -47 = 48$

Еще раз (y/n)?

n

До свидания!

Анализ

Основное различие между кодами листингов 6.7 и 6.1 состоит в том, что последнему требуются два запуска, чтобы позволить пользователю ввести новый набор чисел и увидеть результат их сложения или умножения. Код листинга 6.7 делает это при одном запуске, циклически повторяя запрос пользователю, не желает ли он выполнить другую операцию. Код, фактически обеспечивающий это повторение, находится в строке 20, в которой вызывается инструкция `goto`, если пользователь вводит символ 'y'. В результате использования инструкции `goto` в строке 20 программа переходит к метке `Start`, объявленной в строке 5, что фактически перезапускает программу.

ВНИМАНИЕ!

Использование инструкции `goto` не рекомендуется для создания циклов, поскольку его массовое применение может привести к непредсказуемой последовательности выполнения кода, когда выполнение может переходить с одной строки на другую без всякого очевидного порядка или последовательности, оставляя переменные в непредсказуемых состояниях.

Тяжелый случай программы, использующей инструкции `goto`, называется *запутанной программой*, или *кодом спагетти* (spaghetti code). Объясняемые на следующих страницах циклы `while`, `do...while` и `for` позволяют избежать использования инструкции `goto`.

Единственная причина упоминания здесь инструкции `goto` — объяснить код, который ее использует.

Цикл `while`

Ключевое слово `while` языка C++ позволяет добиться того же, что инструкция `goto` делала в листинге 6.7, но правильным способом. Синтаксис этого цикла имеет следующий вид:

```
while(Выражение)
{
    // Если Выражение == true
    Блок_Инструкций;
}
```

Выполнение блока инструкций многократно повторяется до тех пор, пока *Выражение* имеет значение `true`. Очень важно, чтобы в коде была ситуация, когда *Выражение* получает значение `false`, иначе цикл `while` никогда не завершится.

Листинг 6.8 является эквивалентом листинга 6.7, позволяющим пользователю повторять цикл вычисления, но с использованием цикла `while` вместо инструкции `goto`.

ЛИСТИНГ 6.8. Использование цикла `while` для многократного повторения вычислений

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     char userSelection = 'm'; // Исходное значение
6:
7:     while(userSelection != 'x')
8:     {
9:         cout << "Введите два целых числа: " << endl;
10:        int num1 = 0, num2 = 0;
11:        cin >> num1;
12:        cin >> num2;
13:
14:        cout << num1 << "x" << num2 << " = " << num1*num2 << endl;
15:        cout << num1 << "+" << num2 << " = " << num1+num2 << endl;
16:
17:        cout << "'x' для выхода, иное для повтора" << endl;
18:        cin >> userSelection;
19:    }
20:
21:    cout << "До свидания!" << endl;
22:
23:    return 0;
24: }
```

Результат

Введите два целых числа:

56

25

56x25 = 1400

56+25 = 81

'x' для выхода, иное для повтора

x

Введите два целых числа:

365

-5

365x-5 = -1825

365+-5 = 360

'x' для выхода, иное для повтора

x

До свидания!

Анализ

Цикл `while` в строках 7–19 содержит большую часть логики этой программы. Обратите внимание, что цикл `while` проверяет выражение `(userSelection != 'x')` и продолжает выполнение, только если оно имеет значение `true`. Для обеспечения первого запуска символьная переменная `userSelection` инициализируется в строке 5 значением `'m'`. Это значение может быть любым, кроме `'x'` (иначе условие не будет выполняться с самого первого цикла и приложение закончит выполнение, не позволив пользователю ничего сделать). Первый запуск очень прост, но затем в строке 17 у пользователя спрашивают, не желает ли он выполнить вычисления снова. Строка 18 принимает сделанный пользователем выбор; здесь вы изменяете значение выражения, которое проверяет цикл `while`, давая программе шанс продолжить выполнение или завершить его. По завершении первого цикла выполнение возвращается к проверке выражения цикла `while` в строке 7 и цикл повторяется, если пользователь ввел значение, отличное от `'x'`. Если же пользователь ввел `'x'`, то выражение в строке 7 становится равным `false` и цикл `while` завершается. Приложение после этого также закончит работу, вежливо попрощавшись.

ПРИМЕЧАНИЕ

Выполнение цикла называется также *итерацией* (iteration). Инструкции, включающие циклы `while`, `do...while` и `for`, называют также *итеративными*.

Цикл `do...while`

Бывают ситуации (как в листинге 6.8), когда определенный фрагмент кода (тело цикла) должен гарантированно выполняться по крайней мере один раз. Для этого используется цикл `do...while`.

Его синтаксис имеет следующий вид:

```
do
{
    Блок_Инструкций; // Выполняется как минимум один раз
} while(Условие);    // Цикл завершается, если Условие ложно
```

Обратите внимание, что строка, содержащая часть `while(Условие)`, заканчивается точкой с запятой. Этим данный цикл отличается от цикла `while`, в котором точка с запятой фактически завершила бы цикл в первой строке, приведя к пустой инструкции.

ЛИСТИНГ 6.9. Использование цикла `do...while` для повторного выполнения блока кода

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     char userSelection = 'x'; // Исходное значение
6:     do
7:     {
8:         cout << "Введите два целых числа: " << endl;
9:         int num1 = 0, num2 = 0;
10:        cin >> num1;
11:        cin >> num2;
12:
13:        cout << num1 << "x" << num2 << " = " << num1 * num2 << endl;
14:        cout << num1 << "+" << num2 << " = " << num1 + num2 << endl;
15:
16:        cout << "'x' для выхода, иное для повтора" << endl;
17:        cin >> userSelection;
18:    } while(userSelection != 'x');
19:
20:    cout << "До свидания!" << endl;
21:
22:    return 0;
23: }
```

Результат

Введите два целых числа:

654

-25

654x-25 = -16350

654+-25 = 629

'x' для выхода, иное для повтора

m

Введите два целых числа:

909

101

909x101 = 91809

909+101 = 1010

'x' для выхода, иное для повтора

x

До свидания!

Анализ

Эта программа по своему поведению и выводу очень похожа на предыдущую. Действительно, единственное их различие — в ключевом слове `do` в строке 6 и использовании цикла `while` в строке 18. Выполнение кода происходит последовательно, строка за строкой, пока в строке 18 не встретится заголовок цикла `while`, проверяющий значение выражения `(userSelection != 'x')`. Когда оно равно `true` (т.е. пользователь не ввел 'x' для выхода из программы), цикл повторяется. Когда выражение возвращает значение `false` (т.е. пользователь ввел 'x'), выполнение покидает цикл и продолжается до окончания приложения, включая вывод прощания.

Цикл `for`

Цикл `for` немного сложнее и обладает выражением инициализации, выполняемым только однажды (обычно для инициализации счетчика), условиями выхода (как правило, использующего этот счетчик) и выполняемого в конце каждого цикла действия (обычно инкремента или иного изменения этого счетчика).

Синтаксис цикла `for` таков:

```
for (Выражение инициализации, выполняемое только раз;  
    Условие выхода, проверяемое в начале каждой итерации;  
    Выражение цикла, выполняемое в конце каждой итерации)  
{  
    Блок инструкций;  
}
```

Цикл `for` — это средство, позволяющее программисту определить счетчик с исходным значением, проверять его значение в условии выхода в начале каждого цикла и изменять значение счетчика в конце цикла.

В листинге 6.10 показан эффективный способ доступа к элементам массива с помощью цикла `for`.

ЛИСТИНГ 6.10. Использование цикла `for` для ввода и отображения элементов статического массива

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: int main()
```

```
4: {
5:     const int ARRAY_LENGTH = 5;
6:     int myNums[ARRAY_LENGTH] = {0};
7:
8:     cout << "Заполнение массива " << ARRAY_LENGTH
9:         << " числами" << endl;
10:    for(int counter = 0; counter < ARRAY_LENGTH; ++counter)
11:    {
12:        cout << "Элемент " << counter << ": ";
13:        cin >> myNums[counter];
14:    }
15:
16:    cout << "Вывод содержимого массива: " << endl;
17:
18:    for(int counter = 0; counter < ARRAY_LENGTH; ++counter)
19:        cout << "Элемент " << counter << " = "
20:            << myNums[counter] << endl;
21:    return 0;
22: }
```

Результат

Заполнение массива 5 числами

Элемент 0: **365**

Элемент 1: **31**

Элемент 2: **24**

Элемент 3: **-59**

Элемент 4: **65536**

Вывод содержимого массива:

Элемент 0 = 365

Элемент 1 = 31

Элемент 2 = 24

Элемент 3 = -59

Элемент 4 = 65536

Анализ

Листинг 6.10 содержит два цикла `for` — в строках 10 и 18. Первый помогает ввести элементы в массив целых чисел, а второй — отобразить их. Синтаксис обоих циклов `for` идентичен. Оба объявляют индексную переменную `counter` для доступа к элементам массива. Значение этой переменной увеличивается в конце каждого цикла, поэтому на следующей итерации цикла она позволяет обратиться к очередному элементу. Среднее выражение в цикле `for` — это условие выхода. Оно проверяет, находится ли значение переменной `counter`, увеличенное в конце каждого цикла, все еще в пределах границ массива (сравнивая его со значением `ARRAY_LENGTH`). Этим гарантируется, что цикл `for` никогда не выйдет за границы массива.

ПРИМЕЧАНИЕ

Такая переменная, как `counter` из листинга 6.10, которая позволяет обращаться к элементам коллекции (например, массива), называется *итератором* (*iterator*).

Область видимости итератора, объявленного в пределах конструкции `for`, ограничивается этой конструкцией. Таким образом, во втором цикле `for` листинга 6.10 эта переменная, объявленная повторно, фактически является новой переменной.

Применение инициализации, условия выхода и выражения, выполняемого в конце каждого цикла, является необязательным. Вполне возможно записать цикл `for` без некоторых из указанных выражений (или вовсе без них), как показано в листинге 6.11.

ЛИСТИНГ 6.11. Использование цикла `for`
 без выражения изменения в заголовке цикла

```

0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Без выражения изменения (третье выражение пропущено)
6:     for(char userSelection = 'm'; (userSelection != 'x'); )
7:     {
8:         cout << "Введите два целых числа: " << endl;
9:         int num1 = 0, num2 = 0;
10:        cin >> num1;
11:        cin >> num2;
12:
13:        cout << num1<<"x"<<num2 << " = " << num1 * num2 << endl;
14:        cout << num1<<"+"<<num2 << " = " << num1 + num2 << endl;
15:
16:        cout << "'x' для выхода, иное для повтора" << endl;
17:        cin >> userSelection;
18:    }
19:
20:    cout << "До свидания!" << endl;
21:
22:    return 0;
23: }
```

Результат

Введите два целых числа:

56

25

56x25 = 1400

56+25 = 81

'x' для выхода, иное для повтора

m

Введите два целых числа:

789

-36

789х-36 = -28404

789+ -36 = 753

'х' для выхода, иное для повтора

х

До свидания!

Анализ

Этот листинг идентичен коду листинга 6.8, который использовал цикл `while`; единственное отличие — в использовании цикла `for` в строке 6. Самое интересное в этом цикле `for` то, что он содержит только выражение инициализации и условие выхода, без изменения значения переменной в конце каждой итерации.

ПРИМЕЧАНИЕ

В пределах выражения инициализации цикла `for` можно инициализировать несколько переменных. Цикл `for` в листинге 6.11 при инициализации нескольких переменных выглядел бы следующим образом:

```
for(int counter1 = 0, counter2 = 5; // Инициализация
    counter1 < ARRAY_LENGTH;       // Условие
    ++counter1, --counter2)         // Инкремент, декремент
```

Обратите внимание на дополнительную переменную `counter2`, которая инициализируется значением 5.

В выражении цикла, выполняемом на каждой итерации, вполне можно выполнять и декремент.

Цикл `for` для диапазона

Стандарт C++11 ввел новый вариант цикла `for`, который работает со всеми элементами из диапазона значений, такого как, например, массив, с помощью более простого и удобочитаемого кода.

Синтаксис цикла `for` для диапазона также использует ключевое слово `for`:

```
for (VarType varName : Последовательность)
{
    // varName в теле цикла содержит элемент последовательности
}
```

Например, для данного массива целых чисел `someNums` можно использовать цикл `for` для диапазона, который будет выводить все элементы, содержащиеся в этом массиве, следующим образом:

```
int someNums[] = { 1, 101, -1, 40, 2040 };
cout << "Элементами массива являются:" << endl;
for(int aNum : someNums) // Цикл for для диапазона
    cout << aNum << endl;
```

СОВЕТ

Можно дополнительно упростить приведенную инструкцию с помощью автоматического вывода типа переменной (ключевое слово `auto`) и составить обобщенный цикл, который будет работать с массивами элементов любого типа:

```
for(auto anElement : elements) // Цикл for для диапазона
    cout << anElement << endl;
```

Ключевое слово `auto` и автоматический вывод типа переменной рассматривались на занятии 3, “Использование переменных и констант”.

В листинге 6.12 продемонстрировано применение цикла `for` для диапазона для разных типов.

ЛИСТИНГ 6.12. Применение цикла `for` для диапазона к массивам и `std::string`

```
0: #include<iostream>
1: #include <string>
2: using namespace std;
3:
4: int main()
5: {
6:     int someNums[] = { 1, 101, -1, 40, 2040 };
7:
8:     for(const int& aNum : someNums)
9:         cout << aNum << ' ';
10:    cout << endl;
11:
12:    for(auto anElement : { 5, 222, 110, -45, 2017 })
13:        cout << anElement << ' ';
14:    cout << endl;
15:
16:    char charArray[] = { 'h', 'e', 'l', 'l', 'o' };
17:    for(auto aChar : charArray)
18:        cout << aChar << ' ';
19:    cout << endl;
20:
21:    double moreNums[] = { 3.14, -1.3, 22, 10101 };
22:    for(auto anElement : moreNums)
23:        cout << anElement << ' ';
24:    cout << endl;
25:
26:    string sayHello{ "Hello World!" };
27:    for(auto anElement : sayHello)
28:        cout << anElement << ' ';
29:    cout << endl;
30:
31:    return 0;
32: }
```

Результат

```
1 101 -1 40 2040
5 222 110 -45 2017
h e l l o
3.14 -1.3 22 10101
H e l l o W o r l d !
```

Анализ

Этот код содержит несколько реализаций цикла `for` для диапазона — в строках 8, 12, 17, 22 и 27 соответственно. Каждый из них выводит на консоль содержимое некоторого диапазона по одному элементу. Интересно, что несмотря на разную природу выбранных диапазонов (от массива целых чисел `someNums` в строке 8 и неопределенного диапазона в строке 12 до массива `charArray` элементов типа `char` в строке 17, и даже `std::string` в строке 27) синтаксис цикла `for` для диапазона остается одним и тем же.

Эта простота делает цикл `for` для диапазона одним из наиболее популярных нововведений в C++.

Изменение поведения цикла с использованием операторов `continue` и `break`

В некоторых случаях (особенно в сложных циклах с большим количеством параметров в условии) может не получиться грамотно сформулировать условие выхода из цикла, и тогда вам придется изменять поведение программы уже в пределах цикла. В этом вам могут помочь операторы `continue` и `break`.

Оператор `continue` позволяет возобновить выполнение с начала цикла. Он просто пропускает весь код, расположенный в теле цикла после него. Таким образом, результат выполнения оператора `continue` в цикле `while`, `do...while` или `for` сводится к переходу к условию выхода из цикла и повторному входу в блок цикла, если условие истинно.

ПРИМЕЧАНИЕ

В случае применения оператора `continue` в цикле `for` перед проверкой условия выхода выполняется выражение цикла (третье выражение в цикле `for`, которое обычно увеличивает значение счетчика).

Оператор `break` осуществляет выход из блока цикла, завершая выполнение цикла, в котором он был вызван.

ВНИМАНИЕ!

Обычно программисты полагают, что пока условия цикла выполняются, выполняется и весь код в цикле. Операторы `continue` и `break` изменяют это поведение и могут привести к непонятному интуитивно коду. Поэтому операторы `continue` и `break` следует использовать реже.

Бесконечные циклы, которые никогда не заканчиваются

Вспомните, что у циклов `while`, `do...while` и `for` есть условия, результат вычисления которых, равный `false`, приводит к завершению цикла. Если вы зададите условие, которое всегда возвращает значение `true`, цикл никогда не закончится.

Бесконечный цикл `while` выглядит следующим образом:

```
while(true) // Выражение условия всегда равно true
{
    Сделай_Нечто;
}
```

Бесконечный цикл `do...while` выглядит следующим образом:

```
do
{
    Сделай_Нечто;
} while(true); // Выражение условия всегда равно true
```

Бесконечный цикл `for` создается проще всего:

```
for(;;) // Условия выхода вообще нет
{
    Сделай_Нечто;
}
```

Как ни странно, у таких циклов действительно есть применение. Представьте, например, операционную систему, которая должна непрерывно проверять, подключено ли устройство USB к порту USB. Это действие должно выполняться регулярно, пока запущена операционная система.

Управление бесконечными циклами

Для выхода из бесконечного цикла (например, перед завершением работы операционной системы в предыдущем примере) вы можете использовать оператор `break` (как правило, в пределах блока `if(условие)`).

Вот пример использования оператора `break` для управления бесконечным циклом `while`:

```
while(true) // Выражение условия всегда равно true
{
    Сделай_Нечто;
    if(Выражение)
        break; // Выход из цикла, если Выражение равно true
}
```

Использование оператора break в бесконечном цикле do..while:

```
do
{
    Сделать_Нечто;
    if(Выражение)
        break; // Выход из цикла, когда Выражение равно true
} while(true); // Выражение условия всегда равно true
```

Использование оператора break в бесконечном цикле for:

```
for(;;) // Условия выхода нет, цикл for бесконечный
{
    Сделать_Нечто;
    if(Выражение)
        break; // Выход из цикла, когда Выражение равно true
}
```

Листинг 6.13 демонстрирует использование операторов continue и break для управления бесконечным циклом.

ЛИСТИНГ 6.13. Использование операторов continue и break для управления бесконечным циклом

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     for(;;) // Бесконечный цикл
6:     {
7:         cout << "Введите два целых числа: " << endl;
8:         int num1 = 0, num2 = 0;
9:         cin >> num1;
10:        cin >> num2;
11:
12:        cout << "Вы хотите исправить ввод? (y/n): ";
13:        char changeNumbers = '\0';
14:        cin >> changeNumbers;
15:
16:        if (changeNumbers == 'y')
17:            continue; // Перезапуск цикла!
18:
19:        cout << num1<<"x"<<num2<<" = "<< num1*num2 << endl;
20:        cout << num1<<"+ "<<num2<<" = "<< num1+num2 << endl;
21:
22:        cout << "'x' для выхода, иное для повтора" << endl;
23:        char userSelection = '\0';
24:        cin >> userSelection;
25:
26:        if (userSelection == 'x')
```

```
27:         break; // Выход из бесконечного цикла
28:     }
29:
30:     cout << "До свидания!" << endl;
31:
32:     return 0;
33: }
```

Результат

```
Введите два целых числа:
560
25
Вы хотите исправить ввод? (y/n): y
Введите два целых числа:
56
25
Вы хотите исправить ввод? (y/n): n
56x25 = 1400
56+25 = 81
'x' для выхода, иное для повтора
x
Введите два целых числа:
95
-1
Вы хотите исправить ввод? (y/n): n
95x-1 = -95
95+-1 = 94
'x' для выхода, иное для повтора
x
До свидания!
```

Анализ

Цикл `for` в строке 5 отличается от такового в листинге 6.11 тем, что он бесконечный — в этом цикле отсутствует условие выхода, проверяемое на каждой итерации. Другими словами, без оператора `break` этот цикл (а следовательно, и все приложение) никогда не завершится. Обратите внимание на вывод, который отличается от вывода предыдущих листингов, — он позволяет пользователю исправить введенные числа, прежде чем программа перейдет к вычислению суммы и произведения. Эта логика реализуется в строках 16 и 17 с использованием оператора `continue`, выполняемого при определенном условии. Когда пользователь вводит 'y' в ответ на запрос, хочет ли он исправить числа, условие в строке 16 возвращает значение `true`, а следовательно, выполняется оператор `continue`. Оператор `continue` возвращает выполнение к началу цикла, и пользователя снова просят ввести два целых числа. Аналогично в конце цикла, когда в ответ на предложение выйти из программы пользователь вводит символ 'x', условие в строке 26 становится истинным и выполняется следующий далее оператор `break`, заканчивающий цикл.

ПРИМЕЧАНИЕ

Для создания бесконечного цикла в листинге 6.13 использован пустой цикл `for(;;)`. Вы можете заменить его циклом `while(true)` или `do...while(true);` и получить тот же результат, хотя и будет использован цикл другого вида.

РЕКОМЕНДУЕТСЯ

Используйте цикл `do...while`, когда код в цикле должен быть выполнен по крайней мере один раз.

Используйте циклы `while`, `do...while` и `for` с хорошо продуманным условием выхода.

Используйте отступы в блоке кода, содержащемся в цикле, чтобы улучшить его удобочитаемость.

НЕ РЕКОМЕНДУЕТСЯ

Не используйте инструкцию `goto`.

Не используйте операторы `continue` и `break` без крайней необходимости.

Не используйте бесконечные циклы с оператором `break`, если этого можно избежать.

Программирование вложенных циклов

Вы уже встречались с вложенными инструкциями `if`. Точно так же иногда требуется вложить один цикл в другой. Предположим, у нас есть два массива целых чисел. Если вы хотите вычислить произведения каждого элемента массива `array1` и каждого элемента массива `array2`, то проще всего использовать вложенный цикл. Первый цикл перебирает элементы массива `array1`, а второй цикл, находящийся внутри первого, перебирает элементы массива `array2`.

Листинг 6.14 демонстрирует, как можно вложить один цикл в другой.

ЛИСТИНГ 6.14. Использование вложенных циклов для перемножения всех элементов двух массивов

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     const int ARRAY1_LEN = 3;
6:     const int ARRAY2_LEN = 2;
7:
8:     int myNums1[ARRAY1_LEN] = {35, -3, 0};
9:     int myNums2[ARRAY2_LEN] = {20, -1};
10:
11:     cout << "Перемножение всех элементов myNums1 со всеми "
12:          << "элементами myNums2:" << endl;
13:     for(int index1 = 0; index1 < ARRAY1_LEN; ++index1)
14:         for(int index2 = 0; index2 < ARRAY2_LEN; ++index2)
```

```
15:         cout << myNums1[index1] << " x " << myNums2[index2]
16:         << " = " << myNums1[index1]*myNums2[index2] << endl;
17:
18:     return 0;
19: }
```

Результат

Перемножение всех элементов myNums1 со всеми элементами myNums2:

35 x 20 = 700

35 x -1 = -35

-3 x 20 = -60

-3 x -1 = 3

0 x 20 = 0

0 x -1 = 0

Анализ

Рассматриваемые вложенные циклы for находятся в строках 13 и 14. Первый цикл for перебирает элементы массива myNums1, а второй — массива myNums2. Первый цикл for запускает второй цикл в пределах каждой своей итерации. Второй цикл for перебирает все элементы массива myNums2, причем при каждой итерации он умножает этот элемент на элемент, проиндексированный переменной index1 из первого, внешнего цикла. Для каждого элемента массива myNums1 второй цикл перебирает все элементы массива myNums2, в результате первый элемент массива myNums1 (с индексом 0) перемножается со всеми элементами массива myNums2. Затем второй элемент массива myNums1 перемножается со всеми элементами массива myNums2. И наконец третий элемент массива myNums1 перемножается со всеми элементами массива myNums2.

ПРИМЕЧАНИЕ

Для удобства (и чтобы не отвлекаться от темы рассмотрения) содержимое массивов в листинге 6.14 инициализировано в коде. В предыдущих примерах, например в листинге 6.10, показано, как позволить пользователю ввести числа в целочисленный массив.

Использование вложенных циклов для перебора многомерного массива

Из занятия 4, “Массивы и строки”, вы узнали о многомерных массивах. В листинге 4.3 происходит обращение к элементам двумерного массива из трех строк и трех столбцов. В этом листинге обращение к каждому элементу в каждой строке осуществлялось индивидуально и не была использована никакая автоматизация. Если бы массив стал большего размера или стало больше его размерностей, то для доступа к его элементам понадобился бы существенно более длинный код. Однако все может изменить использование циклов, показанное в листинге 6.15.

ЛИСТИНГ 6.15. Использование вложенных циклов для обхода элементов двумерного массива

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     const int NUM_ROWS = 3;
6:     const int NUM_COLS = 4;
7:
8:     // Двумерный массив целых чисел
9:     int myNums[NUM_ROWS][NUM_COLS] = {{ 34,  -1,  879, 22},
10:                                         { 24, 365, -101, -1},
11:                                         {-20, 40,   90, 97}};
12:
13:     // Обход всех строк массива
14:     for(int row = 0; row < NUM_ROWS; ++row)
15:     {
16:         // Обход каждой строки (по столбцам)
17:         for(int column = 0; column < NUM_COLS; ++column)
18:         {
19:             cout << "Integer[" << row << "][" << column
20:                 << "] = " << myNums[row][column] << endl;
21:         }
22:     }
23:
24:     return 0;
25: }
```

Результат

```
Integer[0][0] = 34
Integer[0][1] = -1
Integer[0][2] = 879
Integer[0][3] = 22
Integer[1][0] = 24
Integer[1][1] = 365
Integer[1][2] = -101
Integer[1][3] = -1
Integer[2][0] = -20
Integer[2][1] = 40
Integer[2][2] = 90
Integer[2][3] = 97
```

Анализ

Строки 14–22 содержат два цикла `for`, необходимых для обхода всех элементов двумерного массива целых чисел один за другим. Двумерный массив — это фактически

массив массивов целых чисел. Обратите внимание, что первый цикл `for` обращается к строкам (каждая из которых представляет собой массив целых чисел), а второй — к столбцам, т.е. осуществляет доступ к каждому элементу в этом массиве, расположенном в строке.

ПРИМЕЧАНИЕ

Скобки в листинге 6.15 вокруг вложенного цикла `for` использованы только для удобочитаемости. Эти конкретные вложенные циклы прекрасно работают и без фигурных скобок, поскольку инструкция цикла — это всего лишь одна инструкция, а не составная инструкция, требующая использования фигурных скобок.

Использование вложенных циклов для вычисления чисел Фибоначчи

Знаменитая последовательность чисел Фибоначчи — это ряд чисел, начинающихся с 0 и 1, где каждое последующее число представляет собой сумму предыдущих двух. Таким образом, ряд Фибоначчи начинается со следующей последовательности чисел:

0, 1, 1, 2, 3, 5, 8, ... и т.д.

В листинге 6.16 показано, как получить ряд Фибоначчи из любого количества чисел, ограниченного только размером целочисленной переменной, хранящей последнее число.

ЛИСТИНГ 6.16. Использование вложенных циклов для вычисления ряда Фибоначчи

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     const int numsToCalculate = 5;
6:     cout << "Вычисление по " << numsToCalculate
7:         << " чисел Фибоначчи" << endl;
8:
9:     int num1 = 0, num2 = 1;
10:    char wantMore = '\0';
11:    cout << num1 << " " << num2 << " ";
12:
13:    do
14:    {
15:        for(int counter=0; counter < numsToCalculate; ++counter)
16:        {
17:            cout << num1 + num2 << " ";
18:
19:            int num2Temp = num2;
20:            num2 = num1 + num2;
```

```

21:         num1 = num2Temp;
22:     }
23:
24:     cout << endl << "Продолжать (y/n)? ";
25:     cin >> wantMore;
26: }while(wantMore == 'y');
27:
28:     cout << "До свидания!" << endl;
29:
30:     return 0;
31: }

```

Результат

```

Вычисление по 5 чисел Фибоначчи
0 1 1 2 3 5 8
Продолжать (y/n)? y
13 21 34 55 89
Продолжать (y/n)? y
144 233 377 610 987
Продолжать (y/n)? y
1597 2584 4181 6765 10946
Продолжать (y/n)? n
До свидания!

```

Анализ

Внешний цикл `do...while` в строке 13 является основным, он запрашивает у пользователя, хочет ли он получать следующие числа. Внутренний цикл `for` в строке 15 решает задачу вычисления и отображения за один раз пяти очередных чисел Фибоначчи. В строке 19 значение переменной `num2` присваивается временной переменной, чтобы использовать его затем в строке 21. Обратите внимание, что без сохранения этого временного значения переменной `num1` было бы присвоено значение, измененное в строке 20, что было бы неправильно. Благодаря этим трем строкам цикл повторяется с новыми значениями в переменных `num1` и `num2`, если пользователь введет 'y' в ответ на вопрос о продолжении работы.

Резюме

На этом занятии вы узнали, что можно писать код, выполняющийся не только последовательно; условные инструкции позволяют создавать альтернативные пути выполнения и повторять блоки кода в циклах. Теперь вы знаете, как использовать конструкции `if...else` и `switch-case`, чтобы справиться с различными ситуациями, когда переменные содержат различные значения.

Для объяснения концепции циклов была представлена инструкция `goto`, однако сразу же было сделано предупреждение о том, что использовать ее не следует — в связи с

возможностью создания запутанного кода. Вы изучили циклы языка C++, использующие конструкции `while`, `do...while` и `for`, и узнали, как заставить циклы выполнять итерации бесконечно, чтобы создать бесконечные циклы, и использовать операторы `continue` и `break` для их контроля.

Вопросы и ответы

■ Что будет, если я пропущу оператор `break` в конструкции `switch-case`?

Оператор `break` позволяет выйти из конструкции `switch`. Без него продолжится выполнение инструкций в следующих частях `case`.

■ Как выйти из бесконечного цикла?

Для выхода из цикла используется оператор `break`. Оператор `return` позволяет выйти также из блока функции.

■ Мой цикл `while` выглядит как `while(Integer)`. Будет ли продолжаться цикл, если значение переменной `Integer` будет равно `-1`?

В идеале выражение выхода из цикла `while` должно возвращать логическое значение `true` или `false`, однако как `false` интерпретируется также значение `0`. Любое другое значение рассматривается как значение `true`. Поскольку `-1` — это не нуль, условие выхода из цикла `while` считается имеющим значение `true`, и цикл продолжает выполняться. Если вы хотите, чтобы цикл выполнялся только для положительных чисел, перепишите выражение как `while(Integer>0)`. Это правило справедливо для всех условных инструкций и циклов.

■ Эквивалентны ли пустой цикл `while` и цикл `for(;;)`?

Нет, цикл `while` всегда нуждается в наличии условия выхода.

■ Я изменил код `do...while(exp)`; на `while(exp)`; с помощью копирования и вставки. Не возникнет ли каких-нибудь проблем?

Да, возникнут — и большие! Код `while(exp);` — вполне допустимый, хотя и пустой цикл `while`, поскольку перед точкой с запятой нет никаких инструкций (даже если за ней следует блок инструкций). Блок кода первого цикла вопроса выполняется как минимум один раз. Будьте внимательны при копировании и вставке кода!

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Зачем беспокоиться об отступах кода в блоках инструкций, вложенных циклов, инструкций `if`, если код вполне нормально компилируется и без них?
2. Вы можете быстро реализовать переход, используя инструкцию `goto`. Почему же тогда следует избегать его применения?
3. Возможно ли написать цикл `for` с уменьшающимся значением счетчика? Как он должен выглядеть?
4. В чем проблема со следующим циклом?

```
for(int counter=0; counter==10; ++counter)
    cout << counter << " ";
```

Упражнения

1. Напишите цикл `for` для доступа к элементам массива в обратном порядке.
2. Напишите вложенный цикл, эквивалентный использованному в листинге 6.14, добавляющий элементы в два массива, но в обратном порядке.
3. Напишите программу, которая, подобно листингу 6.16, выводит числа Фибоначчи, но запрашивая пользователя, сколько чисел он хочет вычислить.
4. Напишите конструкцию `switch-case`, которая сообщает, есть ли в радуге такой цвет. Используйте константы перечисления.
5. **Отладка.** Что не так с этим кодом?

```
for(int counter=0; counter=10; ++counter)
    cout << counter << " ";
```

6. **Отладка.** Что не так с этим кодом?

```
int loopCounter = 0;
while(loopCounter <5);
{
    cout << loopCounter << " ";
    loopCounter++;
}
```

7. **Отладка.** Что не так с этим кодом?

```
cout << "Введите число от 0 до 4" << endl;
int input = 0;
cin >> input;
switch(input)
{
    case 0:
    case 1:
    case 2:
```

```
case 3:  
case 4:  
    cout << "Корректный ввод" << endl;  
default:  
    cout << "Некорректный ввод" << endl;  
}
```

ЗАНЯТИЕ 7

Организация кода с помощью функций

До сих пор в этой книге вы видели простые программы, все действия которых выполнялись в функции `main()`. В больших программах и приложениях это работает достаточно хорошо. Но чем больше и сложнее становится программа, тем длиннее и запутаннее становится содержимое функции `main()`, если только вы не решите структурировать свою программу с помощью функций.

Функции позволяют разделить и организовать логику выполнения программы. Они разделяют содержимое приложения на логические блоки, вызываемые при необходимости.

Таким образом, *функция* (function) — это подпрограмма, которая может получать параметры и возвращает значение. Чтобы выполнить свою задачу, функция должна быть вызвана.

На этом занятии...

- Потребность в функциях
- Прототип и определение функции
- Передача параметров в функции и возвращение значений из них
- Перегрузка функций
- Рекурсивные функции
- Лямбда-функции C++11

Потребность в функциях

Рассмотрим приложение, запрашивающее у пользователя радиус круга, а затем вычисляющее его площадь и периметр. Конечно, можно поместить весь код в функцию `main()`, но лучше разделить это приложение на логические блоки, в данном случае на блок, вычисляющий площадь по данному радиусу, и блок, вычисляющий периметр (листинг 7.1).

ЛИСТИНГ 7.1. Функции, вычисляющие площадь и периметр круга заданного радиуса

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.14159;
4:
5: // Объявления функций (прототипы)
6: double Area(double radius);
7: double Circumference(double radius);
8:
9: int main()
10: {
11:     cout << "Введите радиус: ";
12:     double radius = 0;
13:     cin >> radius;
14:
15:     // Вызов функции "Area"
16:     cout << "Площадь равна: " << Area(radius) << endl;
17:
18:     // Вызов функции "Circumference"
19:     cout << "Периметр равен: " << Circumference(radius) << endl;
20:
21:     return 0;
22: }
23:
24: // Определения функций (реализации)
25: double Area(double radius)
26: {
27:     return Pi * radius * radius;
28: }
29:
30: double Circumference(double radius)
31: {
32:     return 2 * Pi * radius;
33: }
```

Результат

Введите радиус: 6.5
Площадь равна: 132.732
Периметр равен: 40.8407

Анализ

Функция `main()`, которая тоже является функцией, достаточно компактна и делегирует выполнение расчетов функциям `Area()` и `Circumference()`, вызов которых осуществляется в строках 16 и 19 соответственно.

Программа демонстрирует следующие элементы, используемые при применении функций.

- Прототипы функции *объявляются* (declare) в строках 6 и 7, так что компилятор знает о том, что означают термины `Area` и `Circumference`, когда они будут использованы в функции `main()`.
- Функции `Area()` и `Circumference()` *вызываются* (invoke) в функции `main()` в строках 16 и 19.
- Функция `Area()` *определяется* (define) в строках 25–28, а функция `Circumference()` — в строках 30–33.

Вынесение вычисления площади и периметра в различные функции может потенциально повысить повторную используемость этих функций при необходимости вычислять то или иное значение.

Что такое прототип функции

Рассмотрим строки 6 и 7 в листинге 7.1:

```
double Area(double radius);
double Circumference(double radius);
```

Состав прототипа функции представлен на рис. 7.1.

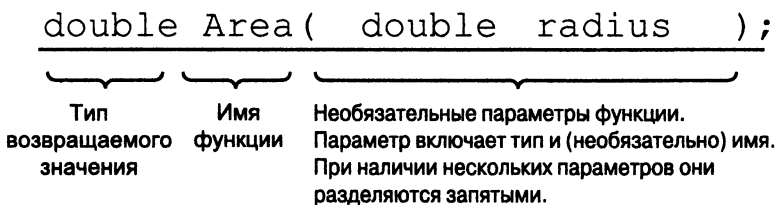


РИС. 7.1. Части прототипа функции

Прототип функции в основном указывает имя функции (в данном случае — `Area`), список получаемых ею параметров (в данном случае это один параметр типа `double` по имени `radius`) и тип возвращаемого функцией значения (в данном случае — `double`).

Без прототипа функции по достижении строк 16 и 19 в функции `main()` компилятор не будет знать, что означают слова `Area` и `Circumference`. Прототипы функции указывают компилятору, что `Area` и `Circumference` — это функции, которые получают один параметр типа `double` и возвращают значение типа `double`. Теперь компилятор распознает эти выражения, считает их допустимыми и обеспечивает связывание вызовов функций с их реализациями, гарантируя, что при выполнении программы будут вызваны именно они.

ПРИМЕЧАНИЕ

Функция может иметь список из нескольких параметров, разделенных запятыми, но только один тип возвращаемого значения.

При создании функции, которая не должна возвращать никаких значений, тип возвращаемого ею значения указывается как `void` (пустота).

Что такое определение функции

Главное в функции заключено в ее *реализации* (implementation), называемой также *определением* (definition). Проанализируем определение функции `Area`:

```
25: double Area(double radius)
26: {
27:     return Pi * radius * radius;
28: }
```

Определение функции всегда состоит из блока инструкций. Если функция не объявлена с типом возвращаемого значения `void`, в теле функции необходимо наличие оператора `return`. В нашем случае функция `Area` нуждается в операторе `return` для возврата значения типа `double`. *Блок инструкций* (statement block) состоит из набора инструкций в фигурных скобках (`{ ... }`), которые выполняются при вызове функции. Функция `Area()` использует входной параметр `radius`, содержащий радиус, в качестве *аргумента*, передаваемый вызывающей стороной для вычисления площади круга.

Что такое вызов функции и аргументы

Для получения результата функции она *вызывается*. Когда функция объявлена так, как в нашем случае — с *параметрами* (parameter), при вызове ей нужно передать *аргументы* (argument), которые представляют собой значения соответствующих параметров. Проанализируем вызов функции `Area()` в листинге 7.1:

```
16:     cout << "Площадь равна: " << Area(radius) << endl;
```

Здесь `Area(radius)` — это вызов функции; `radius` — аргумент, переданный в функцию `Area()`. При вызове функции поток выполнения переходит в функцию `Area()`, которая использует переданное значение радиуса для вычисления площади круга. Завершив работу, функция возвращает значение типа `double`, которое затем отображается на экране с помощью потока `cout`.

Создание функций с несколькими параметрами

Предположим, вы пишете программу, которая вычисляет площадь поверхности цилиндра, показанного на рис. 7.2.

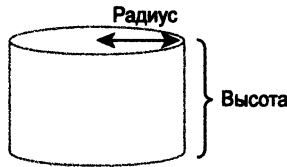


РИС. 7.2. Цилиндр

Вот как выглядит формула для площади поверхности цилиндра:

Площадь цилиндра = Площадь верхнего круга + Площадь нижнего круга +
Площадь боковой поверхности
$$= \text{Pi} \cdot \text{radius}^2 + \text{Pi} \cdot \text{radius}^2 + 2 \cdot \text{Pi} \cdot \text{radius} \cdot \text{height}$$
$$= 2 \cdot \text{Pi} \cdot \text{radius}^2 + 2 \cdot \text{Pi} \cdot \text{radius} \cdot \text{height}$$
$$= 2 \cdot \text{Pi} \cdot \text{radius} \cdot (\text{radius} + \text{height})$$

Таким образом, при вычислении площади поверхности цилиндра необходимо знать две величины — радиус и высоту цилиндра. Поэтому при объявлении такой функции в списке ее параметров указываются по крайней мере два параметра. В списке параметров их разделяют запятой, как показано в листинге 7.2.

ЛИСТИНГ 7.2. Функция, получающая два параметра и вычисляющая площадь поверхности цилиндра

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.141593;
4:
5: // Объявление содержит два параметра
6: double SurfaceArea(double radius, double height);
7:
8: int main()
9: {
10:     cout << "Радиус цилиндра: ";
11:     double radius = 0;
12:     cin >> radius;
13:     cout << "Высота цилиндра: ";
14:     double height = 0;
15:     cin >> height;
16:
17:     cout << "Площадь поверхности: "
18:         << SurfaceArea(radius, height) << endl;
19:     return 0;
20: }
```



```
21:
22: double SurfaceArea(double radius, double height)
23: {
24:     double area = 2*Pi*radius*(radius + height);
25:     return area;
26: }
```

Результат

Радиус цилиндра: 3
Высота цилиндра: 6.5
Площадь поверхности: 179.071

Анализ

Строка 6 содержит объявление функции `SurfaceArea()` с двумя параметрами: `radius` и `height`. Оба параметра имеют тип `double` и в списке параметров разделены запятой. Строки 22–26 содержат определение, т.е. реализацию функции `SurfaceArea()`. Как можно заметить, входные параметры `radius` и `height` используются при вычислении значения переменной `area`, которое затем и возвращается вызывающей функцией.

ПРИМЕЧАНИЕ

Параметры функций похожи на локальные переменные. Они видимы и работают только в пределах функции. Так, параметры `radius` и `height` функции `SurfaceArea()` в листинге 7.2 в пределах самой функции `SurfaceArea()` представляют собой копии переменных с теми же именами в функции `main()`.

Создание функций без параметров и возвращаемых значений

Если вы делегируете задачу вывода фразы "Hello World" функции, которая выполняет только вывод этой строки и ничего больше не делает, то ей не нужны никакие параметры; она не должна также возвращать никакие значения (вы не ожидаете от такой функции никакой полезной информации). Одна из таких функций представлена в листинге 7.3.

ЛИСТИНГ 7.3. Функция без параметров и возвращаемых значений

```
0: #include <iostream>
1: using namespace std;
2:
3: void SayHello();
4:
5: int main()
```

```
6: {  
7:     SayHello();  
8:     return 0;  
9: }  
10:  
11: void SayHello()  
12: {  
13:     cout << "Hello World" << endl;  
14: }
```

Результат

Hello World

Анализ

Прототип функции в строке 3 объявляет функцию SayHello с возвращаемым значением типа void, т.е. не возвращающим никакого значения. Поэтому в определении функции (строки 11–14) нет никакого оператора возврата. Вызов этой функции в строке 7 функции main() не присваивает никакой переменной возвращаемого значения и не использует его ни в каком выражении, поскольку данная функция ничего не возвращает. Некоторые программисты предпочитают вставлять пустой оператор return в конце таких функций:

```
void SayHello()  
{  
    cout << "Hello World" << endl;  
    return; // Пустой возврат  
}
```

Параметры функций со значениями по умолчанию

До сих пор мы использовали в примерах фиксированное значение числа π как константу, не предоставляя пользователю возможности его изменить. Однако пользователь функции может интересоваться более точное или менее точное ее значение¹. Как при создании функции, использующей число π , позволить ее пользователю использовать собственное значение, а при его отсутствии задействовать стандартное?

Один из способов решения этой проблемы подразумевает создание в функции Area() дополнительного параметра для числа π и присваивание ему значения по умолчанию (default value). Такая адаптация функции Area() из листинга 7.1 выглядела бы следующим образом:

```
double Area(double radius, double Pi = 3.14);
```

¹ Естественно, пример крайне надуманный; отнеситесь к нему просто как к абстрактному учебному примеру. Или представьте себе параллельную вселенную, в которой число π имеет иное значение. — *Примеч. ред.*

Обратите внимание на второй параметр π и присвоенное ему по умолчанию значение 3,14. Этот второй параметр является теперь *необязательным параметром* (optional parameter) для вызывающей функции. Вызывающая функция все равно может вызвать функцию `Area()`, используя синтаксис

```
Area(radius);
```

В данном случае второй параметр был проигнорирован, поэтому используется его значение по умолчанию 3,14. Но если пользователь захочет задействовать другое значение числа π , то можно сделать это, вызвав функцию `Area()` следующим образом:

```
Area(radius, Pi); // Pi определяется пользователем
```

Листинг 7.4 демонстрирует возможность создания функции, параметры которой имеют значения по умолчанию, но при необходимости могут быть переопределены пользовательским значением.

ЛИСТИНГ 7.4. Функция, вычисляющая площадь круга
и использующая число π как второй параметр со значением по умолчанию 3,14

```
0: #include <iostream>
1: using namespace std;
2:
3: // Объявление функции (прототип)
4: double Area(double radius,      // Pi со значением
5:              double Pi = 3.14); // по умолчанию
6: int main()
7: {
8:     cout << "Введите радиус: ";
9:     double radius = 0;
10:    cin >> radius;
11:
12:    cout << "Pi равно 3.14. Изменить (y/n)? ";
13:    char changePi = 'n';
14:    cin >> changePi;
15:
16:    double circleArea = 0;
17:    if (changePi == 'y')
18:    {
19:        cout << "Новое значение Pi: ";
20:        double newPi = 3.14;
21:        cin >> newPi;
22:        circleArea = Area (radius, newPi);
23:    }
24:    else
25:        circleArea = Area(radius); // 2-й параметр принимает
26:                                   // значение по умолчанию
27:    // Вызов функции Area
28:    cout << "Площадь равна: " << circleArea << endl;
29:
```

```
30:     return 0;
31: }
32:
33: // В определении функции значение по умолчанию не указывается
34: double Area(double radius, double Pi)
35: {
36:     return Pi * radius * radius;
37: }
```

Результат

Введите радиус: 1
Pi равно 3.14. Изменить (y/n)? n
Площадь равна: 3.14

Следующий запуск:

Введите радиус: 1
Pi равно 3.14. Изменить (y/n)? y
Новое значение Pi: **3.1416**
Площадь равна: 3.1416

Анализ

При обоих запусках в приведенном выше выводе пользователь вводил одинаковый радиус, равный 1. Но при втором запуске пользователь решил изменить точность числа π , а потому вычисленная площадь стала немного иной. Обратите внимание, что в обоих случаях в строках 22 и 25 выполняется вызов одной и той же функции. В строке 25 в вызове нет второго параметра, Pi , поэтому в данном случае используется значение по умолчанию 3,14.

ПРИМЕЧАНИЕ

У функции может быть несколько параметров со значениями по умолчанию; но все они должны быть расположены в заключительной части списка параметров.

Рекурсия — функция, вызывающая сама себя

В некоторых случаях функция может фактически вызывать сама себя. Такая функция называется *рекурсивной* (recursive function). Обратите внимание, что у рекурсивной функции должно быть четко определенное условие выхода, когда она завершает работу и больше себя не вызывает.

ВНИМАНИЕ!

При отсутствии условия выхода или при ошибке в нем выполнение программы застрянет в рекурсивном вызове функции, которая непрерывно будет вызывать сама себя, пока в конечном счете не приведет к переполнению стека и аварийному завершению приложения.

Рекурсивные функции могут пригодиться при вычислении чисел ряда Фибоначчи, представленном в листинге 7.5. Этот ряд начинается с двух чисел, 0 и 1:

$$F(0) = 0$$

$$F(1) = 1$$

Значение каждого следующего числа последовательности — это сумма двух предыдущих чисел. Таким образом, n -е значение последовательности (при $n > 1$) определяется следующей (рекурсивной) формулой:

$$\text{Fibonacci}(n) = \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$$

В результате ряд Фибоначчи расширяется до

$$F(2) = 1$$

$$F(3) = 2$$

$$F(4) = 3$$

$$F(5) = 5$$

$$F(6) = 8 \quad \text{и так далее.}$$

ЛИСТИНГ 7.5. Использование рекурсивной функции для вычисления членов ряда Фибоначчи

```
0: #include <iostream>
1: using namespace std;
2:
3: int GetFibNumber(int fibIndex)
4: {
5:     if (fibIndex < 2)
6:         return fibIndex;
7:     else // рекурсия, если fibIndex >= 2
8:         return GetFibNumber(fibIndex-1)+GetFibNumber(fibIndex-2);
9: }
10:
11: int main()
12: {
13:     cout << "Введите индекс числа Фибоначчи, начиная с 0: ";
14:     int index = 0;
15:     cin >> index;
16:
17:     cout << "Число Фибоначчи: " << GetFibNumber(index) << endl;
18:     return 0;
19: }
```

Результат

Введите индекс числа Фибоначчи, начиная с 0: 6

Число Фибоначчи: 8

Анализ

Функция `GetFibNumber()`, определенная в строках 3–9, рекурсивна, поскольку она вызывает сама себя в строке 8. Обратите внимание на условие выхода в строках 5 и 6: когда индекс становится меньше двух, функция перестает быть рекурсивной. С учетом того, что функция вызывает сама себя, последовательно уменьшая значение индекса `fibIndex`, в определенный момент это значение достигает уровня, когда срабатывает условие выхода и рекурсия останавливается.

Функции с несколькими операторами `return`

Вы не ограничены наличием только одного оператора `return` в определении функции. По желанию вы можете осуществлять выход из функции в любом месте, не обязательно только в одном, как показано в листинге 7.6. В зависимости от логики и задачи приложения это может быть и преимуществом, и недостатком.

ЛИСТИНГ 7.6. Использование нескольких операторов `return` в одной функции

```

0: #include <iostream>
1: using namespace std;
2: const double Pi = 3.14159265;
3:
4: void QueryAndCalculate()
5: {
6:     cout << "Введите радиус: ";
7:     double radius = 0;
8:     cin >> radius;
9:
10:    cout << "Площадь: " << Pi * radius * radius << endl;
11:
12:    cout << "Вычислять периметр (y/n)? ";
13:    char calcCircum = 'n';
14:    cin >> calcCircum;
15:
16:    if (calcCircum == 'n')
17:        return;
18:
19:    cout << "Периметр: " << 2 * Pi * radius << endl;
20:    return;
21: }
22:
23: int main()
24: {
25:     QueryAndCalculate();
26:
27:     return 0;
28: }

```

Результат

```
Введите радиус: 1
Площадь: 3.14159
Вычислять периметр (y/n)? y
Периметр: 6.28319
```

Следующий запуск:

```
Введите радиус: 1
Площадь: 3.14159
Вычислять периметр (y/n)? n
```

Анализ

Функция `QueryAndCalculate()` содержит несколько операторов `return`: один — в строке 17, второй — в строке 20. Эта функция спрашивает у пользователя, не хочет ли он, кроме площади, вычислить периметр. Если пользователь вводит символ 'n', отказываясь от вычисления, происходит выход из программы с использованием первого оператора `return`. В противном случае происходит вычисление периметра окружности, а затем выход с использованием следующего оператора `return`.

ВНИМАНИЕ!

Используйте несколько выходов из функции осторожно. Значительно проще исследовать и понять функцию, которая начинается сверху и заканчивается внизу, чем функцию, которая имеет несколько выходов в разных местах.

Чтобы избежать использования нескольких выходов в листинге 7.6, достаточно изменить условие инструкции `if` на проверку значения 'y':

```
if (calcCircum == 'y')
    cout << "Периметр: " << 2 * Pi * radius << endl;
```

Использование функций для работы с данными различных видов

Функции не ограничивают вас передачей значений по одному; вы можете передавать в функции массивы значений. Вы даже можете создать две функции с одинаковым именем и одинаковым типом возвращаемого значения, но с различными наборами параметров. Можно создать функцию, параметры которой не создаются и не уничтожаются в вызове функции; вместо них используются ссылки, которые остаются корректными и после завершения работы функции, так что такие функции могут работать с большими объемами данных или параметрами больших размеров. В этом разделе вы узнаете о передаче функциям массивов, о перегрузке функций и передаче аргументов в функции по ссылке.

Перегрузка функций

Функции с одинаковым именем и одинаковым типом возвращаемого значения, но с разными наборами параметров называют *перегруженными функциями* (overloaded function). Перегруженные функции могут быть весьма полезными, например, в приложениях, в которых имеется функция с определенным именем, которая осуществляет некоторый вывод, но может быть вызвана с различными наборами параметров. Предположим, необходимо написать приложение, которое вычисляет площадь круга и площадь поверхности цилиндра. Функция, которая вычисляет площадь круга, нуждается в одном параметре — радиусе. Вторая функция, которая вычисляет площадь поверхности цилиндра, нуждается, кроме радиуса, во втором параметре — высоте цилиндра. Обе эти функции должны вернуть данные одного типа, содержащие площадь. Язык C++ позволяет определить две перегруженные функции, обе с именем Area и возвращающие значение типа double, однако одна из них получает только радиус, а другая — радиус и высоту, как показано в листинге 7.7.

ЛИСТИНГ 7.7. Использование перегруженной функции

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.14159265;
4:
5: double Area(double radius); // для круга
6: double Area(double radius, double height); // для цилиндра
7:
8: int main()
9: {
10:     cout << "Введите z для цилиндра, c для круга: ";
11:     char userSelection = 'z';
12:     cin >> userSelection;
13:
14:     cout << "Введите радиус: ";
15:     double radius = 0;
16:     cin >> radius;
17:
18:     if (userSelection == 'z')
19:     {
20:         cout << "Введите высоту: ";
21:         double height = 0;
22:         cin >> height;
23:
24:         // Вызов перегруженной версии Area для цилиндра
25:         cout << "Поверхность цилиндра: " << Area(radius,height);
26:     }
27:     else
28:         cout << "Площадь круга: " << Area(radius);
29:     cout << endl;
```



```
30:     return 0;
31: }
32:
33: // Для круга
34: double Area(double radius)
35: {
36:     return Pi * radius * radius;
37: }
38:
39: // Для цилиндра
40: double Area(double radius, double height)
41: {
42:     // Повторное использование версии для площади круга
43:     return 2 * Area(radius) + 2 * Pi * radius * height;
44: }
```

Результат

Введите z для цилиндра, с для круга: **z**

Введите радиус: **2**

Введите высоту: **5**

Поверхность цилиндра: 87.9646

Следующий запуск:

Введите z для цилиндра, с для круга: **с**

Введите радиус: **1**

Площадь круга: 3.14159

Анализ

В строках 5 и 6 объявлены прототипы перегруженных версий функции `Area()`: первая принимает один параметр (радиус круга), а вторая — два параметра (радиус и высоту цилиндра). У обеих функций одинаковые имена (`Area`) и типы возвращаемого значения (`double`), но разные наборы параметров. Следовательно, это перегруженные функции. Определения перегруженных функций находятся в строках 34–44, где реализованы две функции вычисления площади: площади круга по его радиусу и площади поверхности цилиндра по его радиусу и высоте соответственно. Интересно, что, поскольку площадь цилиндра состоит из площади двух кругов (один сверху, второй снизу) и площади боковой стороны, перегруженная версия для цилиндра может повторно использовать функцию `Area()` для круга, как показано в строке 43.

Передача в функцию массива значений

Функция, которая выводит на консоль целое число, может быть представлена следующим образом:

```
void DisplayInteger(int number);
```

Прототип функции, способной отобразить массив целых чисел, должен быть немного другим:

```
void DisplayIntegers(int[] numbers, int length);
```

Первый параметр указывает, что передаваемые в функцию данные являются массивом, а второй параметр указывает его длину, чтобы, используя массив, вы не вышли за его границы (листинг 7.8).

ЛИСТИНГ 7.8. Функция, получающая массив как параметр

```
0: #include <iostream>
1: using namespace std;
2:
3: void DisplayArray(int numbers[], int length)
4: {
5:     for(int index = 0; index < length; ++index)
6:         cout << numbers[index] << " ";
7:
8:     cout << endl;
9: }
10:
11: void DisplayArray(char characters[], int length)
12: {
13:     for(int index = 0; index < length; ++index)
14:         cout << characters[index] << " ";
15:
16:     cout << endl;
17: }
18:
19: int main()
20: {
21:     int myNums[4] = {24, 58, -1, 245};
22:     DisplayArray(myNums, 4);
23:
24:     char myStatement[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
25:     DisplayArray(myStatement, 7);
26:
27:     return 0;
28: }
```

Результат

```
24 58 -1 245
H e l l o !
```

Анализ

В листинге имеются две перегруженные версии функции `DisplayArray()`: одна отображает содержимое элементов целочисленного массива, а вторая — символьного.

В строках 22 и 25 в вызовах функций в качестве аргументов им передаются массив целых чисел и массив символов соответственно. Обратите внимание, что при объявлении и инициализации массива символов (строка 24) был преднамеренно включен нулевой символ. Это хорошая привычка, даже при том, что в данной ситуации массив не используется в качестве строки, выводимой в `cout`.

Передача аргументов по ссылке

Давайте вернемся к функции вычисления площади круга по радиусу в листинге 7.1:

```
24: // Определения функции (реализация)
25: double Area(double radius)
26: {
27:     return Pi * radius * radius;
28: }
```

Здесь параметр `radius` содержит значение, которое копируется в него при вызове функции `main()`:

```
15: // Вызов функции "Area"
16: cout << "Площадь равна: " << Area(radius) << endl;
```

Это означает, что вызов функции `Area()` никак не воздействует на переменную `radius` в функции `main()`, поскольку эта функция работает с *копией* значения переменной `radius`, содержащейся в параметре `radius`. Однако иногда нужны функции, способные работать с исходной переменной или изменять значение так, чтобы это изменение было доступно вне функции, скажем, в вызывающей функции. В таком случае следует объявить параметр как получающий аргумент *по ссылке* (by reference). Версия функции `Area()`, вычисляющая и возвращающая площадь с помощью передаваемого по ссылке параметра, выглядит следующим образом:

```
// Выходной параметр result по ссылке
void Area(double radius, double& result)
{
    result = Pi * radius * radius;
}
```

Обратите внимание: в этой версии функция `Area()` имеет два параметра. Не пропустите амперсанд (&) рядом со вторым параметром `result`. Именно этот знак указывает компилятору, что второй аргумент должен быть не скопирован в функцию, а передан как ссылка на переменную. Тип возвращаемого значения был изменен на `void`, поскольку теперь функция возвращает вычисленную площадь не как возвращаемое значение, а с помощью выходного параметра, передаваемого по ссылке. Возврат значения через параметр, передаваемый по ссылке, продемонстрирован в листинге 7.9, в котором вычисляется площадь круга.

ЛИСТИНГ 7.9. Возврат результата вычислений с помощью ссылки, а не в качестве возвращаемого значения

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.1416;
4:
5: // Выходной параметр result, передаваемый по ссылке
6: void Area(double radius, double& result)
7: {
8:     result = Pi * radius * radius;
9: }
10:
11: int main()
12: {
13:     cout << "Введите радиус: ";
14:     double radius = 0;
15:     cin >> radius;
16:
17:     double areaFetched = 0;
18:     Area(radius, areaFetched);
19:
20:     cout << "Площадь равна: " << areaFetched << endl;
21:     return 0;
22: }
```

Результат

```
Введите радиус: 2
Площадь равна: 12.5664
```

Анализ

Обратите внимание на строки 17 и 18, в которых функция `Area()` вызывается с двумя параметрами; второй параметр — переменная, в которую должен быть записан результат. Поскольку функция `Area()` получает второй параметр по ссылке, переменная `result`, используемая в строке 8 в функции `Area()`, указывает на ту же область памяти, что и переменная `double areaFetched`, объявленная в строке 17 вызывающей функции `main()`. Таким образом, вычисленный в функции `Area()` результат (строка 8) оказывается размещенным в переменной `areaFetched` функции `main()` и отображается на экране в строке 20.

ПРИМЕЧАНИЕ

Используя оператор `return`, функция может вернуть только одно значение. Но если функция должна возвращать вызывающей функции несколько значений, то передача аргументов по ссылке является единственным способом, обеспечивающим такой возврат информации вызывающей функции.

Как процессор обрабатывает вызовы функций

Хотя знать во всех подробностях, как вызов функции реализуется на уровне процессора, не так уж необходимо, понятие об этом все же стоит иметь. Это поможет понять, почему язык C++ позволяет создавать встраиваемые функции, которые будут рассматриваться в этом разделе позже.

Вызов функции, по существу, означает, что процессор переходит к выполнению следующей команды, принадлежащей вызываемой функции и расположенной в некоторой не последовательной области памяти. После выполнения команд функции поток выполнения возвращается туда, откуда был совершен переход в функцию. Для реализации этой логики компилятор преобразует вызов функции в команду процессора CALL. Данная команда включает адрес следующей команды для выполнения (это адрес команды вызываемой функции). Когда процессор встречает команду CALL, он сохраняет в стеке позицию команды, которая будет выполнена после возврата из функции, и переходит к командам в области памяти, указанной в команде CALL.

Понятие стека

Стек (stack) — это структура памяти, действующая по принципу *последним вошел, первым вышел* (Last-In-First-Out — LIFO), весьма похожая на стопку тарелок, из которой первой вы берете ту тарелку сверху, которую положили последней. Помещение данных в стек называется операцией *заталкивания* (push), а извлечение данных из стека называется операцией *вытягивания* (pop). По мере роста стека вверх происходит приращение указателя вершины стека, так что он всегда указывает на вершину стека (рис. 7.3).

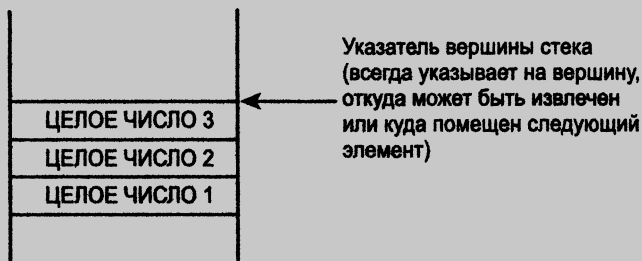


РИС. 7.3. Визуальное представление стека, содержащего три целых числа

Природа стека делает его оптимальным для обработки вызовов функций. При вызове функции все ее локальные переменные создаются в стеке. Когда функция заканчивает работу, они просто снимаются со стека, и указатель вершины стека возвращается к своему первоначальному положению.

Эта область памяти содержит команды, принадлежащие функции. Процессор выполняет их до тех пор, пока не встретит команду RET (машинный код для оператора

return в программе C++). Команда RET требует от процессора извлечь из стека адрес, сохраненный во время выполнения команды CALL, и использовать его в качестве адреса команды в вызывающей функции, которой должно продолжиться выполнение программы. Таким образом, процессор возвращает выполнение вызывающей функции, и оно продолжается с того места, где было прервано вызовом функции.

Встраиваемые функции

Вызов обычной функции преобразуется в команду CALL, которая приводит к выполнению операций со стеком, переходу процессора к выполнению кода функции и т.д. Эта дополнительная работа, выполняемая невидимо для пользователя, в большинстве случаев невелика. Но что если функция очень проста, как эта?

```
double GetPi()  
{  
    return 3.14159;  
}
```

Накладные расходы времени на выполнение фактического вызова функции в этом случае весьма высоки по сравнению со временем, затраченным на выполнение кода функции GetPi(). Вот почему компиляторы C++ позволяют программисту объявлять такие функции как *встраиваемые* (inline). Ключевое слово inline — это просьба построить реализацию функции вместо ее вызова в место ее вызова.

```
inline double GetPi()  
{  
    return 3.14159;  
}
```

Хорошим кандидатом на встраивание являются очень простая функция, например, удваивающая число или выполняющая похожие простые операции. Один из таких случаев приведен в листинге 7.10.

ЛИСТИНГ 7.10. Использование встраиваемой функции, удваивающей целое число

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: // Определение встраиваемой функции удвоения  
4: inline long DoubleNum(int inputNum)  
5: {  
6:     return inputNum * 2;  
7: }  
8:  
9: int main()  
10: {  
11:     cout << "Введите целое число: ";  
12:     int inputNum = 0;  
13:     cin >> inputNum;  
14:
```

```
15:    // Вызов встраиваемой функции
16:    cout << "После удвоения: " << DoubleNum(inputNum) << endl;
17:
18:    return 0;
19: }
```

Результат

Введите целое число: **35**

После удвоения: 70

Анализ

Рассматриваемое ключевое слово `inline` используется в строке 4. Компиляторы рассматривают это ключевое слово как просьбу разместить код функции `DoubleNum()` непосредственно в месте ее вызова (строка 16), что увеличивает скорость выполнения кода.

В то же время указание функций как встраиваемых способно увеличить размер кода, особенно если встраиваемая функция содержит сложную обработку или имеет большой размер. Использовать ключевое слово `inline` следует по минимуму и только для тех функций, которые выполняют простые действия, сравнимые по объему с дополнительными затратами на вызов обычной функции, как упоминалось ранее.

ПРИМЕЧАНИЕ

Большинство современных компиляторов C++ оснащены высококачественными оптимизаторами кода. Многие из них позволяют оптимизировать программу по размеру, создавая приложение минимального размера, или по скорости, обеспечивая максимальную его производительность. Первое весьма важно при разработке программного обеспечения для различных устройств наподобие мобильных, в которых не так уж много памяти. При такой оптимизации компилятор отклоняет большинство просьб о встраивании, поскольку это может увеличить размер кода.

При оптимизации по скорости компилятор обычно удовлетворяет просьбы о встраивании (там, где это имеет смысл), причем делает это зачастую даже в тех случаях, когда никто его об этом не просит.

Автоматический вывод возвращаемого типа

Вы уже знаете о ключевом слове `auto` из занятия 3, “Использование переменных и констант”. Оно позволяет оставить вывод типа переменной компилятору, который делает это на основе инициализирующего значения, присваиваемого переменной. Начиная со стандарта C++14 то же самое возможно и для функций. Вместо указания типа возвращаемого значения можно использовать ключевое слово `auto` и позволить компилятору самому вывести тип возвращаемого значения на основе вашего кода.

В листинге 7.11 показано применение ключевого слова `auto` в функции, вычисляющей площадь круга.

ЛИСТИНГ 7.11. Использование `auto` в качестве возвращаемого типа функции `Area()`

```
0: #include <iostream>
1: using namespace std;
2:
3: const double Pi = 3.14159265;
4:
5: auto Area(double radius)
6: {
7:     return Pi * radius * radius;
8: }
9:
10: int main()
11: {
12:     cout << "Введите радиус: ";
13:     double radius = 0;
14:     cin >> radius;
15:
16:     // Вызов функции "Area"
17:     cout << "Площадь равна: " << Area(radius) << endl;
18:
19:     return 0;
20: }
```

Результат

```
Введите радиус: 2
Площадь равна: 12.5664
```

Анализ

Нас интересует строка 5 исходного текста, в которой ключевое слово `auto` используется в качестве типа возвращаемого значения функции `Area()`. Компилятор выводит возвращаемый тип на основе выражения `return`, которое возвращает из функции значение типа `double`. Таким образом, несмотря на использование ключевого слова `auto` функция `Area()` в листинге 7.11 компилируется так же, как и в листинге 7.1 с явным указанием возвращаемого типа `double`.

ПРИМЕЧАНИЕ

Функции с использованием автоматического вывода типа возвращаемого значения должны быть определены (т.е. реализованы) до того, как вы будете к ним обращаться. Это связано с тем, что компилятор должен знать тип возвращаемого значения функции в точке, где она используется. Если такая функция содержит несколько операторов `return`, все они должны возвращать один и тот же тип. Рекурсивные вызовы должны предваряться по крайней мере одним оператором `return` в теле функции.

Лямбда-функции

Этот раздел — не более чем беглое введение в сложную для новичков концепцию. Попробуйте изучить ее, но не расстраивайтесь, если это у вас не получится. Более подробная информация о лямбда-функциях рассматривается на занятии 22, “Лямбда-выражения языка C++11”.

Лямбда-функции введены стандартом C++11 и очень помогают использовать алгоритмы STL для сортировки и обработки данных. Как правило, функция сортировки требует бинарного предиката, который представляет собой функцию, сравнивающую два аргумента и возвращающую `true`, если первый меньше второго, и `false` в противном случае (тем самым определяя порядок, в котором должны находиться отсортированные элементы). Такие предикаты обычно реализуются в виде операторов класса, что требует весьма кропотливого программирования. Лямбда-функции позволяют сократить определения предикатов, как показано в листинге 7.12.

ЛИСТИНГ 7.12. Использование лямбда-функции
для сортировки и отображения элементов массива

```
0: #include <iostream>
1: #include <algorithm>
2: #include <vector>
3: using namespace std;
4:
5: void DisplayNums(vector<int>& dynArray)
6: {
7:     for_each(dynArray.begin(), dynArray.end(),
8:             [](int e) {cout << e << " ";} );
9:
10:    cout << endl;
11: }
12:
13: int main()
14: {
15:     vector<int> myNums;
16:     myNums.push_back(501);
17:     myNums.push_back(-1);
18:     myNums.push_back(25);
19:     myNums.push_back(-35);
20:
21:     DisplayNums(myNums);
22:
23:     cout << "Сортировка в порядке убывания" << endl;
24:
25:     sort(myNums.begin(), myNums.end(),
26:         [](int num1, int num2) {return (num2 < num1);} );
27:
28:     DisplayNums(myNums);
29:
30:     return 0;
31: }
```

Результат

```
501 -1 25 -35
Сортировка в порядке убывания
501 25 -1 -35
```

Анализ

Программа помещает целые числа в динамический массив, предоставляемый стандартной библиотекой C++ в виде вектора `std::vector` (строки 15–19). Функция `DisplayNums()` использует алгоритм STL `for_each` для вывода значений всех элементов массива. При этом в строке 8 используется лямбда-функция. При сортировке в строке 25 с помощью алгоритма `std::sort` используется бинарный предикат (строка 26), который также имеет вид лямбда-функции, возвращающей значение `true`, если второе число меньше первого, тем самым обеспечивая сортировку коллекции в порядке убывания.

Синтаксис лямбда-функций следующий:

```
[необязательные параметры] (список параметров) { инструкции; }
```

ПРИМЕЧАНИЕ

Предикаты и их использование в алгоритмах, таких как сортировка, подробно рассматриваются на занятии 23, “Алгоритмы библиотеки STL”. В частности, в листинге 23.6 приводится код, который использует как алгоритм с лямбда-функцией, так и вариант без применения лямбда-функции, позволяя сравнить эффективность программирования с применением последних стандартов C++ и без них.

Резюме

На этом занятии вы изучили основы модульного программирования, включая то, как функции могут помочь в структурировании кода, а также многократно использовать созданные вами алгоритмы. Вы узнали, что функции могут получать параметры и возвращать значения, что у параметров могут быть значения по умолчанию, которые вызывающая функция может переопределить, а также что параметры в функции могут передаваться как по значению, так и по ссылке. Вы узнали, как передать в функцию массив и как создавать перегруженные функции с одинаковым именем и типом возвращаемого значения, но с разными списками параметров.

И наконец вы получили некоторое представление о лямбда-функциях. У этой новинки C++11 имеется потенциал, позволяющий совершенно изменить способ программирования приложений C++, особенно при активном использовании библиотеки STL.

Вопросы и ответы

■ Что будет, если я создам рекурсивную функцию без условия выхода?

Выполнение программы никогда не закончится. По существу, в этом нет ничего плохого, поскольку циклы `while(true)` и `for(;;)` делают то же самое; однако рекурсивный вызов функции потребляет все больше стековой памяти, что в конечном счете приводит к аварийному завершению работы приложения в связи с переполнением стека.

■ Почему бы не встраивать каждую функцию? Ведь это увеличит скорость выполнения, не так ли?

Все зависит от обстоятельств. Результатом встраивания всех функций будет многократное повторение их содержимого во множестве мест вызова, что приведет к увеличению объема кода. Поэтому наиболее современные компиляторы сами судят о том, какие вызовы могут быть встроены, в зависимости от настроек производительности.

■ Могу ли я задать значения по умолчанию для всех параметров в функции?

Да, это вполне возможно и рекомендовано, когда в этом есть смысл.

■ У меня есть две функции, обе по имени `Area`. Одна получает радиус, а другая высоту. Я хочу, чтобы одна возвращала тип `float`, а другая тип `double`. Это возможно?

Для перегрузки обе функции нуждаются в одинаковом имени и одинаковом типе возвращаемого значения. В данном случае компилятор сообщит об ошибке, поскольку две функции с разными типами возвращаемых значений не могут иметь одинаковое имя.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Какова область видимости переменных, объявленных в прототипе функции?
2. Какова природа значения, переданного этой функции?

```
int Func(int &someNumber);
```

3. У меня есть функция, которая вызывает сама себя. Как она называется?
4. Я объявил две функции с одинаковым именем и типом возвращаемого значения, но разными списками параметров. Как они называются?
5. Указатель вершины стека указывает на вершину, середину или дно стека?

Упражнения

1. Напишите перегруженные функции, которые вычисляют объемы сферы и цилиндра. Формулы таковы:

Объем сферы = $(4 * \text{Pi} * \text{Радиус} * \text{Радиус} * \text{Радиус}) / 3$
Объем цилиндра = $\text{Pi} * \text{Радиус} * \text{Радиус} * \text{Высота}$

2. Напишите функцию, которая получает массив типа double.
3. **Отладка.** Что не так с этим кодом?

```
#include <iostream>
using namespace std;

const double Pi = 3.1416;

void Area(double radius, double result)
{
    result = Pi * radius * radius;
}

int main()
{
    cout << "Введите радиус: ";
    double radius = 0;
    cin >> radius;

    double areaFetched = 0;
    Area(radius, areaFetched);

    cout << "Площадь равна: " << AreaFetched << endl;
    return 0;
}
```

4. **Отладка.** Что не так в следующем объявлении функции?

```
double Area(double Pi = 3.14, double radius);
```

5. Напишите функцию с типом возвращаемого значения void, которая, тем не менее, способна вернуть вызывающей стороне вычисленную площадь и периметр круга заданного радиуса.

ЗАНЯТИЕ 8

Указатели и ссылки

Одно из самых больших преимуществ языка C++ в том, что он позволяет писать высокоуровневые приложения, абстрагируясь от машинного уровня, и в то же время при необходимости работать близко к аппаратным средствам. Язык C++ позволяет настраивать производительность приложения на уровне байтов и битов. Понимание работы указателей и ссылок — один из этапов на пути к умению писать программы, эффективно использующие системные ресурсы.

На этом занятии...

- Что такое указатель
- Что такое динамическая память
- Как использовать операторы `new` и `delete` для выделения и освобождения памяти
- Как писать стабильные приложения, используя указатели и динамическое распределение памяти
- Что такое ссылка
- Различия между указателями и ссылками
- Когда использовать указатели, а когда ссылки

Что такое указатель

Не усложняя, можно сказать, что *указатель* (pointer) — это переменная, которая хранит адрес области в памяти. Точно так же, как переменная типа `int` используется для хранения целочисленного значения, переменная указателя используется для хранения адреса области памяти (рис. 8.1).

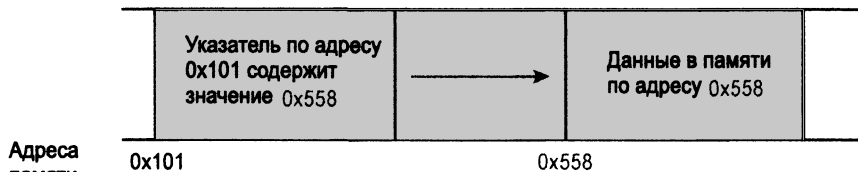


РИС. 8.1. Визуализация указателя

Таким образом, указатель — это переменная, и, как и все переменные, он занимает пространство в памяти (в случае рис. 8.1 — по адресу 0x101). Но особенными указатели делают то, что содержащиеся в них значения (в данном случае — 0x558) интерпретируются как адреса областей памяти. Следовательно, указатель — это специальная переменная, которая *указывает* на область в памяти.

ПРИМЕЧАНИЕ

Адреса памяти обычно представлены в *шестнадцатеричной* записи. Это система счисления с основанием 16, т.е. использующая 16 различных символов — за 0–9 следуют символы A–F. По соглашению перед шестнадцатеричным числом записывается префикс 0x. Таким образом, шестнадцатеричное число 0xA представляет собой 10 в десятичной системе счисления; шестнадцатеричное 0xF — 15; а шестнадцатеричное 0x10 — 16. Дополнительную информацию по этому вопросу можно найти в приложении А, “Двоичные и шестнадцатеричные числа”.

Объявление указателя

Поскольку указатель является переменной, его следует объявить, как и любую иную переменную. Обычно вы объявляете, что указатель указывает на значение определенного типа (например, типа `int`). Это значит, что содержащийся в указателе адрес указывает на область в памяти, содержащую целое число. Можно определить указатель и на нетипизированный блок памяти (называемый также *указателем на void*).

Указатель должен быть объявлен, как и все остальные переменные:

Указываемый_тип * *Имя_Переменной_Указателя*;

Как и в случае с большинством переменных, если не инициализировать указатель, он будет содержать случайное значение. Во избежание обращения к случайной

области памяти указатель инициализируют значением `nullptr`¹. Значение указателя всегда можно проверить на равенство значению `nullptr`, которое не может быть адресом реальной области памяти:

```
Указываемый_тип * Имя_Переменной_Указателя = nullptr;
```

Таким образом, объявление указателя на целое число может иметь следующий вид:

```
int *pInteger = nullptr;
```

ВНИМАНИЕ!

Указатель, как и переменная любого другого изученного на настоящий момент типа данных, до инициализации содержит случайное значение. В случае указателя это случайное значение особенно опасно, поскольку означает некоторый адрес области памяти. Неинициализированные указатели способны заставить вашу программу обратиться к недопустимой области памяти, приводя (в лучшем случае) к аварийному завершению.

Определение адреса переменной с использованием оператора получения адреса &

Переменные — это средство, предоставляемое языком для работы с данными в памяти. Эта концепция была подробно рассмотрена на занятии 3, “Использование переменных и констант”.

Если `varName` — переменная, то выражение `&varName` возвращает адрес места в памяти, где хранится ее значение.

Так, если вы объявили целочисленную переменную, используя хорошо знакомый вам синтаксис

```
int age = 30;
```

то выражение `&age` вернет адрес области памяти, в которую помещается указанное значение 30. Листинг 8.1 демонстрирует концепцию адреса памяти целочисленной переменной, используемого для доступа к хранимому в ней значению.

ЛИСТИНГ 8.1. Определение адресов переменных типа `int` и `double`

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int age = 30;
6:     const double Pi = 3.1416;
7:
8:     // Использование & для поиска адреса в памяти
```

¹ Это значение — нулевой указатель — появилось в языке C++ в стандарте C++11. До этого использовалось совместимое с языком программирования C значение `NULL` (которое можно использовать и сейчас, хотя в новых программах рекомендуется применять значение `nullptr`). —

Примеч. ред.


```
9:     cout << "Адрес age: 0x" << hex << &age << endl;  
10:    cout << "Адрес Pi:  0x" << hex << &Pi << endl;  
11:  
12:    return 0;  
13: }
```

Результат

```
Адрес age: 0x0045FE00  
Адрес Pi:  0x0045FDF8
```

Анализ

Обратите внимание, как оператор получения адреса & используется в строках 9 и 10 для получения адресов переменной `age` и константы `Pi`. Часть `0x` была добавлена по соглашению, используемому для записи шестнадцатеричных чисел.

ПРИМЕЧАНИЕ

Вы уже знаете, что объем памяти, используемый переменной, зависит от ее типа. Применение оператора `sizeof()` в листинге 3.5 показало, что размер целочисленной переменной составляет 4 байта (по крайней мере, на компьютере автора при использовании его компилятора). Приведенный выше вывод свидетельствует о том, что значение целочисленной переменной `age` находится по адресу `0x0045FE08`, а зная, что значение `sizeof(int)` равно 4 байтам, можно сделать вывод, что четыре байта, расположенные в диапазоне `0x0045FE00–0x0045FE04`, принадлежат целочисленной переменной `age`.

ПРИМЕЧАНИЕ

Оператор получения адреса & иногда называют также оператором ссылки (referencing operator).

Использование указателей для хранения адресов

Вы уже знаете, как объявлять указатели и выяснять адрес переменной, а также, что указатели — это переменные, используемые для хранения адреса области памяти. Пришло время объединить эти знания и использовать указатели для хранения адресов, полученных с использованием оператора получения адреса &.

С синтаксисом объявления переменной определенного типа вы уже знакомы:

```
// Объявление переменной  
Тип Имя_Переменной = Начальное_Значение;
```

Чтобы сохранить адрес этой переменной в указателе, следует объявить указатель на тот же *Тип* и инициализировать его, используя оператор получения адреса &:

```
// Объявление указателя на тот же тип и его инициализация  
Тип* Указатель = &Имя_Переменной;
```

Предположим, вы объявили переменную `age` типа `int` так:

```
int age = 30;
```

Указатель на `int`, хранящий для последующего использования адрес значения переменной `age`, при этом объявляется следующим образом:

```
int* pointsToInt = &age; // Указатель на целочисленную переменную age
```

Листинг 8.2 демонстрирует применение указателя для хранения адреса, полученного с помощью оператора `&`.

ЛИСТИНГ 8.2. Объявление и инициализация указателя

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int age = 30;
6:     int* pointsToInt = &age; // Указатель на int,
7:         // инициализированный значением &age
8:     // Вывод значения указателя
9:     cout << "Адрес age: 0x" << hex << pointsToInt << endl;
10:
11:     return 0;
12: }
```

Результат

Адрес age: 0x0045FE00

Анализ

По существу, вывод этого фрагмента кода такой же, как и в предыдущем листинге, поскольку оба примера отображают одну и ту же концепцию — адрес в памяти, где хранится содержимое переменной `age`. Отличие здесь в том, что адрес сначала присваивается указателю (строка 6) и только потом (в строке 9) в поток `cout` выводится значение переменной-указателя.

ПРИМЕЧАНИЕ

У вас выводимый адрес будет иным. Более того, адрес переменной может изменяться при каждом запуске приложения на одном и том же компьютере.

Теперь, когда вы знаете, как сохранить адрес в переменной-указателе, вполне логично предположить, что тому же указателю может быть присвоен другой адрес области памяти и он после этого будет указывать на другое значение, как показано в листинге 8.3.

ЛИСТИНГ 8.3. Присваивание указателю адреса другой переменной

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int age = 30;
6:
7:     int* pointsToInt = &age;
8:     cout << "pointsToInt указывает на age" << endl;
9:
10:    // Вывод значения указателя
11:    cout << "pointsToInt = 0x" << hex << pointsToInt << endl;
12:
13:    int dogsAge = 9;
14:    pointsToInt = &dogsAge;
15:    cout << "pointsToInt указывает на dogsAge" << endl;
16:
17:    cout << "pointsToInt = 0x" << hex << pointsToInt << endl;
18:
19:    return 0;
20: }
```

Результат

```
pointsToInt указывает на age
pointsToInt = 0x002EFB34
pointsToInt указывает на dogsAge
pointsToInt = 0x002EFB1C
```

Анализ

Эта программа демонстрирует, что один и тот же указатель `pointsToInt` на целочисленную переменную способен указывать на любую целочисленную переменную. В строке 7 этот указатель был инициализирован как `&age`, так что он содержал адрес переменной `age`. В строке 14 тому же указателю присваивается результат выражения `&dogsAge`, и после этого он указывает на другую область в памяти, содержащую значение переменной `dogsAge`. Таким образом, приведенный вывод демонстрирует, что значение указателя, первоначально бывшее адресом переменной `age`, сменилось адресом переменной `dogsAge`. Значения этих переменных, естественно, хранятся в разных областях памяти, по адресам `0x002EFB34` и `0x002EFB1C` соответственно.

Доступ к данным с использованием оператора разыменования *

Предположим, у вас есть указатель, содержащий вполне допустимый адрес. Как же теперь обратиться к этой области, чтобы записать или прочитать содержащиеся в ней данные? Для этого используется *оператор разыменования* (dereferencing operator) *. По существу, если есть корректный указатель pData, выражение *pData позволяет получить доступ к значению, хранящемуся по адресу, содержащемуся в этом указателе. Использование оператора * показано в листинге 8.4.

ЛИСТИНГ 8.4. Использование оператора разыменования * для доступа к целочисленному значению

```

0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int age = 30;
6:     int dogsAge = 9;
7:
8:     cout << "age      = " << age << endl;
9:     cout << "dogsAge = " << dogsAge << endl;
10:
11:     int* pointsToInt = &age;
12:     cout << "pointsToInt указывает на age" << endl;
13:
14:     // Вывод значения указателя
15:     cout << "pointsToInt = 0x" << hex << pointsToInt << endl;
16:
17:     // Вывод значения из указанной области
18:     cout << "*pointsToInt = " << dec << *pointsToInt << endl;
19:
20:     pointsToInt = &dogsAge;
21:     cout << "pointsToInt указывает на dogsAge" << endl;
22:
23:     cout << "pointsToInt = 0x" << hex << pointsToInt << endl;
24:     cout << "*pointsToInt = " << dec << *pointsToInt << endl;
25:
26:     return 0;
27: }
```

Результат

```

age      = 30
dogsAge = 9
pointsToInt указывает на age
```

```

pointsToInt = 0x0025F788
*pointsToInt = 30
pointsToInt указывает на dogsAge
pointsToInt = 0x0025F77C
*pointsToInt = 9

```

Анализ

В этом листинге, помимо изменения адреса, хранимого в указателе (как и в предыдущем примере в листинге 8.3), используется также оператор разыменования `*` для отображения значений, хранящихся по двум разным адресам. Обратите внимание на строки 18 и 24. В этих строках осуществляется доступ к целочисленному значению, на которое указывает указатель `pointsToInt`, с использованием оператора разыменования. Поскольку адрес, содержащийся в указателе `pointsToInt`, в строке 20 изменяется, тот же указатель после присваивания позволяет обратиться к переменной `dogsAge` и вывести значение 9.

Когда выполняется оператор разыменования, приложение использует хранящийся в указателе адрес как отправную точку для выборки из памяти 4 байтов, принадлежащих целому числу (поскольку это указатель на тип `int`, а оператор `sizeof(int)` возвращает значение 4). Таким образом, корректность адреса, содержавшегося в указателе, является абсолютно необходимым условием. При инициализации указателя выражением `&age` в строке 11 мы гарантировали, что указатель содержит корректный адрес. Если указатель не инициализировать, он будет содержать случайное значение, которое имелось в области памяти при создании переменной указателя. Обращение к значению такого указателя обычно приводит к ошибке нарушения прав доступа, свидетельствующей о попытке обратиться к области памяти, доступ к которой вашему приложению не разрешен.

ПРИМЕЧАНИЕ

Оператор разыменования `*` называется также *оператором косвенного обращения* (indirection operator).

Указатель в приведенном выше примере использовался для чтения значения из области памяти, на которую он указывает. В листинге 8.5 показано, что происходит, когда оператор `*pointsToInt` используется как l-значение, т.е. для присваивания значения, а не для его чтения.

ЛИСТИНГ 8.5. Работа с данными с помощью указателя и оператора разыменования

```

0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int dogsAge = 30;
6:     cout << "Изначально dogsAge = " << dogsAge << endl;
7:

```

```
8:   int* pAge = &dogsAge;
9:   cout << "pAge указывает на dogsAge" << endl;
10:
11:   cout << "Введите значение dogsAge: ";
12:
13:   // Сохранение значения в области памяти по адресу pAge
14:   cin >> *pAge;
15:
16:   // Вывод адреса
17:   cout << "Значение сохранено по адресу 0x"
18:         << hex << pAge << endl;
19:   cout << "Теперь dogsAge = " << dec << dogsAge << endl;
20:
21:   return 0;
22: }
```

Результат

```
Изначально dogsAge = 30
pAge указывает на dogsAge
Введите значение dogsAge: 10
Значение сохранено по адресу 0x0025FA18
Теперь dogsAge = 10
```

Анализ

Ключевой здесь является строка 14, в которой введенное пользователем целочисленное значение сохраняется в области памяти, на которую указывает указатель `pAge`. Обратите внимание, что несмотря на то, что введенное число сохранено с помощью указателя `pAge`, строка 19 отображает это же значение с помощью переменной `dogsAge`. Дело в том, что указатель `pAge` продолжает указывать на переменную `dogsAge` после своей инициализации в строке 8. Любое изменение в области памяти, в которой хранится значение переменной `dogsAge` и на которую указывает указатель `pAge`, выполненное одним из способов, может быть прочитано другим способом.

Значение `sizeof()` для указателя

Как вы уже знаете, указатель — это просто переменная, содержащая адрес области памяти. Следовательно, независимо от типа, на который он указывает, содержимое указателя — числовое значение адреса. Длина адреса — это количество байтов, необходимых для его хранения; она является постоянной для конкретной системы. Таким образом, результат выполнения оператора `sizeof()` для указателя зависит от компилятора и операционной системы, для которой программа была скомпилирована, и не зависит от характера данных, на которые он указывает, что и продемонстрировано в листинге 8.6.

ЛИСТИНГ 8.6. Одинаковый размер указателей на различные типы данных

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "sizeof для типов:" << endl;
6:     cout << "sizeof(char)      = " << sizeof(char) << endl;
7:     cout << "sizeof(int)       = " << sizeof(int) << endl;
8:     cout << "sizeof(double)    = " << sizeof(double) << endl;
9:
10:    cout << "sizeof для указателей на типы:" << endl;
11:    cout << "sizeof(char*)     = " << sizeof(char*) << endl;
12:    cout << "sizeof(int*)      = " << sizeof(int*) << endl;
13:    cout << "sizeof(double*)    = " << sizeof(double*) << endl;
14:
15:    return 0;
16: }
```

Результат

```
sizeof для типов:
sizeof(char)      = 1
sizeof(int)       = 4
sizeof(double)    = 8
sizeof для указателей на типы:
sizeof(char*)     = 4
sizeof(int*)      = 4
sizeof(double*)   = 4
```

Анализ

Вывод однозначно демонстрирует, что при том, что значение `sizeof(char)` равно 1 байту, а `sizeof(double)` — 8 байтам, размер указателей на них всегда остается постоянным — 4 байта. Это связано с тем, что объем памяти, необходимый для хранения адреса, является одним и тем же независимо от того, содержит ли память по указанному адресу 1 байт или 8 байтов.

ПРИМЕЧАНИЕ

Вывод листинга 8.6 свидетельствует, что размер указателей составляет 4 байта, но в вашей системе результат может быть иным. Приведенный выше вывод был получен при компиляции кода на 32-битовом компиляторе. Если вы используете 64-битовый компилятор и запустите программу на 64-разрядной системе, то размер вашего указателя может составить 64 бита, т.е. 8 байтов.

Динамическое распределение памяти

Когда вы пишете программу, содержащую объявление массива, такое как

```
int Numbers[100]; // Статический массив для 100 целых чисел
```

возникают две проблемы.

1. Вы фактически ограничиваете возможности своей программы, поскольку она не сможет хранить больше 100 чисел.
2. Вы неэффективно используете ресурсы в случае, когда храниться должно, скажем, только 1 число, а память все равно выделяется для 100 чисел.

Причиной этих проблем является статическое, фиксированное выделение памяти для массива компилятором.

Чтобы программа могла оптимально использовать память, в зависимости от конкретных потребностей пользователя, необходимо использовать динамическое распределение памяти. Оно позволяет при необходимости выделять большее количество памяти и освобождать ее, когда необходимости в ней больше нет. Язык C++ предоставляет два оператора, `new` и `delete`, позволяющие управлять использованием памяти в вашем приложении. В эффективном динамическом распределении памяти критически важную роль играют указатели, хранящие адреса памяти.

Использование `new` и `delete` для выделения и освобождения памяти

Оператор `new` используется для выделения новых блоков памяти. Чаще всего используется версия оператора `new`, возвращающая указатель на затребованную область памяти в случае успеха и генерирующая исключение в противном случае. При использовании оператора `new` необходимо указать тип данных, для которого выделяется память:

```
Тип* Указатель = new Тип; // Запрос памяти для одного элемента
```

Вы можете также определить количество элементов, для которых хотите выделить память (если нужно выделять память для массива элементов):

```
Тип* Указатель = new Тип[Количество]; // Запрос памяти для указан-  
// ного количества элементов
```

Таким образом, если необходимо разместить в памяти целые числа, используйте следующий код:

```
int* pointToAnInt = new int;      // Указатель на целое число  
int* pointToNums  = new int[10];  // Указатель на массив из 10  
// целых чисел
```


ПРИМЕЧАНИЕ

Обратите внимание на то, что оператор `new` запрашивает область памяти. Нет никакой гарантии, что запрос всегда будет удовлетворен успешно, поскольку это зависит от состояния системы и доступного количества памяти.

Каждая область памяти, выделенная оператором `new`, должна быть в конечном счете освобождена соответствующим оператором `delete`:

```
Тип* Указатель = new Тип;  
delete Указатель; // Освобождение памяти, выделенной  
                // ранее для одного экземпляра Типа
```

Это справедливо и при запросе памяти для нескольких элементов:

```
Тип* Указатель = new Тип[Количество];  
delete[] Указатель; // освободить выделенный ранее массив
```

ПРИМЕЧАНИЕ

Обратите внимание на применение оператора `delete[]` при выделении блока с использованием оператора `new[...]` и оператора `delete` при выделении только одного элемента с использованием оператора `new`.

Если не освободить выделенную память по окончании ее использования, она останется выделенной и недоступной для последующих выделений вашему или иным приложениям. Такая *утечка памяти* может привести даже к замедлению работы приложения или компьютера в целом, и ее следует избегать любой ценой.

В листинге 8.7 показано динамическое выделение и освобождение памяти.

ЛИСТИНГ 8.7. Использование оператора `*` для доступа к памяти, выделенной с помощью оператора `new`, и ее освобождение оператором `delete`

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: int main()  
4: {  
5:     // Выделение памяти для int  
6:     int* pointsToAnAge = new int;  
7:  
8:     // Использование выделенной памяти  
9:     cout << "Введите возраст собаки: ";  
10:    cin >> *pointsToAnAge;  
11:  
12:    // Применение оператора разыменования *  
13:    cout << "Возраст " << *pointsToAnAge << " хранится по "  
14:        << "адресу 0x" << hex << pointsToAnAge << endl;  
15:  
16:    delete pointsToAnAge; // Освобождение памяти
```

```
17:     return 0;
18: }
```

Результат

Введите возраст собаки: 9

Возраст 9 хранится по адресу 0x00338120

Анализ

Строка 6 демонстрирует использование оператора `new` для выделения памяти под целое число, в которой планируется хранить введенный пользователем возраст собаки. Обратите внимание, что оператор `new` возвращает указатель, который сохраняется путем присваивания. Введенный пользователем возраст сохраняется в этой выделенной области памяти, а поток `cin` обращается к ней в строке 10, используя оператор `*`. Строки 13 и 14 отображают значение возраста; при этом снова используется оператор разыменования. Здесь же выводится адрес области памяти, в которой хранится этот возраст. Содержавшийся в указателе `pointsToAnAge` адрес в строке 13 остается тем же, который был возвращен оператором `new` в строке 6, — он с тех пор не изменялся.

ВНИМАНИЕ!

Оператор `delete` не может быть вызван только для адреса, который был возвращен оператором `new` и еще не был освобожден с помощью оператора `delete`.
Таким образом, несмотря на то что указатели в листинге 8.6 содержат допустимые адреса, их нельзя освобождать с помощью оператора `delete`, поскольку они не были возвращены при вызове оператора `new`.

Обратите внимание, что при выделении памяти для диапазона элементов с использованием оператора `new[]` вы обязаны освобождать ее с помощью оператора `delete[]`, как показано в листинге 8.8.

ЛИСТИНГ 8.8. Выделение памяти оператором `new[]` и освобождение оператором `delete[]`

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Количество чисел в массиве?" << endl;
7:     int numEntries = 0;
8:     cin >> numEntries;
9:
10:    int* myNumbers = new int[numEntries];
11:
```

```
12:     cout << "Выделена память по адресу: 0x"  
13:         << myNumbers << hex << endl;  
14:  
15:     // Освобождение памяти  
16:     delete[] myNumbers;  
17:     return 0;  
18: }
```

Результат

Количество чисел в массиве?

5001

Выделена память по адресу: 0x00C71578

Анализ

Самыми важными являются строки, в которых используются операторы `new[]` и `delete[]`, — строки 10 и 16 соответственно. По сравнению с листингом 8.7, в котором выделялось место только для одного элемента, здесь выделяется блок памяти для массива элементов, количество которых определяет пользователь. В данном примере мы ввели значение 5001, но оно может быть и иным, например 20 или 55000. Эта программа при каждом запуске выделяет разное количество памяти, зависящее от ввода пользователя. Для такого выделения памяти с помощью оператора `new[]` в программе должно иметься соответствующее освобождение памяти с помощью оператора `delete[]` по окончании работы с выделенной памятью.

ПРИМЕЧАНИЕ

Операторы `new` и `delete` выделяют область в динамической памяти. *Динамическая память* (free store) — это абстракция памяти в форме пула памяти, из которой диспетчер памяти может выделять блоки памяти для вешего приложения и освобождать их, возвращая в пул свободной памяти.

Указатели и операции инкремента и декремента

Указатель содержит адрес области памяти. Например, указатель на целое число в листинге 8.3 содержит значение 0x002EFB34 — адрес, по которому размещается целое число. Само целое число имеет длину 4 байта, а следовательно, занимает в памяти четыре ячейки от 0x002EFB34 до 0x002EFB37. Приращение значения этого указателя с использованием оператора инкремента `++` не даст значения 0x002EFB35, указывающего на середину целого числа, так как это было бы бессмысленно.

Операция инкремента (приращения) или декремента с указателем интерпретируется компилятором как потребность перевода указателя на следующее значение в блоке памяти, с учетом того, что все элементы имеют один и тот же тип, а не на следующий байт (если, конечно, тип значения не имеет длину 1 байт, как, например, тип `char`).

Так, результатом инкремента такого указателя, как `pointsToInt` из листинга 8.3, будет увеличение его значения на 4 байта, что соответствует размеру типа `int`.

Использование оператора ++ для этого указателя говорит компилятору, что вы хотите перенести указатель на следующее расположенное далее целое число. Следовательно, после приращения указатель будет указывать на адрес 0x002EFB38. Точно так же добавление 2 к этому указателю приведет к его переносу на 2 целых числа далее, что составит 8 байтов. Впоследствии вы увидите корреляцию между таким поведением, демонстрируемым указателями, и индексами, используемыми в массивах.

Декремент указателя с использованием оператора -- демонстрирует тот же эффект: значение, содержащееся в указателе, уменьшается на размер типа данных, на которые он указывает.

Что происходит при инкременте и декременте указателя

Содержавшийся в указателе адрес увеличивается или уменьшается на размер указываемого типа (а не на один байт). Таким образом, компилятор гарантирует, что указатель никогда не будет указывать на середину или конец данных, помещенных в память, а только на их начало.

Если указатель был объявлен как

Тип pType = Адрес;*

то выражение ++pType означает, что после этого указатель pType содержит значение адреса *Адрес+sizeof(Тип)*.

В листинге 8.9 показан результат инкремента указателей или добавления к ним смещений.

ЛИСТИНГ 8.9. Использование смещений и операторов для инкремента и декремента указателей

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Количество вводимых чисел? ";
6:     int numEntries = 0;
7:     cin >> numEntries;
8:
9:     int* pointsToInts = new int[numEntries];
10:
11:     cout << "Память выделена" << endl;
12:     for(int counter = 0; counter < numEntries; ++counter)
13:     {
14:         cout << "Введите число "<< counter << ": ";
15:         cin >> *(pointsToInts + counter);
16:     }
17:
18:     cout << "Введены числа: " << endl;
19:     for(int counter = 0; counter < numEntries; ++counter)
```

```
20:         cout << *(pointsToInts++) << " ";
21:
22:     cout << endl;
23:
24:     // Возврат указателя в начальную позицию
25:     pointsToInts -= numEntries;
26:
27:     // Освобождение памяти
28:     delete[] pointsToInts;
29:
30:     return 0;
31: }
```

Результат

```
Количество вводимых чисел? 2
Память выделена
Введите число 0: 789
Введите число 1: 575
Введены числа:
789 575
```

Другой запуск:

```
Количество вводимых чисел? 5
Память выделена
Введите число 0: 789
Введите число 1: 12
Введите число 2: -65
Введите число 3: 285
Введите число 4: -101
Введены числа:
789 12 -65 285 -101
```

Анализ

Прежде чем выделять память, программа запрашивает у пользователя количество целых чисел, которые он хочет ввести. Затем в строке 9 программа выделяет запрошенную память. В листинге продемонстрированы два способа увеличения указателя. Первый в строке 15 использует смещение `counter`. Второй в строке 20 использует оператор `++`, который увеличивает содержащийся в переменной указатель так, что он указывает на следующее целочисленное значение в выделенной памяти. Операторы декремента и инкремента были введены на занятии 5, “Выражения, инструкции и операторы”.

Строки 12–16 содержат цикл `for`, в котором пользователя просят ввести числа, которые затем, в строке 15, сохраняются в последовательных позициях в выделенной памяти. Здесь отсчитываемое от нуля значение смещения (`counter`) добавляется

к указателю, заставляя создавать команды, которые вносят введенное пользователем значение в соответствующую область памяти, без перезаписи предыдущего значения. Цикл `for` в строках 19 и 20 аналогичным образом выводит эти значения, сохраненные предыдущим циклом, на консоль.

Исходный адрес, возвращенный оператором `new[]` при выделении памяти, должен использоваться в вызове `delete[]` для освобождения более ненужной памяти. Так как значение указателя, содержащееся в переменной `pointsToInts`, было изменено операторами `++` в строке 20, мы возвращаем указатель в начальную позицию с помощью применения оператора `--` в строке 25, перед вызовом `delete[]` в строке 28.

Использование ключевого слова `const` с указателями

Из занятия 3, “Использование переменных и констант”, вы узнали, что объявление переменной как `const` гарантирует, что значение переменной после ее инициализации останется фиксированным. Значение такой переменной не может быть изменено, и она не может использоваться в качестве l-значения.

Указатели — это тоже переменные, а следовательно, ключевое слово `const` вполне уместно и для них. Однако указатели — это особый вид переменных, которые содержат адреса областей памяти и позволяют модифицировать данные в памяти. Таким образом, когда дело доходит до указателей и констант, возможны следующие комбинации.

- Содержащийся в указателе адрес является константным и не может быть изменен, однако данные, на которые он указывает, вполне могут быть изменены:

```
int daysInMonth = 30;
int* const pDaysInMonth = &daysInMonth;
*pDaysInMonth = 31;           // OK! Значение может быть изменено
int daysInLunarMonth = 28;
pDaysInMonth = &daysInLunarMonth; // Ошибка компиляции: нельзя
                                   // изменить адрес!
```

- Данные, на которые указывает указатель, являются константными и не могут быть изменены, но сам адрес, содержащийся в указателе, вполне может быть изменен (т.е. указатель может указывать и на другое место):

```
int hoursInDay = 24;
const int* pointsToInt = &hoursInDay;
int monthsInYear = 12;
pointsToInt = &monthsInYear; // OK!
*pointsToInt = 13;           // Ошибка времени компиляции:
                              // изменять данные нельзя
int* newPointer = pointsToInt; // Ошибка времени компиляции:
                              // нельзя присваивать указатель на
                              // константу указателю на не константу
```

- И содержащийся в указателе адрес, и значение, на которое он указывает, являются константами и не могут быть изменены (самый ограничивающий вариант):

```
int hoursInDay = 24;
const int* const pHoursInDay = &HoursInDay;
*pHoursInDay = 25; // Ошибка компиляции: нельзя изменять значение, на которое указывает данный указатель
int daysInMonth = 30;
pHoursInDay = &daysInMonth; // Ошибка компиляции: нельзя изменять значение данного указателя
```

Эти разнообразные формы константности особенно полезны при передаче указателей в функции. Параметры функций следует объявлять, обеспечивая максимально ограничивающий уровень константности, чтобы гарантировать невозможность функции изменить значение, на которое указывает указатель, если таковое изменение не предполагается в данной функции. Это предотвратит возможность допущения программистом ошибочного изменения значения указателя или данных, на которые он указывает.

Передача указателей в функции

Указатели — это эффективное средство передачи функции областей памяти, содержащих значения и способных содержать результат. При использовании указателей с функциями важно гарантировать, что вызываемой функции разрешено изменять только те параметры, которые вы хотите позволить ей изменять, но не другие. Например, функции, вычисляющей площадь круга по заданному радиусу, передаваемому через указатель, нельзя позволить изменять этот радиус. В этом случае пригодятся константные указатели, позволяющие эффективно управлять тем, что функции разрешено изменять, а что — нет (листинг 8.10).

ЛИСТИНГ 8.10. Использование ключевого слова `const` при вычислении площади круга

```
0: #include <iostream>
1: using namespace std;
2:
3: void CalcArea(const double* const pPi, // Константный указатель
4:              const double* const pRadius, // на константные данные:
5:              double* const pArea) // Изменяемо значение, но не адрес
6: {
7:     // Проверка корректности указателей перед использованием!
8:     if (pPi && pRadius && pArea)
9:         *pArea = (*pPi) * (*pRadius) * (*pRadius);
10: }
11:
12: int main()
13: {
```

```
14:    const double Pi = 3.1416;
15:
16:    cout << "Введите радиус круга: ";
17:    double radius = 0;
18:    cin >> radius;
19:
20:    double area = 0;
21:    CalcArea (&Pi, &radius, &area);
22:
23:    cout << "Площадь равна " << Area << endl;
24:
25:    return 0;
26: }
```

Результат

Введите радиус круга: 10.5
Площадь равна 346.361

Анализ

Строки 3–5 демонстрируют две разновидности константности, в которых указатели `pRadius` и `pPi` передаются как константные указатели на константные данные, так что ни адрес, хранимый в указателе, ни данные, на которые он указывает, не могут быть изменены. Указатель `pArea` представляет собой параметр, предназначенный для сохранения результата расчета. Значение указателя (адрес памяти) не может быть изменен, но данные, на которые он указывает, могут и должны быть изменены. Строка 8 демонстрирует проверку допустимости переданных функции указателей перед их использованием. Вы вряд ли захотите, чтобы функция вычисляла площадь, если вызывающая функция по неосторожности в качестве любого из параметров получает нулевой указатель, поскольку это привело бы к нарушению прав доступа к памяти.

Сходство между массивами и указателями

Вам не кажется, что у примера в листинге 8.9, в котором для доступа к следующему числу в памяти выполнялся инкремент указателя с использованием индекса, отсчитываемого от нуля, слишком много сходства с индексацией массивов? Объявляя массив целых чисел

```
int myNumbers[5];
```

вы требуете от компилятора выделить фиксированный объем памяти для хранения пяти целых чисел и предоставить вам указатель на первый элемент в этом массиве, идентифицируемый именем, которое вы использовали для переменной массива. Другими словами, `myNumbers` — это указатель на первый элемент `myNumbers[0]`. Листинг 8.11 подчеркивает эту аналогию.

ЛИСТИНГ 8.11. Переменная массива — это указатель на первый его элемент

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Статический массив из 5 целых чисел
6:     int myNumbers[5];
7:
8:     // Массив присваивается указателю на тип int
9:     int* pointToNums = myNumbers;
10:
11:    // Вывод адреса, содержащегося в указателе
12:    cout << "pointToNums    = 0x" << hex << pointToNums << endl;
13:
14:    // Адрес первого элемента массива
15:    cout << "&myNumbers[0] = 0x" << hex << &myNumbers[0] << endl;
16:
17:    return 0;
18: }
```

Результат

```
pointToNums    = 0x004BFE8C
&myNumbers[0] = 0x004BFE8C
```

Анализ

Эта простая программа демонстрирует, что переменная массива может быть присвоена указателю того же типа (см. строку 9), по существу, подтверждая, что переменная массива родственна указателю. Строки 12 и 15 демонстрируют, что хранящийся в указателе адрес является тем же адресом, по которому в памяти находится первый элемент массива (с индексом 0). Тем самым программа подтверждает, что массив — это указатель на его первый элемент.

Если необходимо получить доступ ко второму элементу, `myNumbers[1]`, то можно воспользоваться указателем `pointToNums` с использованием синтаксиса `*(pointToNums+1)`. Обращение к третьему элементу статического массива выглядит как `myNumbers[2]`, тогда как к третьему элементу динамического массива можно обратиться с помощью синтаксиса `*(pointToNums+2)`.

Поскольку переменные массивов, по существу, являются указателями, при работе с массивами вполне можно использовать оператор разыменования `*`, как при работе с обычными указателями. Точно так же можно использовать оператор индексации `[]` при работе с указателями, как показано в листинге 8.12.

ЛИСТИНГ 8.12. Доступ к элементам массива с использованием операторов разыменования и индексации

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     const int ARRAY_LEN = 5;
6:
7:     // Инициализированный статический массив из 5 целых чисел
8:     int myNumbers[ARRAY_LEN] = {24, -1, 365, -999, 2011};
9:
10:    // Указатель, инициализированный первым элементом массива
11:    int* pointToNums = myNumbers;
12:
13:    cout << "Указатель и оператор *:" << endl;
14:    for(int index = 0; index < ARRAY_LEN; ++index)
15:        cout << "Элемент " << index << " = "
16:            << *(myNumbers + index) << endl;
17:    cout << "Указатель и оператор []:" << endl;
18:    for(int index = 0; index < ARRAY_LEN; ++index)
19:        cout << "Элемент " << index << " = "
20:            << pointToNums[index] << endl;
21:    return 0;
22: }
```

Результат

```
Указатель и оператор *:
Элемент 0 = 24
Элемент 1 = -1
Элемент 2 = 365
Элемент 3 = -999
Элемент 4 = 2011
Указатель и оператор []:
Элемент 0 = 24
Элемент 1 = -1
Элемент 2 = 365
Элемент 3 = -999
Элемент 4 = 2011
```

Анализ

Приложение содержит статический массив из пяти целых чисел, инициализированных пятью исходными значениями в строке 8. Приложение отображает содержимое этого массива, используя два альтернативных подхода: с использованием переменной

массива и оператора разыменования в строках 15 и 16, а также с использованием переменной указателя и оператора индексации в строках 19 и 20.

Таким образом, данная программа свидетельствует, что и массив `myNumbers`, и указатель `pointToNums` фактически демонстрируют поведение указателя. Другими словами, объявление массива подобно созданию указателя для работы в пределах фиксированного диапазона памяти. Обратите внимание, что можно присвоить массив указателю, как это сделано в строке 11, но нельзя присвоить указатель массиву, поскольку массив имеет статический характер, а следовательно, не может быть l-значением.

ВНИМАНИЕ!

Не забывайте, что указатели, созданные динамически с помощью оператора `new`, следует освободить с помощью оператора `delete`, даже если вы использовали для обращения к данным тот же синтаксис, что и для статических массивов.

Если вы забудете об этом, то произойдет утечка памяти, а это плохо.

Наиболее распространенные ошибки при использовании указателей

Язык C++ позволяет выделять память динамически, чтобы вы могли оптимизировать использование памяти вашим приложением. В отличие от более новых языков, таких как C# и Java, основанных на среде времени выполнения, язык C++ не использует автоматический сборщик мусора, который освобождает выделенную вашей программе память, когда она уже не используется. Возможность полного управления памятью в C++ с помощью указателей имеет обратную сторону — массу возможностей для программиста сделать ошибку.

Утечки памяти

Вероятно, это одна из самых распространенных проблем приложений C++: чем дольше они выполняются, тем больший объем памяти используют и тем самым замедляют систему. Это, как правило, случается, когда программист не обеспечил освобождение памяти, выделенной динамически оператором `new`, с помощью вызова оператора `delete` по завершении ее использования.

Обеспечение освобождения всей выделенной памяти — задача программиста, т.е. ваша. Вы должны следить, чтобы никогда не происходили некоторые вещи наподобие показанных ниже:

```
int* pointToNums = new int[5]; // Выделение памяти
// Использование указателя pointToNums
...
// Освобождение памяти с помощью delete[] pointToNums не сделано
...
// Очередное выделение и перезапись указателя
pointToNums = new int[10];      // Утечка ранее выделенной памяти
```

Когда указатели указывают на недопустимые области памяти

Когда вы разыменовываете указатель для доступа к значению, на которое он указывает, необходимо гарантировать, что указатель содержит допустимый адрес области памяти, иначе поведение вашей программы будет непредсказуемым. Некорректные указатели являются практически наиболее частой причиной аварийных отказов приложения. Указатели могут оказаться некорректными по ряду причин, связанных с плохим управлением памятью. Типичный случай некорректного указателя приведен в листинге 8.13.

ЛИСТИНГ 8.13. Пример плохого программирования и некорректных указателей

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Неинициализированный указатель (плохо!)
6:     bool* isSunny;
7:
8:     cout << "Сегодня солнечно (y/n)? ";
9:     char userInput = 'y';
10:    cin >> userInput;
11:
12:    if (userInput == 'y')
13:    {
14:        isSunny = new bool;
15:        *isSunny = true;
16:    }
17:
18:    // isSunny содержит некорректное значение при вводе 'n'
19:    cout << "Булев флаг равен " << *isSunny << endl;
20:
21:    // delete вызывается и тогда, когда не был вызван new
22:    delete isSunny;
23:
24:    return 0;
25: }
```

Результат

```
Сегодня солнечно (y/n)? y
Булев флаг равен 1
```

Второй запуск:

```
Сегодня солнечно (y/n)? n
<НЕПРИЯТНОСТИ>
```

Анализ

В программе много проблем, ряд из которых уже прокомментирован в коде. Обратите внимание, что в строке 14 выделяется память, а ее адрес присваивается указателю в блоке `if` только тогда, когда его условие этой конструкции `if` выполняется, т.е. когда пользователь, соглашаясь, введет 'y'. При любом другом вводе пользователя этот блок `if` не выполняется, и указатель `isSunny` остается некорректным. Таким образом, когда во второй раз пользователь вводит 'n', указатель `isSunny` содержит некорректный адрес области памяти, и обращение к значению по этому указателю в строке 19 создает проблему.

Кроме того, вызов оператора `delete` в строке 22 для этого указателя, который не был инициализирован значением, возвращенным оператором `new`, также является некорректным. Обратите внимание, что, если у вас есть копия указателя, вызов оператора `delete` необходимо выполнять только для одного из этих указателей (впрочем, наличия неконтролируемых копий указателей также следует избегать).

Лучшая (более безопасная и стабильная) версия программы из листинга 8.13 использовала бы инициализацию указателей, проверку допустимости их значений и освобождение только однажды и только для корректного значения указателя.

Висячие (беспризорные, дикие) указатели

Любой корректный указатель становится некорректным после того, как он освобождается оператором `delete`. Так, если бы указатель `isSunny` был использован после строки 22, то даже в случае, если пользователь действительно ввел 'y' и указатель был корректным до удаления, после вызова оператора `delete` он становится некорректным и не должен использоваться.

Чтобы избежать этой проблемы, большинство программистов присваивают указателю при инициализации и после его освобождения значение `nullptr`, а затем, используя его, проверяют корректность указателя, прежде чем применить к нему оператор разыменования.

Ознакомившись с возможными проблемами при использовании указателей, давайте исправим дефектный код листинга 8.13 так, как показано в листинге 8.14.

ЛИСТИНГ 8.14. Безопасная версия кода из листинга 8.13

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Сегодня солнечно (y/n)? ";
6:     char userInput = 'y';
7:     cin >> userInput;
8:
9:     // Объявление и инициализация указателя
10:    bool* const isSunny = new bool;
11:    *isSunny = true;
12:
```

```
13:     if (userInput == 'n')
14:         *isSunny = false;
15:
16:     cout << "Булев флаг равен " << *isSunny << endl;
17:
18:     // Освобождение выделенной памяти
19:     delete isSunny;
20:
21:     return 0;
22: }
```

Результат

Сегодня солнечно (y/n)? **y**
Булев флаг равен 1

Следующий запуск:

Сегодня солнечно (y/n)? **n**
Булев флаг равен 0

Анализ

Небольшое изменение кода делает его безопасным при любом вводе пользователя. Обратите внимание, как в строке 10 указатель при объявлении инициализируется корректным значением, получаемым от оператора `new`. Мы использовали модификатор `const`, чтобы позволить изменять данные, на которые указывает данный указатель, но не само его значение. В строке 11 мы инициализировали значение, на которое указывает указатель, значением `true`. Все внесенные изменения гарантируют, что указатель останется корректным до конца программы, а выделенная память будет корректно освобождена в строке 19 независимо от ввода пользователя.

Проверка успешности запроса с использованием оператора `new`

До сих пор в нашем коде мы полагали, что оператор `new` всегда возвращает корректный указатель на блок памяти. На самом деле оператор `new`, как правило, выполняется успешно, если только не был запрошен необычно большой объем памяти или если система не находится в столь критическом состоянии, что у нее не хватает памяти. Существуют приложения, которые должны запрашивать большие объемы памяти (например, приложения баз данных), да и вообще, не следует думать, что распределение памяти всегда осуществляется успешно. Язык C++ предоставляет две возможности удостовериться в успешности выделения памяти. Основной способ, действующий по умолчанию, подразумевает генерацию исключения `std::bad_alloc` при неудачном выделении памяти. Генерация исключения приводит к прерыванию выполнения приложения с сообщением об ошибке наподобие “unhandled exception” (необработанное исключение), если только вы не создали *обработчик исключения* (exception handler).

Исключения подробно описываются на занятии 28, “Обработка исключений”. Листинг 8.15 позволяет увидеть, как может использоваться обработка исключений для выяснения, успешно ли произошло выделение памяти. Не слишком беспокойтесь, если сейчас обработка исключений покажется вам сложной и непонятной — здесь она упоминается только ради полноты изложения темы о выделении памяти. Вы можете вернуться к этому листингу позже, после занятия 28, “Обработка исключений”.

ЛИСТИНГ 8.15. Обработка исключения при неудаче оператора `new`

```
0: #include <iostream>
1: using namespace std;
2:
3: // Удалите блок try-catch, если хотите аварийного завершения
4: int main()
5: {
6:     try
7:     {
8:         // Запрос БОЛЬШОГО количества памяти
9:         int* pointsToManyNums = new int[0x1fffffff];
10:        // Использование памяти
11:
12:        delete[] pointsToManyNums;
13:    }
14:    catch(bad_alloc)
15:    {
16:        cout << "Ошибка выделения памяти" << endl;
17:    }
18:    return 0;
19: }
```

Результат

Ошибка выделения памяти

Анализ

На вашем компьютере эта программа может выполняться по-другому. У автора программа не смогла успешно выделить пространство для 536870911 целых чисел. Без обработчика исключения (блока `catch`, расположенного в строках 14–17) программа завершается аварийно. В режиме отладки интегрированной среды разработки Microsoft Visual Studio выполнение программы приводит к результату, показанному на рис. 8.2.

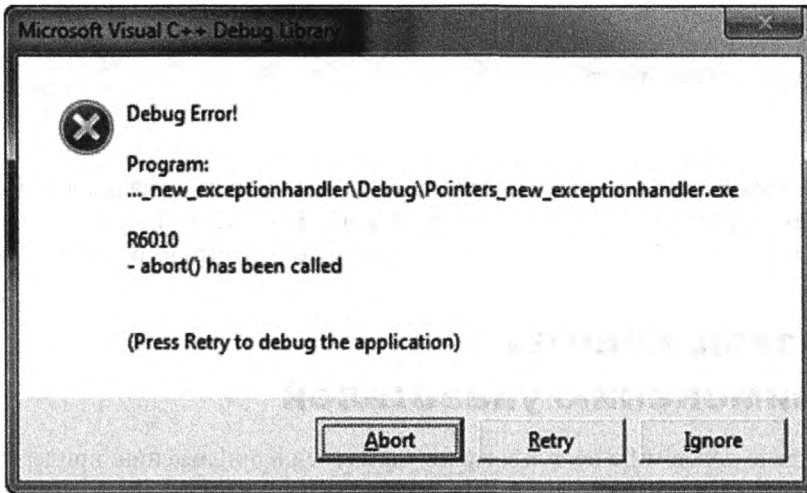


РИС. 8.2. Аварийное завершение программы при отсутствии обработки исключений в листинге 8.15 (отладка в MSVC)

Наличие обработчика исключения предоставило бы приложению способ управляемого выхода после информирования пользователя о возникшей проблеме.

Для тех, кто не хочет работать с исключениями, есть также вариант оператора `new`, называемый `new(nothrow)`, который не генерирует исключение, а возвращает значение `nullptr`, которое можно проверить при проверке корректности указателя до его использования (листинг 8.16).

ЛИСТИНГ 8.16. Использование оператора `new(nothrow)`

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Запрос большого количества памяти с помощью new(nothrow)
6:     int* pointsToNums = new(nothrow) int[0x1fffffff];
7:
8:     if (pointsToNums) // Проверка pointsToNums != nullptr
9:     {
10:        // Использование выделенной памяти
11:        delete[] pointsToNums;
12:    }
13:    else
14:        cout << "Ошибка выделения памяти" << endl;
15:
16:    return 0;
17: }
```


Результат

Ошибка выделения памяти

Анализ

Программа в этом листинге работает так же, как и программа в листинге 8.15, но использует оператор `new(nothrow)`, возвращающий при неудачном выделении памяти значение `nullptr`. Оба варианта корректно работают, так что выбор остается за вами.

Полезные советы по применению указателей

Когда дело доходит до использования указателей в приложении, придерживайтесь простых правил, которые упростят вам жизнь.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
<p>Инициализируйте переменные указателей всегда, иначе они будут содержать случайные значения. Эти случайные значения интерпретируются как адреса областей памяти, и у вашего приложения может не быть прав доступа к ним. Если вы не можете инициализировать указатель допустимым адресом, возвращенным оператором <code>new</code> или другой допустимой переменной, инициализируйте его значением <code>nullptr</code>.</p> <p>Убедитесь в корректности используемых вашей программой указателей, иначе ваша программа может аварийно завершиться.</p> <p>Помните о необходимости освобождать память, выделенную оператором <code>new</code>, с помощью оператора <code>delete</code>. В противном случае ваше приложение создаст утечку памяти и уменьшит производительность системы.</p>	<p>Не используйте указатель для обращения к блоку памяти после того, как он был освобожден оператором <code>delete</code>.</p> <p>Не применяйте оператор <code>delete</code> к одному и тому же адресу повторно.</p> <p>Не допускайте утечки памяти, забыв вызвать оператор <code>delete</code> для выделенного ранее блока памяти.</p>

Что такое ссылка

Ссылка (reference) — это псевдоним переменной. При объявлении ссылки ее необходимо инициализировать переменной. Таким образом, ссылочная переменная — это только другое средство доступа к данным, хранимым в переменной.

Для объявления ссылки используется символ `&`, как в следующем примере:

```
VarType original = Value;
VarType& referenceVariable = original;
```

Чтобы лучше понять, как объявлять ссылки и использовать их, рассмотрим листинг 8.17.

ЛИСТИНГ 8.17. Ссылки — псевдонимы переменных

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int original = 30;
6:     cout << "original = " << original << endl;
7:     cout << "Адрес original: " << hex << &original << endl;
8:
9:     int& ref = original;
10:    cout << "Адрес ref:      " << hex << &ref << endl;
11:
12:    int& ref2 = ref;
13:    cout << "Адрес ref2:      " << hex << &ref2 << endl;
14:    cout << "ref2 = " << dec << Ref2 << endl;
15:
16:    return 0;
17: }
```

Результат

```
original = 30
Адрес original: 0044FB5C
Адрес ref:      0044FB5C
Адрес ref2:      0044FB5C
ref2 = 30
```

Анализ

Вывод показывает, что ссылки (независимо от того, чем они инициализируются, — исходной переменной, как в строке 9, или ссылкой, как в строке 12) адресуют одну и ту же область памяти, в которой содержится исходное значение. Таким образом, ссылки — это настоящие псевдонимы, т.е. просто другие имена переменной `original`. Отображение значения с использованием ссылки `ref2` в строке 14 дает такой же результат, как и при использовании переменной `original` в строке 6, поскольку псевдоним `ref2` и переменная `original` относятся к одной и той же области в памяти.

Зачем нужны ссылки

Ссылки позволяют работать с областью памяти, которой они инициализируются. Это делает ссылки особенно полезными при создании функций. Как вы уже изучали на занятии 7, “Организация кода с помощью функций”, типичная функция объявляется так:

```
Возвращаемый_Тип Сделай_Нечто(Тип Параметр);
```

Функция `Сделай_Нечто()` вызывается так:

```
Возвращаемый_Тип Результат = Сделай_Нечто(Аргумент);
```

Приведенный выше код ведет к копированию аргумента в параметр, который затем используется функцией `Сделай_Нечто()`. Этап копирования может быть весьма продолжительным, если рассматриваемый `Аргумент` занимает много памяти. Аналогично, когда функция `Сделай_Нечто()` возвращает значение, оно копируется в `Результат`. Было бы хорошо, если бы можно было избежать этого копирования, позволяя функции работать непосредственно с данными в стеке вызывающей функции. Ссылки обеспечивают эту возможность.

Версия функции без копирования выглядит следующим образом:

```
Возвращаемый_Тип Сделай_Нечто(Тип& Параметр);
```

Вызывается функция так же, как и раньше:

```
Возвращаемый_Тип Результат = Сделай_Нечто(Аргумент);
```

Но, поскольку аргумент передается по ссылке, параметр будет не копией аргумента, а псевдонимом последнего, как `ref` в листинге 8.17. Кроме того, функция, которая получает параметр как ссылку, может возвращать значение, используя ссылочные же параметры. Рассмотрим листинг 8.18, чтобы понять, как функции могут использовать ссылки вместо возвращаемых значений.

ЛИСТИНГ 8.18. Возврат результата через параметр, передаваемый по ссылке

```
0: #include <iostream>
1: using namespace std;
2:
3: void GetSquare(int& number)
4: {
5:     number *= number;
6: }
7:
8: int main()
9: {
10:     cout << "Введите число: ";
11:     int number = 0;
12:     cin >> number;
13:
14:     GetSquare(number);
```

```
15:     cout << "Квадрат равен: " << number << endl;
16:
17:     return 0;
18: }
```

Результат

Введите число: 5
Квадрат равен: 25

Анализ

Функция, вычисляющая квадрат числа, находится в строках 3–6. Обратите внимание на то, что она получает возводимое в квадрат число по ссылке и возвращает результат таким же образом. Если бы вы забыли указать параметр `number` как ссылку (с помощью `&`), то результат не мог бы вернуться в вызывающую функцию `main()`, поскольку функция `GetSquare()` выполняла бы расчеты с локальной копией переменной `number`, которая будет уничтожена при выходе из функции. Используя ссылку, вы позволяете функции `GetSquare()` работать в том же адресном пространстве, где определена переменная `number` в функции `main()`. Таким образом, результат будет доступен функции `main()` после того, как функция `GetSquare()` закончит работу.

В этом примере было изменено значение входного параметра, содержащего число, переданное пользователем. При необходимости оба значения, исходное и вычисленный квадрат, могут быть двумя параметрами функции: один из них получает входное значение, а другой возвращает вычисленное выходное значение.

Использование ключевого слова `const` со ссылками

Возможны ситуации, когда ссылка не должна позволять изменять значение исходной переменной. При объявлении таких ссылок используют ключевое слово `const`:

```
int original = 30;
const int& constRef = original;
constRef = 40;                                // Недопустимо: constRef не может
                                              // изменить значение в original
int& ref2 = constRef;                          // Недопустимо: ref2 не const
const int& constRef2 = constRef; // OK
```

Передача аргументов в функции по ссылке

Одно из главных преимуществ ссылок в том, что они позволяют вызываемой функции работать с параметрами без копирования из вызывающей функции, что существенно увеличивает производительность. Но поскольку вызываемая функция работает с параметрами, расположенными непосредственно в стеке вызывающей функции, зачастую важно гарантировать невозможность вызываемой функции изменить значение переменной в вызывающей функции. Ссылки, определенные как `const`,

обеспечивают эту возможность, как показано в листинге 8.19. Константная ссылка не может использоваться как l-значение, поэтому любая попытка присваивания ей значения вызовет ошибку компиляции.

ЛИСТИНГ 8.19. Использование константной ссылки в качестве параметра

```
0: #include <iostream>
1: using namespace std;
2:
3: void GetSquare(const int& number, int& result)
4: {
5:     result = number*number;
6: }
7:
8: int main()
9: {
10:     cout << "Введите число: ";
11:     int number = 0;
12:     cin >> number;
13:
14:     int square = 0;
15:     GetSquare(number, square);
16:     cout << number << "^2 = " << square << endl;
17:
18:     return 0;
19: }
```

Результат

```
Введите число: 27
27^2 = 729
```

Анализ

В отличие от предыдущей программы, в которой передававшая число переменная содержала еще и результат, здесь используются две переменные: одна — для передачи значения, которое будет возведено в квадрат, а вторая — для возврата результата вычисления. Для гарантии неизменности передаваемого числа оно было объявлено как константная ссылка с помощью ключевого слова `const` (см. строку 3). Это автоматически делает параметр `number` *входным параметром*, значение которого не может быть изменено.

В качестве эксперимента можете изменить строку 5 так, чтобы вернуть квадрат, используя ту же логику, что и в листинге 8.18:

```
number *= number;
```

Вы получите ошибку при компиляции, сообщающую, что значение константы не может быть изменено. Таким образом, константные ссылки — мощный инструмент, предоставляемый языком C++ для входных параметров и гарантии того, что

передаваемое по ссылке значение не может быть изменено вызываемой функцией. На первый взгляд, это может показаться тривиальным, но в коллективе разработчиков, где один программист пишет первую версию функции, второй ее дополняет, а третий исправляет или совершенствует, использование константных ссылок имеет важное значение для качества программы.

Резюме

На этом занятии вы узнали об указателях и ссылках. Вы научились применять указатели для доступа к областям памяти и работы с ними, а также узнали, что указатели — это инструмент, помогающий использовать динамическое распределение памяти. Вы изучили операторы `new` и `delete`, применяемые при выделении памяти для элемента и ее освобождении, а также версии `new[]` и `delete[]`, позволяющие выделять память для массива данных. Вы ознакомились с возможными проблемами при использовании указателей и динамического распределения, а также узнали о том, что освобождение динамически распределенной памяти важно для предотвращения ее утечек. Вы узнали, что ссылки — это псевдонимы, являющиеся мощной альтернативой использованию указателей при передаче аргументов функциям, поскольку они гарантируют их корректность. Вы узнали о “константной корректности” при использовании указателей и ссылок и, надеюсь, впредь будете объявлять функции с параметрами с наиболее ограничивающим уровнем константности из всех возможных.

Вопросы и ответы

- **Зачем нужно динамическое выделение памяти, если все необходимое можно сделать со статическими массивами и при этом не нужно заботиться об их освобождении?**

Статические массивы имеют фиксированный размер; они не могут увеличиваться, если приложению понадобится больше памяти, и при этом окажутся слишком расточительными, если приложение нуждается в меньшем количестве элементов. В этом случае лучше всего подходит использование динамического распределения памяти.

- **У меня есть два указателя:**

```
int* pointToAnInt = new int;
int* pCopy = pointToAnInt;
```

- **Полагаю, после использования будет лучше освободить их оба, чтобы гарантированно избежать утечки памяти?**

Нет, это неправильное решение. Оператор `delete` можно использовать только один раз для каждого адреса, возвращенного оператором `new`. Кроме того, желательно избегать наличия двух указателей на один и тот же адрес, поскольку выполнение оператора `delete` для любого из них сделает некорректным другой указатель. Не нужно допускать в программах никаких неопределенностей в отношении корректности используемых указателей.

■ Когда стоит использовать оператор `new(nothrow)`?

Если вы не хотите обрабатывать исключение `std::bad_alloc`, то используйте версию `nothrow` оператора `new`, возвращающую значение `nullptr` при неудачном выделении памяти.

■ Я могу вызвать функцию для вычисления площади, используя два следующих способа:

```
void CalculateArea(const double* const pRadius,
                  double* const pArea);
void CalculateArea(const double& radius, double& area);
```

■ Какой из них мне следует предпочесть?

Лучше использовать последний вариант, с использованием ссылок, так как он вполне корректен при отсутствии указателей. Кроме того, он проще.

■ У меня есть два указателя:

```
int number = 30;
const int* pointToAnInt = &number;
```

■ Я понимаю, что не могу изменить значение `number`, используя указатель `pointToAnInt`, поскольку он объявлен константным. Могу ли я присвоить указатель `pointToAnInt` неконстантному указателю, а затем использовать его для работы со значением, хранящимся в целочисленной переменной `number`?

Нет, вы не можете изменять константность указателя:

```
int* pAnother = pointToAnInt; // Нельзя присвоить указатель на
                             // константу неконстантному указателю
```

■ Зачем передавать значения в функцию по ссылке?

Это не нужно, пока не возникнет необходимость. Если параметрами функции являются объекты очень большого размера, то передача по значению требует весьма дорогостоящего копирования. Вызов функции будет намного эффективнее при использовании ссылок. Не забудьте использовать ключевое слово `const`, если только функция не должна изменять значение соответствующей переменной, например, для возврата результата работы.

■ В чем разница между этими двумя объявлениями?

```
int myNumbers[100];
int* myArrays[100];
```

`myNumbers` — это массив целых чисел, т.е. `myNumbers` — это указатель на область памяти, в которой содержится первое (с индексом 0) из 100 целых чисел. Это статическая альтернатива для следующего кода:

```
int* myNumbers = new int[100]; // Динамически выделенный массив
// Использование myNumbers
delete myNumbers;
```

`myArrays` же является массивом из 100 указателей, каждый из которых способен указывать на целое число или массив целых чисел.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Почему вы не можете присвоить константную ссылку неконстантной?
2. Являются ли `new` и `delete` функциями?
3. Каков характер значения, содержащегося в переменной указателя?
4. Какой оператор используется для доступа к данным, на которые указывает указатель?

Упражнения

1. Что будет выведено при выполнении этих инструкций?

```
0: int number = 3;
1: int* pNum1 = &number;
2: *pNum1 = 20;
3: int* pNum2 = pNum1;
4: number *= 2;
5: cout << *pNum2;
```

2. В чем сходство и различие между следующими тремя перегруженными функциями?

```
int DoSomething(int num1, int num2);
int DoSomething(int& num1, int& num2);
int DoSomething(int* pNum1, int* pNum2);
```

3. Как изменить объявление указателя `pNum1` в строке 1 упражнения 1, чтобы сделать присваивание в строке 3 недопустимым? (Подсказка: это имеет некоторое отношение к гарантии невозможности изменения данных, на которые указывает указатель `pNum1`.)
4. **Отладка.** Что не так с этим кодом?

```
#include <iostream>
using namespace std;
int main()
{
    int *pointToAnInt = new int;
    pointToAnInt = 9;
```



```
    cout << "Значение в pointToAnInt: " << *pointToAnInt;  
    delete pointToAnInt;  
    return 0;  
}
```

5. Отладка. Что не так с этим кодом?

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int pointToAnInt = new int;  
    int* pNumberCopy = pointToAnInt;  
    *pNumberCopy = 30;  
    cout << *pointToAnInt;  
    delete pNumberCopy;  
    delete pointToAnInt;  
    return 0;  
}
```

6. Каким будет вывод приведенной выше программы после исправления ошибок?

Часть II

Объектно-ориентированное программирование на C++

В ЭТОЙ ЧАСТИ...

ЗАНЯТИЕ 9. Классы и объекты

ЗАНЯТИЕ 10. Реализация наследования

ЗАНЯТИЕ 11. Полиморфизм

ЗАНЯТИЕ 12. Типы операторов и их перегрузка

ЗАНЯТИЕ 13. Операторы приведения

ЗАНЯТИЕ 14. Введение в макросы и шаблоны

ЗАНЯТИЕ 9

Классы и объекты

До сих пор мы имели дело со структурой простой программы, которая начинает выполнение с функции `main()`, содержит локальные и глобальные переменные и константы и выполняет логику, сосредоточенную в функциях, которые могут получать параметры и возвращать значения. Наш стиль программирования до этого был *процедурным*, и пока что ни с чем объектно-ориентированным мы не сталкивались. Но наконец пришло время узнать об объектно-ориентированном программировании на C++.

На этом занятии...

- Что такое классы и объекты
- Как классы позволяют объединить данные с функциями, работающими с ними
- Конструкторы, копирующие конструкторы и деструкторы
- Что такое перемещающий конструктор
- Объектно-ориентированные концепции инкапсуляции и абстракции
- Что такое указатель `this`
- Что такое `struct` и в чем различие между `struct` и `class`

Концепция классов и объектов

Предположим, вы пишете программу, моделирующую такого человека, как вы сами. У человека должны быть индивидуальные данные: имя, дата рождения, место рождения и пол — информация, которая делает человека уникальным. Человек может выполнять определенные действия, например говорить и представляться другим людям. Таким образом, человека можно моделировать так, как показано на рис. 9.1.

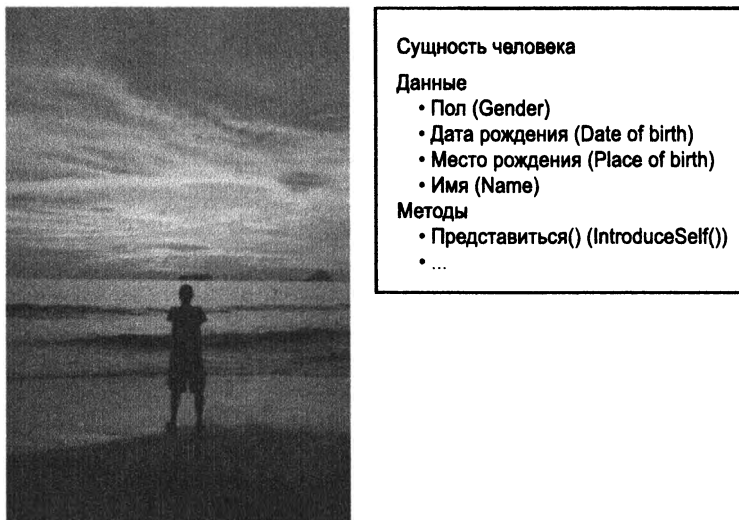


РИС. 9.1. Общее представление человека

Чтобы смоделировать человека в программе, нужна конструкция, позволяющая группировать атрибуты, определяющие человека (данные), и действия, которые он может выполнять (функции), используя доступные атрибуты. Эта конструкция называется *классом* (class).

Объявление класса

При объявлении класса используется ключевое слово `class`, за которым следуют имя класса и блок `{ ... }`, который заключает в фигурные скобки набор атрибутов-членов и функций-членов и который завершается точкой с запятой.

Объявление класса говорит компилятору о классе и его свойствах. Само по себе объявление класса никак не влияет на выполнение программы, поскольку класс следует использовать, как и функцию следует вызывать.

Моделирующий человека класс выглядит следующим образом (пока что игнорируйте непонятные места в синтаксисе):

```
class Human
{
    // Атрибуты данных:
    string name;
```

```
string dateOfBirth;  
string placeOfBirth;  
string gender;  
  
// Методы:  
void Talk(string textToTalk);  
void IntroduceSelf();  
...  
};
```

Необходимо сказать, что функция `IntroduceSelf()` использует `Talk()` и некоторые из атрибутов данных, которые сгруппированы в конструкции `class Human`. Таким образом, ключевое слово `class` языка C++ предоставляет мощный способ создания собственного типа данных, который позволяет *инкапсулировать* атрибуты и функции для работы с ними. Все атрибуты класса, в данном случае `name`, `dateOfBirth`, `placeOfBirth` и `gender`, а также все объявленные в нем функции, `Talk()` и `IntroduceSelf()`, называются *членами класса* (members of class) `Human`.

Инкапсуляция (encapsulation) представляет собой способ логического группирования данных и функций, которые их используют; она является важнейшим свойством объектно-ориентированного программирования.

ПРИМЕЧАНИЕ

Вы можете столкнуться с названием *метод* (method); это, по существу, функции, являющиеся членами класса. В данной книге этот термин используется наряду с термином *функция-член* класса.

Объект как экземпляр класса

Класс похож на чертеж, и одно лишь его объявление не оказывает никакого влияния на выполнение программы. Реальный экземпляр класса, который можно использовать во время выполнения программы, — это *объект* (object). Для использования возможностей класса обычно создается экземпляр этого класса, именуемый объектом и позволяющий получить доступ к его методам и атрибутам.

Создание объекта `class Human` подобно созданию экземпляра любого другого типа, скажем, `double`:

```
double pi = 3.1415; // Переменная типа double  
Human firstMan;    // firstMan — объект класса Human
```

В качестве альтернативы можно создать экземпляр класса `Human` динамически, используя оператор `new`, как и для любого другого типа, например для типа `int`:

```
int* pointsToNum = new int;    // Динамическое создание int  
delete pointsToNum;           // Освобождение памяти  
  
Human* firstWoman = new Human(); // Динамическое создание Human  
delete firstWoman;            // Освобождение памяти
```

Доступ к членам класса с использованием оператора точки (.)

Рассмотрим человека по имени Адам, мужчину, родившегося в 1970 году в Алабаме. Экземпляр `firstMan` — это объект класса `Human`, т.е. экземпляр класса, который существует в действительности и может использоваться во время выполнения приложения:

```
Human firstMan; // Экземпляр класса (объект) Human
```

Как свидетельствует объявление класса, у объекта `firstMan` есть атрибуты, такие как `dateOfBirth`, к которым можно обратиться, используя *оператор выбора поля*, или *оператор точки (dot operator) (.)*:

```
firstMan.dateOfBirth = "1970";
```

Дело в том, что атрибут `dateOfBirth` принадлежит классу `Human` и является его элементом, как видно из объявления класса. Этот атрибут существует в действительности, т.е. во время выполнения приложения — только когда создан объект. Оператор точки позволяет обратиться к атрибутам объекта, а также к его методам, таким как `IntroduceSelf()`:

```
firstMan.IntroduceSelf();
```

Если у вас есть указатель `firstWoman` на экземпляр класса `Human`, то для доступа к его членам можно использовать как описанный в следующем разделе оператор указателя (`->`), так и оператор разыменования с последующим оператором точки:

```
Human* firstWoman = new Human();  
(*firstWoman).IntroduceSelf();
```

ПРИМЕЧАНИЕ

Здесь продолжают действовать наши соглашения об именовании. Имя класса и функции-члены объявляются в стиле Pascal, например `IntroduceSelf()`. Атрибуты членов класса используют “верблюжий стиль”, например `dateOfBirth`.

Создавая объект класса, мы объявляем переменную с типом класса. Поэтому мы используем тот же “верблюжий стиль”, который использовали ранее для имен переменных, например `firstMan`.

Обращение к членам класса с использованием оператора указателя (->)

Если объект был создан в динамической памяти с использованием оператора `new` или если у вас есть указатель на готовый объект, то для доступа к его атрибутам и функциям можно использовать *оператор указателя (pointer operator) (->)*:

```
Human* firstWoman = new Human();  
firstWoman->dateOfBirth = "1970";  
firstWoman->IntroduceSelf();  
delete firstWoman;
```

Готовая для компиляции форма класса `Human` с новым ключевым словом `public` представлена в листинге 9.1.

ЛИСТИНГ 9.1. Готовый для компиляции класс `Human`

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6:     public:
7:         string name;
8:         int age;
9:
10:        void IntroduceSelf()
11:        {
12:            cout << "Я " + name << " и мне ";
13:            cout << age << " лет" << endl;
14:        }
15: };
16:
17: int main()
18: {
19:     // Объект Human с именем "Adam"
20:     Human firstMan;
21:     firstMan.name = "Adam";
22:     firstMan.age = 30;
23:
24:     // Объект Human с именем "Eve"
25:     Human firstWoman;
26:     firstWoman.name = "Eve";
27:     firstWoman.age = 28;
28:
29:     firstMan.IntroduceSelf();
30:     firstWoman.IntroduceSelf();
31: }
```

Результат

```
Я Adam и мне 30 лет
Я Eve и мне 28 лет
```

Анализ

В строках 4–15 продемонстрирован очень простой класс C++ `Human`. Рассмотрим структуру класса `Human` и его использование в функции `main()`.

Этот класс содержит две переменные-члена: одна — типа `string` по имени `name` в строке 7, другая — типа `int` по имени `age` в строке 8, а также функцию (именуемую

также *методом*) `IntroduceSelf()` в строках 10–14. В строках 20 и 25 в функции `main()` создаются два объекта класса `Human` с именами `firstMan` и `firstWoman` соответственно. Следующие строки устанавливают значения переменных-членов объектов `firstMan` и `firstWoman`. Обратите внимание, как в строках 29 и 30 вызывается метод `IntroduceSelf()` для этих двух объектов, что позволяет вывести две разные строки. Программа демонстрирует, что объекты `firstMan` и `firstWoman` являются различными представителями абстрактного типа, определенного классом `Human`.

Вы обратили внимание на ключевое слово `public` в листинге 9.1? Пришло время изучить средства, предоставляемые языком C++, для защиты атрибутов, которые ваш класс должен оставить скрытыми от тех, кто их использует.

Ключевые слова `public` и `private`

Информацию можно поделить как минимум на две категории: ту, которую мы оставляем *открытой* (`public`) для всех, и ту, которую никто знать не должен, т.е. *закрытую* (`private`). Например, утаивать пол или имя обычно не имеет смысла, в то время как мало кто захочет открывать свои доходы.

Язык C++ позволяет объявлять атрибуты и методы класса открытыми или закрытыми. Открытые члены класса могут использоваться кем угодно, обладающим объектом класса. Закрытые члены класса могут использоваться только внутри класса (или “друзьями” класса). Ключевые слова C++ `public` (открытый) и `private` (закрытый) позволяют разработчику класса решать, какая часть класса может быть доступна извне, например из функции `main()`, а какая нет.

Какие преимущества дает возможность отмечать атрибуты и методы как закрытые? Рассмотрим объявление класса `Human`, игнорируя все, кроме переменной-члена `age`:

```
class Human
{
    private:
        // Закрытые данные-члены:
        int age;
        string name;

    public:
        int GetAge()
        {
            return age;
        }
        void SetAge(int humanAge)
        {
            age = humanAge;
        }
        // ...Другие члены и объявления
};
```

Предположим, экземпляр класса `Human` называется `eve`:

```
Human eve;
```

Пользователь этого экземпляра, попытавшись получить доступ к возрасту Евы следующим образом:

```
cout << eve.age; // Ошибка компиляции
```

получит при компиляции ошибку примерно со следующим сообщением: “Error: Human::age — cannot access private member declared in class Human” (Ошибка: Human::age — нельзя обратиться к закрытому члену, объявленному в классе Human). Единственный допустимый способ доступа к атрибуту age подразумевает обращение к открытому методу GetAge(), предоставляемому классом Human и реализованному разработчиком как корректный способ предоставления возраста:

```
cout << Eve.GetAge(); // ОК
```

Если разработчик класса Human пожелает, он может реализовать метод GetAge() так, чтобы Ева представлялась моложе, чем она есть на самом деле! Другими словами, это значит, что язык C++ позволяет разработчику класса контролировать, какие атрибуты предоставлять и каким образом. Если бы класс Human не реализовал открытый метод GetAge(), то он фактически гарантировал бы невозможность обращения пользователя к атрибуту age вообще. Это средство может быть полезным в ситуациях, которые рассматриваются далее на этом занятии.

Аналогично значение атрибуту Human::age также не может быть присвоено непосредственно:

```
eve.age = 22; // Ошибка компиляции
```

Единственным допустимым способом установки возраста является метод SetAge():

```
eve.SetAge(22); // ОК
```

У этого подхода много преимуществ. Текущая реализация метода SetAge() не делает ничего, кроме собственно присваивания значения переменной-члена Human::age. Однако вы можете использовать метод SetAge() для проверки устанавливаемого возраста, не является ли он, например, нулевым или отрицательным:

```
class Human
{
private:
    int age;

public:
    void SetAge(int humanAge)
    {
        if (humanAge > 0)
            age = humanAge;
    }
};
```

Таким образом, язык C++ позволяет разработчику класса управлять возможностью обращения к атрибутам данных класса и работы с ними.

Абстракция данных с помощью ключевого слова `private`

Позволяя разрабатывать класс как контейнер, инкапсулирующий данные и работающие с ними методы, язык C++ разрешает вам с помощью ключевого слова `private` решить, какая информация недоступна внешнему миру (т.е. недоступна вне класса). В то же время с помощью методов, объявленных как `public`, у вас есть возможность предоставить контролируемый доступ даже к информации, объявленной закрытой. Таким образом, реализация вашего класса может абстрагировать информацию о членах, к которой внешний мир (другие классы и функции, такие как `main()`) не должны иметь доступа.

Возвращаясь к примеру с переменной-членом `Human::age`, являющейся закрытым членом, вы знаете, что в действительности многим людям не нравится разглашать свой истинный возраст. Если бы классу `Human` понадобилось сообщить значение возраста, которое на два меньше, чем на самом деле, то это легко позволила бы сделать открытая функция `GetAge()`, использующая параметр `Human::age`, но уменьшающая его на два и возвращающая соответствующий результат, как показано в листинге 9.2.

ЛИСТИНГ 9.2. Модель класса `Human`, в которой сообщается меньший возраст

```
0: #include <iostream>
1: using namespace std;
2:
3: class Human
4: {
5:     private:
6:         // Закрытые данные-члены:
7:         int age;
8:
9:     public:
10:        void SetAge(int inputAge)
11:        {
12:            age = inputAge;
13:        }
14:
15:        // Человек лжет о своем возрасте (если ему за 30)
16:        int GetAge()
17:        {
18:            if (age > 30)
19:                return (age - 2);
20:            else
21:                return age;
22:        }
23: };
24:
25: int main()
26: {
27:     Human firstMan;
```

```
28:     firstMan.SetAge(35);
29:
30:     Human firstWoman;
31:     firstWoman.SetAge(22);
32:
33:     cout << "Возраст firstMan " << firstMan.GetAge() << endl;
34:     cout << "Возраст firstWoman " << firstWoman.GetAge() << endl;
35:
36:     return 0;
37: }
```

Результат

```
Возраст firstMan 33
Возраст firstWoman 22
```

Анализ

Обратите внимание на открытый метод `Human::GetAge()` в строке 16. Поскольку фактический возраст, содержащийся в закрытой целочисленной переменной-члене `Human::age`, недоступен непосредственно, единственным способом для внешнего пользователя объекта класса `Human` обратиться к его атрибуту `age` является вызов метода `GetAge()`. Таким образом, фактический возраст, содержащийся в переменной-члене `Human::age`, абстрагируется от внешнего мира. Фактически наш класс `Human` скрывает свой истинный возраст, и метод `GetAge()` в строках 16–22 возвращает уменьшенное значение возраста, если истинное значение больше 30.

Абстракция (abstraction) представляет собой очень важную концепцию объектно-ориентированных языков. Она позволяет программистам решать, какие атрибуты класса должны оставаться известными только самому классу и его членам, но никому вовне (за исключением его “друзей”).

Конструкторы

Конструктор (constructor) — это специальная функция (или метод), вызываемая во время инстанцирования класса для создания объекта. Так же, как и функции, конструкторы могут быть перегружены.

Объявление и реализация конструктора

Конструктор — это специальная функция, имя которой совпадает с именем класса и не имеет возвращаемого значения. Так, у класса `Human` есть конструктор, который объявляется следующим образом:

```
class Human
{
    public:
        Human(); // Объявление конструктора
};
```

Конструктор может быть реализован как в объявлении класса, так и вне объявления класса. Реализация (определение) в классе выглядит следующим образом:

```
class Human
{
public:
    Human()
    {
        // Код конструктора
    }
};
```

Вариант определения конструктора вне объявления класса выглядит следующим образом:

```
class Human
{
public:
    Human(); // Объявление конструктора
};

// Определение конструктора (реализация)
Human::Human()
{
    // Код конструктора
}
```

ПРИМЕЧАНИЕ

Оператор `::` называется *оператором разрешения области видимости* (scope resolution operator). Например, синтаксис `Human::dateOfBirth` относится к переменной `dateOfBirth`, объявленной в пределах класса `Human`. Синтаксис `::dateOfBirth` же относится к другой переменной `dateOfBirth`, объявленной в глобальной области видимости.

Когда и как использовать конструкторы

Конструктор всегда вызывается при создании объекта, когда строится экземпляр класса. Это делает конструктор наилучшим местом для инициализации исходными значениями переменных-членов класса, таких как целые числа, указатели и т.д. Вернемся к листингу 9.2. Обратите внимание: если бы вы забыли вызвать метод `SetAge()`, то целочисленная переменная `Human::age` содержала бы неизвестное случайное значение, поскольку эта переменная не была инициализирована (посмотрите, что получится, если закомментировать строки 28 и 31). В листинге 9.3 приведена улучшенная версия класса `Human`, в которой переменная `age` инициализируется в конструкторе.

ЛИСТИНГ 9.3. Использование конструктора
для инициализации переменных-членов класса

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6:     private:
7:         string name;
8:         int age;
9:
10:    public:
11:        Human() // Конструктор
12:        {
13:            age = 0; // Инициализация
14:            cout << "Конструирование объекта Human" << endl;
15:        }
16:
17:        void SetName (string humansName)
18:        {
19:            name = humansName;
20:        }
21:
22:        void SetAge(int humansAge)
23:        {
24:            age = humansAge;
25:        }
26:
27:        void IntroduceSelf()
28:        {
29:            cout << "Я " + name << " и мне ";
30:            cout << age << " лет" << endl;
31:        }
32: };
33:
34: int main()
35: {
36:     Human firstWoman;
37:     firstWoman.SetName("Ева");
38:     firstWoman.SetAge(28);
39:
40:     firstWoman.IntroduceSelf();
41: }
```

Результат

Конструирование объекта Human
Я Ева и мне 28 лет

Анализ

В выводе вы видите добавление строки, которая указывает на вызов конструктора. Теперь обратите внимание на функцию `main()`, определенную в строках 34–41. Вы видите, что первая строка вывода является результатом создания объекта `first Woman` в строке 36. Конструктор `Human::Human()` в строках 11–15 содержит инструкцию с выводом в поток `cout`. Кроме того, конструктор инициализирует нулем целочисленную переменную `age`. Если вы забудете установить возраст недавно созданного объекта, то можете быть уверены, что конструктор обеспечит ему не произвольное целочисленное значение (которое могло бы выглядеть правильным), а значение “нуль”.

ПРИМЕЧАНИЕ

Конструктор, который может быть вызван без аргумента, называется *конструктором по умолчанию* (default constructor). Собственноручно создавать конструктор по умолчанию не обязательно.

Если вы не создали ни одного конструктора сами, как в листинге 9.1, то компилятор предоставит его автоматически (он конструирует переменные-члены класса, но не инициализирует никакими значениями простые старые типы данных, такие как `int`).

Перегрузка конструкторов

Конструкторы могут быть перегружены, как и функции. Таким образом, вполне может иметься конструктор, позволяющий создать объект класса `Human` с именем в качестве параметра, например:

```
class Human
{
public:
    Human()
    {
        // Код конструктора по умолчанию
    }

    Human(string humansName)
    {
        // Код перегруженного конструктора
    }
};
```

Листинг 9.4 демонстрирует применение перегруженных конструкторов при создании объекта класса `Human` с предоставленным именем.

ЛИСТИНГ 9.4. Класс Human с несколькими конструкторами

```

0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6:     private:
7:         string name;
8:         int age;
9:
10:    public:
11:        Human() // Конструктор по умолчанию
12:        {
13:            age = 0; // Инициализация
14:            cout << "Конструктор по умолчанию" << endl;
15:        }
16:
17:        Human(string humansName, int humansAge) // Перегруженный
18:        {
19:            name = humansName;
20:            age = humansAge;
21:            cout << "Перегруженный конструктор создал: ";
22:            cout << name << " с возрастом " << age << endl;
23:        }
24: };
25:
26: int main()
27: {
28:     Human firstMan; // Конструктор по умолчанию
29:     Human firstWoman("Ева", 20); // Перегруженный конструктор
30: }

```

Результат

Конструктор по умолчанию

Перегруженный конструктор создал: Ева с возрастом 20

Анализ

Минималистичная функция `main()` в строках 26–30 создает два экземпляра класса `Human`. `firstMan` использует конструктор по умолчанию, а `firstWoman` — перегруженный конструктор, получающий имя и возраст для создания объекта. Весь вывод данного приложения является результатом конструирования объектов! Если бы в классе `Human` не был написан конструктор по умолчанию, то функция `main()` не могла бы создавать объекты иначе как с указанием для перегруженного конструктора имени и возраста, что делало бы невозможным создание объекта `Human` без указания имени или возраста.

СОВЕТ

Вы можете решить не реализовывать конструктор по умолчанию, чтобы гарантировать создание экземпляров объектов с определенным минимальным набором параметров, о чем говорится в следующем разделе.

Класс без конструктора по умолчанию

В листинге 9.5 показан класс `Human` без конструктора по умолчанию, что заставляет создавать его объекты только с предоставлением имени и возраста.

ЛИСТИНГ 9.5. Класс с перегруженным конструктором, но без конструктора по умолчанию

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6:     private:
7:         string name;
8:         int age;
9:
10:    public:
11:        Human(string humansName, int humansAge)
12:        {
13:            name = humansName;
14:            age = humansAge;
15:            cout << "Перегруженный конструктор создал:";
16:            cout << name << " с возрастом " << age << endl;
17:        }
18:
19:        void IntroduceSelf()
20:        {
21:            cout << "Я " + name << " и мне ";
22:            cout << age << " лет" << endl;
23:        }
24: };
25:
26: int main()
27: {
28:     Human firstMan("Адам", 25);
29:     Human firstWoman("Ева", 28);
30:
31:     firstMan.IntroduceSelf();
32:     firstWoman.IntroduceSelf();
33: }
```

Результат

```

Перегруженный конструктор создал: Адам с возрастом 25
Перегруженный конструктор создал: Ева с возрастом 28
Я Адам и мне 25 лет
Я Ева и мне 28 лет

```

Анализ

Эта версия класса `Human` имеет только один конструктор, получающий входные параметры типа `string` и `int`, как видно из строки 11. Никакого конструктора по умолчанию нет, а при наличии перегруженного конструктора компилятор C++ не генерирует конструктор по умолчанию автоматически. Этот пример демонстрирует также возможность создания объектов класса `Human` только с определенными параметрами, в данном случае — `name` и `age`, и невозможность их изменения после создания. Дело в том, что класс `Human` хранит атрибут имени в закрытой переменной типа `string` по имени `name`. К `Human::name` нельзя обратиться или модифицировать ее из функции `main()` или любого другого кода, который не является членом класса `Human`. Другими словами, перегруженный конструктор вынуждает пользователя класса `Human` указать имя (и возраст) для каждого создаваемого объекта и не позволяет впоследствии менять это имя (весьма неплохая модель реального положения вещей, не так ли?). Вы получили имя при рождении; людям разрешено знать его, но ни у кого, кроме вас, нет права его изменить.

Параметры конструктора со значениями по умолчанию

Как и функции, конструкторы способны иметь параметры со значениями по умолчанию. В следующем коде приведена немного модифицированная версия конструктора из строки 11 листинга 9.5, в которой параметр `age` имеет значение по умолчанию 25:

```

class Human
{
    private:
        string name;
        int age;

    public:
        // Перегруженный конструктор (конструктора по умолчанию нет)
        Human(string humansName, int humansAge = 25)
        {
            name = humansName;
            age = humansAge;
            cout << "Перегруженный конструктор создал:";
            cout << name << " с возрастом " << age << endl;
        }
        // ... Другие члены
};

```

Объект такого класса может быть создан следующим образом:

```
Human Adam("Адам"); // Adam.age имеет значение по умолчанию 25
Human Eve("Ева", 18); // Eve.age имеет указанное значение 18
```

ПРИМЕЧАНИЕ

Конструктор по умолчанию — это конструктор, который позволяет создавать экземпляры без аргументов, причем это не обязательно конструктор, который не получает параметров. Ниже приведен конструктор с двумя параметрами, оба они имеют значения по умолчанию, поэтому данный конструктор является конструктором по умолчанию.

```
class Human
{
    private:
        string name;
        int age;

    public:
        // Значения по умолчанию для обоих параметров
        Human(string humansName = "Adam", int humansAge = 25)
        {
            name = humansName;
            age = humansAge;
            cout << "Перегруженный конструктор создал:";
            cout << name << " с возрастом " << age << endl;
        }
};
```

Дело в том, что экземпляр класса Human вполне может быть создан без аргументов:

```
Human Adam; // Human со значениями по умолчанию:
// name = "Adam", age = 25
```

Конструкторы со списками инициализации

Вы уже видели, насколько полезны конструкторы при инициализации переменных-членов. Другой способ инициализации членов — использование *списков инициализации* (initialization list). Ниже показан вариант конструктора из листинга 9.5, получающего два параметра, но использующего списки инициализации:

```
class Human
{
    private:
        string name;
        int age;

    public:
        // Конструктор получает два параметра для
        // инициализации членов age и name
```

```

Human(string humansName, int humansAge)
    :name(humansName), age(humansAge)
{
    cout << "Перегруженный конструктор создал:";
    cout << name << " с возрастом " << age << endl;
}
// ... другие члены класса
};

```

Таким образом, список инициализации характеризуется двоеточием (:) после объявления параметров в круглых скобках (...), за которым идут отдельные переменные-члены и значения для их инициализации. Инициализирующее значение может быть параметром, таким как `humansName`, или даже фиксированным значением. Списки инициализации могут также пригодиться при вызове конструкторов базового класса с определенными аргументами. Этот вопрос рассматривается на занятии 10, “Реализация наследования”.

В листинге 9.6 представлена версия класса `Human` с конструктором по умолчанию с параметрами, заданными по умолчанию, и списком инициализации.

ЛИСТИНГ 9.6. Конструктор со значениями по умолчанию и списком инициализации

```

0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6:     private:
7:         int age;
8:         string name;
9:
10:    public:
11:        Human(string humansName = "Adam", int humansAge = 25)
12:            :name(humansName), age(humansAge)
13:        {
14:            cout << "Перегруженный конструктор создал:";
15:            cout << name << " с возрастом " << age << endl;
16:        }
17: };
18:
19: int main()
20: {
21:     Human firstMan;
22:     Human firstWoman("Eve", 18);
23:
24:     return 0;
25: }

```

Результат

Перегруженный конструктор создал: Adam с возрастом 25

Перегруженный конструктор создал: Eve с возрастом 18

Анализ

Конструктор со списками инициализации находится в строках 11–16, в которых можно также видеть, что параметры имеют значения по умолчанию "Adam" для переменной-члена name и 25 — для переменной-члена age. Следовательно, когда в строке 21 создается экземпляр класса Human с именем firstMan, его членам автоматически присваиваются значения по умолчанию. Экземпляр firstWoman, напротив, имеет явно указанные значения для переменных name и age, как показано в строке 22.

ПРИМЕЧАНИЕ

Можно определить конструктор как константное выражение, используя ключевое слово constexpr. В особых случаях, когда такая конструкция полезна с точки зрения производительности, ее можно использовать в объявлении конструктора.

```
class Sample
{
    const char* someString;
public:
    constexpr Sample(const char* input)
        :someString(input)
    {
        // Код конструктора
    }
};
```

Деструктор

Деструкторы (destructor), как и конструкторы, также являются специальными функциями. В отличие от конструкторов, деструкторы автоматически вызываются при удалении объекта.

Объявление и реализация деструктора

Имя деструктора, подобно конструктору, совпадает с именем класса, но предваряется тильдой (~). Так, у класса Human может быть деструктор, объявленный следующим образом:

```
class Human
{
    ~Human(); // Объявление деструктора
};
```

Этот деструктор может быть реализован в объявлении класса или вне его. Реализация или определение в классе выглядит следующим образом:

```
class Human
{
    public:
        ~Human()
        {
            // Код деструктора
        }
};
```

Определение деструктора вне объявления класса выглядит следующим образом:

```
class Human
{
    public:
        ~Human(); // Объявление деструктора
};

// Определение деструктора (реализация)
Human::~Human()
{
    // Код деструктора
}
```

Как видите, объявление деструктора немного отличается от такового у конструктора, — оно содержит тильду (~). Однако роль деструктора прямо противоположна роли конструктора.

Когда и как использовать деструкторы

Деструкторы всегда вызываются при выходе объекта класса из области видимости или при их удалении оператором `delete`. Это делает деструкторы идеальным местом для сброса переменных и освобождения выделенной динамической памяти и других ресурсов.

В этой книге регулярно рекомендуется применять строки класса `std::string`, а не символьные буфера `char*` в стиле C, где распределением, управлением и освобождением памяти вам придется заниматься самостоятельно. Класс `std::string` и другие подобные классы обладают не только конструкторами и деструкторами, но и массой полезных функций (в дополнение к тем, которые вы изучите на занятии 12, “Типы операторов и их перегрузка”), которые берут на себя заботы по выделению и освобождению памяти. Проанализируем пример класса `MyString`, представленный в листинге 9.7, который выделяет память для строки в конструкторе и освобождает ее в деструкторе.

ЛИСТИНГ 9.7. Пример класса, инкапсулирующего буфер в стиле C

```
0: #include <iostream>
1: #include <string.h>
2: using namespace std;
3: class MyString
4: {
5:     private:
6:         char* buffer;
7:
8:     public:
9:         MyString(const char* initString) // Конструктор
10:        {
11:            if(initString != nullptr)
12:            {
13:                buffer = new char[strlen(initString) + 1];
14:                strcpy(buffer, initString);
15:            }
16:            else
17:                buffer = nullptr;
18:        }
19:
20:        ~MyString()
21:        {
22:            cout << "Вызов деструктора" << endl;
23:            if (buffer != nullptr)
24:                delete[] buffer;
25:        }
26:
27:        int GetLength()
28:        {
29:            return strlen(buffer);
30:        }
31:
32:        const char* GetString()
33:        {
34:            return buffer;
35:        }
36: };
37:
38: int main()
39: {
40:     MyString sayHello("Hello from String Class");
41:     cout << "Содержимое буфера длиной ";
42:     cout << sayHello.GetLength() << " символа." << endl;
43:
44:     cout << "Буфер содержит: " << sayHello.GetString() << endl;
45: }
```

Результат

```
Содержимое буфера длиной 23 символа.
Буфер содержит: Hello from String Class
Вызов деструктора
```

Анализ

Этот класс инкапсулирует строку в стиле С в переменной `MyString::buffer` и освобождает вас от задач выделения и освобождения памяти, необходимых при использовании такой строки в стиле С. Особенно интересен код конструктора `MyString()` в строках 9–18 и деструктора `~MyString()` в строках 20–25. Конструктор требует передачи инициализирующей строки, поскольку у него есть обязательный входной параметр, а затем копирует ее в строку `buffer` в стиле С после выделения памяти для нее в строке 13, где используется функция `strlen` для определения длины входной строки. Для копирования в эту вновь выделенную область памяти в строке 14 используется функция `strcpy`. На случай, если пользователь класса в качестве аргумента для параметра `initString` предоставит значение `nullptr`, переменная `MyString::buffer` также будет инициализирована значением `nullptr` (чтобы указатель не содержал случайное значение, которое может оказаться опасным при попытке использовать его для доступа к области памяти). Код деструктора гарантирует автоматическое освобождение памяти, выделенной в конструкторе, и возвращение ее операционной системе. Он проверяет, не содержит ли переменная `MyString::buffer` значение `nullptr`, и если это не так, то выполняет для нее оператор `delete[]`, соответствующий оператору `new[]` в конструкторе¹. Обратите внимание, что в функции `main()` нигде не используются операторы `new` или `delete`. Этот класс не только абстрагировал реализацию, но и гарантировал при этом техническую правильность освобождения выделенной памяти. Деструктор `~MyString()` вызывается автоматически при выходе из функции `main()`, что и демонстрирует вывод программы (выполняемый в деструкторе).

Классы, облегчающие работу со строками, являются одним из многих примеров использования деструкторов. На занятии 26, “Понятие интеллектуальных указателей”, вы узнаете, что деструкторы играют критически важную роль в работе с интеллектуальными указателями.

ПРИМЕЧАНИЕ

Деструкторы не могут быть перегружены. У класса может быть только один деструктор. Если вы забудете реализовать деструктор, компилятор создаст и вызовет фиктивный деструктор, т.е. пустой деструктор, который не осуществляет никакого освобождения выделенной динамической памяти.

¹ Заметим, что данная проверка излишня, так как согласно стандарту оператор `delete` корректно обрабатывает нулевые указатели. — *Примеч. ред.*

Копирующий конструктор

На занятии 7, “Организация кода с помощью функций”, вы узнали, что переданные функции наподобие `Area()` аргументы копируются (см. листинг 7.1):

```
double Area(double radius);
```

Так, аргумент, переданный в виде параметра `radius`, копируется, когда вызывается функция `Area()`. Это правило относится и к объектам (экземплярам классов).

Поверхностное копирование и связанные с ним проблемы

Такие классы, как `MyString`, представленный в листинге 9.7, содержат в качестве члена указатель, который указывает на область в динамически распределяемой памяти, выделенную в конструкторе с помощью оператора `new[]` и освобождаемую в деструкторе с использованием оператора `delete[]`. Когда объект такого класса копируется, копируется и указатель, являющийся его членом, но не копируется буфер, на который он указывает. В результате два объекта указывают на один и тот же буфер, находящийся в динамически распределяемой памяти. Такое копирование называется *поверхностным копированием* (shallow copy) и представляет угрозу стабильности программы, как показано в листинге 9.8.

ЛИСТИНГ 9.8. Проблема передачи объекта класса по значению

```

0: #include <iostream>
1: #include <string.h>
2: using namespace std;
3: class MyString
4: {
5:     private:
6:         char* buffer;
7:
8:     public:
9:         MyString(const char* initString) // Конструктор
10:        {
11:            buffer = nullptr;
12:            if(initString != nullptr)
13:            {
14:                buffer = new char[strlen(initString) + 1];
15:                strcpy(buffer, initString);
16:            }
17:        }
18:
19:        ~MyString() // Деструктор
20:        {
21:            cout << "Вызов деструктора" << endl;
22:            delete[] buffer;
23:        }
24:

```

```
25:     int GetLength()  
26:     { return strlen(buffer); }  
27:  
28:     const char* GetString()  
29:     { return buffer; }  
30: };  
31:  
32: void UseMyString(MyString str)  
33: {  
34:     cout << "Содержимое буфера длиной " << str.GetLength();  
35:     cout << " символа" << endl;  
36:  
37:     cout << "Буфер содержит: " << str.GetString() << endl;  
38:     return;  
39: }  
40:  
41: int main()  
42: {  
43:     MyString sayHello("Hello from String Class");  
44:     UseMyString(sayHello);  
45:  
46:     return 0;  
47: }
```

Результат

Содержимое буфера длиной 23 символа
Буфер содержит: Hello from String Class
Вызов деструктора
Вызов деструктора
<СВОЙ, КАК ПОКАЗАНО НА РИС. 9.2>

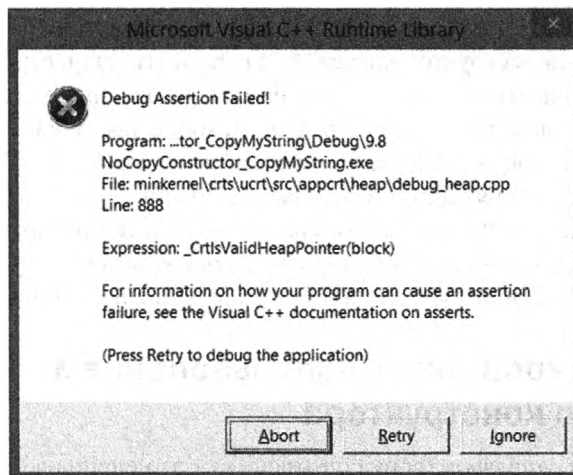


РИС. 9.2. Копия экрана аварийного завершения, происшедшего при выполнении кода листинга 9.8 (режим отладки Visual Studio)

Анализ

Почему класс `MyString`, который только что прекрасно работал в листинге 9.6, привел к отказу в листинге 9.7? Единственное различие между листингами 9.6 и 9.7 в том, что использование объекта `sayHello` класса `MyString`, созданного в функции `main()`, было делегировано функции `UseMyString()`, вызываемой в строке 44. Делегирование выполнения действий с объектом этой функции привело к тому, что объект `sayHello` в функции `main()` копируется в аргумент `str`, используемый в функции `UseMyString()`. Эта копия создается компилятором, поскольку функция была объявлена как получающая параметр `str` по значению, а не по ссылке. Компилятор создает простую бинарную копию данных, таких как целые числа, символы и указатели. Таким образом, значение, содержащееся в указателе `sayHello.buffer`, будет просто скопировано в параметр `str`, т.е. теперь `sayHello.buffer` указывает на ту же область памяти, что и `str.buffer` (рис. 9.3).

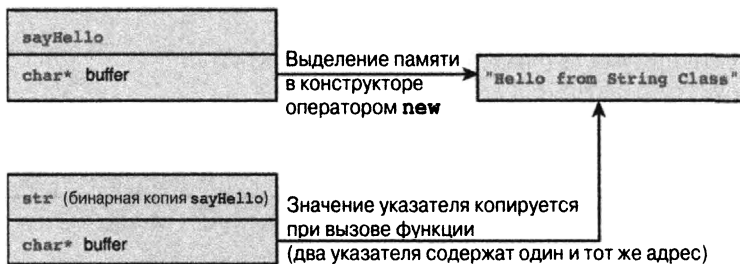


РИС. 9.3. Поверхностное копирование объекта `sayHello` в параметр `str` при вызове функции `UseMyString()`

Бинарная копия не обеспечивает *глубокое копирование* (deep copy) и не распространяется на указываемую область памяти, поэтому теперь есть два объекта класса `MyString`, которые указывают на одну и ту же область памяти. По завершении работы функции `UseMyString()` переменная `str` выходит из области видимости и удаляется. При этом вызывается деструктор класса `MyString` и его код в строке 22 листинга 9.8 освобождает с помощью оператора `delete` память, выделенную для буфера. Обратите внимание, что этот вызов оператора `delete`, освобождая, делает недействительной область памяти, на которую есть указатель в объекте `sayHello`, находящемся в функции `main()`. Когда функция `main()` завершается, объект `sayHello` выходит из области видимости и удаляется. Однако на этот раз строка 22 повторно вызывает оператор `delete` для адреса области памяти, который уже освобожден при удалении параметра `str`. Результатом повторного удаления и будет аварийный отказ приложения.

Глубокое копирование с использованием копирующего конструктора

Копирующий конструктор (copy constructor) — это специальный перегруженный конструктор, предоставляемый разработчиком класса. Компилятор использует копирующий конструктор каждый раз, когда копируется объект класса.

Копирующий конструктор для класса `MyString` можно объявить так:

```
class MyString
{
    MyString(const MyString& CopySource); // Копирующий конструктор
};

MyString::MyString(const MyString& CopySource)
{
    // Код реализации копирующего конструктора
}
```

Таким образом, копирующий конструктор получает в качестве параметра по ссылке объект того же класса. Этот параметр — псевдоним исходного объекта, используемый при написании собственного специального кода копирования. Вы можете использовать копирующий конструктор для гарантии глубокого копирования всех буферов оригинала, как показано в листинге 9.9.

ЛИСТИНГ 9.9. Определение копирующего конструктора с глубоким копированием

```
0: #include <iostream>
1: #include <string.h>
2: using namespace std;
3: class MyString
4: {
5:     private:
6:         char* buffer;
7:
8:     public:
9:         MyString(const char* initString) // Конструктор
10:        {
11:            buffer = nullptr;
12:            cout << "Вызов конструктора по умолчанию" << endl;
13:            if(initString != nullptr)
14:            {
15:                buffer = new char[strlen(initString) + 1];
16:                strcpy(buffer, initString);
17:
18:                cout << "buffer указывает на: 0x" << hex;
19:                cout << (unsigned int*)buffer << endl;
20:            }
21:        }
22:
23:        MyString(const MyString& copySource) // Копирующий конструктор
24:        {
25:            buffer = nullptr;
26:            cout << "Вызов копирующего конструктора" << endl;
27:            if(copySource.buffer != nullptr)
28:            {
29:                // Выделение собственного буфера
```

```
30:         buffer = new char[strlen(copySource.buffer) + 1];
31:
32:         // Глубокое копирование исходного буфера в целевой
33:         strcpy(buffer, copySource.buffer);
34:
35:         cout << "buffer указывает на: 0x" << hex;
36:         cout << (unsigned int*)buffer << endl;
37:     }
38: }
39:
40: // Деструктор
41: ~MyString()
42: {
43:     cout << "Вызов деструктора" << endl;
44:     delete[] buffer;
45: }
46:
47: int GetLength()
48: { return strlen(buffer); }
49:
50: const char* GetString()
51: { return buffer; }
52: };
53:
54: void UseMyString(MyString str)
55: {
56:     cout << "Длина buffer в MyString равна ";
57:     cout << str.GetLength() << " символом" << endl;
58:
59:     cout << "buffer содержит: " << str.GetString() << endl;
60:     return;
61: }
62:
63: int main()
64: {
65:     MyString sayHello("Hello from String Class");
66:     UseMyString(sayHello);
67:
68:     return 0;
69: }
```

Результат

Вызов конструктора по умолчанию
buffer указывает на: 0x01232D90
Вызов копирующего конструктора
buffer указывает на: 0x01232DD8

Длина `buffer` в `MyString` равна 17 символам
`buffer` содержит: `Hello from String Class`
 Вызов деструктора
 Вызов деструктора

Анализ

Большая часть кода подобна коду листинга 9.8 с добавлением нового копирующего конструктора в строках 23–38. Для начала рассмотрим функцию `main()`, которая (как и прежде) создает объект `sayHello` в строке 65. Создание объекта `sayHello` приводит к выводу первой строки в поток `cout` в строке 12 конструктора `MyString`. Для удобства конструктор отображает также адрес области памяти, на которую указывает `buffer`. Затем, в строке 66, функция `main()` передает объект `sayHello` по значению функции `UseMyString()`, что автоматически приводит к вызову копирующего конструктора, о чем и свидетельствует вывод приложения. Код в копирующем конструкторе очень похож на таковой в обычном конструкторе. Основная идея та же — выяснить длину строки в стиле C, которая содержится в буфере оригинала (строка 30), выделить достаточное количество памяти в собственном экземпляре `buffer`, а затем использовать функцию `strcpy` для копирования оригинала в копию (строка 33). Это не поверхностное копирование значения указателя. Это *глубокое копирование* (deep copy), при котором во вновь созданный буфер, принадлежащий объекту, копируется содержимое, на которое указывает указатель (рис. 9.4).

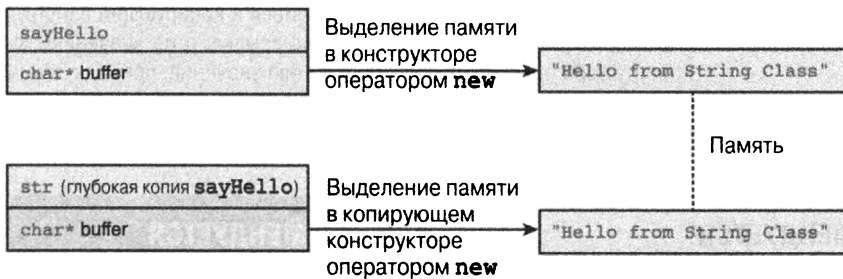


РИС. 9.4. Глубокое копирование аргумента `sayHello` в параметр `str` при вызове функции `UseMyString()`

Вывод листинга 9.9 свидетельствует о том, что адреса памяти, на которые указывает `buffer`, разные у копии и у оригинала, т.е. два объекта больше не содержат указатель на один и тот же адрес памяти. В результате при выходе из функции `UseMyString()` и уничтожении параметра `str` код деструктора выполняет оператор `delete[]` для адреса памяти, который был выделен в копирующем конструкторе и принадлежит данному объекту. При этом область памяти, указатель на которую содержится в объекте `sayHello` функции `main()`, никак не затрагивается. Таким образом, при выходе из обеих функций успешно и без неприятностей освобождаются их собственные области памяти, и приложение успешно завершается.

ПРИМЕЧАНИЕ

Копирующий конструктор гарантирует глубокое копирование в таких случаях, как вызов функции:

```
MyString sayHello("Hello from String Class");  
UseMyString(sayHello);
```

Однако при попытках копировать через присваивание

```
MyString overwrite("who cares?");  
overwrite = sayHello;
```

это все еще будет поверхностным копированием из-за *оператора присваивания* (assignment operator) по умолчанию, генерируемого компилятором (поскольку вы сами его не определили). Чтобы избежать проблемы поверхностного копирования при присваивании, необходимо реализовать копирующий оператор присваивания (operator=).

Копирующий оператор присваивания рассматривается на занятии 12, "Типы операторов и их перегрузка". Листинг 12.7 содержит усовершенствованную версию класса MyString, которая реализует именно этот оператор:

```
MyString::operator=(const MyString& CopySource)  
{  
    //...Код копирующего оператора присваивания  
}
```

ВНИМАНИЕ!

Использование ключевого слова `const` в объявлении копирующего конструктора гарантирует, что он не изменит копируемый объект.

Кроме того, параметр должен передаваться в копирующий конструктор по ссылке. Если бы он передавался не по ссылке, а по значению, то вызов конструктора приводил бы к копированию значения, приводя, таким образом, к вызову самого себя вновь и вновь, до полного исчерпания системных ресурсов памяти.

РЕКОМЕНДУЕТСЯ

Всегда создавайте копирующий конструктор и копирующий оператор присваивания, если ваш класс в качестве члена содержит *простой указатель* (например, `char*` и т.п.).

Всегда создавайте копирующий конструктор с константной ссылкой на оригинал в качестве параметра.

Избегайте неявных преобразований типов, используя ключевое слово `explicit` при объявлении конструкторов.

Используйте в качестве членов класса строк, такие как `std::string`, и классы интеллектуальных указателей вместо простых указателей, поскольку они реализуют копирующие конструкторы и экономят ваши усилия.

НЕ РЕКОМЕНДУЕТСЯ

Не используйте простые указатели в качестве членов класса, если только вы не оказываетесь в ситуации, когда этого совершенно невозможно избежать.

ПРИМЕЧАНИЕ

Класс `MyString` с простым указателем `char* buffer` в качестве члена используется как пример для объяснения необходимости копирующего конструктора.

Если вам нужно создать класс, который должен содержать строковые данные, например, для хранения имени, то используйте класс `std::string`, а не `char*`, поскольку при отсутствии простых указателей не нужен даже копирующий конструктор. Дело в том, что копирующий конструктор по умолчанию, генерируемый компилятором, гарантирует вызов всех доступных копирующих конструкторов членов объектов, таких как `std::string`.

Перемещающий конструктор улучшает производительность

Бывают случаи, когда ваши объекты копируются автоматически, просто в силу природы языка программирования и его потребностей. Рассмотрим следующий код:

```
class MyString
{
    // Реализация из листинга 9.9
};

MyString Copy(MyString& source)
{
    MyString CopyForReturn(source.GetString()); // Создание копии
    return copyForReturn;                       // Возвращение по значению вызывает
                                                // копирующий конструктор
}

int main()
{
    MyString sayHello("Hello World of C++");
    MyString sayHelloAgain(Copy(sayHello)); // Вызов копирующего
                                                // конструктора дважды

    return 0;
}
```

Как свидетельствует комментарий, при создании экземпляра `sayHelloAgain` копирующий конструктор был вызван дважды, следовательно, и глубокое копирование было выполнено дважды из-за вызова функции `Copy(sayHello)`, которая возвращает объект класса `MyString` по значению. Однако возвращаемое значение — временное, недоступное вне данного выражения. Таким образом, вызов копирующего конструктора, добросовестно выполненный компилятором C++, фактически приводит к снижению производительности, которое может оказаться существенным, если массив объектов в динамической памяти имеет большой размер.

Чтобы избежать этого падения производительности, можно в дополнение к копирующему конструктору создать *перемещающий конструктор* (move constructor). Синтаксис перемещающего конструктора имеет следующий вид:

```
// Перемещающий конструктор
MyString(MyString&& MoveSource)
{
    if(MoveSource.buffer != nullptr)
    {
        buffer = MoveSource.buffer; // Принять владение,
                                   // т.е. "переместить"
        MoveSource.buffer = nullptr; // Установить источник
                                   // в nullptr
    }
}
```

При доступности такого конструктора он автоматически выбирается компилятором C++11 для “перемещения” временного ресурса, а следовательно, избегает этапа глубокого копирования. При реализованном перемещающем конструкторе комментарий должен быть заменен следующим:

```
MyString sayHelloAgain(Copy(sayHello)); // По одному вызову
// копирующего и перемещающего конструкторов
```

Перемещающий конструктор обычно реализуют с *перемещающим оператором присваивания* (move assignment operator), который подробно обсуждается на занятии 12, “Типы операторов и их перегрузка”. Улучшенная версия класса MyString, реализующая перемещающий конструктор и оператор присваивания при перемещении, приведена в листинге 12.10.

Способы использования конструкторов и деструктора

На этом занятии вы изучили ряд очень важных, фундаментальных концепций, таких как конструктор, деструктор, а также абстракция данных и методов с помощью таких ключевых слов, как public и private. Эти концепции позволяют создавать классы, которые управляют тем, как они создаются, копируются, уничтожаются и предоставляют данные.

Рассмотрим несколько интересных шаблонов, которые помогут вам справиться с некоторыми проблемами в своих проектах.

Класс, который не разрешает себя копировать

Предположим, вас попросят смоделировать конституцию вашей страны. У страны может быть только один президент. Ваш класс President рискует следующим:

```
President ourPresident;  
DoSomething(ourPresident); // Дублирование при передаче по значению  
President clone;  
clone = ourPresident;      // Дублирование при присваивании
```

Понятно, что таких ситуаций следует избегать. Кроме конституции, вы могли бы моделировать операционную систему, в которой необходимо было бы обеспечить только одну локальную сеть, один процессор и т.д. Словом, иногда необходимо избегать ситуаций, в которых некоторые ресурсы могут быть скопированы. Если вы не объявите копирующий конструктор, компилятор C++ сам сгенерирует открытый копирующий конструктор по умолчанию. Это нарушит ваш проект и создаст угрозу для его реализации. Но язык предоставляет решение этой проблемы.

Объявление закрытого копирующего конструктора позволяет гарантировать, что объект вашего класса не может быть скопирован. Так, вызов функции `DoSomething(ourPresident)` приведет к неудаче при компиляции. Чтобы избежать присваивания, следует объявить закрытым оператор присваивания.

Таким образом, решение следующее:

```
class President  
{  
private:  
    President(const President&);           // Закрытый копирующий  
                                           // конструктор  
    President& operator=(const President&); // Закрытый копирующий  
                                           // оператор присваивания  
  
    // ... Другие атрибуты  
};
```

Нет никакой необходимости в реализации закрытого копирующего конструктора и оператора присваивания. Для предотвращения копирования объектов класса `President` вполне достаточно лишь объявления их как закрытых.

Класс-синглтон, обеспечивающий наличие только одного экземпляра

Обсуждавшийся ранее класс `President` хорош, но у него есть и недостаток: невозможно воспрепятствовать появлению нескольких президентов при создании нескольких объектов:

```
President One, Two, Three;
```

Благодаря закрытым копирующим конструкторам индивидуальные объекты не копируемы, однако в идеале класс `President` нуждается в одном и только одном объекте, а создание дополнительных запрещается. Такова концепция *синглтона* (singleton), подразумевающая использование закрытого конструктора, закрытого оператора присваивания и статического члена экземпляра класса для реализации этого мощного проектного шаблона.

СОВЕТ

Ключевое слово `static`, примененное к переменной-члену класса, гарантирует его совместное использование всеми экземплярами.

Когда ключевое слово `static` применяется к локальной переменной, объявленной в пределах функции, это гарантирует сохранение переменной своего значения между вызовами функции.

Когда ключевое слово `static` применяется к функции-члену (методу), этот метод может использоваться без указания конкретного объекта класса, так как он принадлежит классу, а не конкретному объекту.

Ключевое слово `static` — основной компонент в создании класса *синглтона* (singleton class), как показано в листинге 9.10.

ЛИСТИНГ 9.10. Синглтон `President`, запрещающий копирование, присваивание и создание нескольких экземпляров

```

0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class President
5: {
6:     private:
7:         President() {};           // Закрытые: конструктор по умолчанию
8:         President(const President&); // Копирующий конструктор
9:         const President& operator=(const President&); // Присваивание
10:
11:         string name;
12:
13:     public:
14:         static President& GetInstance()
15:         {
16:             // Статические объекты конструируются только один раз
17:             static President onlyInstance;
18:             return onlyInstance;
19:         }
20:
21:         string GetName()
22:         { return name; }
23:
24:         void SetName(string InputName)
25:         { name = InputName; }
26: };
27:
28: int main()
29: {
30:     President& onlyPresident = President::GetInstance();
31:     onlyPresident.SetName("Авраам Линкольн");
32:
33:     // Раскомментируйте, чтобы убедиться в невозможности дублей:

```

```
34:    // President second;
35:    // President* third= new President();
36:    // President fourth = onlyPresident;
37:    // onlyPresident = President::GetInstance();
38:
39:    cout << "Президента зовут ";
40:    cout << President::GetInstance().GetName() << endl;
41:
42:    return 0;
43: }
```

Результат

Президента зовут Авраам Линкольн

Анализ

Взгляните на функцию `main()`: в ней всего несколько строк кода и несколько закомментированных строк, которые демонстрируют все возможные варианты создания новых экземпляров или копий объектов класса `President`, которые будут отвергнуты компилятором. Давайте проанализируем их одна за другой:

```
34:    // President second;                // Конструктор недоступен
35:    // President* third= new President(); // Конструктор недоступен
```

Строки 34 и 35 — это попытки создания объекта в стеке и динамической памяти соответственно, с использованием конструктора по умолчанию, который недоступен, поскольку в строке 7 он объявлен закрытым.

```
36:    // President fourth = OnlyPresident;
```

В строке 36 предпринимается попытка создания копии существующего объекта с помощью копирующего конструктора (присваивание во время создания вызывает копирующий конструктор), который недоступен в функции `main()`, поскольку в строке 8 он объявлен закрытым.

```
37:    // onlyPresident = President::GetInstance();
```

Строка 37 является попыткой создания копии через присваивание, которое не работает, так как оператор присваивания объявлен закрытым в строке 9. Таким образом, функция `main()` не может создать экземпляр класса `President` никаким способом, а единственная оставшаяся возможность получить экземпляр класса `President` — это использовать статическую функцию `GetInstance()`, как в строке 30. Поскольку функция `GetInstance()` является статическим членом класса, она очень похожа на глобальную функцию, которая может быть вызвана и без наличия объекта. Функция `GetInstance()`, реализованная в строках 14–19, использует статическую переменную `onlyInstance` для гарантии наличия одного и только одного экземпляра класса `President`. Дело в том, что код в строке 17 выполняется только один раз (статическая инициализация), а следовательно, функция `GetInstance()` возвращает единственный доступный экземпляр класса `President`, независимо от того, как часто вызывается `President::GetInstance()`.

ВНИМАНИЕ!

Используйте *проектный шаблон синглтон* (singleton pattern) только там, где это абсолютно необходимо, с учетом будущего развития приложения и его возможностей. Обратите внимание на то, что ограничение на создание нескольких экземпляров может стать узким местом архитектуры, когда впоследствии понадобятся несколько экземпляров класса.

Например, если наш проект перерастет от моделирования одной нации к Организации Объединенных Наций, которая в настоящее время насчитывает 193 члена, архитектурная проблема президента-синглтона станет очевидной.

Класс, запрещающий создание экземпляра в стеке

Пространство в стеке зачастую ограничено. Если вы пишете базу данных, способную содержать терабайт данных в своих внутренних структурах, то имеет смысл гарантировать, что клиент этого класса не сможет создать его экземпляр в стеке, а вынужден будет создавать его только в динамической памяти. Для этого следует объявить закрытым деструктор:

```
class MonsterDB
{
private:
    // Закрытый деструктор
    ~MonsterDB();
    // ... Члены, требующие огромный объем памяти
};
```

При попытке использовать класс MonsterDB следующим образом у вас ничего не получится:

```
int main()
{
    MonsterDB myDatabase; // Ошибка компиляции
    // ... Остальной код
    return 0;
}
```

Этот экземпляр, если бы он был создан, находился бы в стеке. Но поскольку компилятор знает, что при выходе из области видимости экземпляр класса myDatabase должен быть удален, он автоматически пытается вызвать его деструктор в конце функции main(), который оказывается недоступным, так как объявлен закрытым. Это приводит к ошибке времени компиляции.

Однако закрытый деструктор не мешает вам создать экземпляр в распределяемой памяти:

```
int main()
{
    MonsterDB* myDatabase = new MonsterDB(); // Ошибки нет
    // ... Остальной код
    return 0;
}
```

Если вы усмотрите здесь утечку памяти, то не ошибетесь. Поскольку деструктор недоступен для функции `main()`, вы не можете удалить в ней объект. Такой класс, как `MonsterDB`, нуждается в открытой статической функции-члене, которая удаляет экземпляр (как член класса такая функция имеет доступ к закрытому деструктору). Рассмотрим листинг 9.11.

ЛИСТИНГ 9.11. Класс базы данных `MonsterDB`, позволяющий создавать свои объекты только в динамической памяти (используя оператор `new`)

```
0: #include <iostream>
1: using namespace std;
2:
3: class MonsterDB
4: {
5:     private:
6:         ~MonsterDB() {}; // Закрытый деструктор для предотвращения
7:                           // создания объектов в стеке
8:     public:
9:         static void DestroyInstance(MonsterDB* pInstance)
10:        {
11:            delete pInstance; // Вызов закрытого деструктора
12:        }
13:
14:        void DoSomething() {} // Пустой метод (для примера)
15: };
16:
17: int main()
18: {
19:     MonsterDB* myDB = new MonsterDB(); // В динамической памяти
20:     myDB->DoSomething();
21:
22:     // Раскомментируйте, чтобы убедиться в неработоспособности
23:     // delete myDB; // Закрытый деструктор не вызываем
24:
25:     // Использование статического метода для удаления
26:     MonsterDB::DestroyInstance(myDB);
27:
28:     return 0;
29: }
```

Этот фрагмент кода не имеет вывода.

Анализ

Код предназначен только для демонстрации класса, который запрещает создание экземпляра в стеке. Ключевым является закрытый деструктор, показанный в строке 6. Статическая функция `DestroyInstance()` в строках 9–12 требуется для освобождения памяти, поскольку функция `main()` не может вызвать `delete` для уничтожения `myDB`. Вы можете убедиться в этом, раскомментировав строку 23.

Применение конструкторов для преобразования типов

Ранее на этом занятии вы узнали, что конструкторы могут быть перегружены, т.е. могут принимать один или несколько параметров. Эта возможность часто используется для выполнения преобразования данных одного типа в другой. Рассмотрим класс `Human`, который имеет перегруженный конструктор, принимающий целое число.

```
class Human {
    int age;
public:
    Human(int humansAge): age(humansAge) {}
};

// Функция, принимающая Human в качестве параметра
void DoSomething(Human person) {
    cout << "Работаем с Human" << endl;
    return;
}
```

Такой конструктор позволяет выполнить следующее преобразование:

```
Human kid(10); // Преобразование int в Human
DoSomething(kid);
```

ВНИМАНИЕ!

Такой преобразующий конструктор допускает выполнение неявных преобразований:

```
Human anotherKid = 11; // int преобразуется в Human
DoSomething(.10);      // 10 преобразуется в Human!
```

Мы объявили `DoSomething(Human person)` как функцию, которая принимает параметр типа `Human`, а не `int`! Так почему же эта строка компилируется? Компилятор знает, что класс `Human` имеет конструктор, который принимает целое число, и выполняет неявное преобразование вместо вас, создавая объект типа `Human` из переданного целого числа и передавая его в качестве аргумента в функцию.

Чтобы избежать неявных преобразований, при объявлении конструктора следует указывать ключевое слово `explicit`:

```
class Human
{
    int age;
public:
    explicit Human(int humansAge): age(humansAge) {}
};
```

Применение ключевого слова `explicit` не является необходимым, но во многих случаях является хорошей практикой программирования. В следующем примере в листинге 9.12 показана версия класса `Human`, который не допускает неявных преобразований.

ЛИСТИНГ 9.12. Использование ключевого слова `explicit` для блокирования неявного преобразования типов

```
0: #include<iostream>
1: using namespace std;
2:
3: class Human
4: {
5:     int age;
6:     public:
7:         // explicit конструктор блокирует неявные преобразования
8:         explicit Human(int humansAge) : age(humansAge) {}
9: };
10:
11: void DoSomething(Human person)
12: {
13:     cout << "Работа с Human" << endl;
14:     return;
15: }
16:
17: int main()
18: {
19:     Human kid(10);                // Явное преобразование, ОК
20:     Human anotherKid = Human(11); // Явное преобразование, ОК
21:     DoSomething(kid);             // ОК
22:
23:     // Human anotherKid2 = 11; // Ошибка: неявное преобразование
24:     // DoSomething(10);        // Неявное преобразование
25:
26:     return 0;
27: }
```

Результат

Работа с Human

Анализ

Строки кода, которые не выполняют никакого вывода, по меньшей мере так же важны, как и те, которые вывод выполняют. Функция `main()` в строках 17–27 использует разные варианты создания объекта класса `Human`, объявленного с `explicit` конструктором в строке 8. Успешно компилируемые строки представляют собой попытки явного преобразования, где `int` использован для создания объекта типа `Human`. Строки 23 и 24 представляют собой варианты, включающие неявные преобразования. Эти строки закомментированы, но если их раскомментировать, то они будут компилироваться, только когда мы удалим ключевое слово `explicit` в строке 8. Таким образом, этот пример демонстрирует, как ключевое слово `explicit` защищает нас от неявных преобразований.

СОВЕТ

Проблема неявных преобразований и их устранения с помощью ключевого слова `explicit` относится и к операторам. Не забудьте использовать ключевое слово `explicit` при программировании операторов преобразования, с которыми вы познакомитесь на занятии 12, "Типы операторов и их перегрузка".

Указатель `this`

Указатель `this` — это важнейшая концепция языка C++; зарезервированное ключевое слово `this` применимо в рамках класса и содержит адрес текущего объекта. Другими словами, значение указателя `this` — это `&object`. В пределах метода класса, когда вы вызываете другой метод, компилятор неявно передает ему при вызове указатель `this` как невидимый параметр:

```
class Human
{
    private:
        // ... Объявления закрытых членов
        void Talk(string Statement)
        {
            cout << Statement;
        }
    public:
        void IntroduceSelf()
        {
            Talk("Bla bla"); // То же, что Talk(this, "Bla bla")
        }
};
```

Здесь представлен метод `IntroduceSelf()`, использующий закрытый член `Talk()` для вывода строки на экран. В действительности компилятор передает указатель `this` в вызов метода `Talk()`, который вызывается, как если бы имел вид `Talk(this, "Bla bla")`.

С точки зрения программирования у указателя `this` не слишком много областей применения, но иногда он оказывается удобным. Например, в коде функции `SetAge()` в листинге 9.2 для доступа к переменной-члену `age` может использоваться такое выражение:

```
void SetAge(int HumansAge)
{
    this->age = HumansAge; // То же, что и age = HumansAge
}
```

ПРИМЕЧАНИЕ

Указатель `this` не передается статическим методам класса, так как статические функции не связаны с экземпляром класса. Статические методы используются всеми экземплярами.

Чтобы использовать переменные экземпляра в статической функции, требуется явно объявить параметр для передачи статической функции в качестве аргумента указателя `this`.

Размер класса

Вы изучили основные принципы определения собственного типа с использованием ключевого слова `class`, позволяющие инкапсулировать атрибуты данных и методы, работающие с этими данными. Оператор `sizeof()`, описанный на занятии 3, “Использование переменных и констант”, используется для определения объема памяти, занимаемого переменной определенного типа, в байтах. Этот оператор применим и для классов, и сообщает сумму количества байтов, занимаемых каждым атрибутом данных, содержащимся в объявлении класса. В зависимости от используемого компилятора оператор `sizeof()` может включать или не включать для некоторых атрибутов дополнения до границ слова. Функции-члены и их локальные переменные в определении размера класса не участвуют. Рассмотрим листинг 9.13.

ЛИСТИНГ 9.13. Результат применения оператора `sizeof()` к классам и их экземплярам

```

0: #include <iostream>
1: #include <string.h>
2: using namespace std;
3: class MyString
4: {
5:     private:
6:         char* buffer;
7:
8:     public:
9:         MyString(const char* initString) // Конструктор по умолчанию
10:        {
11:            buffer = nullptr;
12:            if(initString != nullptr)
13:            {
14:                buffer = new char[strlen(initString) + 1];
15:                strcpy(buffer, initString);
16:            }
17:        }
18:
19:        MyString(const MyString& copySource) // Копирующий конструктор
20:        {
21:            buffer = nullptr;
22:            if(copySource.buffer != nullptr)
23:            {
24:                buffer = new char[strlen(copySource.buffer) + 1];
25:                strcpy(buffer, copySource.buffer);
26:            }
27:        }
28:
29:        ~MyString()
30:        {
31:            delete[] buffer;

```

```

32:     }
33:
34:     int GetLength()
35:     { return strlen(buffer); }
36:
37:     const char* GetString()
38:     { return buffer; }
39: };
40:
41: class Human
42: {
43:     private:
44:         int age;
45:         bool gender;
46:         MyString name;
47:
48:     public:
49:         Human(const MyString& InputName, int InputAge, bool g)
50:         : name(InputName), age(InputAge), gender(g) {}
51:
52:         int GetAge()
53:         { return age; }
54: };
55:
56: int main()
57: {
58:     MyString mansName("Adam");
59:     MyString womansName("Eve");
60:
61:     cout << "sizeof(MyString) = " << sizeof(MyString) << endl;
62:     cout << "sizeof(mansName) = " << sizeof(mansName) << endl;
63:     cout << "sizeof(womansName) = " << sizeof(womansName) << endl;
64:
65:     Human firstMan(mansName, 25, true);
66:     Human firstWoman(womansName, 18, false);
67:
68:     cout << "sizeof(Human) = " << sizeof(Human) << endl;
69:     cout << "sizeof(firstMan) = " << sizeof(firstMan) << endl;
70:     cout << "sizeof(firstWoman) = " << sizeof(firstWoman) << endl;
71:
72:     return 0;
73: }

```

Результат для 32-разрядного компилятора

```

sizeof(MyString) = 4
sizeof(mansName) = 4
sizeof(womansName) = 4

```

```
sizeof(Human) = 12
sizeof(firstMan) = 12
sizeof(firstWoman) = 12
```

Результат для 64-разрядного компилятора

```
sizeof(MyString) = 8
sizeof(mansName) = 8
sizeof(womansName) = 8
sizeof(Human) = 16
sizeof(firstMan) = 16
sizeof(firstWoman) = 16
```

Анализ

Пример несколько длинноват, поскольку содержит класс `MyString` и вариант класса `Human`, который использует тип `MyString` для хранения имени (`name`), а также имеет новый параметр типа `bool` для пола (`gender`).

Приступим к анализу вывода. Как можно заметить, результат выполнения оператора `sizeof()` для класса совпадает с таковым для объекта класса. Следовательно, `sizeof(MyString)` — то же самое, что и `sizeof(mansName)`, поскольку количество байтов, использованных классом, по существу, фиксируется во время компиляции. Не удивляйтесь, что размер в байтах объектов `firstMan` и `firstWoman` одинаков, несмотря на то, что один содержит имя `Adam`, а другой `Eve`, поскольку они хранятся в переменной `MyString::buffer`, которая фактически является указателем типа `char*`, размер которого составляет 4 байта (на моей 32-разрядной системе) и не зависит от объема данных, на которые указывает.

При вычислении размера типа `Human` получается 12 байт. Строки 44–46 свидетельствуют, что класс `Human` содержит атрибуты типа `int`, `bool` и `MyString`. Чтобы освежить в памяти размер в байтах используемых встроенных типов, обратитесь к листингу 3.5. Тип `int` использует 4 байта, тип `bool` — 1 байт, тип `MyString` — 4 байта на системе автора, что в итоге не равно значению 12, которое выведено программой. Дело в том, что на результат оператора `sizeof()` влияет *дополнение до границ слова* и другие факторы.

Чем структура отличается от класса

Ключевое слово `struct` осталось со времен языка `C` и во всех практических целях обрабатывается компилятором `C++` почти так же, как ключевое слово `class`. Различия кроются лишь в заданном по умолчанию модификаторе доступа (`public` или `private`), когда разработчик не указывает никакого модификатора. По умолчанию, если ничего не указано, члены структуры являются открытыми (`public`), а члены класса — закрытыми (`private`), и если не определено иное, то члены структуры остаются открытыми при наследовании базовой структуры, а члены класса — закрытыми. Наследование рассматривается на занятии 10, “Реализация наследования”.

Вариант структуры класса Human из листинга 9.13 имел бы следующий вид:

```
struct Human
{
    // Конструктор, открытый по умолчанию (поскольку
    // никакой модификатор доступа не упомянут)
    Human(const MyString& inputName, int inputAge, bool inputGender)
        : name(inputName), age(inputAge), gender(inputGender) {}

    int GetAge()
    {
        return age;
    }

private:
    int age;
    bool gender;
    MyString name;
};
```

Как можно заметить, структура Human очень похожа на класс Human, и создание экземпляра объекта структуры очень похоже на таковое для класса:

```
Human firstMan("Adam", 25, true); // Экземпляр структуры Human
```

Объявление друзей класса

Класс не разрешает доступ извне к своим закрытым переменным-членам и методам. Это правило не относится к тем классам и функциям, которые с помощью ключевого слова *friend* объявлены *дружественными* (*friend*), как показано в листинге 9.14.

ЛИСТИНГ 9.14. Использование ключевого слова *friend*

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6:     private:
7:         friend void DisplayAge(const Human& person);
8:         string name;
9:         int age;
10:
11:     public:
12:         Human(string humansName, int humansAge)
13:         {
14:             name = humansName;
```

```
15:         age = humansAge;
16:     }
17: };
18:
19: void DisplayAge(const Human& person)
20: {
21:     cout << person.age << endl;
22: }
23:
24: int main()
25: {
26:     Human firstMan("Adam", 25);
27:     cout << "Доступ друга к закрытым членам: ";
28:     DisplayAge(firstMan);
29:
30:     return 0;
31: }
```

Результат

Доступ друга к закрытым членам: 25

Анализ

Строка 7 содержит объявление, указывающее компилятору, что функция `DisplayAge()` из глобальной области видимости является другом, а значит, ей разрешен специальный доступ к закрытым членам класса `Human`. Закомментировав строку 7, вы сразу получите ошибку компиляции в строке 22.

Как и функции, внешние классы также могут быть объявлены дружественными, как показано в листинге 9.15.

ЛИСТИНГ 9.15. Объявление класса другом

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: class Human
5: {
6:     private:
7:         friend class Utility;
8:         string name;
9:         int age;
10:
11:     public:
12:         Human(string humansName, int humansAge)
13:         {
14:             name = humansName;
15:             age = humansAge;
```

```

16:     }
17: };
18:
19: class Utility
20: {
21:     public:
22:         static void DisplayAge(const Human& person)
23:         {
24:             cout << person.age << endl;
25:         }
26: };
27:
28: int main()
29: {
30:     Human firstMan("Adam", 25);
31:     cout << "Доступ друга к закрытым членам: ";
32:     Utility::DisplayAge(firstMan);
33:
34:     return 0;
35: }

```

Результат

Доступ друга к закрытым членам: 25

Анализ

Строка 7 объявляет класс `Utility` дружественным классу `Human`. Это объявление позволяет всем методам класса `Utility` обращаться даже к закрытым переменным-членам и методам класса `Human`.

Специальный механизм хранения данных — `union`

Объединение (`union`) представляет собой особый тип класса, в котором в каждый момент времени активен только один из нестатических членов-данных. Таким образом, объединение может принимать несколько членов-данных, как и класс, но с тем отличием, что использоваться в каждый момент времени может только один из них.

Объявление объединения

Объединение объявляется с помощью ключевого слова `union`, за которым следуют имя объединения и его члены в фигурных скобках:

```

union Имя_Объединения {
    Тип1 член1;

```

```
    Тип2 член2;  
    ...  
    ТипN членN;  
};
```

Вы можете создавать объекты и использовать объединения следующим образом:

```
UnionName unionObject;  
unionObject.member2 = value; // member2 выбран как активный член
```

ПРИМЕЧАНИЕ

Подобно struct, члены union по умолчанию открыты. Однако, в отличие от struct, объединения не могут использоваться в иерархиях наследования.

Кроме того, при применении sizeof() к объединению всегда возвращается размер наибольшего содержащегося в нем члена, даже если этот член в данном объекте в настоящее время неактивен.

Где используется объединение

Часто объединение используется в качестве члена struct для моделирования сложного типа данных. В некоторых реализациях возможность объединения интерпретировать фиксированное пространство памяти как другой тип используется для преобразования типов или иной интерпретации памяти — практика, которая как минимум является очень спорной и не рекомендуемой к применению.

В листинге 9.16 демонстрируются объявление и применение объединений.

ЛИСТИНГ 9.16. Объявление и инстанцирование объединения и применение sizeof()

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: union SimpleUnion  
4: {  
5:     int num;  
6:     char alphabet;  
7: };  
8:  
9: struct ComplexType  
10: {  
11:     enum DataType  
12:     {  
13:         Int,  
14:         Char  
15:     } Type;  
16:  
17:     union Value  
18:     {  
19:         int num;
```



```

20:         char alphabet;
21:
22:         Value() {}
23:         ~Value() {}
24:     }value;
25: };
26:
27: void DisplayComplexType(const ComplexType& obj)
28: {
29:     switch(obj.Type)
30:     {
31:     case ComplexType::Int:
32:         cout<<"union содержит число: "<<obj.value.num<<endl;
33:         break;
34:
35:     case ComplexType::Char:
36:         cout<<"union содержит символ: "<<obj.value.alphabet<<endl;
37:         break;
38:     }
39: }
40:
41: int main()
42: {
43:     SimpleUnion u1, u2;
44:     u1.num = 2100;
45:     u2.alphabet = 'C';
46:     cout << "sizeof(u1) с числом: " << sizeof(u1) << endl;
47:     cout << "sizeof(u2) с символом: " << sizeof(u2) << endl;
48:
49:     ComplexType myData1, myData2;
50:     myData1.Type = ComplexType::Int;
51:     myData1.value.num = 2017;
52:
53:     myData2.Type = ComplexType::Char;
54:     myData2.value.alphabet = 'X';
55:
56:     DisplayComplexType(myData1);
57:     DisplayComplexType(myData2);
58:
59:     return 0;
60: }

```

Результат

```

sizeof(u1) с числом: 4
sizeof(u2) с символом: 4
union содержит число: 2017
union содержит символ: X

```

Анализ

В приведенном примере показано, что `sizeof()` объединений `u1` и `u2` возвращает одинаковое количество выделенной для обоих объектов памяти, несмотря на то, что `u1` используется для хранения целого числа, а `u2` — для хранения `char`, а размер `char` меньше, чем размер `int`. Дело в том, что компилятор выделяет для объединения количество памяти, которое потребляется крупнейшим объектом, который может содержаться в нем. Структура `ComplexType`, определенная в строках 9–25, содержит перечисление `DataType`, которое используется для указания характера объекта, хранящегося в объединении, помимо данных-члена, который представляет собой объединение с именем `value`. Такое сочетание структуры с перечислением, используемым для указания сведений о хранимом типе, и объединение для хранения значения является распространенным применением объединения. Например, широко используемая в Windows структура `VARIANT` использует аналогичный подход. Эта комбинация применена в функции `DisplayComplexType()`, определенной в строках 27–39, которая использует перечисление в конструкции `switch-case`. В качестве примера мы включили в это объявление конструктор и деструктор — в листинге 9.16 это не обязательно, так как объединение содержит старые простые типы данных. Однако если объединение состоит из пользовательских типов наподобие класса или структуры, такие конструктор и деструктор могут потребоваться.

СОВЕТ

Ожидается, что в стандарт C++17 будет включена безопасная с точки зрения типов альтернатива объединению. Чтобы узнать о `std::variant`, обратитесь к занятию 29, “Что дальше”.

Агрегатная инициализация классов и структур

Показанный далее синтаксис инициализации называется синтаксисом *агрегатной инициализации* (*aggregate initialization*):

```
Тип Имя_объекта = { аргумент1, ..., аргументN};
```

Начиная с C++11 имеется альтернативный вариант:

```
Тип Имя_объекта { аргумент1, ..., аргументN};
```

Агрегатная инициализация может быть применена к агрегатам, а потому важно понимать, какие типы данных попадают в эту категорию.

Вы уже встречались с агрегатной инициализацией при инициализации массивов на занятии 4, “Массивы и строки”.

```
int myNums[] = { 9, 5, -1 }; // myNums имеет тип int[3]
char hello[6] = { 'h', 'e', 'l', 'l', 'o', ' \0' };
```

Однако термин *агрегат* не ограничивается массивами простых типов, таких как целые числа или символы, но распространяется и на классы (а потому на структуры

и объединения тоже). Существуют ограничения, введенные в стандарте на спецификации структуры или класса, который может быть назван агрегатом. Эти ограничения несколько различны в разных версиях стандарта C++. Тем не менее можно с уверенностью утверждать, что классы/структуры, которые состоят из открытых и нестатических данных-членов, не содержащие закрытых или защищенных данных-членов, не содержащие виртуальных функций-членов, использующие только открытое наследование, т.е. не `private`, `protected` или виртуальное наследование (или не использующие никакого), и не имеющие пользовательских конструкторов, являются агрегатами и могут быть инициализированы соответствующим образом.

СОВЕТ

Наследование рассматривается на занятиях 10, “Реализация наследования”, и 11, “Полиморфизм”.

Таким образом, приведенная далее структура удовлетворяет указанным требованиям и, будучи агрегатом, может быть инициализирована как таковой:

```
struct Aggregate1 {
    int num;
    double pi;
};
```

Инициализация:

```
Aggregate1 a1{ 2017, 3.14 };
```

Еще один пример:

```
struct Aggregate2 {
    int num;
    char hello[6];
    int impYears[5];
};
```

Инициализация:

```
Aggregate2 a2 {42, {'h', 'e', 'l', 'l', 'o'},
               {1998, 2003, 2011, 2014, 2017}};
```

Листинг 9.17 содержит пример, демонстрирующий применение агрегатной инициализации к структурам и классам.

ЛИСТИНГ 9.17. Агрегатная инициализация класса

```
0: #include <iostream>
1: #include<string>
2: using namespace std;
3:
4: class Aggregate1
5: {
6:     public:
```

```
7:     int num;
8:     double pi;
9: };
10:
11: struct Aggregate2
12: {
13:     char hello[6];
14:     int impYears[3];
15:     string world;
16: };
17:
18: int main()
19: {
20:     int myNums[] = { 9, 5, -1 }; // myNums имеет тип int[3]
21:     Aggregate1 a1{ 2017, 3.14 };
22:     cout << "Pi приближенно равно: " << a1.pi << endl;
23:
24:     Aggregate2 a2{ {'h', 'e', 'l', 'l', 'o'},
25:                   {2011, 2014, 2017}, "world"};
26:     // Альтернативный вариант
27:     Aggregate2 a2_2{'h', 'e', 'l', 'l', 'o', '\0',
28:                    2011, 2014, 2017, "world"};
29:     cout << a2.hello << ' ' << a2.world << endl;
30:     cout << "Новый стандарт C++ будет принят в "
31:           << a2.impYears[2] << " году" << endl;
32:     return 0;
33: }
```

Результат

```
Pi приближенно равно: 3.14
hello world
Новый стандарт C++ будет принят в 2017 году
```

Анализ

В листинге показано, как можно использовать агрегатную инициализацию для создания экземпляров классов (или структур). Тип `Aggregate1`, определенный в строках 4–9, представляет собой класс с открытыми членами данных, а `Aggregate2`, определенный в строках 11–16, является структурой. Строки 21, 24, 25, 27 и 28 демонстрируют агрегатную инициализацию объектов `class` и `struct` соответственно. Мы обращаемся к членам класса/структуры, демонстрируя, что компилятор корректно размещает инициализирующие значения в соответствующих данных-членах. Обратите внимание, что некоторые члены являются массивами и что член `std::string` в `Aggregate2` также был инициализирован с помощью этой конструкции в строке 24.

ВНИМАНИЕ!

Агрегатная инициализация инициализирует только первый нестатический член объединения. Агрегатная инициализация объединений, объявленных в листинге 9.16, должна иметь следующий вид:

```
43:    SimpleUnion u1{ 2100 }, u2{ 'C' };  
    // В u2 член num(int) инициализируется значением 'C'  
    // (ASCII 67), хотя программист хотел инициализировать  
    // член alphabet (char)
```

Таким образом, для ясности не стоит использовать синтаксис агрегатной инициализации для объединений, несмотря на его применение в листинге 9.16.

constexpr с классами и объектами

Мы уже познакомились с `constexpr` на занятии 3, “Использование переменных и констант”, на котором узнали, что это ключевое слово предлагает мощный способ повысить производительность приложения C++. Помечая функции, работающие с константами или константными выражениями, как `constexpr`, мы поручаем компилятору вычисление этих функций и вставку их результата вместо команд, которые вычисляют результат во время выполнения приложения. Это ключевое слово может также использоваться с классами и объектами, которые рассматриваются как константы, как показано в листинге 9.18. Обратите внимание, что компилятор игнорирует ключевое слово `constexpr`, если класс или функция используется с объектами, которые не являются константными.

ЛИСТИНГ 9.18. Использование `constexpr` с классом `Human`

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: class Human  
4: {  
5:     int age;  
6:     public:  
7:     constexpr Human(int humansAge) :age(humansAge) {}  
8:     constexpr int GetAge() const { return age; }  
9: };  
10:  
11: int main()  
12: {  
13:     constexpr Human somePerson(15);  
14:     const int hisAge = somePerson.GetAge();  
15:  
16:     Human anotherPerson(45); // Неконстантное выражение  
17:  
18:     return 0;  
19: }
```

Результат

<Программа не дает вывода на экран>

Анализ

Обратите внимание на незначительные изменения в классе `Human` в строках 3–9. Теперь он использует `constexpr` в объявлениях конструктора и функции-члена `GetAge()`. Это маленькое дополнение указывает компилятору на то, что он должен создавать и использовать экземпляры класса `Human` как константное выражение, где это возможно. `somePerson` в строке 13 объявляется как константный экземпляр и используется как таковой в строке 14. Поэтому данный код будет выполняться компилятором, который будет генерировать высокопроизводительный код времени выполнения. `anotherPerson` в строке 16 не объявлен как константный экземпляр, так что связанный с ним код может не рассматриваться компилятором как константное выражение.

Резюме

На этом занятии вы познакомились с одной из самых фундаментальных концепций языка C++ — классом. Вы узнали, что класс инкапсулирует данные-члены и функции-члены для работы с ними. Вы увидели, как такие модификаторы доступа, как `public` и `private`, позволяют абстрагировать данные и функции, которые не должны быть видимы сущностям вне класса. Вы изучили концепцию копирующих конструкторов и перемещающих конструкторов, введенных стандартом C++11, которые позволяют оптимизировать нежелательные копирования. Вы также рассмотрели некоторые частные случаи, когда все эти элементы объединяются, позволяя реализовать такие проектные шаблоны, как синглтон.

Вопросы и ответы

■ В чем разница между экземпляром класса и объектом того же класса?

По существу, никакой. Создавая экземпляр класса, вы получаете объект.

■ Как лучше получить доступ к члену: используя оператор точки (.) или оператор указателя (->)?

Если у вас есть указатель на объект, то лучше использовать оператор указателя. Если объект создан в стеке как локальная переменная, то лучше подойдет оператор точки.

■ Должен ли я всегда создавать копирующий конструктор?

Если среди переменных-членов вашего класса есть интеллектуальные указатели, строковые классы или контейнеры STL, такие как `std::vector`, то копирующий конструктор по умолчанию, предоставляемый компилятором, гарантирует вызов их копирующих конструкторов. Однако, если среди членов вашего класса есть простой указатель (такой, как `int*` для динамического массива вместо `std::vector<int>`),

необходимо предоставить копирующий конструктор, гарантирующий глубокое копирование массива при вызове функции, которой объект класса передается по значению.

- **У моего класса есть только один конструктор, параметр которого был определен со значением по умолчанию. Это все еще конструктор по умолчанию?**

Да. Если экземпляр класса может быть создан без аргументов, то считается, что у класса есть конструктор по умолчанию. У класса может быть только один конструктор по умолчанию.

- **Почему в некоторых примерах на данном занятии используются такие функции, как `SetAge()`, для установки значения переменных, как, например, `Human::age`? Почему бы не сделать переменную `age` открытой и не присваивать ей значение, когда нужно?**

С технической точки зрения открытая переменная-член `Human::age` также вполне работоспособна. Однако с точки зрения дизайна данные-члены имеет смысл делать закрытыми. Функции доступа, такие как `GetAge()` или `SetAge()`, являются корректным и рекомендуемым средством обращения к таким закрытым данным-членам, позволяя выполнять проверки на ошибки, прежде чем, например, будет выполнено присваивание значения переменной `Human::age`.

- **Почему оригинал копирующему конструктору передается по ссылке?**

Прежде всего, такой копирующий конструктор ожидается компилятором. Дело в том, что копирующий конструктор, получая оригинал по значению, вызвал бы сам себя, а это привело бы к бесконечной рекурсии.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Когда я создаю экземпляр класса с помощью оператора `new`, где он создается, в стеке или в динамической памяти?
2. В моем классе есть простой указатель `int*` на динамический массив целых чисел. Будет ли размер, возвращаемый оператором `sizeof`, зависеть от количества целых чисел в динамическом массиве?
3. Все члены моего класса являются закрытыми, и для него не объявлен ни один дружественный класс или функция. Кто может обратиться к этим членам?
4. Может ли один метод класса вызвать другой?

5. Для чего используется конструктор?
6. Для чего используется деструктор?

Упражнения

1. **Отладка.** Что не так в следующем объявлении класса?

```
Class Human
{
    int age;
    string name;
public:
    Human() {}
}
```

2. Как пользователь класса из упражнения 1 может обратиться к переменной-члену `Human::age`?
3. Напишите лучшую версию класса из упражнения 1, в которой все параметры инициализируются с использованием списка инициализации в конструкторе.
4. Напишите класс `Circle`, который вычисляет площадь и периметр по радиусу, который передается классу в качестве параметра при создании экземпляра. Число π должно содержаться в константном закрытом члене, к которому нельзя обратиться извне класса.

ЗАНЯТИЕ 10

Реализация наследования

Объектно-ориентированное программирование основано на четырех важных аспектах: *инкапсуляции* (encapsulation), *абстракции* (abstraction), *наследовании* (inheritance) и *полиморфизме* (polymorphism). *Наследование* — это мощнейший способ многократного использования атрибутов и краеугольный камень полиморфизма.

На этом занятии...

- Наследование в контексте программирования
- Синтаксис наследования C++
- Открытое, закрытое и защищенное наследование
- Множественное наследование
- Проблемы, вызванные сокрытием методов базового класса и срезкой

Основы наследования

То, что Том Смит унаследовал от своих предков, — это, прежде всего, фамилию, которая и делает его Смитом. Кроме того, он унаследовал некоторые знания, которые его родители преподали ему, и навыки, приобретенные в лесу, где семья Смита живет на протяжении многих поколений. Все эти атрибуты вместе характеризуют Тома как потомственного лесоруба Смита.

В программировании вы нередко будете встречаться с ситуациями, в которых используемые компоненты обладают сходными атрибутами с незначительными различиями в деталях или поведении. Один из способов работы в такой ситуации — сделать все компоненты классами, каждый из которых реализует все атрибуты, в том числе общие. Другое решение — использовать наследование, чтобы позволить подобным классам получать общие атрибуты и общие функциональные возможности из реализующего их базового класса и переопределять их таким образом, чтобы реализовать поведение, делающее каждый класс индивидуальным. Последнее решение зачастую предпочтительнее. Добро пожаловать в мир наследования объектно-ориентированного программирования (рис. 10.1)!

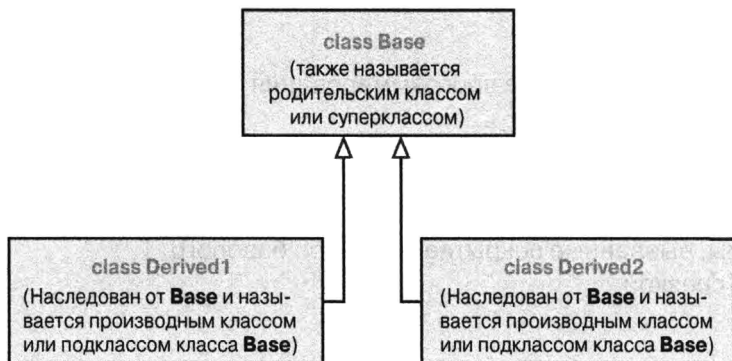


РИС. 10.1. Наследование классов

Наследование и порождение

На рис. 10.1 приведена схема отношений между *базовым классом* (base class) и порожденными из него *производными классами* (derived class). Прямо сейчас трудно представить, чем могут быть базовый и производный классы; пока просто постарайтесь понять, что производный класс унаследован от базового класса и в этом смысле является базовым классом, как Том является Смитом.

ПРИМЕЧАНИЕ

Отношение ЯВЛЯЕТСЯ между производным и базовым классами применимо только к *открытому наследованию* (public inheritance). Данное занятие начинается с рассмотрения открытого наследования, чтобы объяснить саму концепцию наследования на примере его наиболее распространенной формы, прежде чем перейти к закрытому и защищенному наследованию.

ПРИМЕЧАНИЕ

Чтобы проще объяснить эту концепцию, рассмотрим базовый класс Bird (Птица). Из класса Bird порождены классы Crow (Ворона), Parrot (Попугай) и Kiwi (Киви). Класс Bird определяет большинство основных атрибутов птицы, таких как наличие крыльев, откладывание яиц, способность летать (у большинства). Производные классы, такие как Crow, Parrot и Kiwi, наследуют эти атрибуты и корректируют их (например, класс Kiwi, представляющий нелетающую птицу, не имеет реализации метода Fly() (летать)). Еще несколько примеров наследования приведено в табл. 10.1.

ТАБЛИЦА 10.1. Примеры открытого наследования из повседневной жизни

Базовый класс	Примеры производных классов
Fish (Рыба)	Goldfish (Золотая рыбка), Carp (Карп), Tuna (Тунец) (Тунец <i>является</i> рыбой)
Mammal (Млекопитающее)	Human (Человек), Elephant (Слон), Lion (Лев), Platypus (Утконос) (Утконос <i>является</i> млекопитающим)
Bird (Птица)	Crow (Ворона), Parrot (Попугай), Ostrich (Страус), Kiwi (Киви), Platypus (Утконос) (Утконос <i>является</i> также и птицей!)
Shape (Форма)	Circle (Круг), Polygon (Многоугольник) (Круг <i>является</i> формой)
Polygon (Многоугольник)	Triangle (Треугольник), Octagon (Восьмиугольник) (Восьмиугольник <i>является</i> многоугольником, который <i>является</i> формой)

Эти примеры показывают, что если надеть объектно-ориентированные очки, то примеры наследования можно увидеть повсюду вокруг. Fish — это базовый класс для класса Tuna, поскольку тунец, как и карп, является рыбой и имеет все присущие рыбе характеристики, такие как холоднокровность. Однако тунец отличается от карпа внешним видом, скоростью плавания и тем, что он — морская рыба. Таким образом, классы Tuna и Carp наследуют общие характеристики от общего базового класса Fish, но специализируют атрибуты своего базового класса, чтобы отличаться один от другого (рис. 10.2).

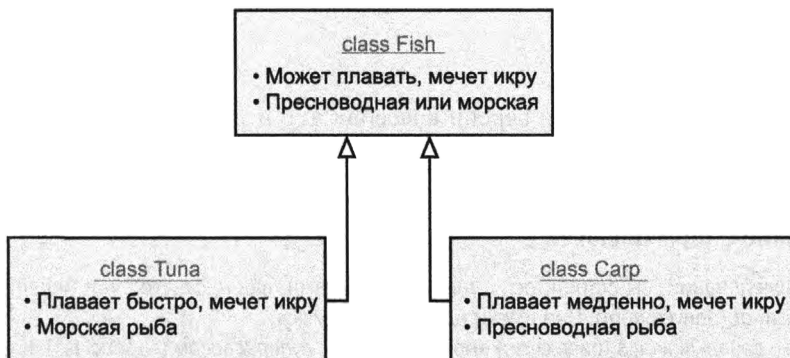


РИС. 10.2. Иерархические отношения между классами Tuna, Carp и Fish

Утконос может плавать, но все же это млекопитающее животное, поскольку кормит детенышей молоком, птица (и похож на птицу), поскольку кладет яйца, и рептилия, поскольку ядовит. Таким образом, класс `Platypus` можно представить как наследника двух базовых классов, класса `Mammal` и класса `Bird`, который наследует возможности как млекопитающих, так и птиц. Такое наследование называется *множественным наследованием* (multiple inheritance) и обсуждается позже.

Синтаксис наследования C++

Как унаследовать класс `Carp` от класса `Fish` и вообще унаследовать класс *Производный* от класса *Базовый*? В языке C++ для этого используется следующий синтаксис:

```
// Объявление базового класса
class Базовый
{
    // ... члены базового класса
};

// Объявление производного класса
class Производный: Модификатор_Доступа Базовый
{
    // ... члены производного класса
};
```

Модификатор_Доступа может быть как `public` (используется чаще всего) для отношений “производный класс является базовым классом”, так и `private` или `protected` для отношений “производный класс имеет базовый класс”.

Иерархическое представление наследования класса `Carp`, производного от класса `Fish`, может иметь следующий вид:

```
class Fish // Базовый класс
{
    // ... члены класса Fish
};

class Carp: public Fish // Производный класс
{
    // ... члены класса Carp
};
```

Пригодные для компиляции версии классов `Carp` и `Tuna`, производных от класса `Fish`, представлены в листинге 10.1.

Примечание о терминологии

Читая о наследовании, вы встретитесь с такими терминами, как *наследуется от* (inherits from) и *производный от* (derives from). Они имеют одинаковый смысл.

Точно так же *базовый класс* (base class) иногда называют *суперклассом* (super class). Класс, производный от базового, называется *производным классом* (derived class), но может упоминаться и как *подкласс* (subclass).

ЛИСТИНГ 10.1. Пример иерархии наследования

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:         bool isFreshWaterFish;
7:
8:         void Swim()
9:         {
10:             if (isFreshWaterFish)
11:                 cout << "Пресноводный" << endl;
12:             else
13:                 cout << "Морской" << endl;
14:         }
15: };
16:
17: class Tuna: public Fish
18: {
19:     public:
20:         Tuna()
21:         {
22:             isFreshWaterFish = false;
23:         }
24: };
25:
26: class Carp: public Fish
27: {
28:     public:
29:         Carp()
30:         {
31:             isFreshWaterFish = true;
32:         }
33: };
34:
35: int main()
36: {
37:     Carp myLunch;
38:     Tuna myDinner;
39:
40:     cout << "Моя еда:" << endl;
41:
42:     cout << "Обед: ";
43:     myLunch.Swim();
44:
45:     cout << "Ужин: ";
46:     myDinner.Swim();
47:
48:     return 0;
49: }
```

Результат

Моя еда:

Обед: Пресноводный

Ужин: Морской

Анализ

Обратите внимание на строки 37 и 38 в функции `main()`, где создаются объекты `myLunch` и `myDinner` классов `Carp` и `Tuna` соответственно. В строках 43 и 46 я запрашиваю свой обед и ужин о среде обитания, вызывая метод `Swim()`, который они должны поддерживать. Теперь посмотрим на определение класса `Tuna` в строках 17–24 и класса `Carp` в строках 26–33. Как можно видеть, эти классы очень компактны, а их конструкторы устанавливают соответствующие значения булева флага `Fish::isFreshWaterFish`. Позднее этот флаг используется в функции `Fish::Swim()`. Но ни один из производных классов, как мы видим, не содержит определение метода `Swim()`, который, тем не менее, успешно вызывается в функции `main()`. Дело в том, что `Swim()` является открытым членом базового класса `Fish` (от которого унаследованы рассматриваемые нами классы), определенного в строках 3–15. Открытое наследование в строках 17 и 26 автоматически предоставляет открытые члены базового класса, включая метод `Swim()`, экземплярам производных классов, с которыми мы и работаем в функции `main()`.

Модификатор доступа `protected`

В листинге 10.1 у класса `Fish` есть открытый атрибут `isFreshWaterFish`, значение которого устанавливается производными классами `Tuna` и `Carp`, чтобы настроить (или *специализировать* (`specialize`)) поведение рыбы и адаптировать ее к морской и пресной воде. Однако в коде листинга 10.1 имеется серьезный недостаток: если вы захотите, то даже в функции `main()` сможете вмешаться в значение этого флага, который помечен как `public`, а следовательно, открыт для изменения извне класса `Fish` с помощью, например, следующего кода:

```
myDinner.isFreshWaterFish = true; // Сделать тунца пресноводной рыбой!
```

Очевидно, что этого следует избегать. Необходим механизм, позволяющий определенным атрибутам в базовом классе быть доступными только для производного класса, но не для внешнего мира. Это означает, что логический флаг `isFreshWaterFish` в классе `Fish` должен быть доступен для классов `Tuna` и `Carp`, которые происходят от него, но не для функции `main()`, в которой создаются экземпляры класса `Tuna` или `Carp`. В этом случае вам пригодится ключевое слово `protected`.

ПРИМЕЧАНИЕ

Ключевое слово `protected` (защищенный), так же, как и слова `public` (открытый) и `private` (закрытый), является модификатором доступа. Объявляя атрибут как `protected`, вы фактически делаете его доступным для производных классов и друзей, одновременно делая его недоступным для всех остальных, включая функцию `main()`.

Если необходимо, чтобы определенный атрибут в базовом классе был доступен для производных классов, следует использовать модификатор доступа `protected`, как показано в листинге 10.2.

ЛИСТИНГ 10.2. Улучшенный класс `Fish`, использующий ключевое слово `protected` для предоставления его переменных-членов только производным классам

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     protected:
6:         bool isFreshWaterFish;
7:
8:     public:
9:         void Swim()
10:        {
11:            if (isFreshWaterFish)
12:                cout << "Пресноводный" << endl;
13:            else
14:                cout << "Морской" << endl;
15:        }
16: };
17:
18: class Tuna: public Fish
19: {
20:     public:
21:         Tuna()
22:         {
23:             isFreshWaterFish = false;
24:         }
25: };
26:
27: class Carp: public Fish
28: {
29:     public:
30:         Carp()
31:         {
32:             isFreshWaterFish = true;
33:         }
34: };
35:
36: int main()
37: {
38:     Carp myLunch;
39:     Tuna myDinner;
40:
41:     cout << "Моя еда:" << endl;
```



```
42:
43:     cout << "Обед: ";
44:     myLunch.Swim();
45:
46:     cout << "Ужин: ";
47:     myDinner.Swim();
48:
49:     // Снимите комментарий со строки ниже, чтобы убедиться в
50:     // недоступности защищенных членов извне иерархии класса
51:     // myLunch.isFreshWaterFish = false;
52:
53: return 0;
54: }
```

Результат

Моя еда:
Обед: Пресноводный
Ужин: Морской

Анализ

Несмотря на совпадение вывода листингов 10.1 и 10.2, здесь в класс `Fish`, определенный в строках 3–16, внесены фундаментальные изменения. Первое и самое очевидное изменение — логическая переменная-член `Fish::isFreshWaterFish` стала защищенной, а следовательно, недоступной из функции `main()`, как свидетельствует строка 51 (снимите комментарий, чтобы увидеть ошибку компиляции). Тем не менее этот параметр с модификатором доступа `protected` доступен из производных классов `Tuna` и `Carp`, что видно из строк 23 и 32 соответственно. Фактически эта небольшая программа демонстрирует использование ключевого слова `protected` для обеспечения защиты атрибута базового класса, который должен быть унаследован, от обращения извне иерархии класса.

Это очень важный аспект объектно-ориентированного программирования — комбинация абстракции данных и наследования для обеспечения безопасного наследования производными классами атрибутов базового класса, в которые не может вмешаться никто извне этой иерархической системы.

Инициализация базового класса — передача параметров базовому классу

Что если базовый класс содержит перегруженный конструктор, которому во время создания экземпляра требуется передать аргументы? Как будет инициализирован такой базовый класс при создании экземпляра производного класса? Фокус — в использовании списков инициализации и вызове соответствующего конструктора базового класса через конструктор производного класса, как демонстрирует следующий код:

```

class Base
{
public:
    Base(int someNumber) // перегруженный конструктор
    {
        // Сделать нечто с someNumber
    }
};
Class Derived: public Base
{
public:
    Derived(): Base(25) // Создать экземпляр Base с аргументом 25
    {
        // Код конструктора производного класса
    }
};

```

Этот механизм может весьма пригодиться в классе `Fish` при предоставлении логического входного параметра для его конструктора, инициализирующего переменную-член `Fish::isFreshWaterFish`. Так базовый класс `Fish` может гарантировать, что каждый производный класс вынужден будет указать, является ли рыба пресноводной или морской, как показано в листинге 10.3.

ЛИСТИНГ 10.3. Конструктор производного класса со списками инициализации

```

0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     protected:
6:         bool isFreshWaterFish; // Доступно только производным классам
7:
8:     public:
9:         // Конструктор класса Fish
10:        Fish(bool IsFreshWater) : isFreshWaterFish(IsFreshWater){}
11:
12:        void Swim()
13:        {
14:            if (isFreshWaterFish)
15:                cout << "Пресноводный" << endl;
16:            else
17:                cout << "Морской" << endl;
18:        }
19: };
20:
21: class Tuna: public Fish
22: {
23:     public:

```

```
24:     Tuna(): Fish(false) {}
25: };
26:
27: class Carp: public Fish
28: {
29:     public:
30:         Carp(): Fish(true) {}
31: };
32:
33: int main()
34: {
35:     Carp myLunch;
36:     Tuna myDinner;
37:
38:     cout << "Моя еда:" << endl;
39:
40:     cout << "Обед: ";
41:     myLunch.Swim();
42:
43:     cout << "Ужин: ";
44:     myDinner.Swim();
45:
46:     return 0;
47: }
```

Результат

```
Моя еда:
Обед: Пресноводный
Ужин: Морской
```

Анализ

Теперь у класса `Fish` есть конструктор, который получает заданный по умолчанию параметр, инициализирующий переменную `Fish::isFreshWaterFish`. Таким образом, единственная возможность создать объект класса `Fish` — это предоставить параметр, который инициализирует данный защищенный член. Так класс `Fish` гарантирует, что защищенный член класса не будет содержать случайного значения, если пользователь производного класса забудет его установить. Теперь производные классы `Tuna` и `Carp` вынуждены определить конструктор, создающий экземпляр базового класса `Fish` с правильным параметром (`true` или `false`, указывающим, пресноводная ли это рыба), как показано в строках 24 и 30 соответственно.

ПРИМЕЧАНИЕ

Как можно заметить в листинге 10.3, производный класс никогда не обращался непосредственно к логической переменной-члену `Fish::isFreshWaterFish`, несмотря на то что она является защищенной, поскольку ее значение было установлено конструктором класса `Fish`. Чтобы гарантировать максимальную безопасность, если производные классы не нуждаются в доступе к атрибуту базового класса, пометьте его как `private`. Таким образом, более безопасную версию класса можно получить, помечая член `Fish::isFreshWaterFish` как `private` (как это сделано в листинге 10.4), после чего доступ к нему имеет только сам класс `Fish`.

Перекрытие методов базового класса в производном

Если производный класс реализует те же функции с теми же возвращаемыми значениями и сигнатурами, что и базовый класс, от которого он порожден, то тем самым он перекрывает этот метод базового класса, как показано в следующем коде:

```
class Base
{
public:
    void DoSomething()
    {
        // Код реализации... Делает нечто
    }
};

class Derived:public Base
{
public:
    void DoSomething()
    {
        // Код реализации... Делает нечто иное
    }
};
```

Таким образом, если метод `DoSomething()` вызывается с использованием экземпляра класса `Derived`, то при этом никак не используется соответствующая функциональность класса `Base`.

Если классы `Tuna` и `Carp` должны реализовать собственный метод `Swim()`, который имеется также и в базовом классе как `Fish::Swim()`, то вызов этого метода в функции `main()` так, как показано в следующем отрывке листинга 10.3

```
36:     Tuna myDinner;
// ... Другие строки
44:     myDinner.Swim();
```

привел бы к выполнению локальной реализации метода `Tuna::Swim()`, которая, по существу, перекрывает метод `Fish::Swim()` базового класса. Это демонстрирует листинг 10.4.

ЛИСТИНГ 10.4. Производные классы Tuna и Carp, перекрывающие метод Swim() базового класса Fish

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: private:
6:     bool isFreshWaterFish;
7:
8: public:
9:     // Конструктор класса Fish
10:    Fish(bool isFreshWater) : isFreshWaterFish(isFreshWater){}
11:
12:    void Swim()
13:    {
14:        if (isFreshWaterFish)
15:            cout << "Пресноводный" << endl;
16:        else
17:            cout << "Морской" << endl;
18:    }
19: };
20:
21: class Tuna: public Fish
22: {
23: public:
24:    Tuna(): Fish(false) {}
25:
26:    void Swim()
27:    {
28:        cout << "Тунец быстро плавает" << endl;
29:    }
30: };
31:
32: class Carp: public Fish
33: {
34: public:
35:    Carp(): Fish(true) {}
36:
37:    void Swim()
38:    {
39:        cout << "Карп медленно плавает" << endl;
40:    }
41: };
42:
43: int main()
44: {
45:    Carp myLunch;
```

```
46:     Tuna myDinner;
47:
48:     cout << "Моя еда:" << endl;
49:
50:     cout << "Обед: ";
51:     myLunch.Swim();
52:
53:     cout << "Ужин: ";
54:     myDinner.Swim();
55:
56:     return 0;
57: }
```

Результат

```
Моя еда:
Обед: Карп медленно плавает
Ужин: Тунец быстро плавает
```

Анализ

Вывод показывает, что вызов метода `myLunch.Swim()` в строке 51 — это вызов метода `Carp::Swim()`, определенного в строках 37–40. Аналогично вызов метода `myDinner.Swim()` в строке 54 — это вызов метода `Tuna::Swim()`, определенного на строках 26–29. Другими словами, реализация метода `Fish::Swim()` в базовом классе `Fish`, показанная в строках 12–18, перекрывается идентичной функцией `Swim()`, определенной в классах `Tuna` и `Carp`, происходящих от класса `Fish`. Единственный способ вызова именно метода `Fish::Swim()` — это использовать в функции `main()` оператор разрешения области видимости (`::`) в явном вызове метода `Fish::Swim()`, как будет показано позже на этом занятии.

Вызов перекрытых методов базового класса

В листинге 10.4 вы видели пример производного класса `Tuna`, переопределяющего функцию `Swim()` из класса `Fish` путем реализации собственной версии той же функции. Таким образом:

```
Tuna myDinner;
myDinner.Swim(); // Будет вызван Tuna::Swim()
```

Если в листинге 10.4 вы захотите вызвать функцию `Fish::Swim()` в функции `main()`, то используйте оператор разрешения области видимости (`::`) со следующим синтаксисом:

```
myDinner.Fish::Swim(); // Вызов Fish::Swim() для экземпляра Tuna
```

В листинге 10.5 показан вызов члена базового класса с использованием экземпляра производного класса.

Вызов методов базового класса в производном классе

Обычно метод `Fish::Swim()` содержит обобщенную реализацию, применимую ко всем рыбам, включая тунцов и карпов. Если специализированные реализации методов `Tuna::Swim()` и `Carp::Swim()` хотят использовать эту обобщенную реализацию метода базового класса `Fish::Swim()`, они могут сделать это с помощью оператора разрешения области видимости (`::`), как показано в следующем фрагменте:

```
class Carp: public Fish
{
    public:
        Carp(): Fish(true) {}

        void Swim()
        {
            cout << "Карп медленно плавает" << endl;
            Fish::Swim(); // Использование оператора ::
        }
};
```

Этот подход использован в листинге 10.5.

ЛИСТИНГ 10.5. Использование оператора `::` для вызова методов базового класса из методов производных классов и функции `main()`

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     private:
6:         bool isFreshWaterFish;
7:
8:     public:
9:         // конструктор класса Fish
10:        Fish(bool isFreshWater) : isFreshWaterFish(isFreshWater){}
11:
12:        void Swim()
13:        {
14:            if (isFreshWaterFish)
15:                cout << "Пресноводный" << endl;
16:            else
17:                cout << "Морской" << endl;
18:        }
19: };
20:
21: class Tuna: public Fish
22: {
```

```
23: public:
24:     Tuna(): Fish(false) {}
25:
26:     void Swim()
27:     {
28:         cout << "Тунец плавает быстро" << endl;
29:     }
30: };
31:
32: class Carp: public Fish
33: {
34:     public:
35:         Carp(): Fish(true) {}
36:
37:         void Swim()
38:         {
39:             cout << "Карп плавает медленно" << endl;
40:             Fish::Swim();
41:         }
42: };
43:
44: int main()
45: {
46:     Carp myLunch;
47:     Tuna myDinner;
48:
49:     cout << "Моя еда:" << endl;
50:
51:     cout << "Обед: ";
52:     myLunch.Swim();
53:
54:     cout << "Ужин: ";
55:     myDinner.Fish::Swim();
56:
57:     return 0;
58: }
```

Результат

```
Моя еда:
Обед: Карп плавает медленно
Пресноводный
Ужин: Морской
```

Анализ

Метод `Carp::Swim()` в строках 37–41 демонстрирует вызов функции `Fish::Swim()` базового класса с использованием оператора разрешения области видимости (`::`).

В строке 55 показана возможность использования оператора разрешения области видимости для вызова метода базового класса `Fish::Swim()` из функции `main()` с использованием объекта производного класса, в данном случае — `Tuna`.

Производный класс, скрывающий методы базового класса

Перекрытие может принять экстремальный характер, когда метод `Tuna::Swim()` потенциально способен скрыть все перегруженные версии функции `Fish::Swim()`, приводя к неудаче компиляции при использовании перегруженных функций, как показано в листинге 10.6.

ЛИСТИНГ 10.6. Соккрытие методом `Tuna::Swim()` перегруженного метода `Fish::Swim(bool)`

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:         void Swim()
7:         {
8:             cout << "Рыба плавает... !" << endl;
9:         }
10:
11:         void Swim(bool isFreshWaterFish)
12:         {
13:             if (isFreshWaterFish)
14:                 cout << "Пресноводный" << endl;
15:             else
16:                 cout << "Морской" << endl;
17:         }
18: };
19:
20: class Tuna: public Fish
21: {
22:     public:
23:         void Swim()
24:         {
25:             cout << "Тунец плавает быстро" << endl;
26:         }
27: };
28:
29: int main()
30: {
31:     Tuna myDinner;
32:
```

```
33:     cout << "Моя еда:" << endl;
34:
35:     // myDinner.Swim(false); // Ошибка компиляции: Fish::Swim(bool)
                                   // скрыт методом Tuna::Swim()
36:     myDinner.Swim();
37:
38:     return 0;
39: }
```

Результат

Моя еда:
Тунец плавает быстро

Анализ

Эта версия класса `Fish` немного отличается от тех, которые вы видели до сих пор. Кроме минимизации версии для объяснения текущей проблемы, данная версия класса `Fish` содержит два перегруженных метода `Swim()`: один не получает никаких параметров (строки 6–9), а другой получает параметр типа `bool` (строки 11–17). Поскольку класс `Tuna` наследуется от класса `Fish` открыто (строка 20), кажется, что обе версии метода `Fish::Swim()` будут доступны через экземпляр класса `Tuna`. Однако в результате того, что класс `Tuna` реализует собственную версию метода `Tuna::Swim()` (строки 23–26), функция `Fish::Swim(bool)` оказывается скрытой от компилятора. Если убрать комментарий из строки 35, будет получена ошибка времени компиляции.

Для того чтобы вызвать функцию `Fish::Swim(bool)` через экземпляр класса `Tuna`, можно прибегнуть к следующим решениям.

- Решение 1. Использовать оператор разрешения области видимости в функции `main()`:

```
myDinner.Fish::Swim();
```

- Решение 2. Использовать в классе `Tuna` ключевое слово `using`, чтобы показать скрытые методы `Swim()` в классе `Fish`:

```
class Tuna: public Fish
{
    public:
        using Fish::Swim; // Раскрытие скрытых методов Swim()
                           // в базовом классе Fish

        void Swim()
        {
            cout << "Тунец плавает быстро" << endl;
        }
};
```

- Решение 3. Переопределить все перегруженные варианты метода `Swim()` в классе `Tuna` (например, при необходимости вызывая метод `Fish::Swim(...)` в `Tuna::Swim(...)`):

```
class Tuna: public Fish
{
public:
    void Swim(bool isFreshWaterFish)
    {
        Fish::Swim(isFreshWaterFish);
    }

    void Swim()
    {
        cout << "Тунец плавает быстро" << endl;
    }
};
```

Порядок конструирования

При создании объекта класса `Tuna`, производного от класса `Fish`, когда будет вызван конструктор класса `Tuna`: до или после конструктора класса `Fish`? Кроме того, каков при конструировании экземпляра порядок создания таких его атрибутов, как `Fish::isFreshWaterFish`? К счастью, последовательность инстанцирования строго стандартизована. Объекты базового класса создаются до производного класса. Таким образом, первой создается часть `Fish` объекта класса `Tuna`, так, чтобы ее члены, в частности открытые и защищенные, были готовы к использованию, когда будет создаваться `Tuna`. При инстанцировании классов `Fish` и `Tuna` атрибуты, такие как `Fish::isFreshWaterFish`, создаются до вызова конструктора `Fish::Fish()`, гарантируя существование атрибутов к моменту начала работы с ними конструктора. То же самое относится и к конструктору `Tuna::Tuna()`.

Порядок деструкции

Когда экземпляр класса `Tuna` выходит из области видимости, последовательность деструкции противоположна последовательности конструкции. В листинге 10.7 приведен простой пример, демонстрирующий последовательность конструкции и деструкции.

ЛИСТИНГ 10.7. Порядок конструкции и деструкции базового класса, производного класса и его членов

```
0: #include <iostream>
1: using namespace std;
2:
3: class FishDummyMember
4: {
```

```
5: public:
6:     FishDummyMember()
7:     {
8:         cout << "Конструктор FishDummyMember" << endl;
9:     }
10:
11:     ~FishDummyMember()
12:     {
13:         cout << "Деструктор FishDummyMember" << endl;
14:     }
15: };
16:
17: class Fish
18: {
19:     protected:
20:         FishDummyMember dummy;
21:
22:     public:
23:         // Конструктор класса Fish
24:         Fish()
25:         {
26:             cout << "Конструктор Fish" << endl;
27:         }
28:
29:         ~Fish()
30:         {
31:             cout << "Деструктор Fish" << endl;
32:         }
33: };
34:
35: class TunaDummyMember
36: {
37:     public:
38:         TunaDummyMember()
39:         {
40:             cout << "Конструктор TunaDummyMember" << endl;
41:         }
42:
43:         ~TunaDummyMember()
44:         {
45:             cout << "Деструктор TunaDummyMember" << endl;
46:         }
47: };
48:
49:
50: class Tuna: public Fish
51: {
52:     private:
53:         TunaDummyMember dummy;
```

```
54:
55: public:
56:     Tuna()
57:     {
58:         cout << "Конструктор Tuna" << endl;
59:     }
60:     ~Tuna()
61:     {
62:         cout << "Деструктор Tuna" << endl;
63:     }
64:
65: };
66:
67: int main()
68: {
69:     Tuna myDinner;
70: }
```

Результат

```
Конструктор FishDummyMember
Конструктор Fish
Конструктор TunaDummyMember
Конструктор Tuna
Деструктор Tuna
Деструктор TunaDummyMember
Деструктор Fish
Деструктор FishDummyMember
```

Анализ

Функция `main()` в строках 67–70 поразительно мала по сравнению с объемом создаваемого ею вывода. Все эти строки выводятся при создании экземпляра класса `Tuna`, поскольку вывод в поток `cout` вставлен в конструкторы и деструкторы всех задействованных при этом объектов. Для демонстрации создания и удаления переменных-членов определены два вымышленных класса, `FishDummyMember` и `TunaDummyMember`, в конструкторах и деструкторах которых осуществляется вывод соответствующих строк в `cout`. Классы `Fish` и `Tuna` содержат члены, которые представляют собой экземпляры этих вымышленных классов (строки 20 и 53). Вывод показывает, что создание объекта класса `Tuna` фактически начинается с вершины иерархии. Так, первой создается часть базового класса `Fish` в составе класса `Tuna`, при этом вначале создаются его члены, такие как `Fish::dummy`. Далее, после создания таких атрибутов, как `dummy`, выполняется конструктор класса `Fish`. После создания экземпляра базового класса продолжается создание экземпляра `Tuna`, которое начинается с создания экземпляра `Tuna::dummy` и завершается выполнением кода конструктора `Tuna::Tuna()`. Вывод также демонстрирует, что последовательность удаления прямо противоположна последовательности создания.

Закрытое наследование

Закрытое наследование (private inheritance) отличается от открытого (рассматривавшегося до сих пор) тем, что в строке объявления производного класса используется ключевое слово `private`:

```
class Base
{
    // ... переменные-члены и методы базового класса
};

class Derived: private Base // закрытое наследование
{
    // ... переменные-члены и методы производного класса
};
```

Закрытое наследование базового класса означает, что все открытые члены и атрибуты базового класса являются закрытыми (т.е. недоступными) для всех, кроме экземпляра производного класса. Другими словами, даже открытые члены и методы класса `Base` могут быть использованы только классом `Derived`, но ни кем иным, владеющим экземпляром класса `Derived`.

Это резко контрастирует с примерами класса `Tuna` и его базового класса `Fish`, которые мы рассматривали начиная с листинга 10.1. Функция `main()` в листинге 10.1 может вызвать функцию `Fish::Swim()` у экземпляра класса `Tuna`, поскольку функция `Fish::Swim()` является открытым методом, а класс `Tuna` является производным от класса `Fish` с использованием открытого наследования. Попробуйте заменить ключевое слово `public` ключевым словом `private` в строке 17, и вы получите сбой компиляции.

Таким образом, для мира вне иерархии наследования закрытое наследование, по существу, не означает отношение *является* (is-a) (вообразите тунца, который не может плавать!). Поскольку закрытое наследование позволяет использовать атрибуты и методы базового класса только производным от него классам, мы получаем отношение *содержит* (has-a). В окружающем мире есть множество примеров закрытого наследования (табл. 10.2).

ТАБЛИЦА 10.2. Примеры закрытого наследования из повседневной жизни

Базовый класс		Примеры производных классов	
Motor	(Мотор)	Car	(Автомобиль <i>содержит</i> мотор)
Heart	(Сердце)	Mammal	(Млекопитающее <i>содержит</i> сердце)
Refill	(Стержень)	Pen	(Ручка <i>содержит</i> стержень)
Moon	(Луна)	Sky	(Небо <i>содержит</i> Луну)

Давайте рассмотрим закрытое наследование на примере отношений автомобиля с его мотором (листинг 10.8).

ЛИСТИНГ 10.8. Класс Car, связанный с классом Motor закрытым наследованием

```
0: #include <iostream>
1: using namespace std;
2:
3: class Motor
4: {
5:     public:
6:         void SwitchIgnition()
7:         {
8:             cout << "Зажигание включено" << endl;
9:         }
10:        void PumpFuel()
11:        {
12:            cout << "Топливо в цилиндрах" << endl;
13:        }
14:        void FireCylinders()
15:        {
16:            cout << "P-p-p-p-p-p-p..." << endl;
17:        }
18: };
19:
20: class Car:private Motor
21: {
22:     public:
23:         void Move()
24:         {
25:             SwitchIgnition();
26:             PumpFuel();
27:             FireCylinders();
28:         }
29: };
30:
31: int main()
32: {
33:     Car myDreamCar;
34:     myDreamCar.Move();
35:
36:     return 0;
37: }
```

Результат

Зажигание включено
Топливо в цилиндрах
P-p-p-p-p-p-p...

Анализ

Класс `Motor`, определенный в строках 3–18, очень прост: он содержит три открытые функции-члена, включая зажигание (`SwitchIgnition()`), подачу топлива (`PumpFuel()`) и запуск (`FireCylinders()`). Класс `Car` наследует класс `Motor` с использованием ключевого слова `private` (строка 20). Таким образом, открытая функция `Car::Move()` обращается к членам базового класса `Motor`. Если вы попытаетесь вставить в функцию `main()` строку

```
myDreamCar.PumpFuel();
```

то получите при компиляции ошибку с сообщением *error C2247: Motor::PumpFuel not accessible because 'Car' uses 'private' to inherit from 'Motor'* (ошибка C2247: `Motor::PumpFuel` недоступен, поскольку `'Car'` использует `'private'` при наследовании от `'Motor'`).

ПРИМЕЧАНИЕ

Если от класса `Car` будет наследован другой класс, например `RaceCar`, то, независимо от характера наследования, у класса `RaceCar` не будет доступа к открытым членам и методам базового класса `Motor`. Дело в том, что отношения наследования между классами `Car` и `Motor` имеют закрытый характер, а значит, доступ для всех остальных, кроме класса `Car`, будет закрытым (т.е. доступа не будет) — даже к открытым членам базового класса. Другими словами, при принятии компилятором решения о том, должен ли у некоего класса быть доступ к открытым или защищенным членам базового класса, доминирует наиболее ограничивающий модификатор доступа.

Защищенное наследование

Защищенное наследование отличается от открытого наличием ключевого слова `protected` в строке объявления производного класса:

```
class Base
{
    // ... переменные-члены и методы базового класса
};

class Derived: protected Base // Защищенное наследование
{
    // ... переменные-члены и методы производного класса
};
```

Защищенное наследование подобно закрытому в следующем отношении.

- Реализует отношение *содержит* (has-a).
- Позволяет производному классу обращаться ко всем открытым и защищенным членам базового класса.
- Вне иерархии наследования с помощью экземпляра производного класса нельзя обратиться к открытым членам базового класса.

Но защищенное наследование все же отличается от закрытого, когда речь идет о следующем производном классе, унаследованном от данного производного класса:

```
class Derived2: protected Derived
{
    // Имеет доступ к открытым и защищенным членам Base
};
```

Иерархия защищенного наследования позволяет подклассу производного класса (т.е. классу Derived2) обращаться к открытым и защищенным членам базового класса (листинг 10.9). Это было бы невозможно, если бы при наследовании классом Derived класса Base использовалось ключевое слово `private`.

ЛИСТИНГ 10.9. RaceCar — подкласс класса Car (наследника Motor)
при защищенном наследовании

```
0: #include <iostream>
1: using namespace std;
2:
3: class Motor
4: {
5: public:
6:     void SwitchIgnition()
7:     {
8:         cout << "Зажигание включено" << endl;
9:     }
10:    void PumpFuel()
11:    {
12:        cout << "Топливо в цилиндрах" << endl;
13:    }
14:    void FireCylinders()
15:    {
16:        cout << "P-p-p-p-p-p-p..." << endl;
17:    }
18: };
19:
20: class Car:protected Motor
21: {
22: public:
23:     void Move()
24:     {
25:         SwitchIgnition();
26:         PumpFuel();
27:         FireCylinders();
28:     }
29: };
30:
```

```

31: class RaceCar:protected Car
32: {
33: public:
34:     void Move()
35:     {
36:         SwitchIgnition(); // RaceCar имеет доступ к членам класса
37:         PumpFuel();       // Motor благодаря защищенному
38:         FireCylinders();  // наследованию между RaceCar и Car и
39:         FireCylinders();  // между Car и Motor
40:         FireCylinders();
41:     }
42: };
43:
44: int main()
45: {
46:     RaceCar myDreamCar;
47:     myDreamCar.Move();
48:
49:     return 0;
50: }

```

Результат

```

Зажигание включено
Топливо в цилиндрах
P-r-p-r-p-r-p-r...
P-r-p-r-p-r-p-r...
P-r-p-r-p-r-p-r...

```

Анализ

Класс Car защищенно наследует класс Motor (строка 20). Класс RaceCar защищенно наследует класс Car (строка 31). Как можно заметить, реализация метода `RaceCar::Move()` использует открытые методы, определенные в базовом классе `Motor`. Этот доступ к первому базовому классу `Motor` через промежуточный базовый класс `Car` обеспечивают отношения между классами `Car` и `Motor`. Если бы это было закрытое наследование, а не защищенное, то у производного класса не было бы доступа к открытым членам `Motor`, поскольку компилятор выбирает самый ограничивающий из использованных модификаторов доступа. Обратите внимание, что характер отношений между классами `Car` и `RaceCar` не имеет значения при доступе к базовому классу. Так, даже если в строке 31 заменить ключевое слово `protected` словом `public` или `private`, исход компиляции этой программы остается неизменным.

ВНИМАНИЕ!

Используйте закрытое или защищенное наследование только по мере необходимости. В большинстве случаев, когда используется закрытое наследование (как у классов `Car` и `Motor`), базовый класс может также быть атрибутом (членом) класса `Car`, а не суперклассом. При наследовании от класса `Motor` вы, по существу, ограничили свой класс `Car` наличием только одного мотора, без какого-либо существенного выигрыша от наличия экземпляра класса `Motor` как закрытого члена.

Автомобили развиваются, и сейчас не редкость гибридные автомобили, например в дополнение к обычному мотору может применяться газовый или электрический. Наша иерархия наследования для класса Car оказалась бы узким местом, попытайтесь мы последовать за такими разработками.

ПРИМЕЧАНИЕ

Наличие экземпляра класса `Motor` как закрытого члена, вместо наследования от него, называется *композицией* (composition) или *агрегацией* (aggregation). Такой класс `Car` выглядел бы следующим образом:

```
class Car
{
private:
    Motor heartOfCar;

public:
    void Move()
    {
        heartOfCar.SwitchIgnition();
        heartOfCar.PumpFuel();
        heartOfCar.FireCylinders();
    }
};
```

Такое решение может оказаться лучшим дизайном, поскольку позволяет легко добавлять к существующему классу `Car` больше моторов как атрибутов, не изменяя его иерархию наследования или предоставляемые клиентам возможности.

Проблема срезки

Что будет, если программист сделает так?

```
Derived objectDerived;
Base objectBase = objectDerived;
```

Или вот так?

```
void FuncUseBase(Base input);  
...  
Derived objectDerived;  
FuncUseBase(objectDerived); // objectDerived будет срезан при  
                             // копировании во время вызова функции
```

В обоих случаях объект производного класса копируется в другой объект базового класса, явно при присваивании или косвенно при передаче в качестве аргумента. В таких случаях компилятор копирует из объекта `objectDerived` только часть, соответствующую классу `Base`, а не весь объект. При этом будет потеряна информация, содержащаяся в членах-данных, относящихся к классу `Derived`. Это непредвиденное и нежелательное сокращение части данных, делающих производный класс специализацией базового, называется *срезкой* (slicing).

ВНИМАНИЕ!

Во избежание срезки не передавайте параметры по значению. Передавайте их как указатели на базовый класс или как (константную) ссылку на него.

Множественное наследование

Ранее в этом занятии упоминалось о том, что иногда может пригодиться *множественное наследование* (multiple inheritance), как в случае с утконосом. Утконос — частично млекопитающее, частично птица, частично рептилия. Для таких случаев язык C++ позволяет унаследовать класс от двух и более базовых классов:

```
class Производный: Модификатор_Доступа Базовый_класс_1,
                  Модификатор_Доступа Базовый_класс_2
{
    // Члены класса
};
```

Схема класса для утконоса на рис. 10.3 выглядит совсем не так, как таковая для классов `Tuna` и `Carp` (см. рис. 10.2).

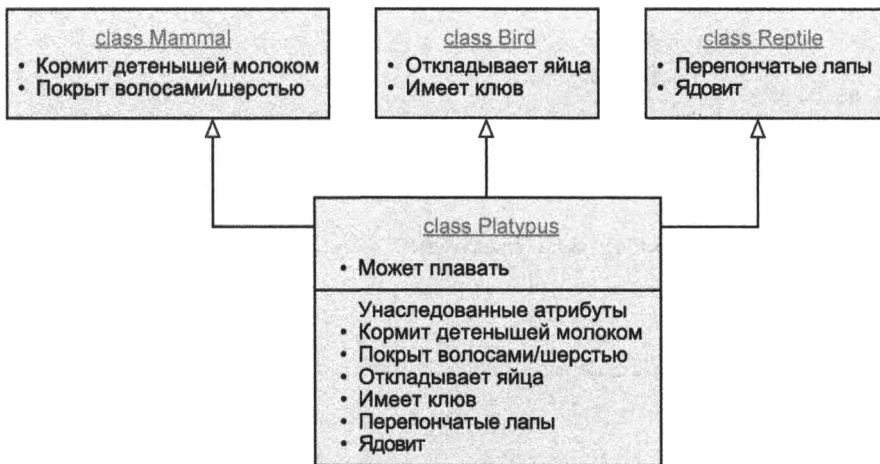


РИС. 10.3. Отношения между классом `Platypus` и классами `Mammal`, `Reptile` и `Bird`

Таким образом, синтаксическое представление C++ класса `Platypus` будет следующим:

```
class Platypus: public Mammal, public Reptile, public Bird
{
    // ... члены класса Platypus
};
```

Класс Platypus, демонстрирующий множественное наследование, представлен в листинге 10.10.

ЛИСТИНГ 10.10. Использование множественного наследования для моделирования утконоса, являющегося млекопитающим, птицей и рептилией

```
0: #include <iostream>
1: using namespace std;
2:
3: class Mammal
4: {
5: public:
6:     void FeedBabyMilk()
7:     {
8:         cout << "Млекопитающее: люблю молоко!" << endl;
9:     }
10: };
11:
12: class Reptile
13: {
14: public:
15:     void SpitVenom()
16:     {
17:         cout << "Рептилия: плюну ядом!" << endl;
18:     }
19: };
20:
21: class Bird
22: {
23: public:
24:     void LayEggs()
25:     {
26:         cout << "Птица: яйца отложены!" << endl;
27:     }
28: };
29:
30: class Platypus: public Mammal, public Bird, public Reptile
31: {
32: public:
33:     void Swim()
34:     {
35:         cout << "Утконос: я умею плавать!" << endl;
36:     }
37: };
38:
```

```
39: int main()
40: {
41:     Platypus realFreak;
42:     realFreak.LayEggs();
43:     realFreak.FeedBabyMilk();
44:     realFreak.SpitVenom();
45:     realFreak.Swim();
46:
47:     return 0;
48: }
```

Результат

```
Птица: яйца отложены!
Млекопитающее: люблю молоко!
Рептилия: плюну ядом!
Утконос: я умею плавать!
```

Анализ

Определение свойств класса `Platypus` весьма компактно (строки 30–37). По существу, класс просто наследует их от трех других: `Mammal`, `Reptile` и `Bird`. Функция `main()` в строках 41–44 способна обратиться к трем индивидуальным возможностям базовых классов, используя объект `realFreak` производного класса `Platypus`. Кроме вызова функций, унаследованных от классов `Mammal`, `Bird` и `Reptile`, функция `main()` в строке 45 вызывает метод `Platypus::Swim()`. Эта программа демонстрирует синтаксис множественного наследования, а также то, что производный класс предоставляет возможность доступа к открытым членам (в данном случае — к методам) своих базовых классов.

Запрет наследования с помощью ключевого слова `final`

Начиная с C++11 компиляторы поддерживают спецификатор `final`. Он используется для указания того, что класс объявлен как последний в иерархии наследования и не может использоваться в качестве базового класса. Например, в листинге 10.10 класс утконоса можно объявить как `final`, тем самым блокируя возможность его наследования. Версия класса утконоса из листинга 10.10, объявленная как `final`, будет выглядеть следующим образом:

```
class Platypus final: public Mammal, public Bird, public Reptile {
public:
    void Swim() {
        cout << "Утконос: я умею плавать!" << endl;
    }
};
```

В дополнение к классам ключевое слово `final` может использоваться с функциями-членами для управления полиморфным поведением. Этот вопрос рассматривается на занятии 11, “Полиморфизм”.

ПРИМЕЧАНИЕ

Утконос может плавать, но он — не рыба. Следовательно, в листинге 10.10 мы не стали наследовать `Platypus` заодно и от класса `Fish`, чтобы просто воспользоваться существующим методом `Fish::Swim()`. Принимая решения о дизайне, не забывайте, что открытое наследование должно также отражать отношение *является* и не должно использоваться без достаточных оснований, просто для решения текущих задач, связанных с повторным использованием кода. Эти цели могут быть достигнуты и по-другому.

РЕКОМЕНДУЕТСЯ

Создавайте открытую иерархию наследования для установки отношений *является*.

Создавайте закрытую или защищенную иерархию наследования для установки отношений *содержит*.

Помните: открытое наследование означает, что у классов, производных от производного класса, есть доступ к открытым и защищенным членам базового класса. Объект производного класса может использоваться для доступа к открытым членам базового класса.

Помните: закрытое наследование означает, что даже классы, производные от производного класса, не имеют доступа к членам базового класса.

Помните: защищенное наследование означает, что у классов, производных от производного класса, есть доступ к открытым и защищенным методам базового класса. Объект производного класса не может использоваться для доступа к открытым членам базового класса.

Помните: независимо от характера наследственных отношений, закрытые члены в базовом классе недоступны никаким производным классам.

НЕ РЕКОМЕНДУЕТСЯ

Не создавайте иерархию наследования только лишь для повторного использования тривиальных функций.

Не используйте закрытое и открытое наследование без разбора, поскольку впоследствии это может стать узким местом архитектуры вашего приложения для будущего масштабирования.

Не создавайте функции производного класса (с тем же именем, но другим набором входных параметров), которые непреднамеренно скрывают таковые в базовом классе.

Резюме

На сегодняшнем занятии рассматривались основы наследования в языке C++. Вы узнали, что открытое наследование — это отношение *является* между производным и базовым классами, а закрытое и защищенное наследование создает отношение *имеет*. Вы видели, что применение модификатора доступа `protected` предоставляет доступ к членам базового класса только для производного класса, оставляя их скрытыми от классов вне иерархии наследования. Вы узнали, что защищенное наследование отличается от закрытого тем, что производные классы производного класса могут обращаться к открытым и защищенным членам базового класса, что невозможно при закрытом наследовании. Вы изучили основы перекрытия методов и их сокрытия, а также узнали, как избежать нежелательного сокрытия метода с помощью ключевого слова `using`.

Теперь вы готовы ответить на несколько вопросов, а затем перейти к изучению следующего столпа объектно-ориентированного программирования — полиморфизму.

Вопросы и ответы

- Меня попросили смоделировать класс `Mammal` наряду с классами еще нескольких млекопитающих: `Human`, `Lion` и `Whale`. Должен ли я использовать иерархию наследования, и если должен, то какую?

Человек, лев и кит — все млекопитающие и, по существу, поддерживают отношение *является*. Используйте открытое наследование, в котором класс `Mammal` будет базовым, а классы `Human`, `Lion` и `Whale` — производными от него.

- В чем разница между терминами *производный класс* и *подкласс*?

По сути, никакой разницы нет. Оба термина подразумевают класс, который порожден от базового класса, т.е. специализирует его.

- Производный класс использует открытое наследование в отношении своего базового класса. Может ли он обратиться к закрытым членам базового класса?

Нет. Компилятор всегда гарантирует, что самые ограничивающие из использованных модификаторов доступа останутся в силе. Независимо от характера наследования закрытые члены класса никогда не предоставляются (т.е. недоступны) вне класса. Исключение — классы и функции, которые были объявлены дружественными (`friend`).

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”.

Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Я хочу, чтобы некоторые члены базового класса были доступны для производного класса, но не вне иерархии классов. Какой модификатор доступа мне использовать?
2. Что будет, если я передам объект производного класса по значению функции, ожидающей в качестве параметра базовый класс?
3. Что лучше — закрытое наследование или композиция?
4. Чем ключевое слово `using` может помочь мне в иерархии наследования?
5. Класс `Derived` закрыто наследуется от класса `Base`. Другой класс, `SubDerived`, открыто наследуется от класса `Derived`. Может ли класс `SubDerived` обратиться к открытым членам класса `Base`?

Упражнения

1. В каком порядке вызываются конструкторы для класса `Platypus` из листинга 10.10?
2. Как классы `Polygon` (Многоугольник), `Triangle` (Треугольник) и `Shape` (Форма) связаны один с другим?
3. Класс `D2` является производным от класса `D1`, который является производным от класса `Base`. Какой модификатор доступа следует использовать и где его расположить, чтобы запретить классу `D2` обращаться к открытым членам класса `Base`?
4. Каков характер наследования в этом фрагменте кода?

```
class Derived: Base
{
    // ... члены класса Derived
};
```

5. **Отладка.** Что неправильно в этом коде:

```
class Derived: public Base
{
    // ... члены класса Derived
};

void SomeFunc(Base value)
{
    // ...
}
```

ЗАНЯТИЕ 11

Полиморфизм

Изучив основы наследования и создания иерархии наследования, а также разобравшись, что открытое наследование, по существу, моделирует отношения *является*, можно переходить к применению этих знаний при изучении святой чаши Грааля объектно-ориентированного программирования — полиморфизма.

На этом занятии...

- Что означает термин *полиморфизм*
- Что делают виртуальные функции и как их использовать
- Что такое абстрактные классы и как их объявлять
- Что такое виртуальное наследование и где оно необходимо

Основы полиморфизма

“Поли” в переводе с греческого языка означает *много*, а “морф” — *форма*. *Полиморфизм* (polymorphism) — это возможность объектно-ориентированных языков, позволяющая аналогичными способами обрабатывать объекты разных типов. Данное занятие посвящено полиморфному поведению, которое может быть реализовано на языке C++ с использованием иерархии наследования, известной также как *полиморфизм подтипов* (subtype polymorphism).

Потребность в полиморфном поведении

На занятии 10, “Реализация наследования”, вы видели, как классы Tuna и Carp наследовали открытый метод Swim() класса Fish (см. листинг 10.1). Однако классы Tuna и Carp могут предоставить собственные методы Tuna::Swim() и Carp::Swim(), чтобы тунец и карп плавали по-разному. Но поскольку каждый из них является также рыбой, пользователь экземпляра класса Tuna вполне может использовать тип базового класса для вызова метода Fish::Swim(), который выполнит только общую часть Fish::Swim(), а не полную Tuna::Swim(), даже при том что этот экземпляр базового класса Fish является частью класса Tuna. Эта проблема представлена в листинге 11.1.

ПРИМЕЧАНИЕ

Во всех примерах кода на этом занятии удалено все, что не является абсолютно необходимым для объяснения рассматриваемой темы, а количество строк кода сведено к минимуму, чтобы улучшить удобочитаемость.

При реальном программировании необходимо создавать классы правильно, а также разрабатывать осмысленные иерархии наследования, учитывающие в перспективе цели проекта и приложения.

ЛИСТИНГ 11.1. Вызов методов с помощью экземпляра базового класса Fish, принадлежащего классу Tuna

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:         void Swim()
7:         {
8:             cout << "Рыба плавает!" << endl;
9:         }
10: };
11:
12: class Tuna:public Fish
13: {
14:     public:
```

```
15:     // Перекрытие Fish::Swim
16:     void Swim()
17:     {
18:         cout << "Тунец плавает!" << endl;
19:     }
20: };
21:
22: void MakeFishSwim(Fish& InputFish)
23: {
24:     // Вызов Fish::Swim
25:     InputFish.Swim();
26: }
27:
28: int main()
29: {
30:     Tuna myDinner;
31:
32:     // Вызов Tuna::Swim
33:     myDinner.Swim();
34:
35:     // Передача Tuna как Fish
36:     MakeFishSwim(myDinner);
37:
38:     return 0;
39: }
```

Результат

```
Тунец плавает!
Рыба плавает!
```

Анализ

Класс `Tuna` специализирует класс `Fish` через открытое наследование, как показано в строке 12. Он также перекрывает метод `Fish::Swim()`. Функция `main()` вызывает метод `Tuna::Swim()` в строке 33 непосредственно и передает объект `myDinner` (класса `Tuna`) как параметр в функцию `MakeFishSwim()`, которая интерпретирует его как ссылку `Fish&`, как видно из ее объявления (строка 22). Другими словами, вызов функции `MakeFishSwim(Fish&)` не заботит, что был передан объект класса `Tuna`; он обрабатывает его как объект класса `Fish` и вызывает метод `Fish::Swim()`. Вторая строка вывода означает, что тот же объект класса `Tuna` в этот раз привел к выводу, как у класса `Fish`, без всякой специализации (с тем же успехом это мог быть класс `Carp`).

Однако в идеале пользователь мог бы ожидать, что объект класса `Tuna` поведет себя, как тунец, даже если вызван метод `Fish::Swim()`. Другими словами, когда в строке 25 вызывается метод `InputFish.Swim()`, пользователь ожидает, что будет выполнен метод `Tuna::Swim()`. Такое полиморфное поведение, когда объект известного типа — `Fish` — может вести себя, как объект фактического типа, а именно — как

объект производного класса `Tuna`, может быть реализован, если сделать функцию `Fish::Swim()` виртуальной.

Полиморфное поведение, реализованное с помощью виртуальных функций

Доступ к объекту класса `Fish` возможен через указатель `Fish*` или по ссылке `Fish&`. Объект класса `Fish` может быть создан отдельно или как часть объекта класса `Tuna` или `Carp`, производного от класса `Fish`. Вы не знаете, как именно (да это и неважно). Вы вызываете метод `Swim()`, используя этот указатель или ссылку:

```
pFish->Swim();
myFish.Swim();
```

Вы ожидаете, что объект класса `Fish` будет плавать, как тунец, если это часть объекта класса `Tuna`, или как карп, если это часть объекта класса `Carp`, или как безымянная рыба, если объект класса `Fish` был создан не как часть специализированного класса, такого как `Tuna` или `Carp`. Вы можете обеспечить такое поведение, объявив функцию `Swim()` в базовом классе `Fish` как *виртуальную функцию* (virtual function):

```
class Базовый
{
    virtual Возвращаемый_Тип Функция(Список_параметров);
};

class Derived
{
    Возвращаемый_Тип Функция (Список_параметров);
};
```

Использование ключевого слова `virtual` означает, что компилятор обеспечивает вызов любого перекрытого варианта запрошенного метода базового класса. Таким образом, если метод `Swim()` объявлен как `virtual`, вызов `myFish.Swim()` (`myFish` имеет тип `Fish&`) приводит к вызову метода `Tuna::Swim()`, как показано в листинге 11.2.

ЛИСТИНГ 11.2. Результат объявления метода `Fish::Swim()` виртуальным

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:         virtual void Swim()
7:         {
8:             cout << "Рыба плавает!" << endl;
9:         }
10: };
11:
```

```

12: class Tuna:public Fish
13: {
14:     public:
15:         // Перекрытие Fish::Swim
16:         void Swim()
17:         {
18:             cout << "Тунец плавает!" << endl;
19:         }
20: };
21:
22: class Carp:public Fish
23: {
24:     public:
25:         // Перекрытие Fish::Swim
26:         void Swim()
27:         {
28:             cout << "Карп плавает!" << endl;
29:         }
30: };
31:
32: void MakeFishSwim(Fish& InputFish)
33: {
34:     // Вызов виртуального метода Swim()
35:     InputFish.Swim();
36: }
37:
38: int main()
39: {
40:     Tuna myDinner;
41:     Carp myLunch;
42:
43:     // Передача в качестве Fish объекта Tuna
44:     MakeFishSwim(myDinner);
45:
46:     // Передача в качестве Fish объекта Carp
47:     MakeFishSwim(myLunch);
48:
49:     return 0;
50: }

```

Результат

```

Тунец плавает!
Карп плавает!

```

Анализ

Реализация функции `MakeFishSwim(Fish&)` никак не изменилась по сравнению с листингом 11.1, но вывод получился совсем иной. Метод `Fish::Swim()` не был вызван

вообще из-за присутствия перекрытых версий метода `Tuna::Swim()` и `Carp::Swim()`, которые получили преимущество при вызове над методом `Fish::Swim()`, поскольку последний был объявлен как виртуальная функция. Это очень важный момент! Он свидетельствует о том, что, даже не зная точный тип обрабатываемого объекта, класс которого происходит от класса `Fish`, реализация метода `MakeFishSwim()` способна привести к вызову разных реализаций метода `Swim()`, определенного в различных производных классах.

Это и есть полиморфизм: обработка различных рыб как общего типа `Fish` при гарантии выполнения правильной реализации метода `Swim()`, предоставляемого производными типами.

Необходимость виртуальных деструкторов

У средств, представленных в листинге 11.1, есть и обратная сторона: непреднамеренный вызов функций базового класса из экземпляра производного, когда доступна специализация. Что будет, если функция применит оператор `delete`, используя указатель типа `Base*`, который фактически указывает на экземпляр производного класса?

Какой деструктор будет вызван? Рассмотрим листинг 11.3.

ЛИСТИНГ 11.3. Функция, вызывающая оператор `delete` для типа `Base*`

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:         Fish()
7:         {
8:             cout << "Создаем Fish" << endl;
9:         }
10:        ~Fish()
11:        {
12:            cout << "Уничтожаем Fish" << endl;
13:        }
14: };
15:
16: class Tuna:public Fish
17: {
18:     public:
19:         Tuna()
20:         {
21:             cout << "Создаем Tuna" << endl;
22:         }
23:        ~Tuna()
24:        {
25:            cout << "Уничтожаем Tuna" << endl;
26:        }
27: };
28:
```

```
29: void DeleteFishMemory(Fish* pFish)
30: {
31:     delete pFish;
32: }
33:
34: int main()
35: {
36:     cout << "Выделение динамической памяти для Tuna:" << endl;
37:     Tuna* pTuna = new Tuna;
38:     cout << "Удаление Tuna: " << endl;
39:     DeleteFishMemory(pTuna);
40:
41:     cout << "Инстанцирование Tuna в стеке:" << endl;
42:     Tuna myDinner;
43:     cout << "Выход из области видимости: " << endl;
44:
45:     return 0;
46: }
```

Результат

```
Выделение динамической памяти для Tuna:
Создаем Fish
Создаем Tuna
Удаление Tuna:
Уничтожаем Fish
Инстанцирование Tuna в стеке:
Создаем Fish
Создаем Tuna
Выход из области видимости:
Уничтожаем Tuna
Уничтожаем Fish
```

Анализ

Функция `main()` создает экземпляр класса `Tuna` в динамической памяти, используя оператор `new` в строке 37, а затем сразу освобождает выделенную память, используя вспомогательную функцию `DeleteFishMemory()` в строке 39. Для сравнения другой экземпляр класса `Tuna` создается в стеке как локальная переменная `myDinner` (строка 42) и выходит из области видимости по завершении функции `main()`. Вывод генерируется в конструкторах и деструкторах классов `Fish` и `Tuna`. Обратите внимание: несмотря на то, что обе части объекта — и часть `Tuna`, и часть `Fish` — были созданы в динамической памяти, поскольку использовался оператор `new`, при удалении был вызван только деструктор класса `Fish`, но не класса `Tuna`. Это сильно отличается от создания и удаления локального объекта `myDinner`, когда вызываются все конструкторы и деструкторы. В листинге 10.7 был представлен правильный порядок создания и удаления классов в иерархии наследования, демонстрирующий, что должны быть вызваны все деструкторы, включая деструктор `~Tuna()`. Здесь явно что-то неправильно.

Дело в том, что код деструктора производного класса, объект которого был создан в динамической памяти с помощью оператора `new`, не будет вызван при применении оператора `delete` к указателю типа `Base*`. В результате ресурсы не будут освобождены, произойдет утечка памяти и другие ненужные неприятности.

Чтобы избежать этой проблемы, следует использовать виртуальные деструкторы, как показано в листинге 11.4.

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя базового типа

```

0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:         Fish()
7:         {
8:             cout << "Создаем Fish" << endl;
9:         }
10:        virtual ~Fish() // Виртуальный деструктор!
11:        {
12:            cout << "Уничтожаем Fish" << endl;
13:        }
14: };
15:
16: class Tuna:public Fish
17: {
18:     public:
19:         Tuna()
20:         {
21:             cout << "Создаем Tuna" << endl;
22:         }
23:         ~Tuna()
24:         {
25:             cout << "Уничтожаем Tuna" << endl;
26:         }
27: };
28:
29: void DeleteFishMemory(Fish* pFish)
30: {
31:     delete pFish;
32: }
33:
34: int main()
35: {
36:     cout << "Выделение динамической памяти для Tuna:" << endl;
37:     Tuna* pTuna = new Tuna;

```

```
38:     cout << "Удаление Tuna: " << endl;
39:     DeleteFishMemory(pTuna);
40:
41:     cout << "Инстанцирование Tuna в стеке:" << endl;
42:     Tuna myDinner;
43:     cout << "Выход из области видимости: " << endl;
44:
45:     return 0;
46: }
```

Результат

Выделение динамической памяти для Tuna:
Создаем Fish
Создаем Tuna
Удаление Tuna:
Уничтожаем Tuna
Уничтожаем Fish
Инстанцирование Tuna в стеке:
Создаем Fish
Создаем Tuna
Выход из области видимости:
Уничтожаем Tuna
Уничтожаем Fish

Анализ

Единственное различие между листингами 11.4 и 11.3 — добавление ключевого слова `virtual` в строке 10, где был объявлен деструктор базового класса `Fish`. Обратите внимание, что это маленькое изменение, по существу, заставило компилятор выполнить деструктор `Tuna::~~Tuna()` в дополнение к деструктору `Fish::~~Fish()` при вызове оператора `delete` для указателя `Fish*` (который фактически указывает на объект класса `Tuna`) в строке 31. Вывод данного кода демонстрирует, что последовательность вызовов конструкторов и деструкторов одинакова независимо от того, создан ли объект класса `Tuna` в динамической памяти с использованием оператора `new`, как показано в строке 37, или в стеке, как локальная переменная (строка 42).

ПРИМЕЧАНИЕ

Всегда объявляйте деструктор базового класса как `virtual`:

```
class Base
{
public:
    virtual ~Base() {}; // Виртуальный деструктор
};
```

Это гарантирует, что будет невозможно вызвать оператор `delete` для указателя `Base*` так, чтобы не были корректно уничтожены объекты производных классов.

Как работают виртуальные функции. Понятие таблицы виртуальных функций

ПРИМЕЧАНИЕ

Необязательно изучать этот раздел, чтобы использовать полиморфизм. Если вам это не интересно, можете его не читать.

Функция `MakeFishSwim(Fish&)` в листинге 11.2 заканчивается вызовом метода `Carp::Swim()` или `Tuna::Swim()` несмотря на то, что программист вызвал в ней метод `Fish::Swim()`. Безусловно, на момент компиляции компилятору ничего не известно о характере объектов, с которыми встретится такая функция, и он не в состоянии гарантировать выполнение различных методов `Swim()` в различные моменты времени. Очевидно, решение о том, какой метод `Swim()` следует вызвать, принимается во время выполнения с использованием скрытой логики, реализующей полиморфизм и предоставляемой компилятором во время компиляции.

Рассмотрим класс `Base`, в котором объявлено N виртуальных функций:

```
class Base
{
    public:
        virtual void Func1()
        {
            // Реализация Func1
        }
        virtual void Func2()
        {
            // Реализация Func2
        }
        // ... и так далее
        virtual void FuncN()
        {
            // Реализация FuncN
        }
};
```

Класс `Derived`, производный от класса `Base`, наследует метод `Base::Func2()`, предоставляя другие виртуальные функции непосредственно из класса `Base`:

```
class Derived: public Base
{
    public:
        virtual void Func1()
        {
            // Func1 переопределяет Base::Func1
        }

        // Реализации для Func2 нет
```

```
virtual void FuncN()
{
    // Реализация FuncN
}
};
```

Компилятор видит иерархию наследования и понимает, что класс Base определяет некоторые виртуальные функции, которые перекрыты в классе Derived. Теперь компилятор должен составить таблицу, называемую *таблицей виртуальных функций* (Virtual Function Table — VFT), для каждого класса, который реализует виртуальную функцию, или производного класса, который перекрывает ее. Другими словами, классы Base и Derived получают экземпляр собственной таблицы виртуальных функций. При создании объектов этих классов инициализируется скрытый указатель (назовем его VFT*) на соответствующую таблицу VFT. Таблицу виртуальных функций можно представить как статический массив, содержащий указатели на функции, каждый из которых указывает на виртуальную функцию (или ее перекрытую версию) (рис. 11.1).

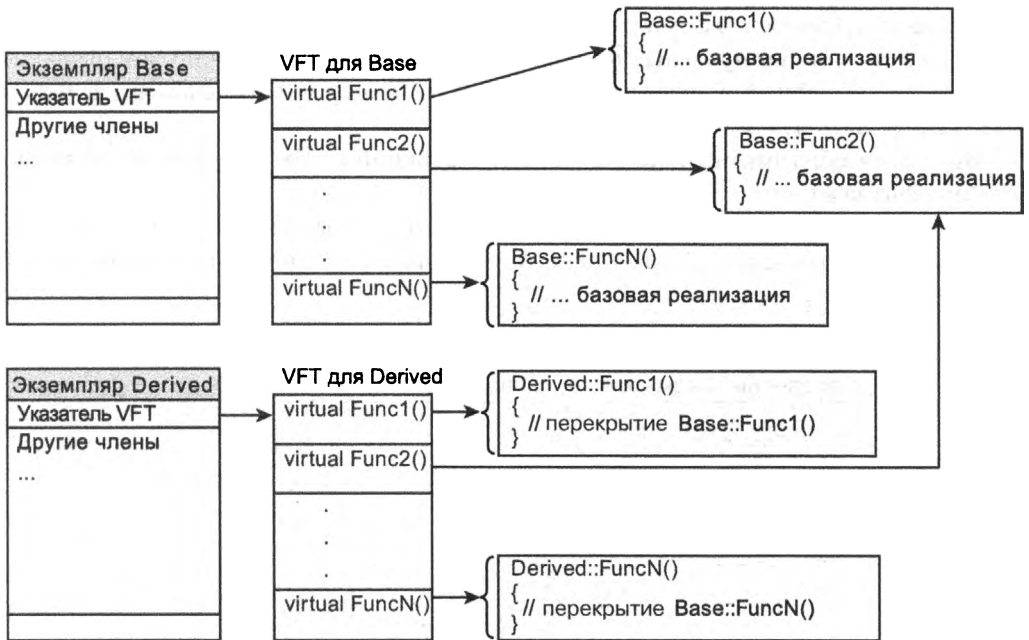


РИС. 11.1. Представление таблицы виртуальных функций для классов Derived и Base

Таким образом, все таблицы состоят из указателей на функции, каждый из которых указывает на доступную реализацию виртуальной функции. В случае класса Derived все, кроме одного указателя на функцию в его таблице VFT, указывают на локальные реализации виртуального метода в классе Derived. Класс Derived не переопределяет метод Base::Func2(), а потому соответствующий указатель на функцию указывает на реализацию в классе Base.

Это означает, что при вызове пользователем класса `Derived`

```
CDerived objDerived;
objDerived.Func2();
```

компилятор осуществляет поиск в таблице VFT класса `Derived` и обеспечивает вызов реализации `Base::Func2()`. Аналогично выполняется вызов переопределенных виртуальных методов:

```
void DoSomething(Base& objBase)
{
    objBase.Func1(); // Вызов Derived::Func1
}
int main()
{
    Derived objDerived;
    DoSomething(objDerived);
};
```

В данном случае, несмотря на то что объект `objDerived` интерпретируется из-за параметра `objBase` как экземпляр класса `Base`, указатель VFT в нем все равно указывает на таблицу виртуальных функций класса `Derived`. Таким образом, функцией `Func1()`, выполняемой через этот указатель VFT, является, конечно же, функция `Derived::Func1()`.

Вот таким образом таблицы виртуальных функций и обеспечивают реализацию полиморфизма в C++.

Листинг 11.5 доказывает существование скрытого указателя VFT, сравнивая размеры двух идентичных классов, у одного из которых есть виртуальная функция, а у другого — нет.

ЛИСТИНГ 11.5. Демонстрация наличия скрытого указателя VFT с помощью сравнения размеров двух классов

```
0: #include <iostream>
1: using namespace std;
2:
3: class SimpleClass
4: {
5:     int a, b;
6:
7: public:
8:     void FuncDoSomething() {}
9: };
10:
11: class Base
12: {
13:     int a, b;
14:
15: public:
```

```
16:     virtual void FuncDoSomething() {}
17: };
18:
19: int main()
20: {
21:     cout << "sizeof(SimpleClass) = " << sizeof(SimpleClass) << endl;
22:     cout << "sizeof(Base) = " << sizeof(Base) << endl;
23:
24:     return 0;
25: }
```

Результат в случае 32-разрядного компилятора

```
sizeof(SimpleClass) = 8
sizeof(Base) = 12
```

Результат в случае 64-разрядного компилятора

```
sizeof(SimpleClass) = 8
sizeof(Base) = 16
```

Анализ

Этот пример ограничен до минимума. Вы видите два класса, SimpleClass и Base, которые идентичны по типам и количеству членов, но функция FuncDoSomething() в классе Base объявлена как виртуальная, а в классе SimpleClass как не виртуальная. Различие лишь в добавлении ключевого слова virtual, но компилятор создает таблицу виртуальных функций для класса Base и резервирует место для указателя на нее в том же классе Base в качестве скрытого члена. Этот указатель использует 4 дополнительных байта в 32-разрядной системе автора, что и является доказательством его существования.

ПРИМЕЧАНИЕ

Язык C++ позволяет запросить указатель Base*, имеет ли он тип Derived*, с помощью оператора приведения dynamic_cast и последующего условного выполнения на основе результата этого запроса.

Этот механизм называется *идентификацией типа времени выполнения* (Run Time Type Identification – RTTI), и в идеале его следует избегать, несмотря на поддержку большинством компиляторов C++. Дело в том, что необходимость выяснять тип объекта производного класса по указателю на базовый класс обычно является плохой практикой программирования и свидетельствует о плохом проектировании.

Более подробно RTTI и оператор dynamic_cast обсуждаются на занятии 13, “Операторы приведения”.

Абстрактные классы и чисто виртуальные функции

Базовый класс, который не может быть инстанцирован (т.е. не может быть создан экземпляр этого класса), называется *абстрактным классом* (abstract base class). Цель существования такого базового класса только одна — получение производных классов. Язык C++ позволяет создать абстрактный класс, используя чисто виртуальные функции.

Функцию называют *чисто виртуальной* (pure virtual), если ее объявление имеет следующий вид:

```
class Абстрактный_Базовый
{
public:
    virtual void Некая_функция() = 0; // Чисто виртуальная функция
};
```

Это объявление, по существу, говорит компилятору о том, что функция *Некая_функция()* должна быть реализована классом, производным от класса *Абстрактный_Базовый*:

```
class Производный: public Абстрактный_Базовый
{
public:
    void Некая_функция() // Реализация функции
    {
        cout << "Реализация виртуальной функции" << endl;
    }
}
```

Таким образом, класс *Абстрактный_Базовый* выполнил свою задачу — заставил класс *Производный* предоставить реализацию виртуального метода *Некая_функция()*. Вернемся к классу *Fish*. Предположим, что тунец не может плавать быстро, поскольку класс *Tuna* не переопределил метод *Fish::Swim()*. Это ошибка реализации и большой недостаток. Сделав класс *Fish* абстрактным базовым классом с чисто виртуальной функцией *Swim()*, мы гарантируем, что класс *Tuna*, производный от класса *Fish*, реализует метод *Tuna::Swim()*, т.е. тунец будет плавать, как тунец, а не как любая рыба. Рассмотрим листинг 11.6.

ЛИСТИНГ 11.6. Класс *Fish* как абстрактный базовый класс для классов *Tuna* и *Carp*

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:         // Определение чисто виртуальной функции Swim
7:         virtual void Swim() = 0;
8: };
```

```

9:
10: class Tuna:public Fish
11: {
12:     public:
13:         void Swim()
14:         {
15:             cout << "Тунец быстро плавает в море!" << endl;
16:         }
17: };
18: class Carp:public Fish
19: {
20:     public:
21:         void Swim()
22:         {
23:             cout << "Карп медленно плавает в озере!" << endl;
24:         }
25: };
26:
27: void MakeFishSwim(Fish& inputFish)
28: {
29:     inputFish.Swim();
30: }
31:
32: int main()
33: {
34:     // Fish myFish; // Нельзя инстанцировать абстрактный класс!
35:     Carp myLunch;
36:     Tuna myDinner;
37:
38:     MakeFishSwim(myLunch);
39:     MakeFishSwim(myDinner);
40:
41:     return 0;
42: }

```

Результат

Карп медленно плавает в озере!
Тунец быстро плавает в море!

Анализ

Существенна первая (закомментированная) строка функции `main()` (строка 34). Она демонстрирует, что компилятор не позволит создать экземпляр класса `Fish`. Он ожидает чего-то более конкретного, такого как специализация класса `Fish` (класса `Tuna`, например), что имеет смысл и в реальности. Благодаря чисто виртуальной функции `Fish::Swim()`, объявленной в строке 7, оба класса, `Tuna` и `Carp`, вынуждены реализовать методы `Tuna::Swim()` и `Carp::Swim()` соответственно. Строки 27–30, в

которых реализован метод `MakeFishSwim(Fish&)`, демонстрируют, что, хотя экземпляр абстрактного класса и не может быть создан, ссылку или указатель на него вполне можно использовать. Таким образом, абстрактные классы — это очень хороший способ потребовать от всех производных классов реализации определенных функций. Если в классе `Trout` (форель), производном от класса `Fish`, забыть реализовать метод `Trout::Swim()`, компиляция потерпит неудачу.

ПРИМЕЧАНИЕ

Для абстрактных базовых классов (Abstract Base Class) часто используется аббревиатура АБК (ABC).

Абстрактные базовые классы накладывают на проект или программу определенные ограничения.

Использование виртуального наследования для решения проблемы ромба

На занятии 10, “Реализация наследования”, мы рассмотрели любопытный случай утконоса, который является млекопитающим, но частично и птицей, и рептилией. В этом случае класс утконоса `Platypus` должен происходить от классов `Mammal`, `Bird` и `Reptile`. Однако каждый из них, в свою очередь, происходит от более обобщенного класса, `Animal` (животное), как показано на рис. 11.2.

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса `Animal` в одном экземпляре класса `Platypus`

```
0: #include <iostream>
1: using namespace std;
2:
3: class Animal
4: {
5:     public:
6:         Animal()
7:         {
8:             cout << "Конструктор Animal" << endl;
9:         }
10:
11:         // Простая переменная
12:         int age;
13: };
14:
15: class Mammal:public Animal
16: {
17: };
```

```
18:
19: class Bird:public Animal
20: {
21: };
22:
23: class Reptile:public Animal
24: {
25: };
26:
27: class Platypus:public Mammal, public Bird, public Reptile
28: {
29:     public:
30:         Platypus()
31:         {
32:             cout << "Конструктор Platypus" << endl;
33:         }
34: };
35:
36: int main()
37: {
38:     Platypus duckBilledP;
39:
40:     // Раскомментировав следующую строку, получим сбой
41:     // компиляции. Есть три экземпляра age в базовых классах
42:     // duckBilledP.age = 25;
43:
44:     return 0;
45: }
```

Результат

```
Конструктор Animal
Конструктор Animal
Конструктор Animal
Конструктор Platypus
```

Анализ

Как показывает приведенный вывод, благодаря множественному наследованию у всех трех базовых классов класса Platypus (производных, в свою очередь, от класса Animal) есть свой экземпляр класса Animal. Следовательно, для каждого экземпляра класса Platypus, как показано в строке 38, автоматически создаются три экземпляра класса Animal. Это просто смешно, поскольку утконос — это одно животное, которое наследует определенные атрибуты классов Mammal, Bird и Reptile. Но проблема с количеством экземпляров базового класса Animal не ограничивается только излишним использованием памяти. У класса Animal есть целочисленный член Animal::age (который для демонстрации был оставлен открытым). При попытке получить доступ к переменной-члену Animal::age через экземпляр класса

Platypus, как показано в строке 42, вы получаете ошибку компиляции, потому что компилятор просто не знает, хотите ли вы установить значение переменной-члена `Mammal::Animal::age`, `Bird::Animal::age` или `Reptile::Animal::age`. Как ни смешно, но при желании вы можете установить значения всех трех членов:

```
duckBilledP.Mammal::Animal::age = 25;
duckBilledP.Bird::Animal::age   = 26;
duckBilledP.Reptile::Animal::age = 27;
```



РИС. 11.2. Схема класса утконоса, демонстрирующего множественное наследование

Так что же произойдет при создании экземпляра класса `Platypus`? Сколько экземпляров класса `Animal` получится в одном экземпляре класса `Platypus`? Листинг 11.7 поможет ответить на этот вопрос.

Безусловно, у одного утконоса должен быть только один возраст. Но все же класс `Platypus` должен быть производным от классов `Mammal`, `Bird` и `Reptile`. Решение — в *виртуальном наследовании* (virtual inheritance). Если вы ожидаете, что производный класс будет использоваться в иерархии наследования в качестве базового, хорошей идеей будет определение его отношения к базовому с использованием ключевого слова `virtual`:

```
class Derived1: public virtual Base
{
    // ... переменные и функции
}
```

```
};  
class Derived2: public virtual Base  
{  
    // ... переменные и функции  
};
```

Улучшенный класс `Platypus` (а фактически — улучшенные классы `Mammal`, `Bird` и `Reptile`) приведен в листинге 11.8.

ЛИСТИНГ 11.8. Как ключевое слово `virtual` в иерархии наследования ограничивает базовый класс одним экземпляром

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: class Animal  
4: {  
5:     public:  
6:         Animal()  
7:         {  
8:             cout << "Конструктор Animal" << endl;  
9:         }  
10:  
11:         // Простая переменная  
12:         int age;  
13: };  
14:  
15: class Mammal:public virtual Animal  
16: {  
17: };  
18:  
19: class Bird:public virtual Animal  
20: {  
21: };  
22:  
23: class Reptile:public virtual Animal  
24: {  
25: };  
26:  
27: class Platypus final:public Mammal, public Bird, public Reptile  
28: {  
29:     public:  
30:         Platypus()  
31:         {  
32:             cout << "Конструктор Platypus" << endl;  
33:         }  
34: };  
35:  
36: int main()  
37: {  
38:     Platypus duckBilledP;
```

```
39:
40:     // OK, есть только один Animal::age
41:     duckBilledP.age = 25;
42:
43:     return 0;
44: }
```

Результат

Конструктор Animal
Конструктор Platypus

Анализ

Бегло сравнивая приведенный вывод с выводом листинга 11.7, можно сразу же заметить, что количество экземпляров класса Animal уменьшилось до одного, что отражает факт создания только одного утконоса. Все дело в ключевом слове `virtual`, использованном в отношениях между классами `Mammal`, `Bird` и `Reptile` и гарантирующем существование только одного экземпляра общего базового класса `Animal`, если они будут объединены классом `Platypus`. Это решает много проблем; одна из них — строка 41, которая теперь компилируется, как представлено в листинге 11.7. Обратите также внимание на ключевое слово `final` в строке 27, которое гарантирует, что класс `Platypus` не может быть использован в качестве базового.

ПРИМЕЧАНИЕ

Проблема иерархии наследования, содержащей два или больше базовых класса, которые происходят от одного общего базового класса, приводящая при отсутствии виртуального наследования к необходимости разрешения неоднозначности, называется *проблемой ромба* (diamond problem).

Название “ромб”, вероятно, возникло благодаря форме схемы классов (см. рис. 11.2), в которой прямоугольники классов и связи между ними создают ромбовидную фигуру.

ПРИМЕЧАНИЕ

Ключевое слово `virtual` в языке C++ нередко используется в различных контекстах для разных целей. (На мой взгляд, кто-то просто хотел сэкономить время на придумывании ключевых слов.) Вот краткое резюме.

Объявление функции *виртуальной* означает, что будет вызвана ее перекрывающая версия из производного класса.

Отношения наследования, объявленные с использованием ключевого слова `virtual`, между классами `Derived1` и `Derived2`, производными от класса `Base`, означают, что экземпляр следующего класса, `Derived3`, производного от классов `Derived1` и `Derived2`, будет содержать только один экземпляр класса `Base`.

Таким образом, одно и то же ключевое слово `virtual` используется для реализации двух разных концепций.

Ключевое слово `override` для указания преднамеренного перекрытия

Наши версии базового класса `Fish` содержат виртуальную функцию `Swim()`, как показано в следующем коде:

```
class Fish {
public:
    virtual void Swim() {
        cout << "Рыба плавает!" << endl;
    }
};
```

Предположим, что производный класс `Tuna` определил функцию `Swim()`, но с немного иной сигнатурой — программист случайно добавил ключевое слово `const`:

```
class Tuna: public Fish {
public:
    void Swim() const {
        cout << "Тунец плавает!" << endl;
    }
};
```

Эта функция `Tuna::Swim()` на самом деле не переопределяет функцию `Fish::Swim()`: у этих функций разные сигнатуры благодаря наличию ключевого слова `const` в определении `Tuna::Swim()`. Компиляция, тем не менее, успешно выполняется, и программист может ошибочно считать, что он успешно перекрыл функцию `Swim()` в классе `Tuna`. Язык C++11 в заботах о программистах дал им новый инструмент — спецификатор `override`, который используется для проверки, была ли перекрытая функция объявлена как виртуальная в базовом классе:

```
class Tuna: public Fish {
public:
    void Swim() const
        override { // Ошибка: виртуальной функции с такой
                    // сигнатурой в классе Fish нет
        cout << "Тунец плавает!" << endl;
    }
};
```

Таким образом, спецификатор `override` предоставляет мощный способ выражения явного намерения перекрытия виртуальной функции базового класса, тем самым позволяя компилятору выполнить проверки того, что

- функция базового класса объявлена как `virtual`;
- сигнатура виртуальной функции базового класса в точности соответствует сигнатуре функции производного класса, объявленной как `override`.

Использование ключевого слова `final` для предотвращения перекрытия функции

Спецификатор `final`, введенный в C++11, был представлен на занятии 10, “Реализация наследования”. Класс, объявленный как `final`, не может использоваться в качестве базового класса. Аналогично виртуальная функция, объявленная как `final`, не может быть перекрыта в производном классе.

Таким образом, версия класса `Tuna`, которая не позволяет никакой дальнейшей специализации виртуальной функции `Swim()`, будет выглядеть следующим образом:

```
class Tuna: public Fish {
public:
    // Перекрываем Fish::Swim и делаем ее final
    void Swim() override final {
        cout << "Тунец плавает!" << endl;
    }
};
```

Эта версия класса `Tuna` может наследоваться, но функция `Swim()` при этом в классах-наследниках не может быть перекрыта:

```
class BluefinTuna final: public Tuna {
public:
    void Swim() { // Ошибка: Swim() в Tuna, объявлена
    }              // как final и не может быть перекрыта
};
```

Применение спецификаторов `override` и `final` показано в листинге 11.9.

ПРИМЕЧАНИЕ

Мы использовали `final` в объявлении класса `BluefinTuna`. Это гарантирует, что класс `BluefinTuna` не может использоваться в качестве базового класса. Таким образом, следующий код приведет к ошибке:

```
class FailedDerivation:public BluefinTuna
{
};
```

Виртуальные копирующие конструкторы?

Обратите внимание на вопросительный знак в конце заголовка данного раздела. Технически в языке C++ невозможно создать виртуальные копирующие конструкторы. Но все же можно создать коллекцию (например, статический массив) типа `Base*`, каждый элемент которого является специализацией этого типа:

```
// Классы Tuna, Carp и Trout открыто наследуют класс Fish
Fish* pFishes[3];
Fishes[0] = new Tuna();
Fishes[1] = new Carp();
Fishes[2] = new Trout();
```

Теперь присвоим его другому массиву того же типа, и виртуальный копирующий конструктор обеспечит глубокое копирование объектов производного класса, а также гарантирует, что объекты классов Tuna, Carp и Trout будут скопированы именно как объекты классов Tuna, Carp и Trout, несмотря на то что будет использован копирующий конструктор для типа Fish*.

Но это все мечты.

Виртуальные копирующие конструкторы невозможны, поскольку ключевое слово `virtual` в контексте методов базового класса, перекрываемых реализациями, доступными в производном классе, свидетельствует о полиморфном поведении во время выполнения. Конструкторы же по своей природе не полиморфны, так как способны создавать экземпляр только фиксированного типа, а следовательно, язык C++ не позволяет использовать виртуальные копирующие конструкторы.

С учетом сказанного выше появляется хороший повод определить собственную функцию клонирования, которая позволит сделать то, что нам нужно:

```
class Fish
{
    public:
        virtual Fish* Clone() const = 0; // Чисто виртуальная функция
};

class Tuna:public Fish
{
    // ... другие члены
    public:
        Tuna * Clone() const // Виртуальная функция клонирования
        {
            return new Tuna(*this); // Вернуть новый объект класса Tuna,
                                   // являющийся копией данного
        }
};
```

Таким образом, виртуальная функция `Clone()` моделирует виртуальный копирующий конструктор, который должен быть вызван явно, как показано в листинге 11.9.

ЛИСТИНГ 11.9. Классы Tuna и Carp с функцией `Clone()`, моделирующей виртуальный копирующий конструктор

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
```



```
5:   public:
6:       virtual Fish* Clone() = 0;
7:       virtual void Swim() = 0;
8:       virtual ~Fish() {};
9:   };
10:
11: class Tuna: public Fish
12: {
13:     public:
14:         Fish* Clone() override
15:         {
16:             return new Tuna(*this);
17:         }
18:
19:         void Swim() override final
20:         {
21:             cout << "Тунец быстро плавает в море" << endl;
22:         }
23: };
24:
25: class BluefinTuna final: public Tuna
26: {
27:     public:
28:         Fish* Clone() override
29:         {
30:             return new BluefinTuna(*this);
31:         }
32:
33:         // Нельзя перекрыть final Tuna::Swim
34: };
35:
36: class Carp final: public Fish
37: {
38:     Fish* Clone() override
39:     {
40:         return new Carp(*this);
41:     }
42:     void Swim() override final
43:     {
44:         cout << "Карп медленно плавает в озере" << endl;
45:     }
46: };
47:
48: int main()
49: {
50:     const int ARRAY_SIZE = 4;
51:
52:     Fish* myFishes[ARRAY_SIZE] = {nullptr};
53:     myFishes[0] = new Tuna();
```

```
54:   myFishes[1] = new Carp();
55:   myFishes[2] = new BluefinTuna();
56:   myFishes[3] = new Carp();
57:
58:   Fish* myNewFishes[ARRAY_SIZE];
59:   for(int index = 0; index < ARRAY_SIZE; ++index)
60:       myNewFishes[index] = myFishes[index]->Clone();
61:
62:   // Вызов виртуального метода для проверки
63:   for(int index = 0; index < ARRAY_SIZE; ++index)
64:       myNewFishes[index]->Swim();
65:
66:   // Освобождение памяти
67:   for(int index = 0; index < ARRAY_SIZE; ++index)
68:   {
69:       delete myFishes[index];
70:       delete myNewFishes[index];
71:   }
72:
73:   return 0;
74: }
```

Результат

```
Тунец быстро плавает в море
Карп медленно плавает в озере
Тунец быстро плавает в море
Карп медленно плавает в озере
```

Анализ

Помимо демонстрации виртуальных копирующих конструкторов посредством виртуальной функции `Fish::Clone()`, листинг 11.9 демонстрирует использование ключевых слов `override` и `final` — для виртуальных функций и классов. Кроме того, в строке 8 имеется виртуальный деструктор класса `Fish`. Строки 52–56 в функции `main()` демонстрируют объявление статического массива указателей на базовый класс `Fish*` и индивидуальное присваивание его элементам вновь созданных объектов класса `Tuna`, `Carp`, `BluefinTuna` и `Carp` соответственно. Обратите внимание на то, что этот массив `myFishes` способен хранить объекты, казалось бы, разных типов, которые связаны общим базовым классом `Fish`. Это уже замечательно — по сравнению с предыдущими массивами в этой книге, которые по большей части имели простой однообразный тип `int`. Если это недостаточно замечательно, то вы можете выполнить копирование в новый массив `myNewFishes` с типом элементов `Fish*` с помощью вызова в цикле виртуальной функции `Fish::Clone()`, как показано в строке 60. Обратите внимание, что массив очень мал: в нем только четыре элемента. Но он может быть намного больше, хотя это и не имеет никакого значения для логики копирования и требует лишь коррекции условия завершения цикла. Строка 64 фактически является

проверкой, которая состоит в вызове виртуальной функции `Fish::Swim()` для каждого хранимого в новом массиве элемента, чтобы проверить, скопировала ли функция `Clone()` объект класса `Tuna` как `Tuna`, а не просто как `Fish`. Показанный вывод демонстрирует, что все скопировано правильно.

РЕКОМЕНДУЕТСЯ

Отмечайте как `virtual` те функции базового класса, которые должны быть перекрыты в производных классах.

Помните, что чисто виртуальные функции делают класс абстрактным базовым классом, а сами эти функции должны быть реализованы в производном классе.

Помечайте в производном классе функцию, предназначенную для перекрытия базовой функциональности, как `override`.

Используйте виртуальное наследование для решения проблемы ромба.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте оснащать базовый класс виртуальным деструктором.

Не забывайте, что компилятор не позволит создать экземпляр абстрактного базового класса.

Не забывайте, что виртуальное наследование спасает общий базовый класс от возникновения проблемы ромба и позволяет создавать только один его экземпляр при множественном наследовании.

Не путайте назначение ключевого слова `virtual` при использовании в создаваемой иерархии наследования с тем же ключевым словом в объявлении функций базового класса.

Резюме

На этом занятии мы изучили мощь иерархий наследования в коде C++ при использовании полиморфизма. Вы узнали, как объявлять и программировать виртуальные функции и как они обеспечивают вызов реализации из производного класса, даже если для вызова виртуального метода используется экземпляр базового класса. Вы узнали, что специальным типом виртуальных функций являются чисто виртуальные функции, гарантирующие невозможность инстанцирования базового класса, что делает его прекрасным местом для определения интерфейсов, которые должны реализовать производные классы. И наконец вы ознакомились с проблемой ромба, вызванной множественным наследованием, и тем, как она решается с помощью виртуального наследования.

Вопросы и ответы

- **Зачем использовать ключевое слово `virtual` в определении функции базового класса, если код компилируется и без этого?**

Без ключевого слова `virtual` вы не в состоянии гарантировать переадресацию вызова `objBase.Function()` функции `Derived::Function()`. Успешная компиляция кода — не единственная мера его качества.

■ Зачем компилятор создает таблицу виртуальных функций?

Для хранения указателей на функцию, чтобы гарантировать вызов правильной версии виртуальной функции.

■ Всегда ли у базового класса должен быть виртуальный деструктор?

В идеале — да. Только тогда вы можете гарантировать, что если некто напишет

```
Base* pBase = new Derived();
delete pBase;
```

то вызов оператора delete для указателя типа Base* приведет к вызову деструктора ~Derived(). Для этого деструктор ~Base() должен быть объявлен виртуальным.

■ Зачем нужен абстрактный базовый класс, если я не могу даже создать его экземпляр?

Абстрактный класс и не предназначен для создания объектов; его задача — быть унаследованным. Он содержит чисто виртуальные функции, определяющие минимальный набор функциональности, который должен реализовываться производными классами, выполняя, таким образом, роль интерфейса.

■ Должен ли я использовать в иерархии наследования ключевое слово virtual в объявлениях всех виртуальных функций, или же достаточно использовать его только в базовом классе?

Достаточно указать ключевое слово virtual в объявлении функции только один раз, в базовом классе.

■ Могу ли я определить функции-члены и переменные-члены в абстрактном классе?

Конечно, можете. Но помните, что нельзя создать экземпляр абстрактного класса, поскольку у него есть по крайней мере одна чисто виртуальная функция, которая должна быть реализована производным классом.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Вы моделируете формы (круг и треугольник) и хотите, чтобы каждый из этих классов обязательно реализовывал функции Area() и Print(). Как это сделать?
2. Для всех ли классов компилятор создает таблицу виртуальных функций?

3. У моего класса `Fish` есть два открытых метода, одна чисто виртуальная функция и несколько переменных-членов. Этот класс все еще остается абстрактным базовым классом?

Упражнения

1. Создайте иерархию наследования, которая реализует контрольный вопрос 1 для круга и треугольника.
2. **Отладка.** Что неправильно в следующем коде?

```
class Vehicle
{
    public:
        Vehicle() {}
        ~Vehicle(){}
};
class Car: public Vehicle
{
    public:
        Car() {}
        ~Car() {}
};
```

3. Каким будет порядок выполнения конструкторов и деструкторов в (неверном) коде упражнения 2, если экземпляр класса `Car` создается и удаляется следующим образом?

```
Vehicle* pMyRacer = new Car;
delete pMyRacer;
```

ЗАНЯТИЕ 12

Типы операторов и их перегрузка

В дополнение к инкапсуляции данных и методов классы позволяют инкапсулировать операторы, которые облегчают работу с экземплярами этого класса. Вы можете использовать эти операторы для выполнения таких операций, как присваивание или сложение объектов класса, как это делается, скажем, с целыми числами, которые мы рассмотрели на занятии 5, “Выражения, инструкции и операторы”. Подобно функциям, операторы могут быть перегружены.

На этом занятии...

- Применение ключевого слова `operator`
- Унарные и бинарные операторы
- Операторы преобразования
- Операторы перемещающего присваивания
- Операторы, которые не могут быть переопределены

Что такое операторы C++

На синтаксическом уровне оператор от функции отличается очень немного — лишь использование ключевого слова `operator`. Объявление оператора очень похоже на объявление функции:

```
Возвращаемый_тип operator Знак (Список_параметров);
```

В данном случае *Знак* может быть любым оператором, который хочет определить программист. Это может быть символ `+` (сложение) или `&&` (логическое И) и т.д. Операнды помогают компилятору отличить один оператор от другого. Встает вопрос — зачем язык C++ предоставляет операторы, когда уже поддерживаются функции?

Рассмотрим вспомогательный класс `Date`, инкапсулирующий день, месяц и год:

```
Date holiday(1, 1, 2017); // Инициализация датой 1 января 2017 года
```

Если теперь понадобится сменить дату на следующий день, 2 января, то какой из предложенных способов удобнее и интуитивно понятнее?

1. Использовать оператор: `++holiday;`
2. Использовать функцию `Date::Increment(): holiday.Increment();`

Конечно, первый способ понятнее и проще, чем применение метода `Increment()`. Выражения с операторами легче в использовании и интуитивно понятнее. Реализация оператора меньше (`<`) в классе `Date` позволяет сравнивать две даты следующим образом:

```
if(Date1 < Date2)
{
    // Сделать нечто
}
else
{
    // Сделать нечто другое
}
```

Операторы могут использоваться во множестве ситуаций, а не только для работы с датами. Представьте оператор суммы (`+`), который обеспечивает простую конкатенацию строк с использованием вспомогательного строкового класса, такого как `MyString` (см. листинг 9.9):

```
MyString sayHello("Hello ");
MyString sayWorld("world");
MyString sumThem(sayHello + sayWorld); // Использование оператора +
```

ПРИМЕЧАНИЕ

Дополнительные усилия по реализации подходящих операторов окупаются легкостью использования вашего класса.

В самом общем смысле операторы C++ можно подразделить на два типа: унарные и бинарные.

Унарные операторы

Как и предполагает их название, *унарные операторы* (unary operator) работают с одиночным операндом. Типичное определение унарного оператора, реализованного как глобальная функция или статическая функция-член, имеет следующий вид:

```
Возвращаемый_тип operator Знак (Тип_параметра)
{
    // ... Реализация
}
```

Унарный оператор, являющийся (нестатическим) членом класса, имеет подобное определение, но у него отсутствует параметр, так как в качестве такового выступает сам объект класса (*this):

```
Возвращаемый_тип operator Знак ()
{
    // ... Реализация
}
```

Типы унарных операторов

Унарные операторы, которые могут быть перегружены (или переопределены), представлены в табл. 12.1.

ТАБЛИЦА 12.1. Унарные операторы

Оператор	Название
++	Инкремент
--	Декремент
*	Разыменование
->	Выбор члена
!	Логическое НЕ
&	Получение адреса
~	Дополнение до единицы
+	Унарный плюс
-	Унарный минус
Операторы преобразования	Преобразование в другой тип

Программирование унарного оператора инкремента или декремента

Унарный префиксный оператор инкремента (++) может быть создан в пределах объявления класса с использованием следующего синтаксиса:

```
// Унарный оператор инкремента (префиксный)
Date& operator ++()
```



```

{
    // Код реализации оператора
    return *this;
}

```

Постфиксный оператор инкремента (++) имеет иной тип возвращаемого значения (которое используется не всегда), а также фиктивный аргумент типа `int`, отличающий постфиксный оператор от префиксного:

```

Date operator ++(int)
{
    // Сохранение текущего объекта до его увеличения
    Date copy(*this);

    // Реализация оператора

    // Возврат сохраненного значения
    return copy;
}

```

Синтаксис префиксного и постфиксного операторов декремента такой же, как и операторов инкремента, только объявление содержит -- там, где объявление инкремента содержит ++. Листинг 12.1 демонстрирует простой класс `Date`, позволяющий увеличивать даты с помощью оператора ++.

ЛИСТИНГ 12.1. Класс даты с операторами инкремента и декремента

```

0: #include <iostream>
1: using namespace std;
2:
3: class Date
4: {
5:     private:
6:         int day, month, year;
7:
8:     public:
9:         Date(int inMonth, int inDay, int inYear)
10:            : month(inMonth), day(inDay), year(inYear) {};
11:
12:         Date& operator ++() // Префиксный инкремент
13:         {
14:             ++day;
15:             return *this;
16:         }
17:
18:         Date& operator --() // Префиксный декремент
19:         {
20:             --day;
21:             return *this;
22:         }

```

```

23:
24:     void DisplayDate()
25:     {
26:         cout << day << "." << . << "." << year << endl;
27:     }
28: };
29:
30: int main()
31: {
32:     Date holiday(1, 7, 2017);
33:
34:     cout << "Дата инициализирована значением: ";
35:     holiday.DisplayDate();
36:
37:     ++holiday; // Вперед на один день
38:     cout << "Дата после префиксного инкремента: ";
39:     holiday.DisplayDate();
40:
41:     --holiday; // Назад на один день
42:     cout << "Дата после префиксного декремента: ";
43:     holiday.DisplayDate();
44:
45:     return 0;
46: }

```

Результат

Дата инициализирована значением: 7.1.2017
 Дата после префиксного инкремента: 8.1.2017
 Дата после префиксного декремента: 7.1.2017

Анализ

Представляющие интерес операторы определены в строках 12–22 и обеспечивают добавление и вычитание дня из экземпляра класса `Date`, как показано в строках 37 и 41 в функции `main()`. Префиксные операторы инкремента и декремента, как видно из примера, возвращают ссылку на тот же экземпляр после выполнения операции увеличения или уменьшения.

ПРИМЕЧАНИЕ

Эта версия класса даты имеет минимальную, практически пустую реализацию, чтобы сократить количество строк при объяснении реализации префиксных операторов `++` и `--`. Серьезная версия должна учитывать возможность перехода даты из месяца в месяц, а также возможный переход между годами и наличие високосных годов.

Для обеспечения постфиксного инкремента и декремента достаточно добавить в класс `Date` следующий код:

```
// Постфиксный оператор отличается от префиксного типом
// возвращаемого значения и параметром
Date operator ++(int)
{
    Date copy(day, month, year);
    ++Day;
    return copy;
}
// Постфиксный оператор декремента
Date operator --(int)
{
    Date copy(day, month, year);
    --Day;
    return copy;
}
```

Теперь вы можете использовать следующие операции с объектами класса Date:

```
Date Holiday(1, 1, 2017); // Создание экземпляра
++Holiday; // Использование префиксного оператора инкремента ++
Holiday++; // Использование постфиксного оператора инкремента ++
--Holiday; // Использование префиксного оператора декремента --
Holiday--; // Использование постфиксного оператора декремента --
```

ПРИМЕЧАНИЕ

Как показано в реализации постфиксных операторов, перед операцией инкремента или декремента создается копия, содержащая текущее состояние объекта; она и будет возвращена после выполнения операции.

Таким образом, если вам нужен *только* инкремент, выбирайте префиксный оператор `++объект`; а не `объект++`; чтобы избежать создания временной копии, которая не используется.

Создание операторов преобразования

Если в код функции `main()` из листинга 12.1 добавить строку

```
cout << Holiday; // Ошибка из-за отсутствия оператора преобразования
```

то произойдет отказ компиляции с сообщением `Error: binary '<<': no operator found which takes a right-hand operand of type 'Date' (or there is no acceptable conversion)` (ошибка: бинарный '<<': не найден оператор, получающий правый операнд типа 'Date' (или нет приемлемого преобразования)). По существу, это сообщение означает, что поток `cout` не знает, как интерпретировать экземпляр класса `Date`, поскольку этот класс не поддерживает операторы, которые могли бы преобразовать его содержимое в тип, который в состоянии вывести поток `cout`.

Однако поток `cout` вполне может работать с константной строкой типа `const char*`:

```
std::cout << "Hello world"; // const char* работает!
```

Поэтому, чтобы поток `cout` работал с объектом класса `Date`, достаточно добавить оператор, который преобразует его в тип `const char*`:

```
operator const char*()
{
    // Реализация оператора, возвращающая const char*
}
```

Листинг 12.2 демонстрирует пример реализации такого оператора.

ЛИСТИНГ 12.2. Реализация оператора преобразования
в `const char*` для класса `Date`

```
0: #include <iostream>
1: #include <sstream> // Заголовочный файл для ostringstream
2: #include <string>
3: using namespace std;
4:
5: class Date
6: {
7:     private:
8:         int day, month, year;
9:         string dateInString;
10:
11:     public:
12:         Date(int inMonth, int inDay, int inYear)
13:             : month(inMonth), day(inDay), year(inYear) {};
14:
15:         operator const char*()
16:         {
17:             ostringstream formattedDate; // Помогает создать строку
18:             formattedDate << day << "." << month << "." << year;
19:
20:             dateInString = formattedDate.str();
21:             return dateInString.c_str();
22:         }
23: };
24:
25: int main()
26: {
27:     Date Holiday(1, 7, 2017);
28:
29:     cout << "Рождество: " << Holiday << endl;
30:
31:     // string strHoliday(Holiday);    // OK!
32:     // strHoliday = Date(11, 7, 2017); // Тоже OK!
33:
34:     return 0;
35: }
```

Результат

Рожество: 7.1.2017

Анализ

Преимущество реализации оператора `const char*` (строки 15–23) проявляется в строке 29 функции `main()`. Теперь экземпляр класса `Date` может непосредственно использоваться потоком `cout` благодаря тому, что этот поток принимает тип `const char*`. Компилятор автоматически использует результат подходящего (а в данном случае единственно доступного) оператора и передает его потоку `cout`, который отображает дату на экране. В нашей реализации оператора `const char*` использован оператор `std::ostringstream`, преобразующий целочисленные члены класса в объект `std::string` (строка 18). Можно было бы сразу вернуть результат метода `formattedDate.str()`, но мы сохраняем его копию в закрытом члене `Date::DateInString` (строка 20), поскольку переменная `formattedDate` является локальной и уничтожается при завершении работы оператора. Поэтому указатель, полученный вызовом метода `str()`, после выхода из оператора будет недействителен.

Этот оператор открывает новые возможности использования класса `Date`. Теперь можно даже присвоить экземпляр даты строке непосредственно:

```
string strHoliday(Holiday);           // OK! Компилятор вызывает
                                     // оператор const char*
strHoliday = Date(11, 11, 2011); // Также OK!
```

ВНИМАНИЕ!

Обратите внимание, что такое присваивание вызывает неявное преобразование, т.е. компилятор использует доступный оператор преобразования (в нашем случае – `const char*`), тем самым разрешая непреднамеренные присваивания, которые компилируются без каких-либо сообщений об ошибках. Чтобы избежать неявных преобразований, используйте ключевое слово `explicit` в начале объявления оператора:

```
explicit operator const char*()
{
    // Код преобразования
}
```

Использование этого ключевого слова заставляет программиста явно указывать свои намерения с использованием операторов приведения:

```
string strHoliday(static_cast<const char*>(Holiday));
strHoliday=static_cast<const char*>(Date(11,11,2016));
```

Операторы приведения, включая `static_cast`, подробнее рассматриваются на занятии 13, “Операторы приведения”.

ПРИМЕЧАНИЕ

Создавайте столько операторов, сколько может понадобиться при использовании класса. Если ваше приложение нуждается в целочисленном представлении даты, то может пригодиться оператор

```
explicit operator int()
{
    // Здесь код преобразования
}
```

Такой оператор позволяет использовать экземпляр класса `Date` в качестве целого числа:

```
FuncTakesInt(static_cast<int>(Date(25,12,2016)));
```

В листинге 12.7 показано применение операторов преобразования с классом строки.

Создание оператора разыменования (*) и оператора выбора члена (->)

Оператор разыменования (*) и оператор обращения к члену класса (выбора члена) (->) чаще всего используются при создании классов интеллектуальных указателей. *Интеллектуальные указатели* (smart pointer) — это вспомогательные классы, являющиеся оболочками для обычных указателей и облегчающие управление памятью (или ресурсом), решая проблемы владения и копирования. В некоторых случаях они даже способны повысить производительность приложения. Подробно интеллектуальные указатели обсуждаются на занятии 26, “Понятие интеллектуальных указателей”, а здесь рассматривается лишь вопрос о том, как перегрузка операторов помогает работе интеллектуальных указателей.

Давайте проанализируем использование указателя `std::unique_ptr` в листинге 12.3 и рассмотрим, как операторы (*) и (->) помогают использовать класс интеллектуального указателя как любой обычный указатель.

ЛИСТИНГ 12.3. Использование интеллектуального указателя `unique_ptr`

```
0: #include <iostream>
1: #include <memory>    // Включение для использования unique_ptr
2: using namespace std;
3:
4: class Date
5: {
6:     private:
7:         int day, month, year;
8:         string dateInString;
9:
10:    public:
11:        Date(int inMonth, int inDay, int inYear)
12:            : month(inMonth), day(inDay), year(inYear) {};
13:
```

```
14: void DisplayDate()
15: {
16:     cout << day << "." << month << "." << year << endl;
17: }
18: };
19:
20: int main()
21: {
22:     unique_ptr<int> smartIntPtr(new int);
23:     *smartIntPtr = 42;
24:
25:     // Использование интеллектуального указателя как int*
26:     cout << "Целое число равно: " << *smartIntPtr << endl;
27:
28:     unique_ptr<Date> smartHoliday(new Date(1, 1, 2017));
29:     cout << "Новый экземпляр даты: ";
30:
31:     // Использование smartHoliday как Date*
32:     smartHoliday->DisplayDate();
33:
34:     return 0;
35: }
```

Результат

Целое число равно: 42
Новый экземпляр даты: 1.1.2017

Анализ

В строке 22 объявляется интеллектуальный указатель типа `int`. Эта строка демонстрирует синтаксис инициализации шаблона для класса интеллектуального указателя `unique_ptr`. Аналогично в строке 28 объявляется интеллектуальный указатель на экземпляр класса `Date`. Пока сосредоточьтесь на использовании указателя и игнорируйте детали.

ПРИМЕЧАНИЕ

Не волнуйтесь, если синтаксис пока что кажется вам непонятным, поскольку шаблоны рассматриваются позже, на занятии 14, "Введение в макросы и шаблоны".

Этот пример демонстрирует не только то, как интеллектуальный указатель позволяет использовать обычный синтаксис указателя (строки 23 и 32). В строке 23 вы можете вывести значение типа `int`, используя синтаксис `*smartIntPtr`, а в строке 32 вы используете вызов `smartHoliday->DisplayData()` так, как будто эти две переменные имеют тип `int*` и `Date*` соответственно. Секрет кроется в классе интеллектуального указателя `std::unique_ptr`, который реализует операторы `(*)` и `(->)`.

ПРИМЕЧАНИЕ

Классы интеллектуальных указателей способны сделать намного больше, чем простая имитация обычных указателей или освобождение памяти при выходе из области видимости. Более подробная информация по этой теме рассматривается на занятии 26, “Понятие интеллектуальных указателей”. Реализация интеллектуальных указателей, перегружающих указанные операторы, приведена в листинге 26.1.

Бинарные операторы

Операторы, работающие с двумя операндами, называются *бинарными операторами* (binary operator). Определение бинарного оператора, реализованного как глобальная функция или статическая функция-член, имеет следующий вид:

```
Возвращаемый_тип operator Знак (Параметр1, Параметр2)
{
    // ... Реализация
}
```

Определение бинарного оператора, реализованного как член класса, имеет вид

```
Возвращаемый_тип operator Знак (Параметр)
{
    // ... Реализация
}
```

Бинарный оператор, реализованный как член класса, получает только один параметр, потому что второй параметр представляет собой сам объект класса.

Типы бинарных операторов

Бинарные операторы, которые могут быть перегружены или переопределены в приложении C++, приведены в табл. 12.2.

ТАБЛИЦА 12.2. Перегружаемые бинарные операторы

Оператор	Название
,	Запятая
!=	Неравенство
%	Деление по модулю
%=	Деление по модулю с присваиванием
&	Побитовое И
&&	Логическое И
&=	Побитовое И с присваиванием
*	Умножение
*=	Умножение с присваиванием
+	Сложение
+=	Сложение с присваиванием

Оператор	Название
-	Вычитание
-=	Вычитание с присваиванием
->*	Косвенное обращение к указателю на член класса
/	Деление
/=	Деление с присваиванием
<	Меньше
<<	Сдвиг влево
<<=	Сдвиг влево с присваиванием
<=	Меньше или равно
=	Присваивание, присваивание копии и присваивание перемещения
==	Равенство
>	Больше
>=	Больше или равно
>>	Сдвиг вправо
>>=	Сдвиг вправо с присваиванием
^	Исключающее ИЛИ
^=	Исключающее ИЛИ с присваиванием
	Побитовое ИЛИ
=	Побитовое ИЛИ с присваиванием
	Логическое ИЛИ
[]	Оператор индексации

Создание бинарных операторов сложения (a+b) и вычитания (a-b)

Подобно операторам инкремента и декремента, бинарные операторы “плюс” и “минус”, будучи определены, позволяют суммировать и вычитать значения поддерживаемого типа данных из объекта класса, который реализует эти операторы. Вернемся к нашему календарному классу Date. Хотя мы уже реализовали в нем возможность инкремента, переводящего календарь на один день вперед, он все еще не поддерживает возможность перевода, скажем, на пять дней вперед. Для этого необходимо реализовать бинарный оператор (+), как сделано в листинге 12.4.

ЛИСТИНГ 12.4. Календарный класс с бинарным оператором суммы

```
0: #include <iostream>
1: using namespace std;
2:
3: class Date
4: {
5:     private:
```

```

6:   int day, month, year;
7:   string dateInString;
8:
9:   public:
10:    Date(int inMonth, int inDay, int inYear)
11:      : month(inMonth), day(inDay), year(inYear) {};
12:
13:    Date operator +(int daysToAdd) // Сложение
14:    {
15:        Date newDate(month, day + daysToAdd, year);
16:        return newDate;
17:    }
18:
19:    Date operator -(int daysToSub) // Вычитание
20:    {
21:        return Date(month, day - daysToSub, year);
22:    }
23:
24:    void DisplayDate()
25:    {
26:        cout << day << "." << month << "." << year << endl;
27:    }
28: };
29:
30: int main()
31: {
32:     Date Holiday(1, 7, 2017);
33:     cout << "Рождество: ";
34:     Holiday.DisplayDate();
35:
36:     Date PreviousHoliday(Holiday - 6);
37:     cout << "Новый год: ";
38:     PreviousHoliday.DisplayDate();
39:
40:     Date NextHoliday(Holiday + 12);
41:     cout << "Крещение: ";
42:     NextHoliday.DisplayDate();
43:
44:     return 0;
45: }

```

Результат

```

Рождество: 7.1.2017
Новый год: 1.1.2017
Крещение: 19.1.2017

```

Анализ

Строки 13–22 содержат реализации бинарных операторов + и –, которые позволяют использовать синтаксис простого сложения и вычитания, показанный соответственно в строках 40 и 36 функции main().

Бинарный оператор сложения также был бы очень полезен в случае строкового класса. На занятии 9, “Классы и объекты”, мы уже анализировали простой класс оболочки строки MyString, инкапсулирующий управление памятью, копирование и другие действия для символьной строки в стиле C с завершающим нулевым символом (см. листинг 9.9). Но этот класс не поддерживает конкатенацию двух строк с использованием синтаксиса сложения:

```
MyString Hello("Hello ");
MyString World(" World");
MyString HelloWorld(Hello + World); // Ошибка: оператор + не определен
```

Само собой разумеется, оператор + чрезвычайно упростил бы использование класса MyString, так что он стоит потраченных на него усилий:

```
MyString operator+(const MyString& addThis)
{
    MyString newString;
    if (addThis.buffer != nullptr)
    {
        newString.buffer = new char[GetLength() +
                                     strlen(addThis.buffer) + 1];
        strcpy(newString.buffer, buffer);
        strcat(newString.buffer, addThis.buffer);
    }

    return newString;
}
```

Чтобы иметь возможность использовать синтаксис сложения, добавьте приведенный выше код в листинг 9.9 с закрытым конструктором по умолчанию MyString() и пустой реализацией. Вы можете увидеть версию класса MyString с оператором + среди прочих в листинге 12.11.

ВНИМАНИЕ!

Операторы обеспечивают удобство и простоту использования класса. Однако необходимо реализовать только те из них, которые имеют смысл. Обратите внимание, что для класса Date мы реализовали операторы сложения и вычитания, а для класса MyString только оператор суммы (+). Поскольку выполнение операций вычитания со строками весьма маловероятно, такой оператор не нашел бы применения.

Реализация операторов сложения с присваиванием (+=) и вычитания с присваиванием (--=)

Операторы сложения с присваиванием обеспечивают такой синтаксис, как “a+=b;”, позволяющий программисту увеличивать значение объекта a на значение b. Преимущество оператора сложения с присваиванием в том, что он может быть перегружен для разных типов параметра b. Приведенный ниже листинг 12.5 позволяет добавлять целочисленное значение к объекту Date.

ЛИСТИНГ 12.5. Определение операторов (+=) и (--=) для добавления и вычитания дней

```

0: #include <iostream>
1: using namespace std;
2:
3: class Date
4: {
5:     private:
6:         int day, month, year;
7:
8:     public:
9:         Date(int inMonth, int inDay, int inYear)
10:            : month(inMonth), day(inDay), year(inYear) {}
11:
12:         void operator+=(int daysToAdd) // Присваивающее сложение
13:         {
14:             day += daysToAdd;
15:         }
16:
17:         void operator--=(int daysToSub) // Присваивающее вычитание
18:         {
19:             day -= daysToSub;
20:         }
21:
22:         void DisplayDate()
23:         {
24:             cout << day << "." << month << "." << year << endl;
25:         }
26: };
27:
28: int main()
29: {
30:     Date holiday(1, 7, 2017);
31:     cout << "Рождество: ";
32:     holiday.DisplayDate();
33:
34:     cout << "holiday -= 6: ";
35:     holiday -= 6;

```

```
36:    holiday.DisplayDate();
37:
38:    cout << "holiday += 18 gives: ";
39:    holiday += 18;
40:    holiday.DisplayDate();
41:
42:    return 0;
43: }
```

Результат

```
Рождество: 7.1.2017
holiday -= 6: 1.1.2017
holiday += 18: 19.1.2017
```

Анализ

Интересующие нас присваивающие операторы сложения и вычитания находятся в строках 12–22. Они обеспечивают добавление и вычитание целочисленных значений к количеству дней в функции `main()`, например:

```
35:    holiday -= 6;
39:    holiday += 18;
```

Теперь класс `Date` позволяет пользователям добавлять и вычитать дни так, как будто это целые числа, используя операторы сложения и вычитания с присваиванием, получающие параметр типа `int`. Вы можете даже предоставить перегруженную версию оператора сложения с присваиванием (`+=`), получающую экземпляр некоего класса `Days`:

```
// Оператор сложения с присваиванием для Days
void operator += (const Days& daysToAdd)
{
    day += daysToAdd.GetDays();
}
```

ПРИМЕЧАНИЕ

Синтаксис присваивающих операторов умножения `*=`, деления `/=`, деления по модулю `%=`, вычитания `-=`, сдвига влево `<<=`, сдвига вправо `>>=`, исключающего ИЛИ `^=`, побитового ИЛИ `|=` и побитового И `&=` подобен синтаксису присваивающего оператора сложения, показанному в листинге 12.5.

Хотя конечная цель перегрузки операторов — сделать класс простым и интуитивно понятным в использовании, есть множество ситуаций, когда реализация оператора может не иметь смысла. Например, у нашего календарного класса `Date` нет никакого смысла в использовании присваивающего оператора побитового И `&=`. Никакому пользователю этого класса никогда не понадобится (даже трудно придумать, зачем) результат такой, например, операции, как `greatDay &= 20`;

Перегрузка операторов равенства (==) и неравенства (!=)

Если пользователю нужно сравнить один экземпляр класса `Date` с другим, он ожидает, что для этого можно применить обычный синтаксис:

```
if (Date1 == Date2)
{
    // Сделать нечто
}
else
{
    // Сделать нечто другое
}
```

Казалось бы, в отсутствие оператора равенства компилятор должен просто выполнить побитовое сравнение двух этих объектов и вернуть значение `true`, если они абсолютно идентичны. Такое сравнение работает для экземпляров классов, содержащих простые типы данных, но оно не годится, если у рассматриваемого класса есть, например, нестатический строковый член, содержащий значение `char*`, как в случае класса `MyString` из листинга 9.9. В таком случае побитовое сравнение фактически сравнит значения указателей, которые будут не равны, даже если строки имеют идентичное содержимое, и всегда будет возвращать значение `false`.

Проблема решается с помощью определения операторов сравнения. В общем виде выражение оператора равенства имеет синтаксис

```
bool operator==(const Тип& объект)
{
    // Код сравнения, возвращающий true при
    // равенстве и false в противном случае
}
```

Оператор неравенства определяется аналогично и может просто использовать оператор равенства:

```
bool operator!=(const Тип& объект)
{
    // Код сравнения, возвращающий false при
    // равенстве и true в противном случае
}
```

Оператор неравенства может быть логическим отрицанием результата оператора равенства. Листинг 12.6 демонстрирует операторы сравнения, определенные в классе `Date`.

ЛИСТИНГ 12.6. Операторы == и !=

```
0: #include <iostream>
1: using namespace std;
2:
3: class Date
4: {
5:     private:
6:         int day, month, year;
```

```
7:
8: public:
9:     Date(int inMonth, int inDay, int inYear)
10:        : month(inMonth), day(inDay), year(inYear) {}
11:
12:     bool operator==(const Date& compareTo)
13:     {
14:         return ((day == compareTo.day)
15:                 && (month == compareTo.month)
16:                 && (year == compareTo.year));
17:     }
18:
19:     bool operator!=(const Date& compareTo)
20:     {
21:         return !(this->operator==(compareTo));
22:     }
23:
24:     void DisplayDate()
25:     {
26:         cout << day << "." << month << "." << year << endl;
27:     }
28: };
29:
30: int main()
31: {
32:     Date holiday1(12, 25, 2016);
33:     Date holiday2(12, 31, 2016);
34:
35:     cout << "Дата 1: ";
36:     holiday1.DisplayDate();
37:     cout << "Дата 2: ";
38:     holiday2.DisplayDate();
39:
40:     if (holiday1 == holiday2)
41:         cout << "Равенство: даты совпадают" << endl;
42:     else
43:         cout << "Равенство: даты не совпадают" << endl;
44:
45:     if (holiday1 != holiday2)
46:         cout << "Неравенство: даты не совпадают" << endl;
47:     else
48:         cout << "Неравенство: даты совпадают" << endl;
49:
50:     return 0;
51: }
```

Результат

Дата 1: 25.12.2016

Дата 2: 31.12.2016

Равенство: даты не совпадают

Неравенство: даты не совпадают

Анализ

Оператор равенства (==) представляет собой простую реализацию, которая возвращает true, если день, месяц и год равны, как показано в строках 12–17. Реализация оператора неравенства (!=) в строке 21 использует оператор равенства. Наличие этих операторов позволяет выполнить сравнение двух объектов (holiday1 и holiday2) класса Date в функции main() (строки 40 и 45).

Перегрузка операторов <, >, <= и >=

Код листинга 12.6 сделал класс Date достаточно интеллектуальным, могущим выяснить, совпадают ли две даты, т.е. равны ли два объекта класса Date. Но чтобы выполнить одну из показанных далее проверок, требуется определить операторы больше (>), меньше (<), больше или равно (>=) и меньше или равно (<=):

```
if (date1 < date2) { /* Сделать нечто */ }
if (date1 <= date2) { /* Сделать нечто */ }
if (date1 > date2) { /* Сделать нечто */ }
if (date1 >= date2) { /* Сделать нечто */ }
```

Эти операторы продемонстрированы в коде, показанном в листинге 12.7.

ЛИСТИНГ 12.7. Реализация операторов <, <=, > и >=

```
0: #include <iostream>
1: using namespace std;
2:
3: class Date
4: {
5:     private:
6:         int day, month, year;
7:
8:     public:
9:         Date(int inMonth, int inDay, int inYear)
10:            : month(inMonth), day(inDay), year(inYear) {}
11:
12:         bool operator<(const Date& compareTo)
13:         {
14:             if (year < compareTo.year)
15:                 return true;
16:             else if (month < compareTo.month)
17:                 return true;
18:             else if (day < compareTo.day)
19:                 return true;
20:             else
21:                 return false;
22:         }
23:
24:         bool operator<=(const Date& compareTo)
25:         {
26:             if (this->operator==(compareTo))
27:                 return true;
```



```
28:         else
29:             return this->operator<(compareTo);
30:     }
31:
32:     bool operator >(const Date& compareTo)
33:     {
34:         return !(this->operator<=(compareTo));
35:     }
36:
37:     bool operator==(const Date& compareTo)
38:     {
39:         return ((day == compareTo.day)
40:                 && (month == compareTo.month)
41:                 && (year == compareTo.year));
42:     }
43:
44:     bool operator>= (const Date& compareTo)
45:     {
46:         if(this->operator==(compareTo))
47:             return true;
48:         else
49:             return this->operator>(compareTo);
50:     }
51:
52:     void DisplayDate()
53:     {
54:         cout << day << "." << month << "." << year << endl;
55:     }
56: };
57:
58: int main()
59: {
60:     Date holiday1(12, 25, 2016);
61:     Date holiday2(12, 31, 2016);
62:
63:     cout << "Дата 1: ";
64:     holiday1.DisplayDate();
65:     cout << "Дата 2: ";
66:     holiday2.DisplayDate();
67:
68:     if (holiday1 < holiday2)
69:         cout << "<: дата 1 раньше" << endl;
70:
71:     if (holiday2 > holiday1)
72:         cout << ">: дата 2 позже" << endl;
73:
74:     if (holiday1 <= holiday2)
75:         cout << "<=: дата 1 не позже даты 2" << endl;
76:
77:     if (holiday2 >= holiday1)
78:         cout << ">=: дата 1 не позже даты 2" << endl;
79:
80:     return 0;
81: }
```

Результат

```
Дата 1: 25.12.2016
Дата 2: 31.12.2016
<: дата 1 раньше
>: дата 2 позже
<=: дата1 не позже даты 2
>=: дата1 не позже даты 2
```

Анализ

Интересующие нас операторы реализованы в строках 12–50 и частично используют оператор `==` из листинга 12.6. Применение этих операторов в строках 68–78 в функции `main()` демонстрирует, насколько реализация этих операторов делает использование класса `Date` простым и интуитивно понятным.

Перегрузка оператора копирующего присваивания (=)

Нередко содержимое экземпляра класса необходимо присвоить другому экземпляру:

```
Date holiday(25, 12, 2016);
Date anotherHoliday(1, 1, 2017);
anotherHoliday = holiday; // Использование оператора копирующего присваивания
```

Это присваивание ведет к вызову оператора копирующего присваивания по умолчанию, который компилятор сгенерирует автоматически, если вы не предоставите такового. В зависимости от природы вашего класса стандартный копирующий конструктор может оказаться не соответствующим стоящей перед ним задаче, особенно если ваш класс задействует ресурс, который не будет скопирован. Данная проблема с копирующим присваиванием по умолчанию аналогична уже рассматривавшейся на занятии 9, “Классы и объекты”, проблеме копирующего конструктора по умолчанию. Чтобы гарантировать глубокое копирование, как и в случае с копирующим конструктором, необходимо определить собственный оператор копирующего присваивания:

```
Тип& operator= (const Тип& исходный_объект)
{
    if(this != &исходный_объект) // Защита от копирования
    {
        // Реализация оператора присваивания
    }
    return *this;
}
```

Глубокое копирование важно, если ваш класс инкапсулирует простой указатель, такой как у класса `MyString` из листинга 9.9. Чтобы гарантировать глубокое копирование во время присваивания, определите оператор копирующего присваивания, как это сделано в листинге 12.8.

ЛИСТИНГ 12.8. Улучшенный класс `MyString` из листинга 9.9 с оператором копирующего присваивания

```
0: #include <iostream>
1: using namespace std;
2: #include <string.h>
3: class MyString
4: {
5:     private:
6:         char* buffer;
7:
8:     public:
9:         MyString(const char* initialInput)
10:        {
11:            if(initialInput != nullptr)
12:            {
13:                buffer = new char[strlen(initialInput) + 1];
14:                strcpy(buffer, initialInput);
15:            }
16:            else
17:                buffer = nullptr;
18:        }
19:
20:        // Оператор копирующего присваивания
21:        MyString& operator=(const MyString& copySource)
22:        {
23:            if ((this != &copySource)&&(copySource.buffer != nullptr))
24:            {
25:                if (buffer != nullptr)
26:                    delete[] buffer;
27:
28:                // Глубокое копирование в свой буфер
29:                buffer = new char[strlen(copySource.buffer) + 1];
30:
31:                // Копирование из исходного объекта в локальный буфер
32:                strcpy(buffer, copySource.buffer);
33:            }
34:
35:            return *this;
36:        }
37:
38:        operator const char*()
39:        {
40:            return buffer;
41:        }
42:
43:        ~MyString()
44:        {
45:            delete[] buffer;
46:        }
47: };
```

```
48:
49: int main()
50: {
51:     MyString string1("Hello ");
52:     MyString string2(" World");
53:
54:     cout << "До присваивания: " << endl;
55:     cout << string1 << string2 << endl;
56:     string2 = string1;
57:     cout << "После присваивания string2 = string1: " << endl;
58:     cout << string1 << string2 << endl;
59:
60:     return 0;
61: }
```

Результат

```
До присваивания:
Hello World
После присваивания string2 = string1:
Hello Hello
```

Анализ

Я преднамеренно опустил копирующий конструктор в этом примере, чтобы сократить объем кода (но при создании подобного класса обязательно добавьте его; см. листинг 9.9). Оператор копирующего присваивания реализован в строках 21–36. Он очень похож на копирующий конструктор, но с предварительной проверкой, гарантирующей, что оригинал и копия не являются одним и тем же объектом. После успешной проверки оператор копирующего присваивания класса `MyString` сначала освобождает свой внутренний буфер, затем повторно резервирует место для текста копии, а потом использует функцию `strcpy()` для копирования, как показано в строке 14.

ПРИМЕЧАНИЕ

Еще одно незначительное различие между листингами 12.8 и 9.9 в том, что функция `GetString()` заменена оператором `const char*`, как видно из строк 38–41. Этот оператор облегчает использование класса `MyString`, как показано в строке 55, где один поток `cout` используется для отображения двух экземпляров класса `MyString`.

ВНИМАНИЕ!

При реализации класса, который управляет динамически распределяемым ресурсом, таким как символьная строка в стиле C, динамический массив и так далее, всегда следует реализовывать (или рассмотреть необходимость такой реализации) копирующий конструктор и оператор копирующего присваивания в дополнение к конструктору и деструктору. Если только вы не решаете проблему владения ресурсом при копировании объектов вашего класса явно, такой класс является неполным и опасным в использовании.

СОВЕТ

Чтобы создать класс, который не может быть скопирован, объявите копирующий конструктор и оператор копирующего присваивания как закрытые. Объявления как `private` при отсутствии реализации вполне достаточно для компилятора, чтобы сообщить об ошибке при любых попытках копирования этого класса, например при передаче в функцию по значению или при присваивании одного экземпляра другому.

Оператор индексации ([])

Оператор `[]`, позволяющий обращаться к классу в стиле массива, называется *оператором* индексации (subscript operator). Типичный синтаксис оператора индексации таков:

```
Возвращаемый_тип& operator[] (Тип_индекса& значение_индекса);
```

Так, при создании такого класса, как `MyString`, инкапсулирующего класс динамического массива символов `char* buffer`, оператор индексации существенно облегчит произвольный доступ к отдельным символам в буфере:

```
class MyString
{
    // ... другие члены класса
public:
    /*const*/ char& operator[](int index) /*const*/
    {
        // Возврат из буфера символа в позиции index
    }
};
```

Пример в листинге 12.9 демонстрирует, как оператор индексации (`[]`) обеспечивает возможность итерации символов, содержащихся в экземпляре класса `MyString` с использованием обычной семантики массива.

ЛИСТИНГ 12.9. Реализация оператора индексации (`[]`) в классе `MyString`, обеспечивающего произвольный доступ к символам в буфере `MyString::buffer`

```
0: #include <iostream>
1: #include <string>
2: #include <string.h>
3: using namespace std;
4: class MyString
5: {
6:     private:
7:         char* buffer;
8:
9:         // Закрытый конструктор по умолчанию
10:        MyString() {}
11:
12: public:
```

```

13: // Конструктор
14: MyString(const char* initialInput)
15: {
16:     if(initialInput != nullptr)
17:     {
18:         buffer = new char[strlen(initialInput) + 1];
19:         strcpy(buffer, initialInput);
20:     }
21:     else
22:         buffer = nullptr;
23: }
24:
25: // Копирующий конструктор: вставить из листинга 9.9
26: MyString(const MyString& copySource);
27:
28: // Оператор копирующего присваивания: вставить из листинга 12.8
29: MyString& operator=(const MyString& copySource);
30:
31: const char& operator[](int index) const
32: {
33:     if (index < GetLength())
34:         return buffer[index];
35: }
36:
37: // Деструктор
38: ~MyString()
39: {
40:     if (buffer != nullptr)
41:         delete[] buffer;
42: }
43:
44: int GetLength() const
45: {
46:     return strlen(buffer);
47: }
48:
49: operator const char*()
50: {
51:     return buffer;
52: }
53: };
54:
55: int main()
56: {
57:     cout << "Введите предложение: ";
58:     string strInput;
59:     getline(cin, strInput);
60:
61:     MyString youSaid(strInput.c_str());
62:
63:     cout << "Ваш ввод с использованием operator[]: " << endl;

```

```
64:     for(int index = 0; index < youSaid.GetLength(); ++index)
65:         cout << youSaid[index] << " ";
66:     cout << endl;
67:
68:     cout << "Введите индекс 0 - " << youSaid.GetLength() - 1 << ": ";
69:     int InIndex = 0;
70:     cin >> InIndex;
71:     cout << "Искомый символ в позиции " << InIndex;
72:     cout << " - " << youSaid[InIndex] << endl;
73:
74:     return 0;
75: }
```

Результат

Введите предложение: **OK, operator[] работает!**

Ваш ввод с использованием operator[]:

O K , o p e r a t o r [] р а б о т а е т !

Введите индекс 0 - 37: 2

Искомый символ в позиции 2 - ,

Анализ

Эта программа получает предложение, которое вы вводите, создает строку `MyString`, использует ее, как показано в строке 61, а затем применяет цикл `for` для посимвольного вывода строки с помощью оператора индексации (`[]`) и использования синтаксиса, как у массива (строки 64 и 65). Сам оператор (`[]`) определяется в строках 31–35; он обеспечивает прямой доступ к символу в определенной позиции после проверки того, что требуемая позиция находится в пределах буфера `char* buffer`.

ВНИМАНИЕ!

При программировании операторов приобретает особую важность использование ключевого слова `const`. Обратите внимание, как листинг 12.9 ограничил возвращаемое значение оператора индексации (`[]`) типом `const char&`. Программа работает и компилируется и без ключевых слов `const`, но причина, по которой они использованы в коде, — избежать модифицирующего кода наподобие

```
MyString sayHello("Hello World");
```

```
sayHello[2] = 'k' // Ошибка: operator[] константный
```

При использовании ключевого слова `const` вы защищаете внутренний член класса `MyString::buffer` от непосредственного изменения извне с помощью оператора `[]`. Кроме объявления возвращаемого значения как `const`, следует объявить как `const` и функцию оператора, чтобы обеспечить ее неспособность изменять члены-данные класса.

Как правило, желательно использовать ограничение `const` везде, где это возможно, чтобы избежать непреднамеренных изменений данных и повысить защиту членов-данных класса.

Код, представленный в листинге 12.9, хотелось бы усовершенствовать, реализовав один оператор индексации, который позволял бы и читать содержимое строки, и записывать значения в элемент динамического массива.

Для этого можно реализовать два оператора индексации: один как константную функцию, а второй — как не константную:

```
char& operator[](int nIndex); // Используется для записи / изменения
                               // буфера по индексу
char& operator[](int nIndex) const; // Используется только для
                                   // чтения символа по индексу
```

Компилятор достаточно интеллектуален, чтобы вызывать константную версию функции для операций чтения и не константную — для операций записи в объект `MyString`. Таким образом, вы можете (если хотите) разделить функциональность между двумя функциями. Существуют и другие бинарные операторы (см. табл. 12.2), которые могут быть переопределены или перегружены, но они на этом занятии не обсуждаются. Однако их реализация подобна уже рассмотренной.

Другие операторы, такие как логические и побитовые, следует создавать, только если они улучшат класс. Конечно, такому календарному классу, как `Date`, не обязательно реализовать логические операторы, тогда как классу, реализующему строку или число, возможно, они будут требоваться достаточно часто.

Принимая решение о перегрузке имеющихся операторов или создании новых, помните о цели вашего класса и способе его использования.

Оператор функции ()

Оператор `()`, заставляющий объекты вести себя, как функции, называется *оператором функции* (function operator). Такие операторы применяются в стандартной библиотеке шаблонов (STL) и обычно используются в алгоритмах STL. Они применимы при принятии решений; такие функциональные объекты обычно называются *унарным* или *бинарным предикатом* (predicate) в зависимости от количества операндов. В листинге 12.10 анализируется настолько простой функциональный объект, что вы сразу сможете понять, почему ему дано такое интригующее название!

ЛИСТИНГ 12.10. Функциональный объект, созданный с использованием оператора `()`

```
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: class Display
6: {
7: public:
8:     void operator()(string input) const
9:     {
10:         cout << input << endl;
11:     }
```



```
12:};  
13:  
14:int main()  
15:{  
16:    Display displayFuncObject;  
17:  
18:    // Эквивалентно displayFuncObject.operator() ("Моя строка!");  
19:    displayFuncObject("Моя строка!");  
20:  
21:    return 0;  
22: }
```

Результат

Моя строка!

Анализ

В строках 8–11 реализуется оператор `()`, который затем используется в строке 19 в функции `main()`. Обратите внимание, что объект `displayFuncObject` используется с синтаксисом вызова функции: компилятор неявно преобразует код, который выглядит как вызов функции, в вызов `operator()`.

Собственно, именно поэтому данный оператор и называется оператором функции `()`, а объект `Display` — функциональным объектом, или *функтором* (functor). Более подробная информация по этой теме рассматривается на занятии 21, “Понятие о функциональных объектах”.

Перемещающий конструктор и оператор перемещающего присваивания

Перемещающий конструктор и оператор перемещающего присваивания представляют собой средства оптимизации производительности, которые стали частью стандарта C++11 и гарантируют, что временные значения (r-значения, которые не существуют вне инструкций) не будут копироваться понапрасну. Это особенно полезно при работе класса, который управляет динамически распределяемым ресурсом, таким как динамический массив или строка.

Проблема излишнего копирования

Обратите внимание на оператор сложения, реализованный в листинге 12.4. Фактически этот оператор создает копию и возвращает ее. Если класс `MyString` из листинга 12.9 поддерживает оператор сложения, то приведенный далее фрагмент исходного текста представляет собой корректный пример конкатенации строк:

```
MyString Hello("Hello ");  
MyString World("World");
```

```
MyString CPP(" of C++");  
MyString sayHello(Hello + World + CPP); // Оператор +,  
                                         // копирующий конструктор  
MyString sayHelloAgain("overwrite this");  
sayHelloAgain = Hello + World + CPP;    // Оператор +,  
                                         // копирующий конструктор,  
                                         // копирующее присваивание
```

Эта простая конструкция, выполняющая конкатенацию трех строк, использует бинарный оператор+:

```
MyString operator+(const MyString& addThis)  
{  
    MyString newStr;  
  
    if (addThis.buffer != nullptr)  
    {  
        // Копирование в newStr  
    }  
    return newStr; // Возврат копии по значению с  
                  // вызовом копирующего конструктора  
}
```

Такой оператор, облегчающий программирование с применением конкатенации с помощью интуитивно понятных выражений, может привести к проблемам производительности. Создание объекта sayHello требует двойного выполнения оператора суммирования; в результате каждого выполнения оператора + создается временная копия, поскольку объект класса MyString возвращается по значению, так что при этом вызывается копирующий конструктор. Он осуществляет глубокое копирование строки во временный объект, который не существует после завершения инструкции. Получается, что это интуитивно понятное выражение приводит к созданию нескольких временных копий (r-значений), которые после завершения выполнения инструкции будут уничтожены, а следовательно, являются узким местом с точки зрения производительности, создаваемым языком C++. По крайней мере, так было до недавнего времени.

Теперь эта проблема решена. Компилятор, соответствующий стандарту C++11, распознает временные объекты и использует для них перемещающий конструктор или оператор перемещающего присваивания, если таковые предоставлены программистом.

Объявление перемещающих конструктора и оператора присваивания

Перемещающий конструктор имеет следующий синтаксис:

```
class Sample {  
    private:  
        Type * ptrRes;  
    public:
```

```

Sample(Sample && moveSource) {    // Перемещающий конструктор,
                                // обратите внимание на &&
    ptrRes = moveSource.ptrRes;   // Получение владения,
    moveSource.ptrRes = nullptr;  // перемещение
}

Sample & operator= (Sample &&    // Оператор перемещающего
                   moveSource) { // присваивания, см. &&
    if (this != & moveSource) {
        delete[] ptrRes;         // Освобождение ресурса.
        ptrRes = moveSource.ptrRes; // Получение владения,
        moveSource.ptrRes = nullptr; // перемещение
    }
}

Sample();                      // Конструктор по умолчанию
Sample(const Sample & copySource); // Копирующий конструктор
Sample & operator= (const Sample & copySource); // Копирующее
};                             // присваивание

```

Таким образом, объявление перемещающих конструктора и оператора присваивания отличается от обычных копирующих конструктора и оператора присваивания тем, что входной параметр имеет тип `MyClass&&`. Кроме того, поскольку входной параметр является исходным объектом для перемещения, он не может быть константным, так как в процессе перемещения он изменяется. Возвращаемые же значения остаются теми же, что и ранее, поскольку это просто перегруженные версии конструктора и оператора присваивания соответственно.

Поддерживающие стандарт C++11 компиляторы гарантируют, что для временных объектов (r-значений) используется перемещающий, а не копирующий конструктор, а также оператор перемещающего присваивания вместо копирующего оператора присваивания. В нашей реализации мы обеспечим вместо копирования простое перемещение ресурса из объекта источника в объект получателя. Листинг 12.11 демонстрирует эффективность этих нововведений C++11 для оптимизации класса `MyString`.

ЛИСТИНГ 12.11. Класс `MyString` с перемещающими конструктором и оператором присваивания в дополнение к копирующим

```

0: #include <iostream>
1: #include <string.h>
2: using namespace std;
3: class MyString
4: {
5:     private:
6:         char* buffer;
7:
8:         MyString(): buffer(nullptr) // Закрытый конструктор
9:         {                          // по умолчанию
10:             cout << "Конструктор по умолчанию" << endl;
11:         }
12:

```

```
13: public:
14:     MyString(const char* initialInput) // Конструктор
15:     {
16:         cout << "Конструктор: " << initialInput << endl;
17:         if(initialInput != nullptr)
18:         {
19:             buffer = new char[strlen(initialInput) + 1];
20:             strcpy(buffer, initialInput);
21:         }
22:     else
23:         buffer = nullptr;
24:     }
25:
26:     MyString(MyString&& moveSrc) // Перемещающий конструктор
27:     {
28:         cout << "Перемещ. конструктор: " << moveSrc.buffer << endl;
29:         if(moveSrc.buffer != nullptr)
30:         {
31:             buffer = moveSrc.buffer; // Получение владения
32:             moveSrc.buffer = nullptr; // Освобождение перемещен-
33:         }                               // ного ресурса
34:     }
35:
36:     MyString& operator=(MyString&& moveSrc) // Перемещающее
37:     {                                       // присваивание
38:         cout << "Перемещ. присваивание: " << moveSrc.buffer << endl;
39:         if((moveSrc.buffer != nullptr) && (this != &moveSrc))
40:         {
41:             delete[] buffer; // release own buffer
42:
43:             buffer = moveSrc.buffer; // Получение владения
44:             moveSrc.buffer = nullptr; // Освобождение перемещен-
45:         }                               // ного ресурса
46:
47:         return *this;
48:     }
49:
50:     MyString(const MyString& copySrc) // Копирующий конструктор
51:     {
52:         cout << "Копир. конструктор: " << copySrc.buffer << endl;
53:         if (copySrc.buffer != nullptr)
54:         {
55:             buffer = new char[strlen(copySrc.buffer) + 1];
56:             strcpy(buffer, copySrc.buffer);
57:         }
58:     else
59:         buffer = nullptr;
60:     }
61:
```

```
62:     MyString& operator=(const MyString& copySrc) // Оператор
63:     {                                           // копирующего присваивания
64:         cout << "Копир. присваивание: " << copySrc.buffer << endl;
65:         if ((this != &copySrc) && (copySrc.buffer != nullptr))
66:         {
67:             if (buffer != nullptr)
68:                 delete[] buffer;
69:
70:             buffer = new char[strlen(copySrc.buffer) + 1];
71:             strcpy(buffer, copySrc.buffer);
72:         }
73:
74:         return *this;
75:     }
76:
77:     ~MyString() // Деструктор
78:     {
79:         if (buffer != nullptr)
80:             delete[] buffer;
81:     }
82:
83:     int GetLength()
84:     {
85:         return strlen(buffer);
86:     }
87:
88:     operator const char*()
89:     {
90:         return buffer;
91:     }
92:
93:     MyString operator+(const MyString& addThis)
94:     {
95:         cout << "operator+: " << endl;
96:         MyString newStr;
97:
98:         if (addThis.buffer != nullptr)
99:         { newStr.buffer =
100:             new char[GetLength()+strlen(addThis.buffer)+1];
101:             strcpy(newStr.buffer, buffer);
102:             strcat(newStr.buffer, addThis.buffer);
103:         }
104:
105:         return newStr;
106:     }
107: };
108:
109: int main()
110: {
```

```
111:    MyString Hello("Hello ");
112:    MyString World("World");
113:    MyString CPP(" of C++");
114:
115:    MyString sayHelloAgain("overwrite this");
116:    sayHelloAgain = Hello + World + CPP;
117:
118:    return 0;
119: }
```

Результат

Вывод без перемещающих конструктора и оператора присваивания (при закомментированных строках 26–48):

```
Конструктор: Hello
Конструктор: World
Конструктор: of C++
Конструктор: overwrite this
operator+:
Конструктор по умолчанию
Копир. конструктор: Hello World
operator+:
Конструктор по умолчанию
Копир. конструктор: Hello World of C++
Копир. присваивание: Hello World of C++
```

Вывод с перемещающими конструктором и оператором присваивания:

```
Конструктор: Hello
Конструктор: World
Конструктор: of C++
Конструктор: overwrite this
operator+:
Конструктор по умолчанию
Перемещ. конструктор: Hello World
operator+:
Конструктор по умолчанию
Перемещ. конструктор: Hello World of C++
Перемещ. присваивание: Hello World of C++
```

Анализ

Код получился действительно весьма длинным, но большая его часть уже была представлена в предыдущих примерах и на занятиях. Самая важная часть этого листинга находится в строках 26–48, где реализованы перемещающий конструктор и оператор перемещающего присваивания соответственно. Те части вывода, на которые влияют нововведения стандарта C++11, выделены полужирным шрифтом. Обратите внимание, насколько существенно изменился вывод по сравнению с тем же классом, но без этих двух средств. Если рассмотреть реализацию перемещающих конструктора

и оператора присваивания, то можно заметить, что семантика перемещения, по существу, реализуется за счет передачи владения ресурсом от источника перемещения (строка 31 в перемещающем конструкторе и строка 43 в перемещающем операторе присваивания). Непосредственно за этим следует присваивание значения `nullptr` исходному указателю (строки 32 и 44). Это присваивание гарантирует, что деструктор экземпляра, из которого выполнено перемещение, не выполняет освобождение памяти с помощью оператора `delete` в строке 80, поскольку владение ресурсом передано целевому объекту. Обратите внимание, что в отсутствии конструктора перемещения вызывается копирующий конструктор, который осуществляет глубокое копирование строки. Таким образом, перемещающий конструктор существенно экономит время работы программы и сокращает количество нежелательных операций распределения памяти и копирования.

Создание перемещающих конструктора и оператора присваивания не является обязательным. В отличие от копирующего конструктора и оператора присваивания копии компилятор не генерирует его реализацию самостоятельно.

Используйте эти возможности для оптимизации работы классов, которые указывают на динамически распределяемые ресурсы и которые в противном случае требуют глубокого копирования даже тогда, когда они используются как временные объекты.

Пользовательские литералы

Литеральные константы были введены на занятии 3, “Использование переменных и констант”. Вот несколько их примеров:

```
int bankBalance = 10000;
double pi = 3.14;
char firstAlphabet = 'a';
const char* sayHello = "Hello!";
```

В приведенном коде 10000, 3.14, ' ' и "Hello!" представляют собой литеральные константы. Стандарт C++11 расширяет поддержку литералов, позволяя определять собственные литералы. Например, если вы работаете над научным приложением, которое занимается термодинамическими расчетами, то можете захотеть хранить ваши температурные данные с использованием шкалы Кельвина. Новый стандарт позволяет объявить ваши температуры с использованием синтаксиса, подобного следующему:

```
Temperature k1 = 32.15_F;
Temperature k2 = 0.0_C;
```

С помощью определенных вами литералов `_F` и `_C` вы делаете ваше приложение проще для чтения, а значит, и для поддержки. Чтобы определить собственный литерал, следует определить `operator""`, как показано далее:

```
Возвращаемый_тип operator "" Ваш_литерал(Тип_значение) {
    // Код преобразования
}
```

ПРИМЕЧАНИЕ

В зависимости от природы пользовательского литерала параметр *Тип* ограничен одним из следующих значений:

unsigned long long int для целочисленного литерала
 long double для литерала с плавающей точкой
 char, wchar_t, char16_t и char32_t для символьного литерала
 const char* для необработанного строкового литерала
 const char* с size_t для строкового литерала
 const wchar_t* с size_t для строкового литерала
 const char16_t* с size_t для строкового литерала
 const char32_t* с size_t для строкового литерала

В листинге 12.12 показаны пользовательские литералы, выполняющие преобразование типа.

ЛИСТИНГ 12.12. Преобразование температур по Фаренгейту и Цельсию в значения по шкале Кельвина

```

0: #include <iostream>
1: using namespace std;
2:
3: struct Temperature
4: {
5:     double Kelvin;
6:     Temperature(long double kelvin) : Kelvin(kelvin) {}
7: };
8:
9: Temperature operator"" _C(long double celcius)
10: {
11:     return Temperature(celcius + 273);
12: }
13:
14: Temperature operator "" _F(long double fahrenheit)
15: {
16:     return Temperature((fahrenheit + 459.67) * 5 / 9);
17: }
18:
19: int main()
20: {
21:     Temperature k1 = 31.73_F;
22:     Temperature k2 = 0.0_C;
23:
24:     cout << "k1 = " << k1.Kelvin << " K" << endl;
25:     cout << "k2 = " << k2.Kelvin << " K" << endl;
26:
27:     return 0;
28: }

```


Результат

```
k1 = 273 K
k2 = 273 K
```

Анализ

В строках 21 и 22 исходного текста показана инициализация двух объектов `Temperature`, один с использованием пользовательского литерала `_F` для объявления значения в градусах Фаренгейта и `_C` для объявления значения в градусах Цельсия. Эти литералы, определенные в строках 9–17, выполняют работу по преобразованию соответствующих значений в температуру по шкале Кельвина и возвращают экземпляры `Temperature`. Обратите внимание, что переменная `k2` преднамеренно инициализирована значением `0.0_C`, а не `0_C`, потому что литерал `_C` определен таким образом, что требует в качестве входного значения `long double`, а `0` будет интерпретироваться как целое число.

Операторы, которые не могут быть перегружены

При всей гибкости, которую предоставляет язык C++ в настройке поведения операторов и классов, он, тем не менее, не разрешает изменять поведение некоторых операторов, которые в любых обстоятельствах должны работать согласованно. Эти операторы, которые не могут быть переопределены, представлены в табл. 12.3.

ТАБЛИЦА 12.3. Операторы, которые не могут быть перегружены или переопределены

Оператор	Название
.	Обращение к члену
.*	Обращение к указателю на член класса
::	Разрешение области видимости
?:	Условный тернарный оператор
sizeof	Размер объекта или типа

РЕКОМЕНДУЕТСЯ

Создавайте столько операторов, сколько необходимо для упрощения использования класса, но не больше.

Маркируйте операторы преобразования как `explicit`, чтобы избежать неявных преобразований.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте, что, если вы не предоставите собственные копирующий оператор присваивания и копирующий конструктор, компилятор сгенерирует их автоматически; однако они не будут выполнять глубокое копирование простых указателей, содержащихся в классе.

РЕКОМЕНДУЕТСЯ

Всегда создавайте копирующий оператор присваивания (с копирующим конструктором и деструктором) для класса, среди членов которого имеется простой указатель.

При использовании компилятора, поддерживающего стандарт C++11 всегда создавайте перемещающий оператор присваивания (и перемещающий конструктор) для классов, которые управляют динамически выделяемыми ресурсами, такими как массивы.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте, что, если вы не предоставите перемещающий оператор присваивания или перемещающий конструктор, компилятор не сгенерирует их автоматически, а использует обычные копирующие оператор присваивания и конструктор.

Резюме

Вы узнали, как создание операторов может существенно упростить использование вашего класса. При разработке класса, который управляет ресурсами, например динамическим массивом или строкой, в дополнение к деструктору необходимо предоставить как минимум копирующие конструктор и оператор присваивания. Вспомогательный класс, который управляет динамическим массивом, стоит снабдить перемещающими конструктором и оператором присваивания, которые гарантируют, что при использовании временных объектов не будет выполняться затратное глубокое копирование хранимого ресурса. Наконец вы узнали, что не могут быть переопределены такие операторы, как `., .*, ::, ?:` и `sizeof`.

Вопросы и ответы

- **Мой класс инкапсулирует динамический массив целых чисел. Какой минимум функций и операторов я должен реализовать?**

При разработке такого класса необходимо четко определить его поведение в случае, когда его экземпляр копируется в другой непосредственно, через присваивание, или косвенно, при передаче в функцию по значению. Как правило, реализуются копирующие конструктор и оператор присваивания, а также деструктор. Если вы хотите повысить производительность вашего класса, имеет смысл снабдить его перемещающими конструктором и оператором присваивания. Для обращения к хранящимся в массиве элементам имеет смысл перегрузить также оператор индексации `operator[]`.

- **У меня есть экземпляр `object` класса. Я хочу обеспечить возможность использования синтаксиса `cout<<object;`. Какой оператор я должен реализовать?** Необходимо реализовать оператор преобразования, который позволит интерпретировать объект вашего класса как тип, который может обработать оператор `std::cout`. Один из способов сделать это — определить оператор `char*()`, как в листинге 12.2.

- Я хочу создать собственный класс интеллектуального указателя. Какой минимум функций и операторов я должен реализовать?

Интеллектуальный указатель должен позволять использовать себя как обычный указатель: `*pSmartPtr` или `pSmartPtr->Func()`. Для этого вы должны реализовать операторы `(*)` и `(->)`. Кроме того, чтобы указатель был интеллектуальным, нужно также позаботиться об автоматическом освобождении ресурсов, предоставив деструктор, а также точно определиться с тем, как именно осуществляется копирование и присваивание, либо реализова копирующие конструктор и оператор присваивания, либо запрещая их (объявив их закрытыми).

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приставайте к изучению материала следующего занятия.

Контрольные вопросы

1. Может ли мой оператор индексации `[]` возвращать константный и не константный типы возвращаемого значения?

```
const Type& operator[](int index);  
Type& operator[](int index); // Это нормально?
```

2. Объявляли бы вы копирующий конструктор или копирующий оператор присваивания как `private`?
3. Имеет ли смысл определять перемещающие конструктор и оператор присваивания для нашего класса `Date`?

Упражнения

1. Напишите для класса `Date` оператор преобразования, который преобразует содержащуюся в нем дату в целое число.
2. Создайте перемещающие конструктор и оператор присваивания для класса `DynIntegers`, который инкапсулирует динамически выделенный массив в виде закрытого члена типа `int*`.

ЗАНЯТИЕ 13

Операторы приведения

Приведение типов (casting) — это механизм, позволяющий программисту изменить интерпретацию объекта компилятором. Приведение не подразумевает изменение самого объекта, изменяется только его интерпретация. Операторы, которые изменяют интерпретацию объекта, называются операторами приведения (casting operator).

На этом занятии...

- Потребность в операторах приведения
- Почему приведение в стиле C не нравится некоторым программистам C++
- Четыре оператора приведения типов C++
- Концепции повышающего и понижающего приведения
- Почему приведение типов C++ — не всегда наилучший выбор

Потребность в приведении типов

В идеальном строго типизированном мире хорошо продуманных приложений C++ не должно быть никакой потребности в приведении типов и операторах приведения. Однако мы живем в реальном мире, где программы разрабатывают по частям множество разных людей и исполнителей, и не редкость необходимость взаимодействия различных систем. Поэтому часто приходится заставлять компилятор интерпретировать данные таким образом, чтобы приложение компилировалось без ошибок и корректно работало.

Рассмотрим реальный пример: хотя большинство компиляторов C++ поддерживают тип `bool` как фундаментальный, множество все еще использующихся библиотек, которые были созданы годы назад на языке C, его не поддерживают. Эти библиотеки созданы для компиляторов C и должны полагаться на использование целочисленного типа для хранения логических данных. Тип `bool` у этих компиляторов выглядит примерно так:

```
typedef unsigned short BOOL;
```

Функция, возвращающая логическое значение, должна быть при этом объявлена как `BOOL IsX()`;

Теперь, если такая библиотека должна использоваться с новым приложением, созданным для последней версии компилятора C++, разработчик должен найти способ сделать логические данные типа `bool`, воспринимаемые компилятором, доступными в виде `BOOL`, понятном библиотеке. Для этого используется приведение типов:

```
bool result = (bool)IsX(); // Приведение в стиле C
```

Развитие языка C++ привело к появлению новых операторов приведения и разделило сообщество разработчиков C++: одни продолжают использовать приведения в стиле C, а другие неукоснительно придерживаются применения ключевых слов приведения типов, введенных новыми стандартами C++. Аргумент первой группы программистов состоит в том, что приведения в стиле C++ громоздки при использовании и иногда немного отличаются по функциональным возможностям от таковых в C (что, впрочем, имеет только теоретическое значение). Вторая группа, которая, очевидно, состоит из фанатиков синтаксиса C++, указывает на возможные недостатки и уязвимости приведения в стиле C.

Поскольку в реальном мире функционирует код обоих видов, имеет смысл прочитать материал этого занятия, чтобы узнать о преимуществах и недостатках каждого стиля и выработать собственное мнение.

Почему приведения в стиле C не нравятся некоторым программистам C++

Безопасность типов (type safety) — один из аргументов, которые приводят программисты C++, восхищаясь качествами этого языка программирования. Фактически большинство компиляторов C++ не позволит вам даже такую мелочь:

```
char* staticStr = "Hello World!";  
int* pBuf = staticStr; // Ошибка: нельзя преобразовать char* в int*
```

...Причем вполне обоснованно!

Современные компиляторы C++ учитывают необходимость обратной совместимости и поддержки устаревшего кода, а потому автоматически разрешают такой синтаксис, как

```
int* pBuf = (int*)pszString; // Устранение одной проблемы создает другую
```

Однако приведения в стиле C фактически вынуждают компилятор интерпретировать целевой тип как тип, который очень удобен разработчику, но в данном случае программист не потрудился задуматься о том, что компилятор не просто так сообщил об ошибке, а имел для этого серьезные основания. Таким образом, программист просто заткнул ему рот, вынуждая повиноваться, и не подумал о возможных последствиях этого шага. Конечно, программиста, стремящегося к неприятностям, не смогут остановить никакие сообщения компилятора об ошибках...

Операторы приведения C++

Несмотря на недостатки приведения типов отказываться от самой их концепции нельзя. Во многих ситуациях приведение — единственная возможность решения важных проблем совместимости. Кроме того, язык C++ предоставляет новый оператор приведения, предназначенный для случаев наследования, которых не существовало в языке C.

В C++ имеется четыре оператора приведения:

- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`
- `const_cast`

Синтаксис их применения одинаков:

```
Целевой_тип результат = Приведение<Целевой_тип>(Приводимый_объект);
```

Использование оператора `static_cast`

Оператор `static_cast` применяется для преобразования указателей связанных типов и выполняет явное преобразование стандартных типов данных, которое в противном случае осуществлялось бы автоматически или неявно. Когда речь идет об указателях, оператор `static_cast` реализует простую проверку времени компиляции приводимости указателя к соответствующему типу. Это является усовершенствованием по сравнению с приведением в стиле C, которое позволяет указателю на один объект быть приведенным к указателю на абсолютно несвязанный тип безо всяких замечаний со стороны компилятора. Используя оператор `static_cast`, указатель можно привести как к базовому классу, так и к производному, что демонстрируется в следующем примере:

```
Base* objBase = new Derived();
Derived* objDer = static_cast<Derived*>(objBase); // OK!

// Класс Unrelated не связан с Base
Unrelated* notRelated = static_cast<Unrelated*>(objBase); // Ошибка!
// Приведение к несвязанному типу не разрешено
```

ПРИМЕЧАНИЕ

Приведение указателя на производный тип к указателю на базовый тип называется *восходящим приведением* (upcasting) и может быть выполнено без явного оператора приведения:

```
Derived objDerived;
Base* pBase = &objDerived; // OK!
```

Приведение указателя на базовый тип к указателю на производный тип называется *нисходящим приведением* (downcasting) и не может быть выполнено без применения явных операторов приведения:

```
Derived objDerived;
Base* pBase = &objDerived; // Восходящее приведение: OK!
Derived* pDerived = pBase; // Ошибка: нисходящее приведение
// должно быть явным
```

Обратите внимание, что оператор `static_cast` проверяет только то, что ссылочные типы связаны. Он *не выполняет* проверок времени выполнения. Таким образом, с оператором `static_cast` разработчик вполне может совершить следующую ошибку:

```
Base* objBase = new Base();
Derived* objDer = static_cast<Derived*>(objBase); // Все OK
```

Здесь указатель `objDer` фактически указывает на частичный объект `Derived`, поскольку фактически объект, на который он указывает, имеет тип `Base`. Так как оператор `static_cast` выполняет только проверку времени компиляции, подтверждая связанность рассматриваемых типов, и не выполняет проверку времени выполнения, вызов `objDer->DerivedFunction()` будет скомпилирован, но, вероятно, приведет к неожиданному поведению во время выполнения.

Помимо помощи в восходящем и нисходящем приведениях, оператор `static_cast` во многих случаях может помочь сделать неявные приведения явными и привлечь к ним внимание разработчика или читателя:

```
double Pi = 3.14159265;
int num = static_cast<int>(Pi); // Делает неявное приведение явным
```

В приведенном выше коде выражение `num=Pi` работало бы не хуже и с тем же успехом. Однако использование оператора `static_cast` привлекает внимание читателя к характеру преобразования и указывает (тому, кто знает оператор `static_cast`), что для выполнения необходимого преобразования типов компилятор выполнил необходимые корректировки на основании информации, доступной во время компиляции.

Оператор `static_cast` необходим также при использовании операторов преобразования или конструкторов, которые были объявлены с использованием ключевого слова `explicit`. Как избежать неявных преобразований с помощью этого ключевого слова, обсуждается на занятиях 9, “Классы и объекты”, и 12, “Типы операторов и их перегрузка”.

Использование оператора `dynamic_cast` и идентификация типа времени выполнения

Динамическое приведение типов, как следует из его названия, является противоположностью статического приведения типов и фактически выполняет приведение времени выполнения. Можно проверить результат выполнения оператора `dynamic_cast` и выяснить, была ли успешной попытка динамического приведения типов. Синтаксис применения оператора `dynamic_cast` имеет следующий вид:

```
Целевой_тип* Dest = dynamic_cast<Целевой_тип*>(Source);
if (Dest) // Проверка успешности приведения типов,
    // прежде чем использовать указатель
    Dest->CallFunc();
```

Например:

```
Base* objBase = new Derived();

// Нисходящее приведение
Derived* objDer = dynamic_cast <Derived*>(objBase);

if (objDer) // Проверка успешности приведения
    objDer->CallDerivedFunction();
```

Как показано в приведенном выше коротком примере, имея указатель на объект базового класса, разработчик может прибегнуть к оператору `dynamic_cast`, чтобы проверить тип целевого объекта, прежде чем перейти к использованию указателя на него. Обратите внимание, что из фрагмента кода кажется, что целевой объект имеет тип `Derived`. Но это пример лишь для демонстрации. Так бывает не всегда, например когда указатель типа `Derived*` передается функции, получающей указатель `Base*`. Функция может применить оператор `dynamic_cast` к переданному указателю типа базового

класса, чтобы выяснить его тип, а затем выполнить операции, специфические для конкретного типа. Таким образом, оператор `dynamic_cast` позволяет определить тип во время выполнения и использовать приведенный указатель, когда это безопасно. Листинг 13.1 использует уже знакомую нам иерархию классов `Tuna` и `Carp`, связанных с базовым классом `Fish`, где функция `DetectFishType()` динамически выясняет, является ли указатель `Fish*` на самом деле указателем `Tuna*` или `Carp*`.

ПРИМЕЧАНИЕ

Данный механизм идентификации типа объекта во время выполнения называется *идентификацией типа времени выполнения* (runtime type identification – RTTI).

ЛИСТИНГ 13.1. Использование динамического приведения типов для выяснения, является ли объект класса `Fish` объектом класса `Tuna` или `Carp`

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:         virtual void Swim()
7:         {
8:             cout << "Рыба плавает в воде" << endl;
9:         }
10:
11:     // Базовый класс должен иметь виртуальный деструктор
12:     virtual ~Fish() {}
13: };
14:
15: class Tuna: public Fish
16: {
17:     public:
18:         void Swim()
19:         {
20:             cout << "Тунец быстро плавает в море" << endl;
21:         }
22:
23:         void BecomeDinner()
24:         {
25:             cout << "Из тунца готовят суши" << endl;
26:         }
27: };
28:
29: class Carp: public Fish
30: {
31:     public:
32:         void Swim()
33:         {
34:             cout << "Карп медленно плавает в озере" << endl;
```

```

35:     }
36:
37:     void Talk()
38:     {
39:         cout << "Карп разговаривает с карпом!" << endl;
40:     }
41: };
42:
43: void DetectFishType(Fish* objFish)
44: {
45:     Tuna* objTuna = dynamic_cast <Tuna*>(objFish);
46:     if (objTuna) // Проверка успешности приведения
47:     {
48:         cout << "Обнаружен тунец: " << endl;
49:         objTuna->BecomeDinner();
50:     }
51:
52:     Carp* objCarp = dynamic_cast <Carp*>(objFish);
53:     if(objCarp)
54:     {
55:         cout << "Обнаружен карп: " << endl;
56:         objCarp->Talk();
57:     }
58:
59:     cout << "Проверка вызовом Fish::Swim: " << endl;
60:     objFish->Swim(); // Вызов виртуальной функции Swim
61: }
62:
63: int main()
64: {
65:     Carp myLunch;
66:     Tuna myDinner;
67:
68:     DetectFishType(&myDinner);
69:     cout << endl;
70:     DetectFishType(&myLunch);
71:
72:     return 0;
73: }

```

Результат

```

Обнаружен тунец:
Из тунца готовят суши
Проверка вызовом Fish::Swim:
Тунец быстро плавает в море

Обнаружен карп:
Карп разговаривает с карпом!
Проверка вызовом Fish::Swim:
Карп медленно плавает в озере

```

Анализ

В этом примере используется иерархия классов `Tuna` и `Carp`, производных от класса `Fish`. С дидактическими целями эти два производных класса не только реализуют виртуальную функцию `Swim()`, но и содержат функции, специфичные для каждого типа, а именно — `Tuna::BecomeDinner()` и `Carp::Talk()`. Особенностью данного примера является то, что, имея указатель на экземпляр базового класса `Fish*`, вы можете динамически обнаружить, не указывает ли он на объект класса `Tuna` или `Carp`. Такое динамическое обнаружение, или идентификация типа времени выполнения, осуществляется в функции `DetectFishType()`, определенной в строках 43–61. В строке 45 оператор `dynamic_cast` используется для проверки входного указателя базового класса типа `Fish*`, не является ли он фактически указателем на тип `Tuna*`. Если этот указатель `Fish*` указывает на тип `Tuna`, оператор возвращает корректный адрес, в противном случае — значение `nullptr`. Следовательно, всегда должна проверяться корректность результата выполнения оператора `dynamic_cast`. После проверки успешности в строке 46 вы знаете, что указатель указывает на допустимый объект класса `Tuna` и его можно использовать для вызова функции `Tuna::BecomeDinner()`, как показано в строке 49. В случае, если передан указатель на объект `Carp`, вы используете его для вызова функции `Carp::Talk()`, как показано в строке 56. Перед выходом функция `DetectFishType()` осуществляет проверку типа, вызвав метод `Fish::Swim()`, который, будучи виртуальным, переадресовывает вызов методу `Swim()`, реализованному в классе `Tuna` или `Carp` соответственно.

ВНИМАНИЕ!

Возвращаемое значение оператора `dynamic_cast` всегда следует проверять на корректность. Если приведение неудачно, возвращается значение `nullptr`.

Использование оператора `reinterpret_cast`

Оператор приведения C++ `reinterpret_cast` ближе всех к приведению в стиле C. Он позволяет разработчику приводить один тип объекта к другому независимо от того, связаны ли их типы:

```
Base * objBase = new Base();
Unrelated * notRelated = reinterpret_cast<Unrelated*>(objBase);
// Код компилируется, но это плохой стиль!
```

Такое приведение фактически заставляет компилятор считать приемлемыми ситуации, которые оператор `static_cast` не пропустил бы. Оно находит применение в некоторых низкоуровневых приложениях (например, таких, как драйверы), в которых данные должны быть преобразованы в простой тип, с которым может работать API (например, ряд функций API работает только с байтовыми потоками, т.е. `unsigned char*`):

```
SomeClass* object = new SomeClass();
// Необходимо передать объект как поток байтов.
unsigned char* bytesForAPI = reinterpret_cast<unsigned char*>(object);
```

Приведение, использованное в показанном фрагменте, не изменяет бинарное представление исходного объекта и фактически обманывает компилятор, разрешая разработчику выбирать отдельные байты, содержащиеся в объекте типа `SomeClass`. Поскольку никакой другой оператор приведения C++ не допускает такого нарушения безопасности типов, оператор `reinterpret_cast` является последним, небезопасным средством.

ВНИМАНИЕ!

По возможности воздержитесь от использования оператора `reinterpret_cast` в ваших приложениях, поскольку он позволяет заставить компилятор рассматривать тип `X` как несвязанный с ним тип `Y`, что плохо и с точки зрения проектирования, и с точки зрения реализации.

Использование оператора `const_cast`

Оператор `const_cast` позволяет отключать модификатор `const` доступа к объекту. Если вы задаетесь вопросом, зачем это приведение нужно вообще, то вы, вероятно, правы. В идеальной ситуации, когда разработчики пишут свои классы правильно, они не забывают использовать ключевое слово `const` и применяют его в правильных местах. На практике все, к сожалению, совсем не так, и код наподобие следующего весьма распространен:

```
class SomeClass
{
public:
    // ...
    void DisplayMembers(); // Хотя функция вывода должна
                          // быть константной
};
```

Если при этом вы создаете такую функцию, как показано ниже, вы сталкиваетесь с запретом компилятора:

```
void DisplayAllData(const SomeClass& object)
{
    object.DisplayMembers(); // Отказ компиляции.
    // Причина отказа: вызов неконстантного члена
    // класса с использованием константной ссылки
}
```

Вы совершенно правы, передавая `object` как константную ссылку. В конце концов, функция отображения предназначена только для чтения и не должна позволять вызывать неконстантные функции-члены, способные изменить состояние объекта. Однако реализация функции `DisplayMembers()`, которая также должна быть константой, к сожалению, таковой не является. Если класс `SomeClass` принадлежит вам и его исходный код находится под вашим контролем, вы можете внести корректирующие изменения в функцию `DisplayMembers()`. Но в большинстве случаев она входит в библиотеку стороннего производителя, и внести в нее изменения вы не в состоянии. В таких ситуациях на выручку приходит оператор `const_cast`.

Синтаксис его применения для функции `DisplayMembers()` следующий:

```
void DisplayAllData(const SomeClass& mData)
{
    SomeClass& refData = const_cast<SomeClass&>(mData);
    refData.DisplayMembers(); // Теперь разрешено!
}
```

Обратите внимание на то, что применение оператора `const_cast` для вызова неконстантных функций должно быть последним средством. Имейте в виду, что использование оператора `const_cast` для изменения константного объекта может привести к неопределенному поведению.

Обратите внимание на то, что оператор `const_cast` применяется и с указателями:

```
void DisplayAllData(const SomeClass* data)
{
    // data->DisplayMembers(); // Ошибка: попытка вызвать
    // неконстантную функцию!
    SomeClass* pCastedData = const_cast<SomeClass*>(data);
    pCastedData->DisplayMembers(); // Разрешено!
}
```

Проблемы с операторами приведения C++

Не все довольны всеми операторами приведения типов C++, даже те, кому они нравятся. Причины недовольства самые разнообразные — от слишком громоздкого и интуитивно непонятного синтаксиса до избыточности.

Просто сравните следующие фрагменты кода:

```
double Pi = 3.14159265;

// Приведение в стиле C++: static_cast
int num = static_cast<int>(Pi); // Результат: num равен 3

// Приведение в стиле C
int num2 = (int)Pi; // Результат: num2 равен 3

// Оставить приведение типов компилятору
int num3 = Pi; // Результат: num3 равен 3
```

Во всех трех случаях достигнут один и тот же результат. На практике, пожалуй, наиболее распространена вторая версия, следующая по распространенности, — третья. Только немногие решаются использовать первый вариант. В любом случае компилятор достаточно интеллектуален, чтобы преобразовывать такие типы правильно. Поэтому, по-видимому, и складывается впечатление, что синтаксис приведения затрудняет чтение кода.

Аналогично другой случай применения оператора `static_cast` также вполне обрабатывается приведениями в стиле C, которые, по общему мнению, выглядят куда проще:

```
// Использование static_cast
Derived* objDer = static_cast <Derived*>(objBase);

// Но этот код работает точно так же:
Derived* objDerSimple = (Derived*)objBase;
```

Таким образом, преимущества использования оператора `static_cast` зачастую омрачаются неуклюжестью его синтаксиса.

Рассмотрим другие операторы. Оператор `reinterpret_cast` позволяет вам все же скомпилировать код, в котором отказывается работать оператор `static_cast`; точно так же применяется оператор `const_cast`, изменяя модификатор доступа `const`. Таким образом, операторов приведения C++, кроме `dynamic_cast`, вполне можно избежать в современных приложениях C++. Применение других операторов приведения становится уместным только тогда, когда требуется использовать устаревшие приложения. В таких случаях предпочтительно использование приведений в стиле C, а использование операторов приведения C++ зачастую является делом вкуса. Однако лучше всего максимально избегать приведений, а когда это не удастся, следует по крайней мере ясно понимать, что при этом происходит.

РЕКОМЕНДУЕТСЯ

Помните, что приведение типа `Derived*` к `Base*` называется восходящим приведением, и оно безопасно.

Помните, что приведение типа `Base*` непосредственно к типу `Derived*` называется нисходящим приведением и может быть небезопасным, если только вы не используете оператор `dynamic_cast` и не проверяете результат его применения.

Помните, что цель создания иерархии наследования обычно заключается в наличии виртуальных функций, при вызове которых с использованием указателей базового класса можно обеспечить доступ к их версии в производном классе.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте проверять корректность указателя, полученного с помощью оператора `dynamic_cast`.

Не проектируйте свои приложения так, чтобы их работоспособность опиралась на применение возможностей RTTI с использованием оператора `dynamic_cast`.

Резюме

На сегодняшнем занятии рассматривались различные операторы приведения C++, аргументы “за” их применение и “против”. Вы также узнали, что применения приведений в общем случае следует избегать.

Вопросы и ответы

- Нормально ли изменять содержимое константного объекта при приведении типа указателя или ссылки на него с использованием оператора `const_cast`?
В большинстве случаев, определенно, нет. Результат такой операции непредсказуем и, определенно, нежелателен.
- Мне нужен указатель `Bird*`, а имеется `Dog*`. Компилятор не позволяет мне использовать указатель на объект `Dog` в качестве `Bird*`. Однако, когда я использую оператор `reinterpret_cast` для приведения типа `Dog*` к типу `Bird*`, компилятор не жалуется, и, кажется, я могу использовать этот указатель для вызова функции-члена `Fly()` класса `Bird`. Все ли у меня в порядке? И вновь, определенно, нет. Оператор `reinterpret_cast` изменяет только интерпретацию указателя, но не объект, на который он указывает (он все еще остается объектом класса `Dog`). Вызов функции `Fly()` объекта класса `Dog` не даст ожидаемых результатов и может привести к неработоспособности приложения.
- У меня есть объект класса `Derived` и указатель на него `objBase`, типа `Base*`. Я уверен, что указатель `objBase` указывает на объект класса `Derived`. Должен ли я использовать оператор `dynamic_cast`?
Если вы абсолютно уверены, что типом объекта, на который он указывает, является `Derived`, можете сэкономить ресурсы исполняющей среды и использовать оператор `static_cast`.
- Язык C++ предоставляет несколько операторов приведения, но автор настоятельно советует их не использовать. Почему?
Вы храните дома аспирин, но не едите его ложкой каждый день только потому, что он есть, правда же? Используйте их, но только когда это на самом деле необходимо и оправданно.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. У вас есть указатель `objBase` на объект базового класса. Какой оператор приведения следует использовать, чтобы выяснить, является ли его типом `Derived1` или `Derived2`?
2. У вас есть константная ссылка на объект, и вы пытаетесь вызвать открытую функцию-член, написанную вами. Компилятор не позволяет сделать это, поскольку рассматриваемая функция — неконстантный член класса. Как вы поступите — исправите функцию или используете оператор `const_cast`?
3. Оператор `reinterpret_cast` для приведения следует использовать только тогда, когда оператор `static_cast` не работает, но при этом известно, что данное приведение необходимо и безопасно. Верно ли данное утверждение?
4. Верно ли, что большинство преобразований, выполняемых оператором `static_cast`, в особенности между простыми типами данных, компилятор C++ способен выполнить автоматически?

Упражнения

1. **Отладка.** Что неправильно в следующем коде?

```
void DoSomething(Base* objBase)
{
    Derived* objDer = dynamic_cast <Derived*>(objBase);
    objDer->DerivedClassMethod();
}
```

2. У вас есть указатель `objFish*`, указывающий на объект класса `Tuna`.

```
Fish* objFish = new Tuna;
Tuna* pTuna = <Какое приведение использовать?>objFish;
```

Какой оператор приведения следует использовать, чтобы получить указатель `Tuna*` на этот объект типа `Tuna`? Продемонстрируйте использующий его код.

ЗАНЯТИЕ 14

Введение в макросы и шаблоны

К настоящему моменту у вас уже должно быть четкое понимание основ синтаксиса языка C++. Вам должны быть понятны исходные тексты, написанные на языке C++, и теперь вы готовы изучать те средства языка, которые помогут писать приложения эффективнее.

На этом занятии...

- Введение в препроцессор
- Ключевое слово `#define` и макросы
- Введение в шаблоны
- Как писать шаблоны функций и классов
- Различие между макросами и шаблонами
- Как использовать ключевое слово C++11 `static_assert` для выполнения проверок времени компиляции

Препроцессор и компилятор

Впервые о препроцессоре вы узнали на занятии 2, “Структура программы на C++”. *Препроцессор* (preprocessor), как свидетельствует его название, запускается перед компилятором. Другими словами, на основании полученных от программиста указаний препроцессор фактически решает, как будет выглядеть компилируемый исходный текст. Все *директивы препроцессора* (preprocessor directive) начинаются со знака #, например:

```
// Указание вставить содержимое заголовочного файла iostream
#include <iostream>

// Определить макрос константы
#define ARRAY_LENGTH 25
int MyNumbers[ARRAY_LENGTH]; // Массив из 25 целых чисел

// Определить макрофункцию
#define SQUARE(x) ((x) * (x))
int TwentyFive = SQUARE(5);
```

На этом занятии рассматриваются два типа директив препроцессора, показанных в представленном выше фрагменте кода; в одном случае директива #define используется для определения константы, а в другом — для определения макрофункции. Обе эти директивы, независимо от их роли, фактически указывают препроцессору заменить каждый экземпляр макроса (ARRAY_LENGTH или SQUARE) значением, которое они определяют.

ПРИМЕЧАНИЕ

Макрос называют также текстовой подстановкой. Препроцессор не делает ничего интеллектуального, просто заменяя некий идентификатор другим текстом.

Использование #define для определения констант

Синтаксис применения директивы #define для определения константы очень прост:

```
#define Идентификатор Значение
```

Например, константа ARRAY_LENGTH, заменяемая значением 25, выше была объявлена следующим образом:

```
#define ARRAY_LENGTH 25
```

Теперь этот идентификатор заменяется текстом 25 везде, где препроцессор его встретит:

```
int numbers[ARRAY_LENGTH] = {0};
double radiuses[ARRAY_LENGTH] = {0.0};
std::string names[ARRAY_LENGTH];
```

После запуска препроцессора эти три строки будут переданы компилятору в таком виде:

```
int numbers[25] = {0};           // Массив из 25 целых чисел
double radiuses[25] = {0.0};    // Массив из 25 чисел типа double
std::string names[25];          // Массив из 25 std::strings
```

Замена применима к любому разделу кода, включая, например, цикл `for`, как показано далее:

```
for(int index = 0; index < ARRAY_LENGTH; ++index)
    numbers[index] = index;
```

Этот цикл `for` компилятор видит таким:

```
for(int index = 0; index < 25; ++index)
    numbers[index] = index;
```

Чтобы увидеть все это в действии, рассмотрим листинг 14.1.

ЛИСТИНГ 14.1. Объявление и использование макросов, определяющих константы

```
0: #include <iostream>
1: #include<string>
2: using namespace std;
3:
4: #define ARRAY_LENGTH 25
5: #define PI          3.1416
6: #define MY_DOUBLE    double
7: #define FAV_WHISKY   "Jack Daniels"
8:
9: int main()
10: {
11:     int numbers[ARRAY_LENGTH] = {0};
12:     cout << "Длина массива: "<<sizeof(numbers)/sizeof(int)<<endl;
13:
14:     cout << "Введите радиус: ";
15:     MY_DOUBLE radius = 0;
16:     cin >> radius;
17:     cout << "Площадь: " << PI * radius * radius << endl;
18:
19:     string favoriteWhisky (FAV_WHISKY);
20:     cout << "Предпочитаю: " << FAV_WHISKY << endl;
21:
22:     return 0;
23: }
```

Результат

Длина массива: 25
Введите радиус: 2.1569
Площадь: 14.7154
Предпочитаю: Jack Daniels

Анализ

ARRAY_LENGTH, PI, MY_DOUBLE и FAV_WHISKY являются четырьмя макроконстантами, определенными в строках 4–7 соответственно. Как можно заметить, первая используется при определении длины массива в строке 11 (которая проверяется с помощью оператора sizeof() в строке 12). MY_DOUBLE используется при объявлении переменной radius типа double в строке 15, а константа PI используется при вычислении площади круга в строке 17. И наконец константа FAV_WHISKY используется при инициализации объекта класса std::string в строке 19 и непосредственно для вывода в поток cout (строка 20). Все эти случаи демонстрируют, что препроцессор осуществляет простую текстовую замену.

У такой “тупой” текстовой замены, которая нашла применение в листинге 14.1, есть и недостатки.

СОВЕТ

Поскольку препроцессор делает лишь простую текстовую подстановку, он не проверяет корректность такой подстановки (что делает компилятор). Вы могли бы определить константу FAV_WHISKY в строке 7 листинга 14.1 так:

```
#define FAV_WHISKY 42 // "Jack Daniels"
```

Это могло бы закончиться ошибкой компиляции в строке 19 при создании экземпляра класса std::string, но при ее отсутствии компилятор продолжил бы работу и вывел следующий текст:

```
Предпочитаю: 42
```

Это, конечно, не имело бы смысла, но важнее всего то, что эта бессмыслица осталась бы необнаруженной компилятором. Кроме того, у вас нет особого контроля над определением константы PI: какой у нее тип – double или float? Ответ: ни тот, ни другой. PI для препроцессора – только текст, заменяемый текстом 3.1416. Ни о каком типе данных нет и речи.

Константы лучше определять, используя ключевое слово const с типами данных. Так намного лучше:

```
const int ARRAY_LENGTH = 25;  
const double PI = 3.1416;  
const char* FAV_WHISKY = "Jack Daniels";  
typedef double MY_DOUBLE; // typedef для псевдонима типа
```

Использование макроса для защиты от множественного включения

Программисты C++, как правило, объявляют свои классы и функции в файлах с расширением .h, называемых *заголовочными файлами* (header file). Соответствующие функции определяются в файлах с расширением .cpp, в которые включают заголовочные файлы, используя директиву препроцессора `#include <заголовочный_файл>`. Если один заголовочный файл (назовем его `class1.h`) объявляет класс, членом которого является другой класс, объявленный в заголовочном файле `class2.h`, то файл `class1.h` должен включать файл `class2.h`. В случае сложного проекта возможна ситуация, когда заголовочный файл `class2.h` требует включения заголовочного файла `class1.h`!

Но для препроцессора два заголовочных файла, которые включают один другой, являются проблемой рекурсивного характера. Чтобы избежать этой проблемы, можно использовать макрос вместе с директивами препроцессора `#ifndef` и `#endif`.

Заголовочный файл `header1.h`, включающий заголовочный файл `header2.h`, выглядит так:

```
#ifndef HEADER1_H_ // Защита от множественного включения:
#define HEADER1_H_ // препроцессор будет читать эту и
                  // последующие строки только один раз
#include <header2.h>

class Class1
{
    // Члены класса
};
#endif // Конец header1.h
```

Заголовочный файл `header2.h` выглядит похоже, но с другим макроопределением, и включает заголовочный файл `header1.h`:

```
#ifndef HEADER2_H_ // Защита от множественного включения
#define HEADER2_H_
#include <header1.h>

class Class2
{
    // Члены класса
};
#endif // Конец header2.h
```

ПРИМЕЧАНИЕ

Директиву `#ifndef` можно прочесть как “если не определено”. Это директива условного выражения, требующая от препроцессора продолжить выполнение, только если идентификатор не был определен.

Директива `#endif` отмечает конец этой условной инструкции препроцессора.

Таким образом, если препроцессор встречает первым заголовочный файл `header1.h`, он выполняет директиву `#ifndef` и, заметив, что идентификатор `HEADER1_H` не был определен, продолжает выполнение. Первая строка после директивы `#ifndef` определяет идентификатор `HEADER1_H`, гарантируя, что вторая попытка препроцессора загрузить этот файл закончится первой же строкой, содержащей директиву `#ifndef`, поскольку теперь это условие будет ложным. То же самое справедливо и для заголовочного файла `header2.h`. Этот простой механизм, возможно, — одна из наиболее часто используемых возможностей макросов при программировании на языке C++.

Использование директивы `#define` для написания макрофункции

Способность препроцессора к простой замене текстовых элементов, идентифицируемых макросом, позволяет писать простые макрофункции, например:

```
#define SQUARE(x) ((x) * (x))
```

Эта функция вычисляет квадрат числа. Аналогично макрос, вычисляющий площадь круга, выглядит следующим образом:

```
#define PI 3.1416
#define AREA_CIRCLE(r) (PI*(r)*(r))
```

Макрофункции (macro function) нередко используются для подобных очень простых вычислений. Они предоставляют то преимущество, что обычный вызов функций, которым они выглядят, раскрывается в код, встраивающийся в исходный текст перед компиляцией, а следовательно, макрофункции могут в определенных ситуациях повысить производительность кода. Листинг 14.2 демонстрирует использование этих макрофункций.

ЛИСТИНГ 14.2. Использование макрофункций, вычисляющих квадрат числа, площадь круга, а также наибольшее и наименьшее из двух чисел

```
0: #include <iostream>
1: #include<string>
2: using namespace std;
3:
4: #define SQUARE(x) ((x) * (x))
5: #define PI 3.1416
6: #define AREA_CIRCLE(r) (PI*(r)*(r))
7: #define MAX(a, b) (((a) > (b)) ? (a) : (b))
8: #define MIN(a, b) (((a) < (b)) ? (a) : (b))
9:
10: int main()
11: {
12:     cout << "Введите целое число: ";
13:     int num = 0;
```

```
14:    cin >> num;
15:
16:    cout << "SQUARE(" << num << ") = " << SQUARE(num) << endl;
17:    cout << "Площадь круга с радиусом " << num << " равна: ";
18:    cout << AREA_CIRCLE(num) << endl;
19:
20:    cout << "Введите другое целое число: ";
21:    int num2 = 0;
22:    cin >> num2;
23:
24:    cout << "MIN(" << num << ", " << num2 << ") = ";
25:    cout << MIN(num, num2) << endl;
26:
27:    cout << "MAX(" << num << ", " << num2 << ") = ";
28:    cout << MAX(num, num2) << endl;
29:
30:    return 0;
31: }
```

Результат

```
Введите целое число: 36
SQUARE(36) = 1296
Площадь круга с радиусом 36 равна: 4071.51
Введите другое целое число: -101
MIN(36, -101) = -101
MAX(36, -101) = 36
```

Анализ

Строки 4–8 содержат несколько вспомогательных макрофункций, которые возвращают квадрат числа, площадь круга, а также наибольшее и наименьшее из двух чисел. Обратите внимание, что функция `AREA_CIRCLE` в строке 6 вычисляет площадь с использованием константы `PI`, свидетельствуя, таким образом, что один макрос может повторно использовать другой макрос. В конце концов, это всего лишь команды препроцессора для простой замены одного текста другим. Давайте рассмотрим строку 25, в которой используется макрофункция `MIN`:

```
cout << MIN(num, num2) << endl;
```

После раскрытия макроса эта строка передается компилятору в следующем виде:

```
cout << ((num) < (num2)) ? (num) : (num2) << endl;
```

ВНИМАНИЕ!

Макрофункции ничего не знают о типах, а потому могут быть опасны. Например, макрофункция `AREA_CIRCLE` в идеале должна возвращать значение типа `double` независимо от того, какое значение радиуса ей передается.

Зачем все эти скобки?

Еще раз взгляните на макрофункцию вычисления площади круга:

```
#define AREA_CIRCLE(r) (PI*(r)*(r))
```

У этого выражения странный синтаксис с множеством скобок. Сравните его с функцией `Area()` из листинга 7.1 занятия 7, “Организация кода с помощью функций”.

```
// Определения функций (реализации)
double Area(double radius)
{
    return Pi * radius * radius; // Никаких скобок
}
```

Так почему же для макроса мы так усердствуем в расставлении скобок, если та же формула в функции обходится без них? Причина — в способе обработки макроса препроцессором, т.е. в механизме текстовой подстановки.

Рассмотрим макрос без множества скобок:

```
#define AREA_CIRCLE(r) (PI*r*r)
```

Что произойдет при следующем использовании этого макроса?

```
cout << AREA_CIRCLE (4+6);
```

Он будет развернут препроцессором в такой код:

```
cout << (PI*4+6*4+6); // Совсем не то же, что и PI*10*10
```

Таким образом, с учетом приоритета операций, согласно которым умножение выполняется до сложения, компилятор фактически вычисляет площадь так:

```
cout << (PI*4+24+6); // 42.5664 (что явно неправильно)
```

Без круглых скобок преобразование текста привело к искажению логики программы! Применение круглых скобок позволяет избежать этой проблемы:

```
#define AREA_CIRCLE(r) (PI*(r)*(r))
cout << AREA_CIRCLE (4+6);
```

Выражение после подстановки воспринимается компилятором как следующее:

```
cout << (PI*(4+6)*(4+6)); // PI*10*10, как и ожидалось
```

Скобки автоматически обеспечивают правильное вычисление площади, делая код макроса независимым от приоритета операторов.

Использование макроса `assert` для проверки выражений

Хотя, конечно, следует тестировать каждый путь выполнения кода непосредственно после его написания, это зачастую оказывается физически невозможным. Но можно хотя бы вставить в код проверки, тестирующие значения выражений или переменных.

Макрос `assert` позволяет выполнять такие проверки. Чтобы использовать макрос `assert`, необходимо включить в код заголовочный файл `assert.h`; синтаксис же использования этого макроса следующий:

```
assert (выражение, возвращающее true или false);
```

Вот простой пример использования макроса `assert()`, проверяющего содержимое указателя:

```
#include <assert.h>
int main()
{
    char* sayHello = new char[25];
    assert(sayHello != nullptr); // Сообщить о нулевом указателе

    // Прочий код

    delete[] sayHello;
    return 0;
}
```

Макрос `assert()` гарантирует вывод уведомления, если указатель окажется нулевым. Для проверки я инициализировал указатель `sayHello` значением `nullptr` и при запуске в режиме отладки Visual Studio немедленно получил на экране окно, показанное на рис. 14.1.

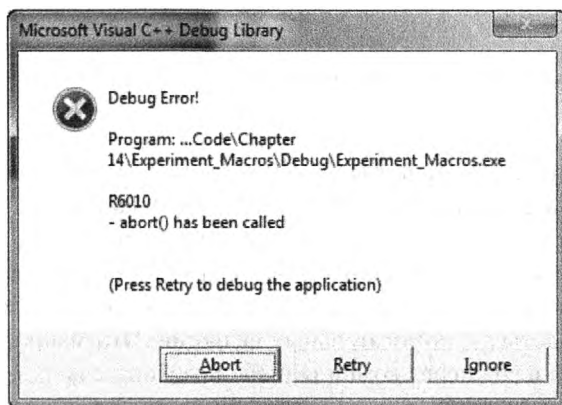


РИС. 14.1. Макрос `assert` обнаружил некорректный указатель

Таким образом, макрос `assert()`, реализованный в отладочном режиме Microsoft Visual Studio, позволяет щелкнуть на кнопке `Retry` (Повторить) и вернуться в приложение, а стек вызовов укажет строку, приведшую к нарушению условия макроса `assert`. Это делает макрос `assert()` весьма удобным средством отладки; например, с его помощью можно проверять входные параметры функций. Это настоятельно рекомендуется и позволяет улучшить качество вашего кода.

ПРИМЕЧАНИЕ

В производственной версии макрос `assert()` обычно отключен и выводит сообщения об ошибках и иную информацию только в отладочном режиме большинства сред разработки.

Кроме того, некоторые среды реализуют его как функцию, а не как макрос.

ВНИМАНИЕ!

Поскольку макрос `assert` не работает в производственных версиях, тесты, критически важные для функционирования приложения (например, возвращаемое значение оператора `dynamic_cast`), следует выполнять с использованием инструкции `if`. Макрос `assert` позволяет обнаруживать проблемы, но не является заменой проверки значения указателя, необходимой в коде.

Преимущества и недостатки использования макрофункций

Макросы позволяют повторно использовать некоторые вспомогательные функции независимо от типа используемых переменных. Вернемся к следующей строке из листинга 14.2:

```
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
```

Эту макрофункцию `MIN` можно использовать для целых чисел:

```
cout << MIN(25, 101) << endl;
```

Но ее же можно использовать для типа `double`:

```
cout << MIN(0.1, 0.2) << endl;
```

Обратите внимание, что, если бы функция `MIN()` была обычной функцией, вам пришлось бы создать два ее варианта: `MIN_INT()`, получающий параметры типа `int` и возвращающий тип `int`, и `MIN_DOUBLE()`, делающий то же самое, но с типом `double`. Такая оптимизация и сокращение количества кода представляют собой определенное преимущество и соблазняют некоторых программистов на использование макросов для определения простых вспомогательных функций. Эти макрофункции раскрываются и встраиваются в код перед компиляцией, а следовательно, производительность простого макроса выше, чем обычного вызова функции, решающего ту же задачу. Это связано с тем, что вызов функции требует создания стека вызовов, передачи аргументов и так далее, так что дополнительные затраты зачастую отнимают больше процессорного времени, чем работа самой функции `MIN`.

Несмотря на все эти преимущества макросы представляют серьезную проблему: они не поддерживают никаких форм безопасности типов. Если этого недостаточно, то учтите, что отладка макроса — также весьма непростое дело.

Если необходимо создание обобщенных функций, которые не зависят от типа, но при этом поддерживают безопасность типов, вместо макрофункции лучше использовать шаблон функции. Если необходимо повышение производительности, объявите свою функцию встраиваемой (`inline`).

Вы уже познакомились со встраиваемыми функциями и использованием ключевого слова `inline` в листинге 7.10 на занятии 7, “Организация кода с помощью функций”.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
<p>Создавайте собственные макрофункции по возможности реже.</p> <p>Используйте по возможности константы вместо макросов.</p> <p>Помните, что макросы небезопасны с точки зрения типов, а препроцессор не выполняет никаких проверок типов.</p>	<p>Не забывайте заключать в скобки каждую переменную в определении макрофункции.</p> <p>Не забывайте использовать в ваших заголовочных файлах защиту от множественного включения, используя директивы <code>#ifndef</code>, <code>#define</code> и <code>#endif</code>.</p> <p>Не забывайте снабжать свой код макросами <code>assert()</code> — в окончательной версии они не выполняют никаких действий, но облегчают отладку и тестирование вашего кода в процессе разработки.</p>

Теперь пришло время изучения практики обобщенного программирования с использованием шаблонов!

Введение в шаблоны

Шаблоны (template) — это, вероятно, одно из самых мощных средств языка C++, на которые зачастую не обращают внимания или не понимают. Прежде чем заняться этой темой, сначала посмотрим, как определяется термин “шаблон” в словаре Вебстера.

Функция: существительное

Этимология: вероятно, от французского *templet*, уменьшительное от *temple* (храм), часть ткацкого станка, вероятно, от латинского *templum*

Дата: 1677

1. Короткий элемент или блок, располагающийся горизонтально в стене под балкой, чтобы распределить ее вес или давление (как над дверью).
2. (1). Лекало, выкройка или шаблон (как тонкая пластина или лист), используемые как направляющие при вырезании детали сложной формы; (2) а: молекула (как ДНК), которая служит шаблоном для создания другой макромолекулы (как РНК); b: перекрытие.
3. Нечто устанавливающее или служащее образцом.

Последнее определение, вероятно, ближе всего к интерпретации слова *шаблон* при использовании в языке C++. Шаблоны в языке C++ позволяют определить действие, которое можно применить к объектам разных типов. Это звучало бы зловеще близко к тому, что позволяет делать макрос (посмотрите на простой макрос `MAX`, который

определял большее из двух чисел), если бы не тот факт, что в отличие от макросов шаблоны обеспечивают безопасность типов.

Синтаксис объявления шаблона

Объявление шаблона начинается с ключевого слова `template`, сопровождаемого списком *параметров типа* (type parameter). Формат объявления таков:

```
template <список параметров>
объявление шаблона функции / класса..
```

Ключевое слово `template` отмечает начало объявления шаблона, а далее следует список параметров шаблона. Этот список параметров содержит ключевое слово `typename`, которое определяет параметр шаблона, делая его заполнителем для типа объекта, для которого создается экземпляр шаблона.

```
template <typename T1, typename T2 = T1>
bool TemplateFunction(const T1& param1, const T2& param2);
```

```
// Шаблон класса
template <typename T1, typename T2 = T1>
class Template
{
private:
    T1 member1;
    T2 member2;
public:
    T1 GetObj1() {return member1; }
    // ... Другие члены
};
```

Здесь представлены шаблон функции и шаблон класса, каждый из которых получает два параметра шаблона `T1` и `T2`, где параметр `T2` имеет заданный по умолчанию тип `T1`.

Типы объявлений шаблонов

Объявление шаблона может быть следующим:

- объявление или определение функции;
- объявление или определение класса;
- определение функции-члена или класса-члена шаблона класса;
- определение статической переменной-члена шаблона класса;
- определение статической переменной-члена класса, вложенного в шаблон класса;
- определение шаблона-члена класса или шаблона класса.

Шаблонные функции

Вообразите функцию, которая сама приспосабливается к параметрам различных типов. Такая функция вполне возможна — при использовании синтаксиса шаблона! Давайте проанализируем типичное объявление шаблона, который является эквивалентом обсуждавшегося ранее макроса `MAX`, который возвращает больший из двух переданных параметров:

```
template <typename objType>
const objType& GetMax(const objType& value1,
                     const objType& value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

Вот как выглядят примеры его применения:

```
int num1 = 25;
int num2 = 40;
int maxVal = GetMax<int>(num1, num2);
double double1 = 1.1;
double double2 = 1.001;
double MaxDVal = GetMax<double>(double1, double2);
```

Обратите внимание на фрагмент `<int>`, использованный в вызове функции `GetMax()`. Фактически это определение параметра шаблона `objType` как типа `int`. Приведенный выше код заставляет компилятор создать две версии функции `GetMax()`, которые можно представить так:

```
const int& GetMax(const int& value1, const int& value2)
{
    //...
}
const double& GetMax(const double& value1, const double& value2)
{
    // ...
}
```

Однако в действительности шаблонные функции не обязательно нуждаются в соответствующем спецификаторе типа. Так, отлично сработает следующий вызов:

```
int MaxValue = GetMax(num1, num2);
```

Компиляторы достаточно интеллектуальны, чтобы понять, что шаблон функции вызывается для того или иного типа, как показано в листинге 14.3.

ЛИСТИНГ 14.3. Шаблон функции GetMax для вычисления большего из двух чисел

```
0: #include<iostream>
1: #include<string>
2: using namespace std;
3:
4: template <typename Type>
5: const Type& GetMax(const Type& value1, const Type& value2)
6: {
7:     if (value1 > value2)
8:         return value1;
9:     else
10:        return value2;
11: }
12:
13: template <typename Type>
14: void DisplayComparison(const Type& value1, const Type& value2)
15: {
16:     cout << "GetMax(" << value1 << ", " << value2 << ") = ";
17:     cout << GetMax(value1, value2) << endl;
18: }
19:
20: int main()
21: {
22:     int num1 = -101, num2 = 2011;
23:     DisplayComparison(num1, num2);
24:
25:     double d1 = 3.14, d2 = 3.1416;
26:     DisplayComparison(d1, d2);
27:
28:     string name1("Jack"), name2("John");
29:     DisplayComparison(name1, name2);
30:
31:     return 0;
32: }
```

Результат

```
GetMax(-101, 2011) = 2011
GetMax(3.14, 3.1416) = 3.1416
GetMax(Jack, John) = John
```

Анализ

Это пример возможностей двух шаблонов функций: GetMax() (строки 4–11), которая используется функцией DisplayComparison() (строки 13–18). Строки 23, 26 и 29 функции main() демонстрируют многократное использование одного и того же шаблона функции для совершенно разных типов данных: int, double и std::string. Кроме того что этот шаблон функции используется повторно (точно так же, как и

его аналог-макрос), его проще создать и поддерживать и он к тому же обеспечивает безопасность типов!

Обратите внимание, что функцию `DisplayComparison()` вполне возможно вызвать с явным указанием типа:

```
23:     DisplayComparison<int>(num1, num2);
```

Однако при вызове шаблонных функций это условие не является обязательным. Вы не обязаны указывать тип(ы) параметров шаблона, поскольку компилятор в состоянии вывести их самостоятельно. Тем не менее это необходимо при работе с шаблонами классов.

Шаблоны и безопасность типов

Шаблонные функции `DisplayComparison()` и `GetMax()`, представленные в листинге 14.3, обеспечивают безопасность типов. Это значит, что они не позволят такой, например, бессмысленный вызов:

```
DisplayComparison(num1, name1);
```

Это немедленно привело бы к ошибке компиляции.

Шаблонные классы

На занятии 9, “Классы и объекты”, упоминалось, что классы — это программные блоки, инкапсулирующие определенные атрибуты и методы, работающие с этими атрибутами. Атрибуты, как правило, — это закрытые члены, такие как `int Age` в классе `Human`. Классы — это только чертежи проекта, а реальным представлением класса являются его объекты. Так, например, Том может быть объектом класса `Human` с атрибутом `Age`, содержащим значение 15. Мы подразумеваем, что это годы. Но что если по каким-то причинам возраст нужно хранить в секундах, и типа `int` окажется недостаточно, так что вместо него потребуется использовать тип `long long`? Здесь могли бы пригодиться шаблонные классы. *Шаблонный класс* (template class) представляет собой шаблонную версию классов C++. Фактически это чертежи чертежей. При использовании шаблона класса появляется возможность определить *тип*, который специализирует класс. Это позволяет создать одни объекты класса `Human` с параметром шаблона `Age` типа `long long`, другие — типа `int`, а третьи — типа `short`.

Простой пример шаблона класса с одним параметром шаблона `T` для переменной-члена может быть написан следующим образом:

```
template <typename T>
class HoldVarTypeT {
private:
    T value;
public:
    void SetValue(const T & newValue) {
        value = newValue;
    }
}
```



```

    T & GetValue() {
        return value;
    }
};

```

Типом переменной `value` является `T` — тип, который назначается во время использования шаблона, т.е. его *инстанцирования*. Рассмотрим типичное применение этого шаблона класса:

```

HoldVarTypeT <int> holdInt; // Инстанцирование шаблона для int
holdInt.SetValue(5);
cout << "Сохраненное значение - " << holdInt.GetValue() << endl;

```

Мы использовали этот шаблон класса для хранения и возврата объекта типа `int`, т.е. шаблон класса инстанцируется для параметра шаблона `int`. Точно так можно использовать этот же класс для работы с символьными строками:

```

HoldVarTypeT <char * > holdStr;
holdStr.SetValue("Sample string");
cout << "Сохраненное значение - " << holdStr.GetValue() << endl;

```

Таким образом, шаблонный класс определяет схему, которую многократно использует для различных типов данных, с которыми инстанцируется этот шаблон.

СОВЕТ

Шаблонные классы могут быть инстанцированы не только простыми типами наподобие `int`, `char*` или классами стандартной библиотеки. Вы можете инстанцировать шаблонные классы с помощью собственноручно написанных классов. Например, внеся код, который определяет шаблонный класс `HoldVarTypeT` в листинг 9.1 занятия 9, «Классы и объекты», вы сможете инстанцировать шаблон для класса `Human`, добавляя следующий код в функцию `main()`:

```

HoldVarTypeT<Human> holdHuman;
holdHuman.SetValue(firstMan);
holdHuman.GetValue().IntroduceSelf();

```

Объявление шаблонов с несколькими параметрами

Список параметров шаблона может быть расширен и содержать несколько параметров, разделенных запятой. Так, если вы хотите объявить обобщенный класс, содержащий два объекта, типы которых могут различаться, можете использовать конструкции, показанные в следующем примере (в котором приведен шаблон класса с двумя параметрами шаблона):

```

template <typename T1, typename T2>
class HoldsPair
{
private:
    T1 value1;

```

```
T2 value2;
public:
    // Конструктор, инициализирующий переменные-члены
    HoldsPair(const T1& val1, const T2& val2)
    {
        value1 = val1;
        value2 = val2;
    };
    // ... Другие функции-члены
};
```

Здесь класс `HoldsPair` получает два параметра шаблона с именами `T1` и `T2`. Мы можем использовать этот класс для хранения двух объектов одинаковых или разных типов:

```
// Создание экземпляра шаблона для типов int и double
HoldsPair <int, double> pairIntDouble(6, 1.99);

// Создание экземпляра шаблона для типов int и int
HoldsPair <int, int> pairIntDouble(6, 500);
```

Объявление шаблонов параметрами по умолчанию

Можно изменить предыдущую версию шаблона `HoldsPair` <...> так, чтобы объявить тип `int` как заданный по умолчанию тип параметра шаблона:

```
template <typename T1=int, typename T2=int>
class HoldsPair
{
    // ... Объявления методов
};
```

Это очень похоже на определение значений по умолчанию для входных параметров функций, но в данном случае мы определяем заданные по умолчанию *типы*. В этом случае применение шаблона `HoldsPair` может быть сжато до следующего:

```
// Создание экземпляра шаблона для типов int и int (тип по умолчанию)
HoldsPair <> pairIntDouble(6, 500);
```

Простой шаблон класса `HoldsPair`

Пришло время дальнейшего усовершенствования версии шаблона `HoldsPair`. Рассмотрим листинг 14.4.

ЛИСТИНГ 14.4. Шаблон класса с двумя атрибутами

```
0: #include <iostream>
1: using namespace std;
2:
3: // Шаблон с параметрами по умолчанию: int и double
```

```
4: template <typename T1=int, typename T2=double>
5: class HoldsPair
6: {
7:     private:
8:         T1 value1;
9:         T2 value2;
10:    public:
11:        HoldsPair(const T1& val1, const T2& val2) // Конструктор
12:            : value1(val1), value2(val2) {}
13:
14:        // Функции доступа
15:        const T1 & GetFirstValue() const
16:        {
17:            return value1;
18:        }
19:
20:        const T2 & GetSecondValue() const
21:        {
22:            return value2;
23:        }
24: };
25:
26: int main()
27: {
28:     HoldsPair<> pairIntDbl(300, 10.09);
29:     HoldsPair<short, const char*> pairShortStr(25, "Шаблон");
30:
31:     cout << "Первый объект содержит:" << endl;
32:     cout << "value1: " << pairIntDbl.GetFirstValue() << endl;
33:     cout << "value2: " << pairIntDbl.GetSecondValue() << endl;
34:
35:     cout << "Второй объект содержит:" << endl;
36:     cout << "value1: " << pairShortStr.GetFirstValue() << endl;
37:     cout << "value2: " << pairShortStr.GetSecondValue() << endl;
38:
39:     return 0;
40: }
```

Результат

```
Первый объект содержит:
value1: 300
value2: 10.09
Второй объект содержит:
value1: 25
value2: Шаблон
```

Анализ

Эта простая программа демонстрирует объявление шаблона класса `HoldsPair`, содержащего значения двух типов, зависящих от списка параметров шаблона. В строке 1 содержится список параметров шаблона, определяющий два параметра шаблона, `T1` и `T2`, с заданными по умолчанию типами `int` и `double` соответственно. Функции доступа, `GetFirstValue()` и `GetSecondValue()`, применяются для доступа к значениям, содержащимся в объекте. Обратите внимание, как функции `GetFirstValue()` и `GetSecondValue()` адаптированы для возвращения объектов соответствующих типов на основании синтаксиса создания экземпляра шаблона. Теперь у нас определен шаблон `HoldsPair`, который можно повторно использовать для предоставления одинаковой логики обработки переменных различных типов. Таким образом, шаблоны повышают уровень повторного использования кода.

Инстанцирование и специализация шаблона

Шаблонный класс является схемой класса, а потому не существует для компилятора в реальности до тех пор, пока не будет использован в том или ином виде. Что касается компилятора, то шаблонный класс, который вы определили, но не использовали в коде, просто игнорируется. Однако вы *инстанцируете* шаблонный класс, такой как `HoldsPair`, указывая аргументы шаблона, например, следующим образом:

```
HoldsPair<int, double> pairIntDbl;
```

Вы поручаете компилятору создать класс с использованием шаблона и инстанцировать его для типов, указанных в качестве аргументов шаблона (в данном случае — `int` и `double`). Таким образом, для шаблонов *инстанцирование* представляет собой акт, или процесс, создания определенного типа с использованием одного или нескольких аргументов шаблона.

С другой стороны, могут быть ситуации, которые требуют явного определения (различного) поведения шаблона при инстанцировании с определенным типом. Этот процесс называется *специализацией* шаблона для данного типа. Специализация шаблона класса `HoldsPair` при инстанцировании с параметрами типа `int` будет выглядеть следующим образом:

```
template<> class HoldsPair<int, int> {  
    // Код реализации для данных типов  
};
```

Излишне говорить, что код, который специализирует шаблон, должен соответствовать определению шаблона. Листинг 14.5 является примером специализации шаблона, демонстрирующим, насколько сильно различные специализированные версии могут отличаться от шаблона, который они специализируют.

ЛИСТИНГ 14.5. Демонстрация специализации шаблона

```
0: #include <iostream>  
1: using namespace std;  
2:
```

```
3: template <typename T1 = int, typename T2 = double>
4: class HoldsPair
5: {
6:     private:
7:         T1 value1;
8:         T2 value2;
9:     public:
10:         HoldsPair(const T1& val1, const T2& val2) // Конструктор
11:             : value1(val1), value2(val2) {}
12:
13:         // Функции доступа
14:         const T1 & GetFirstValue() const;
15:         const T2& GetSecondValue() const;
16:     };
17:
18: // Специализация HoldsPair для двух int
19: template<> class HoldsPair<int, int>
20: {
21:     private:
22:         int value1;
23:         int value2;
24:         string strFun;
25:     public:
26:         HoldsPair(const int& val1, const int& val2) // Конструктор
27:             : value1(val1), value2(val2) {}
28:
29:         const int & GetFirstValue() const
30:         {
31:             cout << "Возвращает " << value1 << endl;
32:             return value1;
33:         }
34: };
35:
36: int main()
37: {
38:     HoldsPair<int, int> pairIntInt(222, 333);
39:     pairIntInt.GetFirstValue();
40:
41:     return 0;
42: }
```

Вывод

Возвращает 222

Анализ

Очевидно, что если вы сравните поведение класса `HoldsPair` в листинге 14.4 и в этом листинге, то заметите, что шаблон ведет себя совсем иначе. В самом деле, функция

`GetFirstValue()` изменена при инстанцировании шаблона `HoldsPair<int, int>` так, что не только возвращает значение, но и выводит его. Внимательное рассмотрение кода специализации в строках 18–34 показывает, что данная версия шаблона имеет дополнительный член-строку, объявленный в строке 24, — член, который отсутствует в определении исходного шаблона `HoldsPair<>` в строках 3–16. Более того, определение исходного шаблона даже не предоставляет реализацию функций доступа `GetFirstValue()` и `GetSecondValue()`, но программа по-прежнему компилируется. Дело в том, что компилятору требуется рассмотреть лишь инстанцирование шаблона для параметров `<int, int>`, для которых имеется достаточно полная специализированная реализация. Таким образом, этот пример демонстрирует не только специализацию шаблона, но и то, как рассматривается (или даже игнорируется) код шаблона в зависимости от его применения.

Шаблонные классы и статические члены

Как уже упоминалось, код в шаблонах начинает свое существование для компилятора только тогда, когда используется в программе, и никак иначе. А что можно сказать о статическом члене шаблонного класса? Из занятия 9, “Классы и объекты”, вы знаете, что объявление члена статическим приводит к тому, что он совместно используется всеми экземплярами класса. То же самое справедливо и для шаблонного класса — статический член совместно используется всеми экземплярами класса с одними и теми же параметрами. Так, статический член `X` в шаблонном классе является статическим для всех экземпляров класса, инстанцированных для типа `int`. Такой же статический член `X` является статическим для всех экземпляров класса, инстанцированных для типа `double`, и при этом никак не связан со статическим членом `X` для `int`. Другими словами, вы можете представить это как создание компилятором двух версий статического члена шаблонного класса: `X_int` — для первого и `X_double` — для второго случая (листинг 14.6).

ЛИСТИНГ 14.6. Статические переменные шаблонного класса

```

0: #include <iostream>
1: using namespace std;
2:
3: template <typename T>
4: class TestStatic
5: {
6:     public:
7:         static int staticVal;
8: };
9:
10: // Инициализация статического члена
11: template<typename T> int TestStatic<T>::staticVal;
12:
13: int main()
14: {
15:     TestStatic<int> intInstance;
```

```

16:     cout << "staticVal для int равен 2011" << endl;
17:     intInstance.staticVal = 2011;
18:
19:     TestStatic<double> dblInstance;
20:     cout << "staticVal для double равен 1011" << endl;
21:     dblInstance.staticVal = 1011;
22:
23:     cout << "intInstance: " << intInstance.staticVal << endl;
24:     cout << "dblInstance: " << dblInstance.staticVal << endl;
25:
26:     return 0;
27: }

```

Результат

```

staticVal для int равен 2011
staticVal для double равен 1011
intInstance: 2011
dblInstance: 1011

```

Анализ

В строках 17 и 21 устанавливаются значения члена `staticVal` экземпляров шаблона для типов `int` и `double` соответственно. Вывод на экран демонстрирует, что компилятор хранит два разных значения в двух разных статических членах, имена которых — `staticVal`. Таким образом, компилятор гарантирует, что поведение статической переменной остается неизменным для специализации шаблонного класса для конкретного типа.

ПРИМЕЧАНИЕ

Обратите внимание на синтаксис создания экземпляра статического члена для шаблона класса в строке 11 листинга 14.6.

```
template<typename T> int TestStatic<T>::staticVal;
```

Он следует общей схеме:

```
template<параметры шаблона>Тип_члена
Имя_Класса<Аргументы шаблона>::Имя_Статического_члена;
```

Шаблоны с переменным количеством параметров (вариадические шаблоны)

Предположим, что вы хотите написать обобщенную функцию, которая суммирует два значения. Шаблон функции `Sum()` делает только это:

```

template <typename T1, typename T2, typename T3>
void Sum(T1 & result, T2 num1, T3 num2) {

```

```
    result = num1 + num2;
    return;
}
```

Здесь все просто. Однако, если вам требуется написать одну функцию, которая могла бы складывать любое количество значений, каждое из которых передается в качестве аргумента, то вам нужно использовать в определении такой функции вариадические шаблоны. Такие шаблоны являются частью C++ начиная со стандарта C++ 14, выпущенного в 2014 году. В листинге 14.7 демонстрируется использование вариадических шаблонов в определении обобщенной функции.

ЛИСТИНГ 14.7. Применение вариадических шаблонов

```
0: #include <iostream>
1: using namespace std;
2:
3: template <typename Res, typename ValType>
4: void Sum(Res& result, ValType& val)
5: {
6:     result = result + val;
7: }
8:
9: template <typename Res, typename First, typename... Rest>
10: void Sum(Res& result, First vall, Rest... valN)
11: {
12:     result = result + vall;
13:     return Sum(result, valN ...);
14: }
15:
16: int main()
17: {
18:     double dResult = 0;
19:     Sum(dResult, 3.14, 4.56, 1.1111);
20:     cout << "dResult = " << dResult << endl;
21:
22:     string strResult;
23:     Sum(strResult, "Hello ", "World");
24:     cout << "strResult = " << strResult.c_str() << endl;
25:
26:     return 0;
27: }
```

Вывод

```
dResult = 8.8111
strResult = Hello World
```


Анализ

В этом примере демонстрируется, что функция `Sum()`, которую мы определили с использованием вариадических шаблонов, может не только работать с совершенно различными типами аргументов, как показано в строках 19 и 23, но и справиться с различным числом аргументов. Функция `Sum()`, вызванная в строке 19, получает четыре аргумента, а в строке 23 — три аргумента, один из которых представляет собой `std::string`, а следующие два — `const char *`. Во время компиляции компилятор создает код функции `Sum()`, который корректно выполняет все требуемые вычисления с помощью рекурсивных вызовов, пока не будут обработаны все переданные функции аргументы.

ПРИМЕЧАНИЕ

В примере кода вы могли заметить использование многоточия. В шаблонах C++ оно используется для того, чтобы сообщить компилятору, что шаблон класса или функции может принимать произвольное количество аргументов шаблона любого типа.

Такие шаблоны с переменным количеством аргументов являются мощным дополнением к C++, которое находит применение как при математической обработке данных, так и при выполнении ряда простых задач. Программисты с помощью шаблонов с переменным количеством аргументов спасаются от повторяющихся действий по реализации функций, выполняющих задачи в различных перегруженных версиях, создавая более короткий код, который проще поддерживать.

ПРИМЕЧАНИЕ

C++14 обеспечивает вас оператором, который может сообщить вам количество аргументов, переданных в вызове шаблона с переменным количеством аргументов. В листинге 14.7 этот оператор можно использовать внутри функции как `Sum()` следующим образом:

```
int arrNums[sizeof...(Rest)];  
// Длина массива вычисляется с помощью sizeof...() во время  
компиляции
```

Не путайте `sizeof...()` с `sizeof(Type)!` Последнее выражение возвращает размер типа, в то время как первое выражение возвращает количество аргументов шаблона, переданных вариадическому шаблону.

Поддержка шаблонов с переменным числом аргументов открыла возможность стандартной поддержки *кортежей*. Шаблонный класс, реализующий кортежи, — `std::tuple`. Он может быть создан с различным количеством элементов и их типов. Эти элементы могут быть доступны индивидуально с помощью функции стандартной библиотеки `std::get`. В листинге 14.8 демонстрируется создание и использование экземпляра `std::tuple`.

ЛИСТИНГ 14.8. Инстанцирование и использование `std::tuple`

```
0: #include <iostream>
1: #include <tuple>
2: #include <string>
3: using namespace std;
4:
5: template <typename tupleType>
6: void DisplayTupleInfo(tupleType& tup)
7: {
8:     const int numMembers = tuple_size<tupleType>::value;
9:     cout<<"Элементов в кортеже: "<< numMembers << endl;
10:    cout<<"Последний элемент: "<< get<numMembers-1>(tup) << endl;
11: }
12:
13: int main()
14: {
15:     tuple<int,char,string>tup1(make_tuple(101,'s',"Hello Tuple!"));
16:     DisplayTupleInfo(tup1);
17:
18:     auto tup2(make_tuple(3.14, false));
19:     DisplayTupleInfo(tup2);
20:
21:     auto concatTup(tuple_cat(tup2, tup1)); // Члены tup2, tup1
22:     DisplayTupleInfo(concatTup);
23:
24:     double pi;
25:     string sentence;
26:     tie(pi, ignore, ignore, ignore, sentence) = concatTup;
27:     cout << "Pi: " << pi << " и \"" << sentence << "\"" << endl;
28:
29:     return 0;
30: }
```

Вывод

```
Элементов в кортеже: 3
Последний элемент: Hello Tuple!
Элементов в кортеже: 2
Последний элемент: 0
Элементов в кортеже: 5
Последний элемент: Hello Tuple!
Pi: 3.14 и "Hello Tuple!"
```

Анализ

Прежде всего, если код в листинге 14.8 кажется для вас слишком сложным, не волнуйтесь. Кортежи представляют собой новейшую концепцию и обычно находят

применение в обобщенном программировании. В этой книге мы просто хотим дать вам представление об этой развивающейся концепции. Строки 15, 18 и 21 содержат три различных экземпляра `std::tuple`. `tup1` содержит три члена: `std::string`, `int` и `char`. `tup2` содержит значения `double` и `bool`, а также использует возможности автоматического вывода типа компилятором с помощью ключевого слова `auto`. `tup3` представляет собой кортеж из пяти членов: `double`, `bool`, `int`, `char` и `string` — результат конкатенации с использованием шаблонной функции `std::tuple_cat`.

Шаблонная функция `DisplayTupleInfo()` в строках 5–11 демонстрирует использование шаблона `tuple_size`, который разрешается в количество элементов, содержащихся в экземпляре `std::tuple` во время компиляции. Функция `std::get`, использованная в строке 10, предоставляет механизм доступа к отдельным значениям, хранящимся в кортеже, с помощью их индексов (как обычно в C++, нумерация начинается с нуля). Наконец функция `std::tie` в строке 26 демонстрирует, как содержимое кортежа может быть распаковано или скопировано в отдельные объекты. Значение `std::ignore` используется для того, чтобы указать `std::tie` те элементы кортежа, которые не интересуют наше приложение.

Использование `static_assert` для выполнения проверок времени компиляции

Эта возможность появилась в языке программирования начиная со стандарта C++11 и позволяет блокировать компиляцию в случае, когда указанные программистом тесты во время компиляции не выполняются. Несмотря на кажущуюся странность этой возможности, она может оказаться очень полезной при разработке шаблонов классов. Например, вы можете захотеть гарантировать, что ваш шаблон класса не будет инстанцирован для типа `int`! Использование `static_assert` позволяет отобразить специальное сообщение времени компиляции в вашей среде разработки (или выдать его на консоль):

```
static_assert(Проверяемое_выражение, "Сообщение об ошибке");
```

Чтобы предотвратить инстанцирование вашего шаблона класса для типа `int`, можно, например, использовать `static_assert()` с оператором `sizeof(T)`, сравнивая возвращаемое им значение с результатом выражения `sizeof(int)` и отображая сообщение об ошибке, если проверка на неравенство терпит неудачу:

```
static_assert(sizeof(T) != sizeof(int), "int не разрешен!");
```

Такой шаблон класса, использующий `static_assert` для блокировки компиляции при определенных условиях, показан в листинге 14.9.

ЛИСТИНГ 14.9. Шаблон класса, не работающий с типами с размером, равным размеру `int`

```
0: template <typename T>
1: class EverythingButInt
2: {
3: public:
4:     EverythingButInt()
5:     {
6:         static_assert(sizeof(T) != sizeof(int), "int запрещен!");
7:     }
8: };
9:
10: int main()
11: {
12:     EverythingButInt<int> test; // Инстанцирование для int.
13:     return 0;
14: }
```

Результат

Вывода нет, поскольку компиляция неудачна — отображается указанное вами сообщение:

```
error: int запрещен!
```

Анализ

Запрет на инстанцирование запрограммирован в строке 6. `static_assert` — это языковое средство C++11, которое помогает, в частности, защитить свой код шаблона от нежелательного инстанцирования.

Использование шаблонов в практическом программировании на C++

Самое важное и мощное применение шаблоны нашли в *стандартной библиотеке шаблонов* (Standard Template Library — STL). Библиотека STL состоит из коллекции шаблонов классов и функций, содержащей обобщенные вспомогательные классы и алгоритмы. Шаблонные классы библиотеки STL позволяют реализовать динамические массивы, списки и контейнеры пар “ключ–значение”, в то время как алгоритмы, как, например, алгоритм сортировки, работают с этими контейнерами и обрабатывают содержащиеся в них данные.

Знание синтаксиса шаблонов, с которым вы познакомились, очень поможет далее, при использовании контейнеров и функций STL, которые будут рассматриваться на следующих занятиях. Хорошее понимание контейнеров и алгоритмов библиотеки STL, в свою очередь, поможет вам создавать эффективные приложения на языке программирования C++ с использованием проверенной и надежной реализации библиотеки STL, а также избежать долгих часов копания в дебрях кода.

РЕКОМЕНДУЕТСЯ

Используйте шаблоны для реализации обобщенных концепций.

Предпочитайте шаблоны макросам.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте использовать константность при разработке шаблонов функций и классов.

Не забывайте, что статический член, содержащийся в шаблоне класса, является статическим для каждой специализации класса.

Резюме

На сегодняшнем занятии представлено большое количество подробностей о работе препроцессора. Каждый раз, когда вы запускаете компилятор, сначала запускается препроцессор, преобразующий в исходный текст такие директивы, как `#define`.

Препроцессор осуществляет только простую текстовую подстановку, хотя использование макросов может давать достаточно сложные результаты. Макрофункции обеспечивают сложную текстовую подстановку на основании аргументов, передаваемых макросу во время компиляции. Каждый аргумент в макросе следует помещать в круглые скобки, чтобы гарантировать правильность подстановки.

Шаблоны помогают обеспечить повторное использование кода, применимого для множества различных типов данных. Они также являются альтернативой макросам, обеспечивающей безопасность типов. Со знанием шаблонов, полученным на этом занятии, вы готовы приступить к изучению библиотеки STL.

Вопросы и ответы

■ **Почему я должен использовать защиту от повторного включения в своих заголовочных файлах?**

Защита от повторного включения с использованием директив `#ifndef`, `#define` и `#endif` защищает ваш заголовочный файл от ошибок, неизбежных при множественном или рекурсивном включении, и в некоторых случаях даже ускоряет компиляцию.

■ **Когда я должен предпочитать макрофункции шаблонам, если необходимая функциональность может быть реализована обоими способами?**

Как правило, желательно использовать шаблоны, поскольку они, обеспечивая обобщенность, являются безопасными с точки зрения типов. Макросы не позволяют получить безопасную с точки зрения типов реализацию, так что их лучше избегать.

■ **Должен ли я определять аргументы шаблона при вызове шаблона функции?**

Обычно нет, поскольку компилятор способен вывести их самостоятельно на основе переданных при вызове аргументов функции.

■ **Сколько экземпляров статических переменных имеется для данного шаблона класса?**

Все зависит от количества типов, для которых шаблон класса был инстанцирован. Так, если шаблон инстанцирован для типов `int`, `string` и пользовательского типа `X`, то будут доступны три экземпляра статической переменной — по одному для каждого инстанцирования.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Что такое защита от повторного включения?
2. Рассмотрим следующий макрос:

```
#define SPLIT(x) x / 5
```

 Каков будет его результат при вызове со значением 20?
3. Каков будет результат, если вызвать макрос `SPLIT` из вопроса 2 со значением 10+10?
4. Как изменить макрос `SPLIT`, чтобы избежать ошибочных результатов?

Упражнения

1. Напишите макрос, умножающий два числа.
2. Напишите шаблонную версию макроса из упражнения 1.
3. Реализуйте шаблонную функцию `swap()` для обмена значений двух переменных.
4. **Отладка.** Как улучшить следующий макрос, вычисляющий четверть исходного значения?

```
#define QUARTER(x) (x / 4)
```
5. Напишите простой шаблон класса, хранящий два массива элементов с типами, которые определены в списке параметров шаблона класса. Размер массива — 10; шаблон класса должен быть оснащен функциями доступа, обеспечивающими работу с элементами массива.
6. Напишите шаблонную функцию `Display()`, которая может быть вызвана с разными количеством и типами аргументов и выводит на консоль каждый из них.

Часть III

Стандартная библиотека шаблонов

В ЭТОЙ ЧАСТИ...

ЗАНЯТИЕ 15. Введение в стандартную библиотеку шаблонов

ЗАНЯТИЕ 16. Класс строки библиотеки STL

ЗАНЯТИЕ 17. Классы динамических массивов библиотеки STL

ЗАНЯТИЕ 18. Классы `list` и `forward_list`

ЗАНЯТИЕ 19. Классы множеств STL

ЗАНЯТИЕ 20. Классы отображений библиотеки STL

ЗАНЯТИЕ 15

Введение в стандартную библиотеку шаблонов

Попросту говоря, стандартная библиотека шаблонов (STL) — это набор шаблонов классов и функций, который предоставляет программистам следующее:

- контейнеры для хранения информации,
- итераторы для доступа к хранимой информации,
- алгоритмы для манипуляции содержимым контейнеров.

На этом занятии вы получите общее представление об этих трех китах библиотеки STL.

Контейнеры STL

Контейнеры (container) — это классы библиотеки STL, предназначенные для хранения данных. Библиотека STL предоставляет два типа контейнерных классов:

- последовательные контейнеры;
- ассоциативные контейнеры.

В дополнение к ним библиотека STL предоставляет классы, называемые *адаптерами контейнеров* (container adapter), являющиеся версиями имеющихся контейнеров с ограниченными функциональными возможностями, предназначенные для специфических целей.

Последовательные контейнеры

Как и подразумевает их название, *последовательные контейнеры* (sequential container) используются для хранения данных в последовательном виде, таком как массивы и списки. Последовательные контейнеры характеризуются быстрым выполнением вставки, но относительно медленным поиском.

Ниже приведены последовательные контейнеры библиотеки STL.

- `std::vector`. Работает как динамический массив и увеличивается с конца. Вектор похож на книжную полку, книги на которую можно добавлять или удалять по одной с конца.
- `std::deque`. Подобен контейнеру `std::vector`, но новые элементы можно вставлять и удалять также в начало контейнера.
- `std::list`. Работает как двухсвязный список. Список похож на цепочку, в которой каждый объект связан с предыдущим и последующим звеньями. Вы можете добавить или удалить звено (т.е. объект) в любой позиции.
- `std::forward_list`. Подобен списку `std::list`, но односвязный список позволяет осуществлять проход по списку только в одном направлении.

Класс `vector` библиотеки STL сродни массиву и обеспечивает произвольный доступ к элементам, т.е. вы можете обращаться к элементам вектора непосредственно с использованием их позиции в векторе, используя оператор индексации (`[]`), и работать с этими данными. Кроме того, вектор STL является динамическим массивом и, таким образом, может изменять свои размеры, чтобы соответствовать требованиям приложения. Для обеспечения этой возможности при сохранении способности произвольного обращения к элементам массива по индексу контейнер `vector` библиотеки STL хранит все элементы последовательно, в непрерывной области памяти. Поэтому вектор должен уметь изменять свои размеры (что может отрицательно влиять на производительность приложения в зависимости от типа объектов, которые он содержит).

Вкратце вектор был представлен в листинге 4.4, а более подробно контейнер `vector` рассматривается на занятии 17, “Классы динамических массивов библиотеки STL”.

Контейнер `list` библиотеки STL является реализацией обычного связанного списка. Хотя к элементам списка нельзя обращаться произвольно, как в векторе STL, список может хранить элементы в несмежных блоках памяти. Поэтому у контейнера `std::list` нет присущих вектору проблем с производительностью, связанных с перераспределением его внутреннего массива. Подробно класс списка библиотеки STL обсуждается на занятии 18, “Классы `list` и `forward_list`”.

Ассоциативные контейнеры

Ассоциативные контейнеры (associative container), хранящие данные в отсортированном виде, сродни словарю. В результате вставка в них осуществляется медленнее, чем в последовательные контейнеры, но когда дело доходит до поиска, преимущества ассоциативных контейнеров оказываются существенными.

Библиотека STL предоставляет следующие ассоциативные контейнеры.

- `std::set`. Уникальные значения хранятся в контейнере в отсортированном порядке; вставка в контейнер и поиск в нем являются операцией с логарифмической сложностью.
- `std::unordered_set`. Уникальные значения хранятся в данном контейнере неотсортированными, но вставка и поиск осуществляются за время, близкое к константному. Этот контейнер доступен начиная с версии C++11.
- `std::map`. Хранит пары “ключ–значение” с уникальными ключами, отсортированными по значениям ключей; вставка в контейнер и поиск в нем являются операцией с логарифмической сложностью.
- `std::unordered_map`. Хранит пары “ключ–значение” с уникальными ключами в неотсортированном порядке, но вставка и поиск осуществляются за время, близкое к константному. Этот контейнер доступен, начиная с версии C++11.
- `std::multiset`. Похож на контейнер `set`; дополнительно обеспечивает возможность хранить несколько элементов с одинаковыми значениями, т.е. значение не обязательно должно быть уникальным.
- `std::unordered_multiset`. Похож на контейнер `unordered_set`; дополнительно обеспечивает возможность хранить несколько элементов с одинаковыми значениями, т.е. значение не обязательно должно быть уникальным. Этот контейнер доступен начиная с версии C++11.
- `std::multimap`. Похож на контейнер `map`; дополнительно обеспечивает возможность хранить пары “ключ–значение” с одинаковыми ключами.
- `std::unordered_multimap`. Похож на контейнер `unordered_map`; дополнительно обеспечивает возможность хранить пары “ключ–значение” с одинаковыми ключами. Этот контейнер доступен начиная с версии C++11.

ПРИМЕЧАНИЕ

Сложность в данном случае является показателем производительности контейнера с учетом количества содержащихся в нем элементов. Говоря о *константной сложности*, как в случае `std::unordered_map`, мы подразумеваем, что производительность контейнера не связана с количеством содержащихся в нем элементов. Такой контейнер, содержащий тысячу элементов, потребует столько же времени на выполнение операции, как и контейнер с миллионом элементов.

Логарифмическая сложность (как в случае с `std::map`) указывает, что время выполнения операции пропорционально логарифму количества элементов, содержащихся в контейнере. Время выполнения операции таким контейнером с тысячей элементов будет в два раза меньше времени работы контейнера с миллионом элементов.

Линейная сложность означает, что время выполнения операции пропорционально количеству элементов в контейнере. Такой контейнер будет в тысячу раз медленнее при обработке миллиона элементов, чем при обработке тысячи элементов.

У одного и того же контейнера сложность может быть разной для различных операций. Например, вставка элемента может иметь константную сложность, в то время как операция поиска элемента — линейную сложность. Таким образом, знание, помимо доступных операций, как именно контейнер может их выполнять, является ключом к выбору контейнера, наилучшим образом подходящего для вашей задачи.

Сортировка контейнеров STL может быть настроена программистом посредством написания соответствующих предикатных функций.

СОВЕТ

Некоторые реализации библиотеки STL предоставляют и такие ассоциативные контейнеры, как `hash_set`, `hash_multiset`, `hash_map` и `hash_multimap`. Они подобны контейнерам `unordered_*`, которые поддерживаются в соответствии со стандартом. В некоторых сценариях варианты `hash_*` и `unordered_*` могут оказаться лучше при поиске элемента, поскольку они предоставляют операции с константным временем выполнения (не зависящие от количества элементов в контейнере). Как правило, эти контейнеры предоставляют также открытые методы, идентичные методам в их стандартных аналогах, а следовательно, столь же удобные.

Использование стандартных контейнеров приводит к созданию кода, который проще переносить на другие платформы и компилировать, а потому их применение предпочтительнее. Кроме того, логарифмическая производительность операций у ряда стандартных контейнеров может быть вполне адекватной для ваших приложений.

Адаптеры контейнеров

Адаптеры контейнеров (container adapter) — это версии последовательных и ассоциативных контейнеров с ограниченными функциональными возможностями, предназначенные для специфических целей. Основные классы адаптеров приведены ниже.

- `std::stack`. Хранит элементы в порядке LIFO (Last-In-First-Out — последним вошел, первым вышел), позволяя вставлять и извлекать элементы из вершины стека.
- `std::queue`. Хранит элементы в порядке FIFO (First-In-First-Out — первым вошел, первым вышел), позволяя извлекать элементы в порядке их вставки в очередь.
- `std::priority_queue`. Элементы хранятся в отсортированном порядке, так что первым в очереди всегда располагается элемент, значение приоритета которого считается самым высоким.

Более подробная информация по этой теме приведена на занятии 24, “Адаптивные контейнеры: стек и очередь”.

Итераторы STL

Самый простой пример *итератора* (iterator) — это указатель на первый элемент в массиве. Вы можете выполнить инкремент этого указателя, и он будет указывать на следующий элемент массива.

Итераторы библиотеки STL — это шаблоны классов, которые в определенной степени являются обобщением указателей. Такие шаблоны классов предоставляют разработчикам средство, позволяющее работать с элементами в контейнерах STL и выполнять над ними те или иные операции. Заметим, что эти операции могут быть алгоритмами STL, которые представляют собой шаблонные функции. Итераторы — это своего рода мост, позволяющий шаблонным функциям единообразно и согласованно работать с самыми разными контейнерами.

Предоставляемые библиотекой STL итераторы глобально можно классифицировать следующим образом.

- *Итератор ввода* (input iterator). Такой итератор может быть разыменован для получения ссылки на объект. Этот объект может, например, находиться в коллекции. Классический итератор ввода гарантирует только доступ для чтения значения объекта.
- *Итератор вывода* (output iterator). Этот итератор обеспечивает запись в коллекцию. Классический итератор вывода гарантирует доступ только для записи.

Основные типы итераторов, упомянутые в предыдущем списке, можно подразделять на следующие разновидности.

- *Однонаправленный итератор* (forward iterator). Усовершенствованный итератор, обеспечивающий как ввод, так и вывод. Такие итераторы могут быть константными, обеспечивающими доступ только для чтения к объекту, на который указывает итератор, либо неконстантными, обеспечивающими операции чтения и записи. Как правило, однонаправленный итератор используется в односвязном списке.

- *Двунаправленный итератор* (bidirectional iterator). Усовершенствованный однонаправленный итератор, допускающий переход как к следующему, так и к предыдущему элементам. Двунаправленный итератор, как правило, используется в двусвязном списке.
- *Итератор произвольного доступа* (random access iterator). Усовершенствованный итератор, допускающий прибавление и вычитание смещения, а также вычитание одного итератора из другого для поиска относительного смещения (дистанции) между двумя объектами коллекции. Итератор произвольного доступа, как правило, используется с массивами.

ПРИМЕЧАНИЕ

На уровне реализации усовершенствование можно рассматривать как *наследование* или *специализацию*.

Алгоритмы STL

Поиск, сортировка, изменение порядка на обратный и тому подобные действия являются стандартными операциями при программировании, реализацию которых разработчик не должен изобретать заново. Библиотека STL предоставляет возможность выполнения этих операций в форме алгоритмов STL, которые, работая с контейнерами посредством итераторов, помогают программисту решить многие распространенные задачи, не изобретая велосипед.

Ниже приведены некоторые из наиболее популярных алгоритмов STL.

- `std::find`. Позволяет найти значение в коллекции.
- `std::find_if`. Позволяет найти значение в коллекции с применением пользовательского предиката.
- `std::reverse`. Обращает порядок элементов в коллекции.
- `std::remove_if`. Позволяет удалить элемент из коллекции с применением пользовательского предиката.
- `std::transform`. Позволяет применить определенную пользователем функцию преобразования к элементам контейнера.

Эти алгоритмы представляют собой шаблоны функций из пространства имен `std`. Для их применения требуется включить в код стандартный заголовочный файл `<algorithm>`.

Взаимодействие контейнеров и алгоритмов с использованием итераторов

Рассмотрим конкретный пример, как использование итераторов соединяет контейнеры и алгоритмы STL. Программа, представленная в листинге 15.1, использует последовательный контейнер STL `std::vector`, работающий как динамический массив,

хранящий несколько целых чисел, а затем использует алгоритм `std::find` для поиска одного из них. Обратите внимание, как итераторы соединяют контейнеры и алгоритмы STL. Не обращайте внимания на сложности синтаксиса или функциональность. Контейнеры, такие как `std::vector`, и алгоритмы, такие как `std::find`, еще будут подробно рассматриваться на занятиях 17, “Классы динамических массивов библиотеки STL”, и 23, “Алгоритмы библиотеки STL”, соответственно. Если эта часть покажется вам слишком сложной, можете ее пропустить.

ЛИСТИНГ 15.1. Поиск элемента по его позиции в векторе

```
1: #include <iostream>
2: #include <vector>
3: #include <algorithm>
4: using namespace std;
5:
6: int main()
7: {
8:     // Динамический массив целых чисел
9:     vector<int> intArray;
10:
11:     // Вставить примеры целых чисел в массив
12:     intArray.push_back(50);
13:     intArray.push_back(2991);
14:     intArray.push_back(23);
15:     intArray.push_back(9999);
16:
17:     cout << "Содержимое вектора: " << endl;
18:
19:     // Обход вектора и чтение значений с помощью итератора
20:     vector<int>::iterator arrIterator = intArray.begin();
21:
22:     while(arrIterator != intArray.end())
23:     {
24:         // Вывод значения на экран
25:         cout << *arrIterator << endl;
26:
27:         // Инкремент итератора для доступа к следующему элементу
28:         ++arrIterator;
29:     }
30:
31:     // Поиск элемента (скажем, 2991) с помощью алгоритма 'find'
32:     vector<int>::iterator elFound =
33:         find(intArray.begin(), intArray.end(), 2991);
34:
35:     // Проверить, найдено ли значение
36:     if (elFound != intArray.end())
37:     {
38:         // Значение найдено. Определяем позицию в массиве:
39:         int elPos = distance(intArray.begin(), elFound);
40:         cout << "Значение "<< *elFound;
41:         cout << " находится в позиции " << elPos << endl;
```



```
42:     }  
43:  
44:     return 0;  
45: }
```

Результат

Содержимое вектора:

```
50  
2991  
23  
9999
```

Значение 2991 находится в позиции 1

Анализ

В листинге 15.1 показано применение итераторов для обхода вектора и в качестве интерфейса, позволяющего использовать такие алгоритмы, как `find`, с разными контейнерами, например `vector`. Объект итератора `arrIterator` объявлен в строке 20 и инициализирован начальной позицией в контейнере (возвращаемым значением функции-члена `begin()` контейнера `vector`). Строки 22–29 демонстрируют использование этого итератора в цикле отображения элементов вектора таким же способом, как и элементов статического массива. Итераторы используются совершенно одинаково всеми контейнерами STL. Все контейнеры предоставляют функцию `begin()`, указывающую на первый элемент, и функцию `end()`, указывающую на конец контейнера — на позицию *после* последнего элемента. Вот почему цикл `while` в строке 22 останавливается на элементе перед указанным функцией `end()`, а не на нем. В строке 32 показано использование алгоритма `find` для поиска значения в контейнере `vector`. Результат операции поиска — итератор, а ее успешность проверяется путем сравнения итератора с итератором конца контейнера (строка 36). Если элемент найден, он может быть отображен с помощью разыменования итератора (как и при использовании указателя). Алгоритм `distance()` применяется для вычисления позиции (смещения) найденного элемента.

Если не глядя заменить в листинге 15.1 все слова `vector` словами `deque`, код все равно будет компилироваться и прекрасно работать благодаря итераторам, которые обеспечивают взаимосвязь между контейнерами и алгоритмами.

Использование ключевого слова `auto` для определения типа

В листинге 15.1 использовано несколько объявлений итератора. Они выглядят подобно следующему:

```
20:     vector<int>::iterator arrIterator = intArray.begin();
```

Определение типа итератора выглядит пугающе. Если вы используете компилятор, совместимый со стандартом C++11, то можете упростить эту строку до следующей:

```
20:     auto arrIterator = intArray.begin(); // Компилятор выводит тип
```

Обратите внимание, что переменная, объявленная с типом `auto`, должна быть инициализирована, так как именно инициализирующее значение используется компилятором для определения типа переменной, объявленной как `auto`.

Выбор правильного контейнера

Очевидно, что требованиям вашего приложения могут удовлетворять контейнеры STL нескольких типов. В таком случае следует сделать правильный выбор, так как неправильный выбор может привести к неоправданной потере производительности приложения.

Очень важно оценить все преимущества и недостатки контейнера, показанные в табл. 15.1, прежде чем остановить выбор на нем.

ТАБЛИЦА 15.1. Свойства контейнерных классов STL

Контейнер	Преимущества	Недостатки
<code>std::vector</code> (Последовательный контейнер)	Быстрая (константная по продолжительности) вставка в конец. Доступ, как у массива	Изменение размеров может привести к потере производительности. Время поиска пропорционально количеству элементов в контейнере. Вставка только в конец контейнера
<code>std::deque</code> (Последовательный контейнер)	Все преимущества контейнера <code>vector</code> , а также постоянная по продолжительности вставка в начало контейнера	Недостатки вектора по производительности и поиску относятся также к деку. В отличие от вектора, дек не обязан предоставлять функцию <code>reserve()</code> , которая резервирует область памяти, позволяя избежать изменения размеров для повышения производительности
<code>std::list</code> (Последовательный контейнер)	Константная продолжительность вставки в любое место списка. Константная продолжительность удаления элементов из списка независимо от позиции. Вставка и удаление элементов не влияют на итераторы, указывающие на другие элементы списка	К элементам нельзя обращаться произвольно по индексу, как в массиве. Доступ к элементам может быть медленнее, чем у вектора, поскольку они хранятся не в смежных областях памяти. Время поиска пропорционально количеству элементов в контейнере
<code>std::forward_list</code> (Последовательный контейнер)	Односвязный список, допускающий итерацию только в одном направлении	Допускает вставку только в начало списка с помощью метода <code>push_front()</code>

Продолжение табл. 15.1

Контейнер	Преимущества	Недостатки
<code>std::set</code> (Ассоциативный контейнер)	Поиск пропорционален не количеству элементов в контейнере, а его логарифму, а потому зачастую выполняется значительно быстрее, чем в последовательных контейнерах	Вставка элементов осуществляется медленнее, чем в последовательных аналогах, поскольку элементы при вставке сортируются
<code>std::unordered_set</code> (Ассоциативный контейнер)	Поиск, вставка и удаление в контейнере этого типа почти не зависят от количества элементов	Нельзя использовать относительную позицию элементов в пределах контейнера — она не определена
<code>std::multiset</code> (Ассоциативный контейнер)	Используется, когда множество должно содержать не уникальные значения	Вставка элементов осуществляется медленнее, чем в последовательных аналогах, поскольку элементы при вставке сортируются
<code>std::unordered_multiset</code> (Ассоциативный контейнер)	Когда необходимо содержать не уникальные значения, предпочтительнее использования контейнера <code>unordered_set</code> . Производительность, как у контейнера <code>unordered_set</code> , а именно: постоянное среднее время поиска, вставки и удаления элементов не зависит от размера контейнера	Нельзя использовать относительную позицию элементов в пределах контейнера — она не определена
<code>std::map</code> (Ассоциативный контейнер)	Контейнер пар “ключ–значение”, поиск в котором пропорционален не количеству элементов в контейнере, а его логарифму, а потому зачастую значительно быстрее, чем в последовательных контейнерах	Вставка элементов осуществляется медленнее, чем в последовательных аналогах, поскольку элементы при вставке сортируются
<code>std::unordered_map</code> (Ассоциативный контейнер)	Поиск, вставка и удаление в контейнере этого типа почти не зависят от количества элементов	Нельзя использовать относительную позицию элементов в пределах контейнера — она не определена
<code>std::multimap</code> (Ассоциативный контейнер)	Используется, когда отображение должно содержать не уникальные значения ключей	Вставка элементов осуществляется медленнее, чем в последовательных аналогах, поскольку элементы при вставке сортируются
<code>std::unordered_multimap</code> (Ассоциативный контейнер)	Когда необходимо содержать не уникальные значения ключей, предпочтительнее использования контейнера <code>unordered_map</code> .	Нельзя использовать относительную позицию элементов в пределах контейнера — она не определена

Контейнер	Преимущества	Недостатки
	Поиск, вставка и удаление в контейнере этого типа почти не зависят от количества элементов	

Классы строк библиотеки STL

Библиотека STL предоставляет шаблон класса, специально предназначенного для строковых операций. Шаблон `std::basic_string<T>` используется обычно в двух своих специализациях.

- `std::string`. Специализация шаблона `std::basic_string` для типа `char`, используемая для работы с простыми символьными строками.
- `std::wstring`. Специализация шаблона `std::basic_string` для типа `wchar_t`, используемая для работы с широкосимвольными строками, обычно для хранения символов Unicode.

Эти вспомогательные классы подробно обсуждаются на занятии 16, “Класс строки библиотеки STL”; вы увидите, насколько они упрощают работу со строками.

Резюме

На сегодняшнем занятии рассматривались фундаментальные концепции библиотеки STL, такие как контейнеры, итераторы и основные алгоритмы. Вы познакомились также с шаблоном `basic_string<T>`, который подробно обсуждается на следующем занятии. Контейнеры, итераторы и алгоритмы — это одни из самых важных концепций библиотеки STL, и их понимание поможет вам эффективно использовать библиотеку STL в своих приложениях. Более подробная информация об этих концепциях и их применении приводится на занятиях 17–25.

Вопросы и ответы

- Мне нужно использовать массив, но количество элементов, которые он должен содержать, заранее неизвестно. Какой контейнер STL мне следует использовать?

Вам отлично подойдут контейнеры `std::vector` и `std::deque`. Их самостоятельное управление памятью и динамическое масштабирование улучшат приложение.

- В моем приложении довольно часто используется поиск. Какой контейнер мне следует выбрать?

Для частых поисков лучше всего подойдут такие ассоциативные контейнеры, как `std::map` и `std::set`, или их неупорядоченные варианты.

- Я должен хранить пары “ключ–значение” для быстрого поиска. Но может случиться так, что ключи будут не уникальными. Какой контейнер мне следует выбрать?

Вам подойдет ассоциативный контейнер типа `std::multimap`. Контейнер `multimap` может содержать не уникальные пары “ключ–значение” и в состоянии обеспечить быстрый поиск, характерный для ассоциативных контейнеров.

- Мое приложение предназначено для разных платформ и компиляторов. Мне необходим контейнер с быстрым поиском на основании ключа. Должен ли я использовать контейнер `std::map` или `std::hash_map`?

Переносимость — важный ограничивающий фактор, поэтому вам необходимо использовать только стандартные контейнеры. `hash_map` не является частью стандарта C++11 и может не поддерживаться на всех платформах, для которых предназначено ваше приложение. Если на всех интересующих платформах используются компиляторы, совместимые со стандартом C++11, вы могли бы использовать контейнер `std::unordered_map`.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Какой контейнер вы выберете, если хранимый массив объектов требует возможности вставки и в начало, и в конец?
2. Необходимо хранить элементы для быстрого поиска. Какой контейнер вы выбрали бы в этом случае?
3. Необходимо хранить элементы в контейнере `std::set`, но при этом необходима возможность изменения критериев поиска на основании условия, которое не обязательно связано со значением элементов. Возможно ли удовлетворить этим требованиям?
4. Какая возможность библиотеки STL позволяет соединить алгоритмы с контейнерами?
5. Выбрали бы вы контейнер `hash_set` для приложения, которое должно быть перенесено на различные платформы и компилироваться разными компиляторами C++?

ЗАНЯТИЕ 16

Класс строки библиотеки STL

Стандартная библиотека шаблонов (STL) предоставляет программистам контейнерный класс, облегчающий операции со строками и манипулирование ими. Класс `string` не только динамически изменяет свои размеры, чтобы удовлетворять требованиям приложения, но и предоставляет полезные вспомогательные функции (или методы), помогающие манипулировать строками. Таким образом, он позволяет программистам использовать стандартные, переносимые и проверенные функциональные возможности в своих приложениях.

На этом занятии...

- Зачем нужны классы обработки строк
- Как работать с классом `string` библиотеки STL
- Как библиотека STL облегчает такие операции со строками, как конкатенация, добавление, поиск и др.
- Как использовать шаблонную реализацию строк библиотеки STL
- Оператор `""s`, поддерживаемый STL `string` (со стандарта C++14)

Потребность в классах обработки строк

Строка в языке C++ — это массив символов. Как вы уже видели на занятии 4, “Массивы и строки”, простейший символьный массив может быть определен следующим образом:

```
char staticName[20];
```

Здесь объявляется символьный массив (именуемый также строкой) фиксированной (статический) длины в 20 элементов. Очевидно, что этот массив может содержать строку ограниченной длины; он окажется переполненным при попытке сохранить в нем большее количество символов. Изменение размеров такого статического массива невозможно. Для преодоления этого ограничения язык C++ предоставляет динамическое распределение памяти для данных. Вот более динамичное представление строкового массива:

```
char* dynamicName = new char[arrayLen];
```

Это динамически распределенный символьный массив, длина экземпляра которого может быть задана при создании значением переменной `arrayLen`, определяемым во время выполнения, а следовательно, способным содержать данные переменной длины. Но если понадобится изменить длину массива во время выполнения, то придется сначала освободить распределенную память, а затем повторно выделить ее для содержания необходимых данных.

Ситуация усложняется, если такие символьные строки используются как данные-члены класса. В ситуациях, когда объект такого класса присваивается другому, при отсутствии грамотно созданного копирующего конструктора и оператора присваивания оба эти объекта будут содержать копии указателя, указывающего на один и тот же строковый буфер, т.е. на одну и ту же область памяти. В результате удаления одного объекта указатель в другом объекте оказывается недействительным (указывающим на освобожденную область памяти, которая может быть использована для других нужд), а ваша программа сталкивается с нешуточными неприятностями.

Строковые классы решают эти проблемы самостоятельно. Строковый класс `std::string` библиотеки STL моделирует символьную строку, а класс `std::wstring` — широкосимвольную строку, помогая вам следующим образом.

- Сокращает усилия по созданию строк и управлению ими.
- Увеличивает стабильность приложения за счет инкапсуляции подробностей распределения памяти.
- Встроенный копирующий конструктор и оператор присваивания автоматически гарантируют корректность копирования строковых членов классов.
- Предоставляет полезные вспомогательные функции, помогающие в копировании, усечении, поиске и удалении.
- Предоставляет операторы для сравнения.
- Позволяет сосредоточить усилия на основных требованиях вашего приложения, а не на подробностях обработки строк.

ПРИМЕЧАНИЕ

Фактически классы `std::string` и `std::wstring` являются специализациями одного и того же шаблона класса `std::basic_string<T>` для типов `char` и `wchar_t` соответственно. Изучив его использование подробнее, вы сможете использовать те же методы и операторы и для других типов.

Давайте на примере класса `std::string` изучим некоторые из вспомогательных функций, предоставляемых строковыми классами библиотеки STL.

Работа с классом строки STL

Наиболее популярные строковые функции приведены ниже.

- Копирование
- Конкатенация
- Поиск символов и подстрок
- Усечение
- Обращение строк и смены регистра символов с использованием алгоритмов, предоставляемых стандартной библиотекой

Для использования строковых классов STL необходимо включить в код заголовочный файл `<string>`.

Создание экземпляров и копий строк STL

Класс `string` предоставляет множество перегруженных конструкторов, а потому его экземпляр может быть создан и инициализирован различными способами. Например, можно инициализировать объект класса `std::string` строкой или присвоить ему постоянный символьный строковый литерал:

```
const char* constCStyleString = "Hello String!";
std::string strFromConst(constCStyleString);
```

или

```
std::string strFromConst = constCStyleString;
```

Приведенный выше фрагмент аналогичен следующему коду:

```
std::string str2("Hello String!");
```

Как можно заметить, создание объекта класса `string` и его инициализация значением не требовали указания длины строки или подробностей распределения памяти — конструктор класса `string` делает все это автоматически.

Точно так же вполне возможно использовать один объект класса `string` для инициализации другого:

```
std::string str2Copy(str2);
```


Вы можете также указать конструктору класса `string`, что для инициализации строки следует принять только n первых символов передаваемой исходной строки:

```
// Инициализировать строку первыми 5 символами другой строки
std::string strPartialCopy(constCStyleString, 5);
```

Можно также инициализировать строку некоторым количеством определенного символа:

```
// Инициализировать строку 10 символами 'a'
std::string strRepeatChars(10, 'a');
```

В листинге 16.1 анализируются некоторые наиболее популярные способы создания экземпляров класса `std::string` и копирования строк.

ЛИСТИНГ 16.1. Создание экземпляров строк STL и их копирование

```
0: #include <string>
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:     const char* constCStyleString = "Hello String!";
7:     cout << "Константная строка: " << constCStyleString << endl;
8:
9:     std::string strFromConst(constCStyleString); // Конструктор
10:    cout << "strFromConst: " << strFromConst << endl;
11:
12:    std::string str2("Hello String!");
13:    std::string str2Copy(str2);
14:    cout << "str2Copy: " << str2Copy << endl;
15:
16:    // Инициализировать строку первыми 5 символами другой строки
17:    std::string strPartialCopy(constCStyleString, 5);
18:    cout << "strPartialCops: " << strPartialCopy << endl;
19:
20:    // Инициализировать строку 10 символами 'a'
21:    std::string strRepeatChars(10, 'a');
22:    cout << "strRepeatChars: " << strRepeatChars << endl;
23:
24:    return 0;
25: }
```

Результат

```
Константная строка: Hello String!
strFromConst: Hello String!
str2Copy: Hello String!
strPartialCopy: Hello
strRepeatChars: aaaaaaaaaa
```

Анализ

Приведенный выше код демонстрирует способы создания экземпляров класса `string` и его инициализации другой строкой, частичной копией и набором повторяющихся символов. Символьная строка `constCStyleString` в стиле C инициализирована значением в строке 6. Строка 9 демонстрирует, насколько просто конструктор класса `std::string` позволяет создать копию этого значения. Строка 12 копирует в объект `str2` класса `std::string` другую постоянную строку, а в строке 13 представлен другой перегруженный конструктор класса `std::string`, позволяющий скопировать объект класса `std::string` и получить новую строку `str2Copy`, являющуюся точной копией исходной. Строка 17 демонстрирует частичное копирование, а строка 21 возможность создания экземпляра класса `std::string` и его инициализацию повторяющимся символом. Этот пример кода демонстрирует отнюдь не все способы того, как класс `std::string` и его многочисленные копирующие конструкторы облегчают разработчику создание строк, их копирование и отображение.

ПРИМЕЧАНИЕ

Если бы вы должны были использовать для подобного копирования строки в стиле C, то эквивалент строки 9 листинга 16.1 выглядел бы следующим образом:

```
const char* constCStyleString = "Hello World!";
// Выделение памяти для создания строки
char * pszCopy = new char[strlen(constCStyleString)+1];
strcpy(pszCopy, constCStyleString); // Копирование
```

...

```
// Освобождение памяти после использования pszCopy
delete[] pszCopy;
```

Как видите, здесь куда больше строк кода и выше вероятность ошибки. Кроме того, необходимо позаботиться об управлении памятью и ее освобождении. Класс `string` библиотеки STL делает все это — и еще многое — вместо вас!

Доступ к символу в строке `std::string`

К символьному содержимому строки STL можно обратиться с помощью итератора или синтаксиса в стиле массива, в котором используется оператор индексации `[]`. Получить представление строки в стиле C можно с помощью функции-члена `c_str()` (листинг 16.2).

ЛИСТИНГ 16.2. Два способа обращения к символу строки STL

```
0: #include <string>
1: #include <iostream>
2:
3: int main()
```

```
4: {
5:     using namespace std;
6:
7:     string stlString("Hello String"); // Пример строки
8:
9:     // Доступ к содержимому строки: синтаксис обращения к массиву
10:    cout<<"Синтаксис обращения к массиву:" << endl;
11:    for(size_t charCounter = 0;
12:        charCounter < stlString.length();
13:        ++charCounter )
14:    {
15:        cout << "Символ[" << charCounter << "] = ";
16:        cout << stlString[charCounter] << endl;
17:    }
18:    cout << endl;
19:
20:    // Доступ к содержимому строки с использованием итератора
21:    cout << "Вывод с использованием итератора:" << endl;
22:    int charOffset = 0;
23:    string::const_iterator charLocator;
24:    for(auto charLocator = stlString.cbegin();
25:        charLocator != stlString.cend();
26:        ++charLocator )
27:    {
28:        cout << "Символ[" << charOffset++ << "] = ";
29:        cout << *charLocator << endl;
30:    }
31:    cout << endl;
32:
33:    // Обращение к содержимому строки в стиле C
34:    cout << "Представление строки как char* = ";
35:    cout << stlString.c_str() << endl;
36:
37:    return 0;
38: }
```

Результат

Синтаксис обращения к массиву:

```
Символ[0] = H
Символ[1] = e
Символ[2] = l
Символ[3] = l
Символ[4] = o
Символ[5] = 
Символ[6] = S
Символ[7] = t
Символ[8] = r
Символ[9] = i
Символ[10] = n
Символ[11] = g
```

Вывод с использованием итератора:

```
Символ[0] = H
Символ[1] = e
Символ[2] = l
Символ[3] = l
Символ[4] = o
Символ[5] = 
Символ[6] = S
Символ[7] = t
Символ[8] = r
Символ[9] = i
Символ[10] = n
Символ[11] = g
```

Представление строки как `char*` = `Hello String`

Анализ

Код демонстрирует несколько способов обращения к содержимому строки. Итераторы важны в том смысле, что большинство функций-членов класса `string` возвращают свои результаты в форме итераторов. Строки 11–17 отображают символы строки с использованием предоставляемого классом `std::string` оператора индексации `[]`, как у массива. Обратите внимание, что этому оператору нужно передавать смещение символа от начала массива, как это делается в строке 16. Очень важно не пересечь границы строки, т.е. вы не должны читать символы со смещением, большим, чем длина строки. Строки 24–30 также посимвольно отображают содержимое строки, но уже с использованием итератора.

СОВЕТ

Избежать длинного объявления типа итератора, показанного в строке 23, можно, прибегнув к помощи ключевого слова `auto`, тем самым поручая компилятору вывести тип переменной на основе возвращаемого значения метода `std::string::cbegin()`, как это сделано в строке 24.

Конкатенация строк

Конкатенация строк может быть осуществлена с помощью либо оператора `+=`, либо функции-члена `append()`:

```
string sampleStr1("Hello");
string sampleStr2(" String!");
sampleStr1 += sampleStr2; // Использование std::string::operator+=

// Альтернативный вариант — функция std::string::append()
sampleStr1.append(sampleStr2); // (Перегружена также для char*)
```

В листинге 16.3 демонстрируется применение этих двух вариантов.

ЛИСТИНГ 16.3. Конкатенация строк с использованием оператора сложения с присваиванием (+=) или метода append()

```
0: #include <string>
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     string sampleStr1("Hello");
8:     string sampleStr2(" String!");
9:
10:    // Конкатенация
11:    sampleStr1 += sampleStr2;
12:    cout << sampleStr1 << endl << endl;
13:
14:    string sampleStr3(" Указатели можно не использовать!");
15:    sampleStr1.append(sampleStr3);
16:    cout << sampleStr1 << endl << endl;
17:
18:    const char* constCStyleString = " Но можно и использовать!";
19:    sampleStr1.append(constCStyleString);
20:    cout << sampleStr1 << endl;
21:
22:    return 0;
23: }
```

Результат

Hello String!

Hello String! Указатели можно не использовать!

Hello String! Указатели можно не использовать! Но можно и использовать!

Анализ

Строки 11, 15 и 19 демонстрируют различные способы конкатенации строк STL. Обратите внимание на использование оператора += и на возможность функции append() (у которой есть множество перегруженных версий) получать как строковые объекты (как показано в строке 11), так и символьные строки в стиле C.

Поиск символа или подстроки в строке

Класс string библиотеки STL предоставляет несколько перегруженных версий функции-члена find(), которая позволяет найти символ или подстроку в данном объекте класса string.

```
// Найти подстроку "day" в строке sampleStr, начиная поиск с позиции 0
size_t charPos = sampleStr.find("day", 0);

// Удостовериться, что подстрока найдена, сравнивая с string::npos
if (charPos != string::npos)
    cout << "Подстрока \"day\" найдена в позиции " << charPos;
else
    cout << "Подстрока не найдена." << endl;
```

В листинге 16.4 демонстрируется удобство применения метода `std::string::find()`.

ЛИСТИНГ 16.4. Использование метода `string::find()` для поиска подстроки или символа

```
0: #include <string>
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     string sampleStr("Good day String! Today is beautiful!");
8:     cout << "Исходная строка:" << "\n" << sampleStr << "\n\n";
9:
10:    // Поиск "day" - find() возвращает позицию
11:    size_t charPos = sampleStr.find("day", 0);
12:
13:    // Проверка, что подстрока найдена...
14:    if (charPos != string::npos)
15:        cout << "\"day\" найдено в позиции " << charPos << endl;
16:    else
17:        cout << "Подстрока не найдена." << endl;
18:
19:    cout << "Поиск всех подстрок \"day\"" << endl;
20:    size_t subStrPos = sampleStr.find("day", 0);
21:
22:    while(subStrPos != string::npos)
23:    {
24:        cout << "Найден \"day\" в позиции " << subStrPos << endl;
25:
26:        // Продолжаем поиск со следующего символа
27:        size_t searchOffset = subStrPos + 1;
28:
29:        subStrPos = sampleStr.find("day", searchOffset);
30:    }
31:
32:    return 0;
33: }
```

Результат

Исходная строка:

```
Good day String! Today is beautiful!
```

"day" найдено в позиции 5

Поиск всех подстрок "day"

Найден "day" в позиции 5

Найден "day" в позиции 19

Анализ

Строки 11–17 демонстрируют простейший случай применения функции `find()` — поиск в строке определенной подстроки. Результат вызова метода `find()` сравнивается со значением `std::string::npos` (фактически это значение `-1`); если они равны, искомый элемент в строке не найден. Если функция `find()` возвращает значение, не равное `npos`, это значение является смещением, указывающим позицию найденных подстроки или символа в строке.

Далее код демонстрирует применение функции `find()` в цикле `while` для поиска всех вхождений символа или подстроки в строку STL. Здесь используется версия функции `find()`, получающая два параметра: искомую подстроку или символ и смещение поиска, означающее точку, начиная с которой осуществляется поиск. В строке 29 в качестве второго параметра для поиска очередного вхождения подстроки мы передаем увеличенную на единицу позицию предыдущего вхождения искомой подстроки.

ПРИМЕЧАНИЕ

Строки STL предоставляют также функции, родственные функции `find()`, такие как `find_first_of()`, `find_first_not_of()`, `find_last_of()` и `find_last_not_of()`, и предоставляющие программисту дополнительные возможности поиска.

Усечение строк STL

Класс `string` библиотеки STL предоставляет функцию-член `erase()`, осуществляющую удаление:

- некоторого количества символов, если заданы смещение позиции и количество удаляемых символов:

```
string sampleStr("Hello String! Wake up to a beautiful day!");  
sampleStr.erase(13, 28); // Hello String!
```

- отдельного символа при наличии указывающего на него итератора:

```
sampleStr.erase(iCharS); // Итератор указывает удаляемый символ
```

- множества символов, находящихся между двух итераторов:

```
// Удалить все символы от начала до конца  
sampleStr.erase(sampleStr.begin(), sampleStr.end());
```

Пример в листинге 16.5 демонстрирует применение различных версий функции `string::erase()`.

ЛИСТИНГ 16.5. Использование функции `string::erase()` для усечения строки

```

0: #include <string>
1: #include <algorithm>
2: #include <iostream>
3:
4: int main()
5: {
6:     using namespace std;
7:
8:     string sampleStr("Hello String! Wake up to a beautiful day!");
9:     cout << "Исходная строка: " << endl;
10:    cout << sampleStr << endl << endl;
11:
12:    // Удалить из строки символы, заданные позицией и количеством
13:    cout << "Удаление второго предложения: " << endl;
14:    sampleStr.erase(13, 28);
15:    cout << sampleStr << endl << endl;
16:
17:    // Найти в строке символ 'S', используя алгоритм поиска
18:    string::iterator iCharS = find(sampleStr.begin(),
19:                                   sampleStr.end(), 'S');
20:
21:    // Если символ найден, удаляем его
22:    cout << "Удаление 'S' из исходной строки:" << endl;
23:    if (iCharS != sampleStr.end())
24:        sampleStr.erase(iCharS);
25:
26:    cout << sampleStr << endl << endl;
27:
28:    // Удаление диапазона символов
29:    cout << "Удаление символов от begin() до end(): " << endl;
30:    sampleStr.erase(sampleStr.begin(), sampleStr.end());
31:
32:    // Проверка длины строки после операции erase()
33:    if (sampleStr.length() == 0)
34:        cout << "Строка пуста" << endl;
35:
36:    return 0;
37: }
```

Результат

```

Исходная строка:
Hello String! Wake up to a beautiful day!
```


Удаление второго предложения:

Hello String!

Удаление 'S' из исходной строки:

Hello tring!

Удаление символов от `begin()` до `end()`:

Строка пуста

Анализ

Листинг демонстрирует три версии функции `erase()`. Одна версия удаляет набор символов, заданных начальным смещением и количеством, как показано в строке 14. Вторая версия удаляет определенный символ, заданный указывающим на него итератором, как показано в строке 24. Последняя версия удаляет диапазон символов, заданных парой итераторов, определяющих границы этого диапазона (строка 30). Поскольку границы этого диапазона предоставлены функциями-членами `begin()` и `end()` класса `string`, диапазон включает все содержимое строки, и вызов метода `erase()` для него полностью удаляет все содержимое строки. Обратите внимание, что класс `string` предоставляет также функцию `clear()`, которая эффективно очищает внутренний буфер и выполняет сброс объекта класса `string`.

СОВЕТ

Стандарт C++11 позволяет упростить пространное объявление итератора, представленное в листинге 16.5:

```
string::iterator iCharS = find(sampleStr.begin(),
                              sampleStr.end(), 'S');
```

Чтобы сократить его, можно использовать ключевое слово `auto`, как было продемонстрировано на занятии 3, "Использование переменных и констант":

```
auto iCharS = find(sampleStr.begin(), sampleStr.end(), 'S');
```

Компилятор автоматически выводит тип переменной `iCharS`, получая информацию о типе возвращаемого значения от функции `std::find`.

Обращение строки

Иногда необходимо изменить порядок символов в строке на обратный. Предположим, необходимо определить, не является ли введенная пользователем строка палиндромом, т.е. строкой, одинаково читаемой как с начала, так и с конца. Один из способов сделать это подразумевает изменение порядка букв в копии содержимого строки на обратный и сравнение с оригиналом. Обобщенный алгоритм `std::reverse()` библиотеки STL позволяет обратить содержимое строки:

```
string sampleStr("Hello String! We will reverse you!");
reverse(sampleStr.begin(), sampleStr.end());
```

В листинге 16.6 показано применение алгоритма `std::reverse()` к объекту класса `std::string`.

ЛИСТИНГ 16.6. Обращение строки с использованием алгоритма `std::reverse()`

```

0: #include <string>
1: #include <iostream>
2: #include <algorithm>
3:
4: int main()
5: {
6:     using namespace std;
7:
8:     string sampleStr("Hello String! We will reverse you!");
9:     cout << "Исходная строка: " << endl;
10:    cout << sampleStr << endl << endl;
11:
12:    reverse(sampleStr.begin(), sampleStr.end());
13:
14:    cout << "После применения алгоритма std::reverse: " << endl;
15:    cout << sampleStr << endl;
16:
17:    return 0;
18: }
```

Результат

Исходная строка:
Hello String! We will reverse you!

После применения алгоритма `std::reverse`:
!uoY esrever lliw eW !gnirts olleH

Анализ

Алгоритм `std::reverse()`, использованный в строке 12, работает в контейнере в пределах, заданных двумя входными параметрами. В нашем случае эти пределы — это начало и конец строкового объекта, так что обращается порядок символов всей строки. Строку можно обратить и частично, задавая соответствующие границы. Обратите внимание: границы не должны превышать значение `end()`.

Смена регистра символов

Для смены регистра символов используется алгоритм `std::transform()`, применяющий определенную пользователем функцию к каждому элементу коллекции. В данном случае коллекция — это не что иное, как объект класса `string`. Пример в листинге 16.7 демонстрирует смену регистра символов в строке.

ЛИСТИНГ 16.7. Преобразование строки в верхний регистр с использованием алгоритма `std::transform()`

```
0: #include <string>
1: #include <iostream>
2: #include <algorithm>
3:
4: int main()
5: {
6:     using namespace std;
7:
8:     cout << "Введите строку для преобразования:" << endl;
9:     cout << "> ";
10:
11:     string inStr;
12:     getline(cin, inStr);
13:     cout << endl;
14:
15:     transform(inStr.begin(), inStr.end(), inStr.begin(), ::toupper);
16:     cout << "Преобразованная в верхний регистр строка:" << endl;
17:     cout << inStr << endl << endl;
18:
19:     transform(inStr.begin(), inStr.end(), inStr.begin(), ::tolower);
20:     cout << "Преобразованная в нижний регистр строка:" << endl;
21:     cout << inStr << endl << endl;
22:
23:     return 0;
24: }
```

Результат

Введите строку для преобразования:
> Convert this StrIng!

Преобразованная в верхний регистр строка:
CONVERT THIS STRING!

Преобразованная в нижний регистр строка:
convert this string!

Анализ

Строки 15 и 19 демонстрируют, насколько эффективно можно применить алгоритм `std::transform()` для изменения регистра содержимого строки.

Реализация строки на базе шаблона STL

Как уже упоминалось, класс `std::string` фактически представляет собой специализацию шаблонного класса STL `std::basic_string<T>`. Объявление шаблона контейнерного класса `basic_string` имеет следующий вид:

```
template<class _Elem,
        class _Traits,
        class _Ax>
class basic_string
```

В этом определении шаблона крайне важен первый параметр: `_Elem`. Это тип объектов, хранимых коллекцией `basic_string`. Таким образом, класс `std::string` — это специализация шаблона `basic_string` для `_Elem=char`, в то время как класс `wstring` — это специализация того же шаблона для `_Elem=wchar_t`.

Другими словами, класс `string` библиотеки STL определяется так:

```
typedef basic_string<char, char_traits<char>, allocator<char>> string;
```

Класс `wstring` библиотеки STL определяется следующим образом:

```
typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t>>
wstring;
```

Значит, все функциональные возможности класса `string`, рассмотренные к этому моменту, фактически предоставляются шаблоном `basic_string`, а потому имеются и у класса `wstring`.

СОВЕТ

Используйте класс `std::wstring` для приложений, которые должны поддерживать не латинские символы, такие как японские или китайские иероглифы.

Оператор `"s` в `std::string` в C++14

Отвечающие стандарту C++14 версии стандартной библиотеки поддерживают оператор `"s`, который преобразует строки в кавычках в целом в `std::basic_string<t>`. Это делает некоторые строковые операции интуитивно понятными и простыми, как демонстрируется в листинге 16.8.

ЛИСТИНГ 16.8. Использование оператора `"s` в C++14

```
0: #include<string>
1: #include<iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     string str1("Традиционная \0 строка");
```

```
7:     cout << "Str1: " << str1 << " Длина: " << str1.length() << endl;
8:
9:     string str2("Инициализация \0 в C++14"s);
10:    cout << "Str2: " << str2 << " Длина: " << str2.length() << endl;
11:
12:    return 0;
13: }
```

Вывод

Str1: Традиционная Длина: 13

Str2: Инициализация в C++14 Длина: 23

Анализ

В строке 6 выполняется инициализация экземпляра `std::string` с помощью обычного строкового литерала. Обратите внимание на нулевой символ посреди строки, который приводит к тому, что слово "строка" полностью отсутствует в `str1`. В строке 9 использован оператор `""s`, введенный в стандарте C++14, чтобы продемонстрировать, что экземпляр `str2` может содержать в своем символьном буфере нулевой символ.

ВНИМАНИЕ!

Стандарт C++14 вводит оператор `""s` и в `std::chrono`, как показано далее:

```
std::chrono::seconds timeInSec(100s); // 100 секунд
std::string timeInText = "100"s;      // Строка "100"
```

В первой строке с использованием целочисленного литерала указано время в секундах, а во второй определена строка.

СОВЕТ

Ожидается, что в стандарте C++17 появится класс `std::string_view`, который должен будет повысить производительность, избегая излишних выделений памяти. Чтобы узнать, что еще ожидается в стандарте C++17, обратитесь к занятию 29, "Что дальше".

Резюме

На сегодняшнем занятии был рассмотрен класс `string` библиотеки STL. Это предоставляемый стандартной библиотекой шаблон контейнер, обеспечивающий разработчику множество функциональных возможностей для работы со строками. При использовании данный класс предоставляет программисту очевидные преимущества, беря на себя заботу об управлении памятью, о сравнении строк и многих других функциях для работы со строками.

Вопросы и ответы

- Я должен обратить порядок символов в строке, используя алгоритм `std::reverse()`. Какой заголовочный файл я должен включить в свою программу, чтобы воспользоваться этой функцией?

Чтобы функция `std::reverse()` стала доступной, следует включить заголовочный файл `<algorithm>`.

- Какую роль играет алгоритм `std::transform()` в преобразовании символов строки в нижний регистр с использованием функции `tolower()`?

Функция `std::transform()` вызывает функцию `tolower()` для всех символов объекта `string` в пределах границ, переданных этой функции.

- Почему классы `std::wstring` и `std::string` демонстрируют одинаковое поведение и одинаковые функции-члены?

Они оба являются специализацией шаблона класса `std::basic_string`.

- Чувствительны ли к регистру операторы сравнения (наподобие оператора `<`) строковых классов библиотеки STL?

Получаемые результаты основаны на сравнении с учетом регистра символов.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Какой шаблон класса STL специализирует класс `std::string`?
2. Как бы вы осуществили сравнение двух строк, не зависящее от регистра символов?
3. Похожи ли строки STL и строки в стиле C с завершающими нулевыми символами?

Упражнения

1. Напишите программу проверки, не являются ли вводимые пользователем слова палиндромами. Например, слово АТОУОТА — палиндром, поскольку при обращении порядка символов оно не изменяется.
2. Напишите программу, подсчитывающую количество гласных во введенном предложении.

3. Преобразуйте каждый второй символ строки в верхний регистр.
4. В вашей программе должно быть четыре строковых объекта, инициализированных строками "I", "Love", "STL" и "String". Добавьте к ним промежуточные пробелы и выведите получившееся предложение на консоль.
5. Напишите программу, которая выводит позиции всех букв 'a' в строке "Good day String! Today is beautiful!"

ЗАНЯТИЕ 17

Классы динамических массивов библиотеки STL

В отличие от статических массивов, динамические массивы обладают большей гибкостью, позволяя хранить данные, даже если их точный объем во время разработки приложения неизвестен. Естественно, это очень распространенное требование, и стандартная библиотека шаблонов (STL) предоставляет готовое к применению решение в виде класса `std::vector`.

На этом занятии...

- Характеристики класса `std::vector`
- Типичные операции с вектором
- Концепция размера и емкости вектора
- Класс `deque` библиотеки STL

Характеристики класса `std::vector`

Шаблонный класс `vector` предоставляет обобщенные функциональные возможности динамического массива и характеризуется следующими свойствами.

- Продолжительность операции добавления элемента в конец массива — константная, не зависящая от размера массива. То же самое относится к получению значения элемента.
- Продолжительность операции вставки элементов в середину массива или удаления их оттуда пропорциональна количеству элементов за этим элементом.
- Количество содержащихся в векторе элементов является динамическим; при этом класс `vector` сам управляет операциями по использованию памяти.

Вектор (`vector`) — это динамический массив, который может быть представлен, как на рис. 17.1.

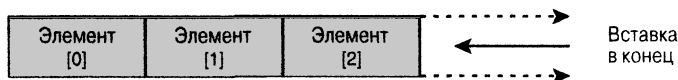


РИС. 17.1. Внутренняя организация вектора

СОВЕТ

Чтобы использовать класс `std::vector`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <vector>
```

Типичные операции с вектором

Открытые методы и члены класса `std::vector` определены стандартом C++. Следовательно, операции с вектором, рассматриваемые на этом занятии, должны поддерживаться всеми платформами программирования C++, соответствующими стандарту C++.

Создание экземпляра вектора

Экземпляры шаблона класса `vector` создаются в соответствии с методиками, описанными на занятии 14, “Введение в макросы и шаблоны”. При создании экземпляра шаблона `vector` следует определить тип объекта, для которого нужно получить динамический массив.

Экземпляр шаблона класса `vector` может быть создан следующим образом:

```
std::vector<int> dynIntArray;    // Вектор для целых чисел
std::vector<float> dynFloatArray; // Вектор для float
std::vector<Tuna> dynTunaArray;  // Вектор для Tuna
```

Для объявления итератора, указывающего на элементы вектора, используется код

```
std::vector<int>::const_iterator elementInVec;
```

Если необходим итератор для изменения значений или вызова неконстантных функций, вместо `const_iterator` используйте ключевое слово `iterator`.

Поскольку класс `std::vector` имеет несколько перегруженных конструкторов, можно создать его экземпляр, указав начальное количество элементов и их исходные значения, либо использовать часть одного вектора (или весь) для создания экземпляра другого.

Создание нескольких экземпляров вектора представлено в листинге 17.1.

ЛИСТИНГ 17.1. Различные формы создания экземпляров класса `std::vector`

```

0: #include <vector>
1:
2: int main()
3: {
4:     // Вектор целых чисел
5:     std::vector<int> integers;
6:
7:     // Использование списка инициализации (C++11)
8:     std::vector<int> initVector{ 202, 2017, -1 };
9:
10:    // Вектор с 10 элементами (может расти во время выполнения)
11:    std::vector<int> tenElements(10);
12:
13:    // Вектор с 10 элементами, каждый равен 90
14:    std::vector<int> tenElemInit(10, 90);
15:
16:    // Инициализация вектора содержимым другого вектора
17:    std::vector<int> copyVector(tenElemInit);
18:
19:    // Вектор инициализирован 5 элементами другого вектора
20:    std::vector<int> partialCopy(tenElements.cbegin(),
21:                                tenElements.cbegin() + 5);
22:
23:    return 0;
24: }
```

Анализ

В приведенном выше коде используется специализация шаблона класса `vector` для типа `int`; другими словами, создается экземпляр вектора целых чисел. Этот вектор с именем `integers` использует конструктор по умолчанию, полезный в случае неизвестного минимального размера контейнера, т.е. когда заранее неизвестно, сколько целых чисел предстоит в нем хранить. Второй способ создания экземпляра вектора в строке 8 использует инициализацию списком значений, появившуюся в стандарте C++11, и инициализирует три элемента вектора значениями 202, 2017 и -1 соответственно. Инициализация вектора в строках 11 и 14 используется, когда разработчику известно, что он нуждается в векторе, способном хранить по крайней мере 10 элементов. Обратите внимание, что окончательный размер контейнера это не ограничивает, устанавливая только начальный размер. Наконец в строках 17 и 20 показано, как для

инициализации вектора используется содержимое другого вектора, другими словами, как создать вектор, являющийся копией или частью другого. Эта конструкция работает со всеми контейнерами библиотеки STL. Последняя форма инициализации использует итераторы. Вектор `partialCopy` содержит первые пять элементов вектора `tenElements`.

ПРИМЕЧАНИЕ

Четвертая конструкция способна работать только с объектами подобных типов. Так, вы можете создать экземпляр `vecArrayCopy` вектора целочисленных объектов, используя другой вектор целочисленных объектов. Если один из них был бы вектором, скажем, элементов типа `float`, такой код не компилировался бы.

Вставка элементов в конец вектора с помощью `push_back()`

Следующая вполне очевидная задача после создания экземпляра вектора целых чисел — это вставка в него элементов. Вставка значений в вектор осуществляется в конец массива с использованием метода `push_back()`:

```
vector<int> integers; // Объявление вектора для типа int

// Вставка целых чисел в вектор:
integers.push_back(50);
integers.push_back(1);
```

В листинге 17.2 демонстрируется использование метода `push_back()` для динамического добавления элементов в вектор.

ЛИСТИНГ 17.2. Вставка элементов в вектор с использованием `push_back()`

```
0: #include <iostream>
1: #include <vector>
2: using namespace std;
3:
4: int main()
5: {
6:     vector<int> integers;
7:
8:     // Вставка целых чисел в вектор:
9:     integers.push_back(50);
10:    integers.push_back(1);
11:    integers.push_back(987);
12:    integers.push_back(1001);
13:
14:    cout << "Вектор содержит ";
15:    cout << integers.size() << " элемента" << endl;
16:
17:    return 0;
18: }
```

Результат

Вектор содержит 4 элемента

Анализ

Метод `push_back()` в строках 9–12 представляет собой открытый член класса `vector`, вставляющий элементы в конец динамического массива. Обратите внимание на использование функции `size()`, которая возвращает количество элементов, хранящихся в векторе.

Инициализация списком

Язык C++11 предоставляет класс списков инициализации `std::initialize_list<>`, позволяющий создать экземпляр вектора и инициализировать его элементы так, как будто это статический массив. Подобно большинству стандартных контейнеров, вектор поддерживает инициализацию списком, позволяя инициализировать содержимое вектора одной строкой:

```
vector<int> integers = {50, 1, 987, 1001};  
// Альтернатива:  
vector<int> vecMoreIntegers {50, 1, 987, 1001};
```

Вставка элементов в определенную позицию с помощью `insert()`

Метод `push_back()` позволяет вставить элементы в конец вектора. Но что если нужно вставить элемент в середину вектора? Многие контейнеры библиотеки STL, включая класс `std::vector`, предоставляют функцию `insert()` со множеством перегруженных версий.

Одна позволяет задать позицию вставки элемента в последовательность:

```
// Вставить элемент в начало  
integers.insert(integers.begin(), 25);
```

Другая позволяет определить позицию и количество элементов с одинаковым значением, которые должны быть вставлены:

```
// Вставить в конец 2 числа со значением 45  
integers.insert(integers.end(), 2, 45);
```

Вы можете также вставить содержимое одного вектора в некоторую позицию другого:

```
// Другой вектор, содержащий два элемента со значением 30  
vector<int> another(2, 30);
```

```
// Вставить два элемента из этого вектора в позицию [1]  
integers.insert(integers.begin() + 1,  
                another.begin(), another.end());
```

Для указания функции `insert()` позиции вставки новых элементов, как правило, используется итератор, возвращаемый функцией `begin()` или `end()`.

СОВЕТ

Такой итератор может быть возвращен некоторым алгоритмом STL, таким, например, как `std::find()`, применяемым для поиска элемента и последующей вставки другого в эту позицию (вставка сдвинет найденный элемент). Алгоритмы стандартной библиотеки более подробно рассматриваются на занятии 23, "Алгоритмы библиотеки STL".

Указанные перегрузки метода `vector::insert()` представлены в листинге 17.3.

ЛИСТИНГ 17.3. Использование `vector::insert()` для вставки элементов в определенную позицию

```
0: #include <vector>
1: #include <iostream>
2: using namespace std;
3:
4: void DisplayVector(const vector<int>& inVec)
5: {
6:     for(auto element = inVec.cbegin();
7:         element != inVec.cend();
8:         ++element )
9:         cout << *element << ' ';
10:
11:     cout << endl;
12: }
13:
14: int main()
15: {
16:     // Создать экземпляр вектора с 4 элементами со значением 90
17:     vector <int> integers(4, 90);
18:
19:     cout << "Начальное содержимое вектора: ";
20:     DisplayVector(integers);
21:
22:     // Вставить 25 в начало
23:     integers.insert(integers.begin(), 25);
24:
25:     // Вставить в конец 2 числа со значением 45
26:     integers.insert(integers.end(), 2, 45);
27:
28:     cout << "Содержимое вектора после вставок: ";
29:     DisplayVector(integers);
30:
31:     // Другой вектор, содержащий два элемента со значением 30
32:     vector <int> another(2, 30);
```

```
33:
34:     // Вставить два элемента из другого контейнера в позицию [1]
35:     integers.insert(integers.begin() + 1,
36:                    another.begin(), another.end());
37:
38:     cout << "Вектор после вставки элементов из ";
39:     cout << "другого вектора в середину: " << endl;
40:     DisplayVector(integers);
41:
42:     return 0;
43: }
```

Результат

Начальное содержимое вектора: 90 90 90 90

Содержимое вектора после вставок: 25 90 90 90 90 45 45

Вектор после вставки элементов из другого вектора в середину:
25 30 30 90 90 90 90 45 45

Анализ

Этот код демонстрирует мощь функции `insert()`, позволяющей помещать значения в середину контейнера. Вектор в строке 17 содержит четыре элемента, которые инициализированы значением 90. Взяв этот вектор в качестве отправной точки, используем различные перегруженные версии функции-члена `vector::insert()`. В строке 23 один элемент добавляется в начало. В строке 26 используется перегруженная версия, добавляющая в конец два элемента со значением 45. Строка 35 демонстрирует возможность вставки элементов из одного вектора в середину другого (в этом примере во вторую позицию со смещением 1).

Хотя метод `vector::insert()` весьма универсален, для добавления элементов в вектор предпочтительнее использовать `push_back()`. Дело в том, что `insert()` — неэффективное средство добавления элементов в вектор (при добавлении в позицию, отличную от конца последовательности), поскольку добавление элементов в начало или середину вектора сдвигает все последующие элементы назад (после создания места для последних в конце). Таким образом, в зависимости от типа объектов, содержащихся в последовательности, продолжительность этой операции может оказаться существенной (поскольку она будет вызывать копирующие конструкторы или операторы присваивания). В данном примере вектор содержит объекты типа `int`, перемещение которых осуществляется относительно быстро. Однако в других случаях все могло бы быть вовсе не так хорошо.

СОВЕТ

Если в вашем приложении требуются частые вставки в середину контейнера, имеет смысл использовать список `std::list`, рассматриваемый на странице 18, “Классы `list` и `forward_list`”.

Доступ к элементам вектора с использованием семантики массива

К элементам вектора можно обратиться, используя семантику массива с оператором индексации (`[]`), с помощью функции-члена `at()` или итераторов.

В листинге 17.1 показано создание экземпляра вектора, изначально хранящего 10 элементов:

```
std::vector<int> tenElements(10);
```

К индивидуальным элементам вектора можно обратиться, используя такой же синтаксис, как и в случае массива:

```
tenElements[3] = 2011; // Присваивание четвертому элементу
```

В листинге 17.4 демонстрируется обращение к элементам вектора с использованием оператора индексации (`[]`).

ЛИСТИНГ 17.4. Доступ к элементам вектора с использованием семантики массива

```
0: #include <iostream>
1: #include <vector>
2:
3: int main()
4: {
5:     using namespace std;
6:     vector<int> integers{ 50, 1, 987, 1001 };
7:
8:     for(size_t index = 0; index < integers.size(); ++index)
9:     {
10:         cout << "Элемент[" << index << "] = " ;
11:         cout << integers[index] << endl;
12:     }
13:
14:     integers[2] = 2011; // Изменение значения 3-го элемента
15:     cout << "После замены: " << endl;
16:     cout << "Элемент[2] = " << integers[2] << endl;
17:
18:     return 0;
19: }
```

Результат

```
Элемент[0] = 50
Элемент[1] = 1
Элемент[2] = 987
Элемент[3] = 1001
После замены:
Элемент[2] = 2011
```

Анализ

В строках 11, 14 и 16 вектор используется для доступа к элементам таким же способом, как и при использовании статического массива, — с помощью оператора индексации (`[]`). Этот оператор получает индекс элемента, который указывает отсчитываемую от нуля позицию, как и в статическом массиве. Обратите внимание, как в строке 8 записан цикл `for`: в нем индекс сравнивается со значением, возвращаемым `vector::size()`, чтобы индекс не вышел за пределы вектора.

ВНИМАНИЕ!

Доступ к элементам вектора с использованием оператора `[]` чреват теми же неприятностями, что и при обращении к элементам массива, — вы не должны пересекать границы контейнера. Если использовать оператор индексации (`[]`) для доступа к элементу вектора в позиции, находящейся за его границей, результат операции будет непредсказуемым (может случиться что угодно, например нарушение прав доступа к памяти).

Более безопасной альтернативой является функция-член `at()`:

```
// Получить элемент в позиции 2
cout << integers.at(2);
// Альтернативный код строки 11 листинга 17.4:
cout << integers.at(Index);
```

Во время выполнения метод `at()` проверяет размер контейнера и генерирует исключение при выходе за границы.

Оператор индексации (`[]`) безопасен, если используется способом, гарантирующим целостность границ, как в приведенном выше примере.

Доступ к элементам вектора с использованием семантики указателя

К элементам вектора можно также обращаться с использованием семантики в стиле указателей — с помощью итераторов (листинг 17.5).

ЛИСТИНГ 17.5. Доступ к элементам вектора с помощью итераторов

```
0: #include <iostream>
1: #include <vector>
2:
3: int main()
4: {
5:     using namespace std;
6:     vector<int> integers{ 50, 1, 987, 1001 };
7:
8:     vector<int>::const_iterator element = integers.cbegin();
9:     // auto element = integers.cbegin(); // Вывод типа
10:
11:     while(element != integers.end())
12:     {
```



```
13:         size_t index = distance(integers.cbegin(), element);
14:
15:         cout << "Элемент в позиции ";
16:         cout << index << " равен " << *element << endl;
17:
18:         // Переход к следующему элементу
19:         ++element;
20:     }
21:
22:     return 0;
23: }
```

Результат

```
Элемент в позиции 0 равен 50
Элемент в позиции 1 равен 1
Элемент в позиции 2 равен 987
Элемент в позиции 3 равен 1001
```

Анализ

Итератор в этом примере ведет себя так же, как указатель, и характер его применения сходен с использованием арифметики указателей, как можно увидеть в строке 16, где обращение к хранящемуся в векторе значению выполняется с использованием оператора разыменования (*), и в строке 19, где инкремент итератора с помощью оператора ++ переносит его к следующему элементу. Обратите внимание на использование в строке 13 метода `std::distance()` для вычисления отсчитываемой от нуля позиции смещения элемента в векторе (т.е. позиции относительно его начала) по значению, возвращаемому методом `cbegin()` и указывающему на элемент итератору. В строке 9 показана более краткая альтернатива объявлению итератора в строке 8 с использованием автоматического вывода типа переменной компилятором, о котором говорилось на занятии 3, “Использование переменных и констант”.

Удаление элементов из вектора

Подобно тому, как метод `push_back()` обеспечивает вставку в конец вектора, метод `pop_back()` обеспечивает удаление элемента из него. Удаление элемента из вектора с помощью метода `pop_back()` выполняется за константное время, т.е. за время, которое не зависит от количества хранящихся в векторе элементов. Код в листинге 17.6 демонстрирует использование функции `pop_back()` для удаления элементов с конца вектора.

ЛИСТИНГ 17.6. Использование метода `pop_back()` для удаления последнего элемента

```
0: #include <iostream>
1: #include <vector>
2: using namespace std;
```

```
3:
4: template <typename T>
5: void DisplayVector(const vector<T>& inVec)
6: {
7:     for(auto element = inVec.cbegin();
8:         element != Input.cend();
9:         ++element )
10:         cout << *element << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main()
16: {
17:     vector <int> integers;
18:
19:     // Вставка в вектор целых чисел:
20:     integers.push_back(50);
21:     integers.push_back(1);
22:     integers.push_back(987);
23:     integers.push_back(1001);
24:
25:     cout << "Вектор содержит " << integers.size() << " элемента: ";
26:     DisplayVector(integers);
27:
28:     // Удалить один элемент в конце
29:     integers.pop_back();
30:
31:     cout << "После вызова pop_back()" << endl;
32:     cout << "Вектор содержит " << integers.size() << " элемента: ";
33:     DisplayVector(integers);
34:
35:     return 0;
36: }
```

Результат

Вектор содержит 4 элемента: 50 1 987 1001

После вызова pop_back()

Вектор содержит 3 элемента: 50 1 987

Анализ

Вывод программы показывает, что метод `pop_back()`, использованный в строке 29, уменьшил количество элементов в векторе, удалив последний вставленный в него элемент. В строке 32 очередной вызов метода `size()` показывает, что количество элементов в векторе сократилось на один.

ПРИМЕЧАНИЕ

Функция `DisplayVector()` в строках 4–13 листинга 17.6 стала шаблонной, что делает ее более универсальной по сравнению с листингом 17.3, в котором она принимала только вектор целых чисел. Шаблонность данной функции позволяет нам повторно использовать ее для вектора с элементами типа `float` (вместо типа `int`):

```
vector<float> vecFloats;  
DisplayVector(vecFloats); // Обобщенная функция для любого  
вектора
```

Теперь она в состоянии работать с вектором элементов любого типа, лишь бы его элементы могли быть выведены в поток `cout`.

Концепции размера и емкости

Размер (`size`) вектора — это количество хранимых в нем элементов. *Емкость* (`capacity`) вектора — это общее количество элементов, которые могут быть сохранены в векторе, прежде чем новое выделение памяти позволит сохранить большее количество элементов. Поэтому размер вектора никогда не превышает его емкость.

Количество элементов в векторе можно выяснить, вызвав функцию `size()`:

```
cout << "Размер: " << integers.size();
```

Емкость возвращает функция `capacity()`:

```
cout << "Емкость: " << integers.capacity() << endl;
```

Если вектору требуется частое перераспределение памяти, это может создать проблемы с производительностью. Данная проблема в значительной степени может быть решена с помощью функции-члена `reserve(число)`, резервирующей память для будущего использования. Эта функция увеличивает объем памяти, выделяемой для внутреннего массива вектора, что позволяет сохранять больше элементов без повторных перераспределений памяти. Такое сокращение количества перераспределений памяти и копирования объектов в новое место повышает общую производительность. Пример кода в листинге 17.7 демонстрирует различие между размером и емкостью.

ЛИСТИНГ 17.7. Демонстрация применения методов `size()` и `capacity()`

```
0: #include <iostream>  
1: #include <vector>  
2:  
3: int main()  
4: {  
5:     using namespace std;  
6:  
7:     // Создание вектора с 5 элементами-целыми числами  
8:     vector<int> integers(5);  
9:  
10:    cout << "Вектор инстанцирован с параметрами: " << endl;
```

```
11:     cout << "размер: " << integers.size();
12:     cout << ", емкость: " << integers.capacity() << endl;
13:
14:     // Вставка в вектор 6-го элемента
15:     integers.push_back(666);
16:
17:     cout << "После вставки нового элемента..." << endl;
18:     cout << "размер: " << integers.size();
19:     cout << ", емкость: " << integers.capacity() << endl;
20:
21:     // Вставка другого элемента
22:     integers.push_back(777);
23:
24:     cout << "После вставки еще одного элемента..." << endl;
25:     cout << "размер: " << integers.size();
26:     cout << ", емкость: " << integers.capacity() << endl;
27:
28:     return 0;
29: }
```

Результат

```
Вектор инстанцирован с параметрами:
размер: 5, емкость: 5
После вставки нового элемента...
размер: 6, емкость: 7
После вставки еще одного элемента...
размер: 7, емкость: 7
```

Анализ

В строке 8 создан вектор целых чисел, изначально содержащий пять целых чисел со значением по умолчанию (0). В строках 11 и 12 выводятся соответственно размер и емкость этого вектора, свидетельствуя, что при создании вектора они равны. В строке 9 в вектор добавляется шестой элемент. Поскольку емкость вектора до вставки была равна 5, во внутреннем буфере вектора больше нет места для сохранения этого нового элемента. Другими словами, для сохранения шестого элемента класс `vector` должен расширить свой внутренний буфер, что делается путем выделения новой памяти и копирования в нее старого содержимого. Реализация логики повторного выделения памяти достаточно интеллектуальна, чтобы избежать перераспределения при вставке каждого элемента; как правило, при расширении буфера выделяется память, превосходящая требования текущего момента.

Вывод доказывает это при вставке в вектор емкостью пять элементов шестого элемента — выделение памяти увеличивает емкость до семи элементов. Метод `size()` всегда отображает количество элементов, хранящихся в векторе, и в данный момент оно равно шести. Добавление седьмого элемента в строке 22 не приводит к увеличению емкости, так как имеющейся памяти пока достаточно. На этом этапе и размер, и

емкость имеют одинаковое значение, а это означает, что вектор заполнен полностью, и вставка следующего элемента приведет к выделению нового буфера большего размера, копированию в него существующих элементов и вставке нового значения.

ПРИМЕЧАНИЕ

Величина избыточного увеличения емкости внутреннего буфера вектора не регулируется никакими стандартными директивами. Она зависит от конкретной реализации используемой библиотеки STL.

Класс deque библиотеки STL

Дек — класс `deque` — является классом динамического массива библиотеки STL, похожим на класс `vector`, но обеспечивающим быструю вставку и удаление элементов как в конец, так и в начало данного массива. Экземпляр класса `deque` для целых чисел можно создать следующим образом:

```
// Определение двухсторонней очереди целых чисел
deque<int> intDeque;
```

СОВЕТ

Чтобы использовать класс `std::deque`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include<deque>
```

Схематически *двухстороннюю очередь* (`deque` — *дек*) можно представить так, как показано на рис. 17.2.



РИС. 17.2. Внутренняя организация двухсторонней очереди

Двухсторонняя очередь очень похожа на вектор в том, что она обеспечивает вставку и извлечение элементов с конца с помощью методов `push_back()` и `pop_back()`. Подобно вектору, двухсторонняя очередь обеспечивает также доступ к элементам с использованием семантики массива и оператора индексации (`[]`). Отличается же двухсторонняя очередь от вектора тем, что позволяет вставлять элементы и в начало массива и извлекать их оттуда, используя методы `push_front()` и `pop_front()` (листинг 17.8).

ЛИСТИНГ 17.8. Создание и использование дека STL

```
0: #include <deque>
1: #include <iostream>
2: #include <algorithm>
3:
4: int main()
```

```

5: {
6:     using namespace std;
7:
8:     // Определение двухсторонней очереди целых чисел
9:     deque<int> intDeque;
10:
11:    // Вставка целых чисел в конец массива
12:    intDeque.push_back(3);
13:    intDeque.push_back(4);
14:    intDeque.push_back(5);
15:
16:    // Вставка целых чисел в начало массива
17:    intDeque.push_front(2);
18:    intDeque.push_front(1);
19:    intDeque.push_front(0);
20:
21:    cout << "Содержимое дека после вставки элементов ";
22:    cout << "в начало и конец:" << endl;
23:
24:    // Вывод содержимого дека на экран
25:    for(size_t count = 0;
26:        count < intDeque.size();
27:        ++count )
28:    {
29:        cout << "Элемент[" << count << "] = ";
30:        cout << intDeque[count] << endl;
31:    }
32:
33:    cout << endl;
34:
35:    // Извлечение элемента из начала
36:    intDeque.pop_front();
37:
38:    // Извлечение элемента из конца
39:    intDeque.pop_back();
40:
41:    cout << "Содержимое дека после удаления элементов ";
42:    cout << "из начала и из конца:" << endl;
43:
44:    // Отображение содержимого с помощью итераторов
45:    // Следующая строка – для устаревших компиляторов
46:    // deque<int>::iterator element;
47:    for(auto element = intDeque.begin();
48:        element != intDeque.end();
49:        ++element )
50:    {
51:        size_t offset = distance(intDeque.begin(), element);
52:        cout << "Элемент[" << offset << "] = " << *element << endl;
53:    }

```

```
54:
55:     return 0;
56: }
```

Результат

Содержимое дека после вставки элементов в начало и конец:

```
Элемент[0] = 0
Элемент[1] = 1
Элемент[2] = 2
Элемент[3] = 3
Элемент[4] = 4
Элемент[5] = 5
```

Содержимое дека после удаления элементов из начала и из конца:

```
Элемент[0] = 1
Элемент[1] = 2
Элемент[2] = 3
Элемент[3] = 4
```

Анализ

В строке 9 создается экземпляр дека целых чисел. Обратите внимание, насколько этот синтаксис похож на синтаксис создания вектора целых чисел. Строки 12–14 демонстрируют применение функции-члена `push_back()` класса `deque`, а строки 17–19 — функции-члена `push_front()`. Наличие последней отличает дек от вектора. Применение метода `pop_front()` показано в строке 36. Первый механизм отображения содержимого дека использует для доступа к элементам синтаксис обращения к элементам массива, а второй — итераторы. В последнем случае, как показано в строках 47–53, для вычисления позиции смещения элемента в двухсторонней очереди используется алгоритм `std::distance()`, точно так же, как и при работе с вектором в листинге 17.5.

СОВЕТ

Если вам нужно опустошить контейнер STL, такой как вектор или дек (т.е. удалить все хранящиеся в нем элементы), вы можете воспользоваться функцией-членом `clear()`.

Приведенный далее код удаляет все элементы из вектора `vector<int> integers` из листинга 17.7:

```
integers.clear();
```

И вектор, и дек имеют функцию-член `empty()`, которая возвращает `true`, если контейнер пуст. Сама она не удаляет элементы из контейнера — это делает функция `clear()`, как показано далее для дека `deque<int> intDeque` из листинга 17.8:

```
intDeque.clear();
if (intDeque.empty())
    cout << "Контейнер пуст" << endl;
```

РЕКОМЕНДУЕТСЯ

Используйте динамические массивы `vector` или `deque`, если количество элементов, которые необходимо хранить в массиве, заранее неизвестно.

Помните, что вектор может расти только с конца, с помощью функции-члена `push_back()`.

Помните, что дек может расти в обоих направлениях, с помощью функций-членов `push_back()` и `push_front()`.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте, что функция-член `pop_back()` извлекает последний элемент из коллекции.

Не забывайте, что функция-член `pop_front()` удаляет из дека первый элемент.

Не обращайтесь к динамическому массиву за его пределами.

Резюме

На сегодняшнем занятии рассматривались основы использования таких динамических массивов, как вектор и двухсторонняя очередь (дек). Здесь были объяснены концепции размера и емкости, а также пояснено, как можно оптимизировать применение вектора для уменьшения количества повторных выделений памяти для внутреннего буфера, копирование объектов в котором способно отрицательно повлиять на производительность. Вектор является самым простым из контейнеров библиотеки STL, но при этом самым эффективным и часто используемым.

Вопросы и ответы

■ Изменяет ли вектор порядок хранящихся в нем элементов?

Вектор — последовательный контейнер, его элементы хранятся в порядке их вставки.

■ Какая функция используется для вставки элементов в вектор и куда вставляется объект?

Функция-член `push_back()` вставляет элементы в конец вектора.

■ Какая функция возвращает количество хранимых в векторе элементов?

Функция-член `size()` возвращает количество элементов, хранимых в векторе. Это утверждение справедливо для всех контейнеров STL.

■ Вставка и извлечение элементов из вектора отнимают больше времени, если вектор содержит больше элементов?

Нет. Время выполнения операций вставки в конец вектора и извлечения элементов оттуда не зависит от количества элементов в векторе.

■ В чем заключается преимущество использования функции-члена `reserve()`?

Она резервирует пространство для внутреннего буфера вектора, снижая частоту повторных выделений памяти при вставке элементов в вектор. В зависимости от характера хранимых в векторе объектов выделение памяти для нового буфера вектора и копирование в него элементов из прежнего буфера может существенно снизить производительность.

■ Есть ли различие между вектором и деком, когда речь идет о вставке элементов?

Пока речь идет о вставке в конец с постоянным временем выполнения и вставке в середину с пропорциональным размеру контейнера временем выполнения, никакой разницы нет. Но вектор допускает быструю вставку только в конец, а двухсторонняя очередь — и в конец, и в начало контейнера.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Могут ли элементы вектора быть вставлены в его середину или начало за константное время?
2. Метод `size()` моего вектора возвращает значение 10, а метод `capacity()` — 20. Сколько еще элементов я могу вставить в него до очередного выделения памяти для внутреннего буфера вектора?
3. Что делает функция `pop_back()`?
4. Если `vector<int>` является динамическим массивом целых чисел, то динамическим массивом какого типа является вектор `vector<Mammal>`?
5. Можно ли обращаться к элементам вектора в произвольном порядке? Если да, то как?
6. Какой тип итератора обеспечивает произвольный доступ к элементам вектора?

Упражнения

1. Напишите интерактивную программу, которая считывает введенное пользователем целое число и сохраняет его в векторе. Пользователь должен быть в состоянии в любой момент обратиться к хранящемуся в векторе значению, указав его индекс.
2. Усовершенствуйте программу из упражнения 1 так, чтобы она могла сообщить пользователю, имеется ли запрашиваемое значение в векторе.
3. Джек продает кувшины на eBay. Чтобы помочь ему с упаковкой и отгрузкой, напишите программу, в которой он может вводить размеры каждого из изделий, сохранять их в векторе и выводить на экран.
4. Напишите приложение, которое инициализирует дек тремя строками. Их следует вывести на экран с помощью шаблонной функции, которая может работать с деками любого вида. Ваше приложение должно продемонстрировать инициализацию списком из стандарта C++11 и использование оператора `""s` из стандарта C++14.

ЗАНЯТИЕ 18

Классы `list` и `forward_list`

Стандартная библиотека шаблонов (STL) предоставляет программистам *двухсвязный список* (doubly linked list) в форме шаблона класса `std::list`. Основное преимущество связанного списка — в быстрой вставке и извлечении элементов за константное время. Начиная со стандарта C++11, вы можете использовать также *односвязный список* (singly linked list) в виде шаблонного класса `std::forward_list`, проход по которому обеспечивается только в одном направлении.

На этом занятии...

- Как создать экземпляр классов `list` и `forward_list`
- Как использовать списки библиотеки STL, включая вставку и удаление
- Как обратиться и отсортировать элементы

Характеристики класса `std::list`

Связанный список (linked list) — это коллекция узлов (node), каждый из которых, кроме представляющего интерес значения или объекта, содержит указатели на следующий и предыдущий узлы, как показано на рис. 18.1.

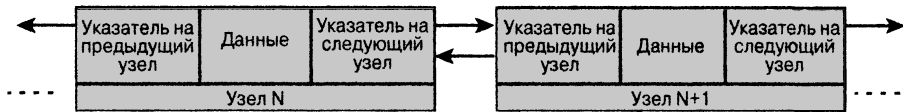


РИС. 18.1. Визуальное представление двухсвязного списка

Реализация класса `list` библиотеки STL обеспечивает константную продолжительность вставки в начало, конец и середину списка.

СОВЕТ

Чтобы использовать класс `std::list`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <list>
```

Основные операции со списком

Чтобы использовать класс `list` библиотеки STL, включите в код заголовочный файл `<list>`. Шаблон класса `list`, находящийся в пространстве имен `std`, является обобщенной реализацией, которая должна быть инстанцирована, прежде чем вы сможете использовать любую из его функций-членов.

Инстанцирование класса `std::list`

При инстанцировании шаблона класса `std::list` нужно указать тип объектов, которые предполагается хранить в списке. Так что инициализация списка должна иметь примерно следующий вид:

```
std::list<int>    linkInts;    // Список целых чисел
std::list<float> listFloats;   // Список чисел типа float
std::list<Tuna>  listTunas;    // Список объектов типа Tuna
```

Для объявления итератора, указывающего на элементы в списке, используется следующий код:

```
std::list<int>::const_iterator elementInList;
```

Если необходим итератор, допускающий изменение значений или вызов неконстантных функций, используйте ключевое слово `iterator` вместо `const_iterator`.

Класс `std::list` имеет несколько перегруженных конструкторов, так что его экземпляр можно создать, в том числе указав начальное количество элементов и их исходные значения, как показано в листинге 18.1.

ЛИСТИНГ 18.1. Способы создания экземпляров класса `std::list`

```
0: #include <list>
1: #include <vector>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     // Создание пустого списка
8:     list<int> linkInts;
9:
10:    // Создание списка с 10 целыми числами
11:    list<int> listWith10Integers(10);
12:
13:    // Создание списка с 4 целыми числами, равными 99
14:    list<int> listWith4IntegerEach99(10, 99);
15:
16:    // Создание точной копии существующего списка
17:    list<int> listCopyAnother(listWith4IntegerEach99);
18:
19:    // Вектор из 10 целых чисел со значением 2017 каждый
20:    vector<int> vecIntegers(10, 2017);
21:
22:    // Создание списка со значениями из другого контейнера
23:    list<int> listContainsCopyOfAnother(vecIntegers.cbegin(),
24:                                       vecIntegers.cend());
25:
26:    return 0;
27: }
```

Анализ

Эта программа не генерирует никакого вывода, она лишь демонстрирует применение различных перегруженных конструкторов для создания списка целых чисел. В строке 8 создается пустой список, а в строке 11 — список, содержащий 10 целых чисел. В строке 14 создается список `listWith4IntegerEach99`, содержащий 4 целых числа, каждое из которых инициализировано значением 99. Строка 17 демонстрирует создание списка, являющегося точной копией другого. Строки 20–24 весьма любопытны! Вы создаете экземпляр вектора, который содержит 10 целых чисел, со значением 2017, а затем, в строке 23, создается экземпляр списка, содержащий элементы, скопированные из вектора с использованием константных итераторов, возвращенных методами `vector::cbegin()` и `vector::cend()` (введены стандартом C++11). Листинг 18.1 демонстрирует также то, как итераторы позволяют отделить реализацию одного контейнера от другого, позволяя при этом использовать их обобщенные функциональные возможности для инстанцирования списка с использованием значений из вектора (строки 23 и 24).

ПРИМЕЧАНИЕ

Сравнивая листинг 18.1 с листингом 17.1 из занятия 17, “Классы динамических массивов библиотеки STL”, можно заметить общую схему и подобие способов инстанцирования контейнеров различных типов. Чем чаще вы будете использовать контейнеры STL, тем быстрее убедитесь в простоте и единообразии их использования.

Вставка элементов в начало и в конец списка

Подобно двухсторонней очереди, вставка в начало (или вершину, в зависимости от вашей точки зрения) осуществляется с использованием метода `push_front()`, а вставка в конец — с использованием метода `push_back()`. Эти два метода получают один входной параметр, содержащий вставляемое значение:

```
linkInts.push_back(-1);
linkInts.push_front(2001);
```

В листинге 18.2 показан результат использования этих двух методов для вставки значений в список целых чисел.

ЛИСТИНГ 18.2. Вставка элементов в список с использованием методов `push_front()` и `push_back()`

```
0: #include <list>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& container)
6: {
7:     for(auto element = container.cbegin();
8:         element != container.cend();
9:         ++element )
10:         cout << *element << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main()
16: {
17:     std::list <int> linkInts { -101, 42 };
18:
19:     linkInts.push_front(10);
20:     linkInts.push_front(2011);
21:     linkInts.push_back(-1);
22:     linkInts.push_back(9999);
23:
24:     DisplayContents(linkInts);
25:
26:     return 0;
27: }
```

Результат

```
2011 10 -101 42 -1 9999
```

Анализ

В строке 17 показано инстанцирование класса `list` для типа `int` с использованием синтаксиса инициализации списком `{...}` из стандарта C++11, который позволяет создать список `linkInts` со значениями `-101` и `42` в нем. Строки 19–22 демонстрируют применение методов `push_front()` и `push_back()`. Значение, переданное как аргумент методу `push_front()`, вставляется в первую позицию списка, а переданное методу `push_back()` — в последнюю. Вывод отображает содержимое списка с помощью шаблонной функции `DisplayContents()`, демонстрируя, что элементы хранятся не в порядке их вставки в список.

ПРИМЕЧАНИЕ

Функция `DisplayContents()` в строках 4–13 листинга 18.2 является более обобщенной версией метода `DisplayVector()` из листинга 17.6 (обратите внимание на измененный список параметров). Хотя предыдущий шаблон, работающий только с векторами, обобщал тип хранимых в нем элементов, новый шаблон обеспечивает полностью обобщенную версию, работающую с контейнерами разных типов.

Вы можете вызвать версию метода `DisplayContents()` из листинга 18.2 для вектора, списка или дека в качестве аргумента, и он будет прекрасно работать.

Вставка в середину списка

Контейнер `std::list` характеризуется возможностью вставки элементов в середину коллекции за константное время. Для этого используется его функция-член `insert()`.

Функция `list::insert()` доступна в трех версиях.

■ Вариант 1

```
iterator insert(iterator pos, const T& x)
```

Этот вариант функции `insert()` получает в качестве первого параметра позицию вставки, и вставляемое значение в качестве второго. Функция возвращает итератор, указывающий на элемент, только что вставленный в список.

■ Вариант 2

```
void insert(iterator pos, size_type n, const T& x)
```

Эта функция получает позицию вставки в качестве первого параметра, вставляемое значение в качестве последнего параметра и количество вставляемых значений в параметре `n`.

■ Вариант 3

```
template <class InputIterator>
void insert(iterator pos, InputIterator f, InputIterator l)
```

Эта перегруженная версия представляет собой шаблон функции, который получает, кроме позиции вставки, два входных итератора, указывающих границы вставляемой в список коллекции. Обратите внимание: входной тип, InputIterator, является параметром типа шаблона, а потому второй и третий параметры функции могут указывать границы любой коллекции значений, будь то массив, вектор или другой список.

В листинге 18.3 показано применение этих перегруженных вариантов функции `list::insert()`.

ЛИСТИНГ 18.3. Способы вставки элементов в список

```
0: #include <list>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& container)
6: {
7:     for(auto element = container.cbegin();
8:         element != container.cend();
9:         ++element )
10:         cout << *element << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main()
16: {
17:     list <int> linkInts1;
18:
19:     // Вставка элементов в начало...
20:     linkInts1.insert(linkInts1.begin(), 2);
21:     linkInts1.insert(linkInts1.begin(), 1);
22:
23:     // Вставка элементов в конец...
24:     linkInts1.insert(linkInts1.end(), 3);
25:
26:     cout << "Список 1 после вставки элементов:" << endl;
27:     DisplayContents(linkInts1);
28:
29:     list <int> linkInts2;
30:
31:     // Вставка 4 элементов с одинаковым значением 0...
32:     linkInts2.insert(linkInts2.begin(), 4, 0);
33:
34:     cout << "Список 2 после вставки ";
35:     cout << linkInts2.size() << "' элементов:" << endl;
36:     DisplayContents(linkInts2);
37:
```

```
38:     list <int> linkInts3;
39:
40:     // Вставка элементов из другого списка в начало...
41:     linkInts3.insert(linkInts3.begin(),
42:                     linkInts1.begin(), linkInts1.end());
43:
44:     cout << "Список 3 после вставки содержимого ";
45:     cout << "списка 1 в начало:" << endl;
46:     DisplayContents(linkInts3);
47:
48:     // Вставка элементов из другого списка в конец...
49:     linkInts3.insert(linkInts3.end(),
50:                     linkInts2.begin(), linkInts2.end());
51:
52:     cout << "Список 3 после вставки содержимого ";
53:     cout << "списка 2 в конец:" << endl;
54:     DisplayContents(linkInts3);
55:
56:     return 0;
57: }
```

Результат

Список 1 после вставки элементов:

1 2 3

Список 2 после вставки '4' элементов:

0 0 0 0

Список 3 после вставки содержимого списка 1 в начало:

1 2 3

Список 3 после вставки содержимого списка 2 в конец:

1 2 3 0 0 0 0

Анализ

Функции-члены `begin()` и `end()` возвращают итераторы, указывающие на начало и конец списка соответственно. В общем случае это справедливо для всех контейнеров STL, включая `std::list`. Функция `list::insert()` получает итератор, указывающий позицию, перед которой должны быть вставлены элементы. Используемый в строке 24 итератор, возвращаемый функцией `end()`, указывает на позицию после последнего элемента в списке. Поэтому эта строка вставляет целочисленное значение 3 перед концом, как последнее значение. В строке 32 список инициализируется четырьмя помещаемыми в начало списка элементами со значением 0. Строки 41 и 42 демонстрируют применение функции `list::insert()` для вставки элементов из одного списка в конец другого. Хотя в этом примере в список вставляются значения из другого списка, с тем же успехом может быть вставлен диапазон значений, например, из вектора, определяемый теми же методами `begin()` и `end()`, как в листинге 18.1, или из обычного статического массива.

Удаление элементов из списка

Функция `erase()` класса `list` имеет два перегруженных варианта: удаляющий один элемент, на который указывает переданный функции итератор, и удаляющий диапазон элементов из списка. В действии функцию `list::erase()` можно увидеть в листинге 18.4, где из списка удаляется как один элемент, так и целый диапазон элементов.

ЛИСТИНГ 18.4. Удаление элементов из списка

```
0: #include <list>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& container)
6: {
7:     for(auto element = container.cbegin();
8:         element != container.cend();
9:         ++element )
10:        cout << *element << ' ';
11:
12:        cout << endl;
13: }
14:
15: int main()
16: {
17:     std::list<int> linkInts{ 4, 3, 5, -1, 2017 };
18:
19:     // Сохраняем итератор, полученный от функции insert()
20:     auto val2 = linkInts.insert(linkInts.begin(), 2);
21:
22:     cout << "Исходное содержимое списка:" << endl;
23:     DisplayContents(linkInts);
24:
25:     cout << "После удаления элемента '" << *val2 << "':" << endl;
26:     linkInts.erase(val2);
27:     DisplayContents(linkInts);
28:
29:     linkInts.erase(linkInts.begin(), linkInts.end());
30:     cout << "Количество элементов после удаления диапазона: ";
31:     cout << linkInts.size() << endl;
32:
33:     return 0;
34: }
```

Результат

Исходное содержимое списка:

```
2 4 3 5 -1 2017
```

После удаления элемента '2':

```
4 3 5 -1 2017
```

Количество элементов после удаления диапазона: 0

Анализ

В строке 20 использованная для вставки значения функция `insert()` возвращает итератор, указывающий на вставленный элемент. В нашем случае этот итератор указывает на элемент со значением 2 и он сохраняется в переменной `val2`, а позже используется в строке 26 при вызове функции `erase()` для удаления этого элемента из списка. В строке 29 продемонстрировано применение метода `erase()` для удаления диапазона элементов. Мы удаляем все элементы от `begin()` до `end()`, т.е. фактически из списка удаляются все элементы.

СОВЕТ

Простейший способ удаления всех элементов списка состоит в вызове функции-члена `list::clear()`. Поэтому строку 29 листинга 18.4 можно записать кратко как

```
linkInts.clear();
```

ПРИМЕЧАНИЕ

Строка 31 листинга 18.4 демонстрирует, что количество элементов списка может быть определено с помощью метода `size()` класса `std::list`, как и для вектора. Этот метод применим ко всем контейнерным классам библиотеки STL.

Обращение списка и сортировка его элементов

У списка есть одна особенность: итераторы, указывающие на элементы списка, остаются корректными несмотря на перестановку существующих элементов или вставку новых. Для обеспечения этой особенности класс `list` предоставляет собственные методы `sort()` и `reverse()`, хотя библиотека STL предлагает обобщенные алгоритмы, способные работать с классом `list` в том числе. Эти алгоритмы в качестве членов класса гарантируют, что итераторы, указывающие на элементы списка, останутся корректными при изменении относительной позиции элементов.

Обращение элементов списка с помощью list::reverse()

Функция-член reverse() класса list не получает никаких параметров и обращает порядок содержимого списка:

```
linkInts.reverse(); // Обратить порядок элементов
```

Применение метода reverse() приведено в листинге 18.5.

ЛИСТИНГ 18.5. Обращение элементов списка

```
0: #include <list>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& container)
6: {
7:     for(auto element = container.cbegin();
8:         element != container.cend();
9:         ++element )
10:        cout << *element << ' ';
11:
12:        cout << endl;
13: }
14:
15: int main()
16: {
17:     std::list<int> linkInts{ 0, 1, 2, 3, 4, 5 };
18:
19:     cout << "Исходное содержимое списка:" << endl;
20:     DisplayContents(linkInts);
21:
22:     linkInts.reverse();
23:
24:     cout << "Список после вызова reverse():" << endl;
25:     DisplayContents(linkInts);
26:
27:     return 0;
28: }
```

Результат

```
Исходное содержимое списка:
0 1 2 3 4 5
Список после вызова reverse():
5 4 3 2 1 0
```

Анализ

Как показано в строке 22, метод `reverse()` просто обращает порядок элементов списка. Это простой вызов без параметров, гарантирующий, что указывающие на элементы итераторы, если они были сохранены, останутся корректными.

Сортировка элементов

Функция-член `sort()` класса `list` может быть вызвана без параметров:

```
linkInts.sort(); // Сортировка в порядке возрастания
```

Другая версия функции позволяет определять собственный критерий сортировки с помощью бинарного предиката, передаваемого как параметр:

```
bool SortPredicate_Descending(const int& lsh, const int& rsh)
{
    // Определение критерия сортировки:
    // возврат true в случае верного упорядочения lsh и rsh
    return (lsh > rsh);
}
// Использование предиката для сортировки списка:
linkInts.sort(SortPredicate_Descending);
```

Эти два варианта показаны в листинге 18.6.

ЛИСТИНГ 18.6. Сортировка списка целых чисел по возрастанию и по убыванию с использованием метода `list::sort()`

```
0: #include <list>
1: #include <iostream>
2: using namespace std;
3:
4: bool SortPredicate_Descending(const int& lhs, const int& rhs)
5: {
6:     // Определение критерия сортировки
7:     return (lhs > rhs);
8: }
9:
10: template <typename T>
11: void DisplayContents(const T& container)
12: {
13:     for(auto element = container.cbegin();
14:         element != container.cend();
15:         ++element )
16:         cout << *element << ' ';
17:
18:     cout << endl;
19: }
20:
21: int main()
```

```
22: {  
23:     list<int> linkInts{ 0, -1, 2011, 444, -5 };  
24:  
25:     cout << "Исходное содержимое списка:" << endl;  
26:     DisplayContents(linkInts);  
27:  
28:     linkInts.sort();  
29:  
30:     cout << "Содержимое после sort():" << endl;  
31:     DisplayContents(linkInts);  
32:  
33:     linkInts.sort(SortPredicate_Descending);  
34:     cout << "Содержимое после sort() с предикатом:" << endl;  
35:     DisplayContents(linkInts);  
36:  
37:     return 0;  
38: }
```

Результат

```
Исходное содержимое списка:  
0 -1 2011 444 -5  
Содержимое после sort():  
-5 -1 0 444 2011  
Содержимое после sort() с предикатом:  
2011 444 0 -1 -5
```

Анализ

Данный пример демонстрирует сортировку списка целых чисел. Строка 28 показывает применение функции `sort()` без параметров, что по умолчанию означает сортировку элементов в порядке возрастания; для сравнения целых чисел используется оператор `<` (который в случае целых чисел реализуется компилятором). Но если разработчик захочет переопределить это стандартное поведение, он должен снабдить функцию сортировки бинарным предикатом, как показано в строке 33. Функция `SortPredicate_Descending()`, определенная в строках 4–8, представляет собой такой бинарный предикат, который позволяет функции `list::sort()` выяснить, меньше ли первый из элементов второго. В противном случае он меняет их местами. Другими словами, вы указываете списку, как именно следует интерпретировать понятие “меньше” (в данном случае для этого первый параметр должен быть больше второго). Предикат возвращает значение `true`, только если первое значение больше второго. Таким образом, использующая предикат функция `sort()` интерпретирует первое значение (`lsh`) как логически меньшее второго (`rsh`), только если числовое значение первого больше второго. На основе этой интерпретации, согласно определенному предикатом критерию, она меняет позиции элементов.

Сортировка и удаление элементов из списка, который содержит объекты класса

Что делать в случае, если у вас есть список объектов некоторого класса, а не таких простых встроенных типов, как `int`? Скажем, список записей адресной книги, в которой каждая запись — объект класса, содержащий имя, адрес и т.д. Как удостовериться, что этот список будет отсортирован по имени?

Решение может иметь два варианта.

- Реализуйте в пределах класса, объекты которого содержатся в списке, оператор `<`.
- Предоставьте соответствующий *бинарный предикат* (binary predicate) — функцию, получающую на входе два значения и возвращающую логическое значение, указывающее, меньше ли первое значение второго.

Реальные приложения, использующие контейнеры STL, редко хранят целые числа; как правило, это пользовательские типы, такие как классы или структуры. Листинг 18.7 демонстрирует пример списка контактов. На первый взгляд, он кажется довольно длинным, но содержит очень простой код.

ЛИСТИНГ 18.7. Список объектов класса: создание списка контактов

```
0: #include <list>
1: #include <string>
2: #include <iostream>
3: using namespace std;
4:
5: template <typename T>
6: void displayAsContents(const T& container)
7: {
8:     for(auto element = container.cbegin();
9:         element != container.cend();
10:         ++element )
11:         cout << *element << endl;
12:
13:     cout << endl;
14: }
15:
16: struct ContactItem
17: {
18:     string name;
19:     string phone;
20:     string displayAs;
21:
22:     ContactItem(const string& conName, const string & conNum)
23:     {
24:         name = conName;
25:         phone = conNum;
26:         displayAs = (name + ": " + phone);
27:     }
```

```

28:
29:     // Используется в list::remove()
30:     bool operator ==(const ContactItem& itemToCompare) const
31:     {
32:         return (itemToCompare.name == this->name);
33:     }
34:
35:     // Используется в list::sort() без параметров
36:     bool operator <(const ContactItem& itemToCompare) const
37:     {
38:         return (this->name < itemToCompare.name);
39:     }
40:
41:     // Используется в displayAsContents для вывода в cout
42:     operator const char*() const
43:     {
44:         return displayAs.c_str();
45:     }
46: };
47:
48: bool SortOnphoneNumber(const ContactItem& item1,
49:                        const ContactItem& item2)
50: {
51:     return (item1.phone < item2.phone);
52: }
53:
54: int main()
55: {
56:     list <ContactItem> contacts;
57:     contacts.push_back(ContactItem("Jack Welsch","+17889879879"));
58:     contacts.push_back(ContactItem("Bill Gates","+197789787998"));
59:     contacts.push_back(ContactItem("Angi Merkel","+49234565466"));
60:     contacts.push_back(ContactItem("Dim Medvedev","+766454564797"));
61:     contacts.push_back(ContactItem("Ben Affleck","+1745641314"));
62:     contacts.push_back(ContactItem("Dan Craig","+44123641976"));
63:
64:     cout << "Список в исходном порядке: " << endl;
65:     displayAsContents(contacts);
66:
67:     contacts.sort();
68:     cout << "Сортировка с помощью оператора <:" << endl;
69:     displayAsContents(contacts);
70:
71:     contacts.sort(SortOnphoneNumber);
72:     cout << "Сортировка с помощью предиката:" << endl;
73:     displayAsContents(contacts);
74:
75:     cout << "Убираем Медведева из списка: " << endl;
76:     contacts.remove(ContactItem("Dim Medvedev", ""));
77:     displayAsContents(contacts);

```

```
78:
79:     return 0;
80: }
```

Результат

Список в исходном порядке:

```
Jack Welsch: +17889879879
Bill Gates: +197789787998
Angi Merkel: +49234565466
Dim Medvedev: +766454564797
Ben Affleck: +1745641314
Dan Craig: +44123641976
```

Сортировка с помощью оператора <:

```
Angi Merkel: +49234565466
Ben Affleck: +1745641314
Bill Gates: +197789787998
Dan Craig: +44123641976
Dim Medvedev: +766454564797
Jack Welsch: +17889879879
```

Сортировка с помощью предиката:

```
Ben Affleck: +1745641314
Jack Welsch: +17889879879
Bill Gates: +197789787998
Dan Craig: +44123641976
Angi Merkel: +49234565466
Dim Medvedev: +766454564797
```

Убираем Медведева из списка:

```
Ben Affleck: +1745641314
Jack Welsch: +17889879879
Bill Gates: +197789787998
Dan Craig: +44123641976
Angi Merkel: +49234565466
```

Анализ

Сначала сосредоточимся на строках 54–80 функции `main()`. В строке 56 создается экземпляр списка элементов книги адресов типа `ContactItem`. В строках 57–62 этот список заполняется вымышленными номерами телефонов и именами нескольких знаменитостей и политических деятелей. В строке 65 осуществляется вывод этого списка на экран. Функция `list::sort` без предиката используется в строке 67. При отсутствии предиката эта функция сортировки ищет оператор `operator<` в классе `ContactItem`, где он был определен в строках 36–39. Оператор `ContactItem::operator<` позволяет контейнеру `list::sort` сортировать свои элементы в алфавитном порядке хранимых имен (а не номеров телефонов или как-то еще). Чтобы отсортировать тот же список по

номерам телефонов, используется функция `list::sort()` с переданным ей в качестве аргумента бинарным предикатом `SortOnphoneNumber()` (строка 71). Эта функция, реализованная в строках 48–52, гарантирует, что входные аргументы типа `ContactItem` сравниваются один с другим по номеру телефона, а не по имени. Это позволяет функции `list::sort()` сортировать список знаменитостей на основе их номеров телефонов, о чем и свидетельствует вывод на экран. И наконец в строке 76 используется метод `list::remove()` для удаления конкретного контакта из списка. Имя контакта передается как параметр. Метод `list::remove()` сравнивает этот объект с другими элементами списка с использованием оператора `ContactItem::operator==`, реализованного в строках 30–33. Этот оператор возвращает значение `true`, если имена совпадают, предоставляя методу `list::remove()` критерий для принятия решения о соответствии.

Данный пример демонстрирует не только применение шаблона связанного списка библиотеки STL для создания списка объектов любого типа, но и важность операторов и предикатов.

Шаблон класса `std::forward_list`

Начиная со стандарта C++11 в языке появилась возможность использовать класс `std::forward_list` вместо двухсвязного списка класса `std::list`. Класс `std::forward_list` предоставляет односвязный список, допускающий перебор только в одном направлении (рис. 18.2).

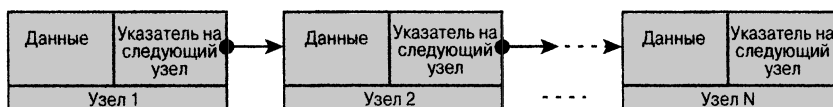


РИС. 18.2. Визуальное представление односвязного списка

СОВЕТ

Для использования `std::forward_list` в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <forward_list>
```

Использование класса `forward_list` очень похоже на использование класса `list`, но перемещение итераторов в нем возможно только в одном направлении. Для вставки элементов есть функция `push_front()`, но нет функции `push_back()`. Конечно, вы всегда можете использовать функцию `insert()` и ее перегруженные версии для вставки элементов в указанную позицию.

В листинге 18.8 показаны некоторые функции класса `forward_list`.

ЛИСТИНГ 18.8. Простые операции вставки и извлечения из односвязного списка

```
0: #include <forward_list>
1: #include <iostream>
2: using namespace std;
```

```

3:
4: template <typename T>
5: void DisplayContents(const T& container)
6: {
7:     for(auto element = container.cbegin();
8:         element != container.cend();
9:         ++element)
10:        cout << *element << ' ';
11:
12:    cout << endl;
13: }
14:
15: int main()
16: {
17:     forward_list<int> flistIntegers{ 3, 4, 2, 2, 0 };
18:     flistIntegers.push_front(1);
19:
20:     cout << "Содержимое forward_list:" << endl;
21:     DisplayContents(flistIntegers);
22:
23:     flistIntegers.remove(2);
24:     flistIntegers.sort();
25:     cout << "Содержимое после удаления и сортировки:" << endl;
26:     DisplayContents(flistIntegers);
27:
28:     return 0;
29: }

```

Результат

```

Содержимое forward_list:
1 3 4 2 2 0
Содержимое после удаления и сортировки:
0 1 3 4

```

Анализ

Как видно из приведенного примера, функционально класс `forward_list` очень похож на класс `list`. Поскольку класс `forward_list` не поддерживает двунаправленный перебор, к итератору можно применять оператор `operator++`, но не `operator--`. Этот пример демонстрирует применение функции `remove(2)` для удаления всех элементов со значением 2 (строка 23). Строка 24 демонстрирует функцию `sort()` для сортировки с использованием предиката по умолчанию `std::less<T>`.

Преимущество класса `forward_list` заключается в том, что это односвязный список, он использует несколько меньше памяти, чем класс `list` (поскольку элемент содержит ссылку только на следующий элемент).

РЕКОМЕНДУЕТСЯ

Используйте класс `std::list` вместо класса `std::vector`, когда необходимы частое удаление и вставка элементов, особенно в середине, поскольку вектор должен изменять размеры своего внутреннего буфера для обеспечения семантики массива (что вызывает продолжительные операции копирования), а список только меняет значения указателей.

Помните, что методы `push_front()` и `push_back()` позволяют вставлять элементы в начало и в конец списка соответственно.

Помните о необходимости реализации операторов `<` и `==` в классе, объекты которого будут храниться в контейнере STL, таком как `list`, чтобы предоставить предикаты по умолчанию для сортировки и удаления.

Помните, что вы всегда можете выяснить количество элементов в списке, используя метод `list::size()`, как и в любом другом контейнере STL.

Помните, что вы можете очистить список, используя метод `list::clear()`, как и в любом другом контейнере STL.

РЕКОМЕНДУЕТСЯ

Не используйте `std::list` при редких вставках и удалениях в конце и отсутствии вставки и удаления в середине; в этом случае лучше использовать классы `vector` и `deque`.

Не забывайте предоставить функцию предиката, если хотите использовать метод `sort()` или `remove()` класса `list` с критериями, отличными от принятых по умолчанию.

Резюме

На этом занятии мы рассмотрели свойства класса `list` и различные операции с ним. Теперь вам известны некоторые из наиболее полезных функций класса `list`, и вы можете создать список объектов любого типа.

Вопросы и ответы

■ **Зачем класс `list` предоставляет такие функции-члены, как `sort()` и `remove()`?**

Класс `list` STL гарантирует, что указывающие на его элементы итераторы останутся корректными независимо от позиций элементов в списке. Хотя обобщенные алгоритмы STL могут работать с классом `list`, функции-члены обеспечивают способность итераторов списка указывать на одни и те же элементы даже после сортировки списка, а также более высокую эффективность операций.

- Вы используете список, элементы которого имеют тип класса `Animal`. Какие операторы должен определять класс `Animal`, чтобы функции-члены списка были в состоянии работать с ним правильно?

Вы должны предоставить операторы сравнения `==` и `<` для любого класса, объекты которых планируется хранить в контейнерах STL.

- Как можно заменить ключевое слово `auto` явным описанием типа в следующей строке:

```
list<int> linkInts(10); // Список из 10 целых чисел
auto firstElement = linkInts.begin();
```

Ключевое слово `auto` можно заменить следующим явным описанием типа:

```
list<int> linkInts(10); // Список из 10 целых чисел
list<int>::iterator firstElement = linkInts.begin();
```

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Будет ли происходить потеря производительности при вставке элементов в середину списка по сравнению со вставкой в его начало или в конец?
2. Два итератора указывают на два элемента в `list` STL, затем между ними вставляется новый элемент. Сделает ли вставка эти итераторы некорректными?
3. Как очистить содержимое списка?
4. Можно ли вставить в список несколько элементов?

Упражнения

1. Напишите короткую программу, которая получает введенные пользователем числа и вставляет их в начало списка.
2. Используя эту программу, продемонстрируйте, что итератор, указывающий на элемент в списке, продолжает оставаться корректным, несмотря на вставку до или после этого элемента других элементов, изменяющую относительную позицию элемента в списке.
3. Напишите программу, которая вставляет содержимое вектора в список, используя функцию вставки класса `list`.
4. Напишите программу, обращающую список строк.

ЗАНЯТИЕ 19

Классы множеств STL

Стандартная библиотека шаблонов (STL) предоставляет разработчикам классы контейнеров, обеспечивающих возможность частого и быстрого поиска. Классы `std::set` и `std::multiset` используются для хранения отсортированного множества элементов и предоставляют возможность поиска элементов с логарифмической сложностью. Их неупорядоченные аналоги обеспечивают возможность вставки и поиска за константное время.

На этом занятии...

- Чем могут быть полезны контейнеры `set`, `multiset`, `unordered_set` и `unordered_multiset` STL
- Вставка, извлечение и поиск элементов
- Преимущества и недостатки использования этих контейнеров

Введение в классы множеств STL

Контейнеры `set` и `multiset` обеспечивают быстрый поиск ключей в содержащем их контейнере, т.е. ключи представляют собой значения, хранящиеся в одномерном контейнере. Различие между *множеством* (`set`) и *мультимножеством* (`multiset`) в том, что последнее допускает наличие нескольких одинаковых элементов, тогда как первое позволяет хранить только уникальные значения.

Рис. 19.1 чисто иллюстративный, показывающий, что множество имен содержит только уникальные имена, тогда как мультимножество допускает дубликаты. Будучи шаблонными классами, контейнеры STL могут хранить объекты любых типов, включая строки, целые числа, структуры или классы.

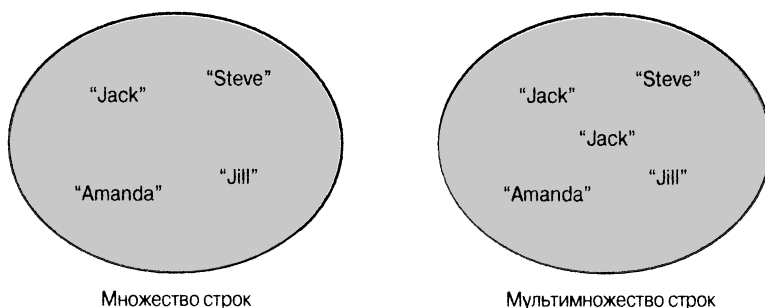


РИС. 19.1. Визуальное представление множества и мультимножества имен

Чтобы облегчить быстрый поиск, классы `set` и `multiset` внутренне реализованы в виде сбалансированного бинарного дерева. Это означает, что для ускорения поиска при вставке элементы сортируются. Это также означает, что, в отличие от вектора, в котором элемент в определенной позиции может быть заменен другим, элемент множества в определенной позиции не может быть заменен новым элементом с другим значением. Дело в том, что класс `set` должен разместить его в другой области внутреннего дерева в соответствии со значением.

СОВЕТ

Чтобы использовать классы `std::set` и `std::multiset`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <set>
```

Фундаментальные операции с классами `set` и `multiset`

Прежде чем вы сможете использовать любую из функций-членов шаблонных классов `set` и `multiset`, необходимо их инстанцировать.

Инстанцирование объекта `std::set`

Для создания экземпляра множества или мультимножества следует специализировать шаблон класса `std::set` или `std::multiset` для конкретного типа:

```
std::set<int> setInts;  
std::multiset<int> msetInts;
```

Чтобы определить множество или мультимножество, содержащее объекты класса `Tuna`, используется следующий код:

```
std::set<Tuna> setTuna;  
std::multiset<Tuna> msetTuna;
```

Чтобы объявить итератор, указывающий на элементы множества или мультимножества, используется следующий код:

```
std::set<int>::const_iterator element;  
std::multiset<int>::const_iterator element;
```

Если необходим итератор для изменения значений или вызова неконстантных функций, вместо `const_iterator` используйте ключевое слово `iterator`.

Поскольку контейнеры `set` и `multiset` сортируют элементы при вставке, они используют заданный по умолчанию предикат `std::less<>`, если только вы не предоставляете иной критерий сортировки. Это гарантирует, что ваше множество будет содержать элементы отсортированными в порядке возрастания.

Вы создаете бинарный предикат сортировки, определяя класс с оператором `operator()`, который получает два содержащихся во множестве значения и возвращает значение `true` в зависимости от выполнения вашего критерия. Вот один из таких предикатов для сортировки в порядке убывания:

```
// Используется как параметр шаблона при инстанцировании  
template <typename T>  
struct SortDescending  
{  
    bool operator()(const T& lhs, const T& rhs) const  
    {  
        return (lhs > rhs);  
    }  
};
```

Затем этот предикат используется при создании экземпляра множества или мультимножества:

```
// Множество и мультимножество целых чисел,  
// использующие предикат сортировки  
set<int, SortDescending<int>> setInts;  
multiset<int, SortDescending<int>> msetInts;
```

Кроме этих вариантов, вы всегда можете создать множество или мультимножество как полную или частичную копию содержимого другого контейнера (листинг 19.1).

ЛИСТИНГ 19.1. Способы инстанцирования множества и мультимножества

```
0: #include <set>
1:
2: // Используется как параметр при инстанцировании
3: template <typename T>
4: struct SortDescending
5: {
6:     bool operator()(const T& lhs, const T& rhs) const
7:     {
8:         return (lhs > rhs);
9:     }
10: };
11:
12: int main()
13: {
14:     using namespace std;
15:
16:     // Множество целых чисел с сортировкой по умолчанию
17:     set<int> setInts1;
18:     multiset<int> msetInts1;
19:
20:     // Инстанцирование с пользовательским предикатом сортировки
21:     set<int, SortDescending<int>> setInts2;
22:     multiset<int, SortDescending<int>> msetInts2;
23:
24:     // Создание множества из другого контейнера или его части
25:     set<int> setInts3(setInts1);
26:     multiset<int> msetInts3(setInts1.cbegin(), setInts1.cend());
27:
28:     return 0;
29: }
```

Анализ

Эта программа не имеет вывода, она просто демонстрирует применение различных способов создания экземпляров множества и мультимножества, специализированных для хранения элементов типа `int`. В строках 17 и 18 показана самая простая форма инстанцирования, в которой проигнорированы все параметры шаблона, кроме типа, что подразумевает использование предиката сортировки `std::less<T>`, заданного по умолчанию при реализации структуры (или класса). Если вы хотите переопределить сортировку по умолчанию, определите предикат, как это сделано в строках 3–10, и используйте его в функции `main()`, как в строках 21 и 22. Приведенный здесь предикат обеспечивает сортировку по убыванию (сортировка по умолчанию — по возрастанию). И наконец, строки 25 и 26 демонстрируют способы создания экземпляра множества как копии другого экземпляра мультимножества из диапазона значений, взятых из множества (но это может быть вектор, список или любой другой контейнер STL, возвращающий итераторы, которые описывают границы с помощью методов `cbegin()` и `cend()`).

Вставка элементов в множество и мультимножество

Большинство функций классов set и multiset работают одинаково. Они получают подобные параметры и возвращают значения одинаковых типов. Например, для вставки элементов в контейнеры обоих видов может быть использована функция insert(), получающая вставляемое значение или диапазон значений из другого контейнера:

```
setInts.insert(-1);
msetInts.insert(setInts.begin(), setInts.end());
```

В листинге 19.2 показана вставка элементов в эти контейнеры.

ЛИСТИНГ 19.2. Вставка элементов в множество и мультимножество библиотеки STL

```
0: #include <set>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& container)
6: {
7:     for(auto element = container.cbegin();
8:         element != container.cend();
9:         ++element)
10:        cout << *element << ' ';
11:
12:        cout << endl;
13: }
14:
15: int main()
16: {
17:     set<int> setInts{ 202, 151, -999, -1 };
18:     setInts.insert(-1); // Дубликат
19:     cout << "Содержимое множества: " << endl;
20:     DisplayContents(setInts);
21:
22:     multiset<int> msetInts;
23:     msetInts.insert(setInts.begin(), setInts.end());
24:     msetInts.insert(-1); // Дубликат
25:
26:     cout << "Содержимое мультимножества: " << endl;
27:     DisplayContents(msetInts);
28:
29:     cout << "Количество экземпляров '-1' в мультимножестве: ";
30:     cout << msetInts.count(-1) << " " << endl;
31:
32:     return 0;
33: }
```

Результат

Содержимое множества:

-999 -1 151 202

Содержимое мультимножества:

-999 -1 -1 151 202

Количество экземпляров '-1' в мультимножестве: '2'

Анализ

В строках 4–13 содержится шаблон обобщенной функции `DisplayContents()` (этот шаблон вы уже видели на занятиях 17, “Классы динамических массивов библиотеки STL”, и 18, “Классы `list` и `forward_list`”), предназначенной для вывода содержимого контейнера на консоль или экран. Строки 17 и 22, как вы уже знаете, определяют объекты классов `set` и `multiset` соответственно, причем в первой строке используется синтаксис инициализации списком. В строках 18 и 24 выполняется вставка значения в множество с помощью функции-члена `insert()`. Строка 23 демонстрирует применение функции `insert()` для вставки содержимого множества в мультимножество (в данном случае — содержимого множества `setInts` в мультимножество `msetInts`). В строке 24 к мультимножеству добавляется элемент, уже имеющийся в нем. Вывод демонстрирует, что мультимножество в состоянии содержать несколько одинаковых значений. Строка 30 демонстрирует функцию-член `multiset::count()`, которая возвращает количество элементов с указанным значением в мультимножестве.

СОВЕТ

Для выяснения количества элементов с одинаковым значением в мультимножестве используйте функцию-член `multiset::count()`.

Поиск элементов в множестве и мультимножестве

Ассоциативные контейнеры, такие как `set`, `multiset`, `map` и `multimap`, предоставляют функцию-член `find()`, позволяющую находить значение по ключу:

```
auto elementFound = setInts.find(-1);

// Проверяем, найден ли искомый элемент...
if (elementFound != setInts.end())
    cout << "Элемент " << *elementFound << " найден!" << endl;
else
    cout << "Элемент не найден!" << endl;
```

Использование функции `find()` продемонстрировано в листинге 19.3. В случае мультимножества, допускающего несколько элементов с одинаковым значением, эта функция находит первый элемент, соответствующий заданному ключу.

ЛИСТИНГ 19.3. Использование функции-члена find()

```
0: #include <set>
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     set<int> setInts{ 43, 78, -1, 124 };
7:
8:     // Вывод содержимого множества
9:     for(auto element = setInts.cbegin();
10:         element != setInts.cend();
11:         ++element )
12:         cout << *element << endl;
13:
14:     // Пытаемся найти элемент
15:     auto elementFound = setInts.find(-1);
16:
17:     // Проверка, найден ли...
18:     if (elementFound != setInts.end())
19:         cout << "Элемент " << *elementFound << " найден!" << endl;
20:     else
21:         cout << "Элемент не найден!" << endl;
22:
23:     // Поиск другого элемента
24:     auto anotherFind = setInts.find(12345);
25:
26:     // Проверка, найден ли...
27:     if (anotherFind != setInts.end())
28:         cout << "Элемент " << *anotherFind << " найден!" << endl;
29:     else
30:         cout << "Элемент 12345 не найден!" << endl;
31:
32:     return 0;
33: }
```

Результат

```
-1
43
78
124
Элемент -1 найден!
Элемент 12345 не найден!
```

Анализ

В строках 15–21 показано применение функции-члена `find()`, возвращающей итератор, сравнение которого с результатом вызова `end()`, как показано в строке 18, позволяет проверить, был ли найден искомый элемент. Если итератор корректен, можно обратиться к указываемому значению с использованием синтаксиса `*elementFound`.

ПРИМЕЧАНИЕ

Пример в листинге 19.3 работает и для мультимножества, т.е. если в строке 6 множество `set` заменить мультимножеством `multiset`, то код будет работать правильно. С учетом того, что мультимножество может хранить несколько значений с одним и тем же значением в соседних местоположениях, к ним можно обратиться с помощью итератора, возвращенного функцией `find()`, если перемещать его поочередно к следующему элементу (`count() - 1`) раз. Функция-член `count()` была продемонстрирована в листинге 19.2.

Удаление элементов из множества и мультимножества

Ассоциативные контейнеры, такие как `set`, `multiset`, `map` и `multimap`, предоставляют функцию-член `erase()`, позволяющую удалять значение с определенным значением ключа:

```
setObject.erase(key);
```

Другая версия функции `erase()` позволяет удалить определенный элемент, заданный указывающим на него итератором:

```
setObject.erase(element);
```

Вы можете удалить целый диапазон элементов множества или мультимножества, используя итераторы, определяющие границы диапазона:

```
setObject.erase(iLowerBound, iUpperBound);
```

Пример в листинге 19.4 демонстрирует использование метода `erase()` для удаления элементов из множества или мультимножества.

ЛИСТИНГ 19.4. Использование функции-члена `erase()` мультимножества

```
0: #include <set>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& Input)
6: {
7:     for(auto element = Input.cbegin();
```

```
8:         element != Input.cend();
9:         ++element )
10:         cout << *element << ' ';
11:
12:     cout << endl;
13: }
14:
15: typedef multiset<int> MSETINT;
16:
17: int main()
18: {
19:     MSETINT msetInts{ 43, 78, 78, -1, 124 };
20:
21:     cout << "multiset содержит " << msetInts.size() << " элементов:";
22:     DisplayContents(msetInts);
23:
24:     cout << "Введите удаляемое число:";
25:     int input = 0;
26:     cin >> input;
27:
28:     cout << "Удаляем " << msetInts.count(input);
29:     cout << " экземпляра значения " << input << endl;
30:
31:     msetInts.erase(input);
32:
33:     cout << "multiset содержит " << msetInts.size() << " элемента: ";
34:     DisplayContents(msetInts);
35:
36:     return 0;
37: }
```

Результат

```
multiset содержит 5 элементов:-1 43 78 78 124
Введите удаляемое число: 78
Удаляем 2 экземпляра значения 78
multiset содержит 3 элемента: -1 43 124
```

Анализ

Обратите внимание на использование в строке 15 ключевого слова `typedef`. Строка 28 демонстрирует применение функции `count()` для выяснения количества элементов с определенным значением. Фактическое удаление осуществляется в строке 31, где удаляются все элементы, которые соответствуют введенному пользователем числу.

СОВЕТ

Функция `erase()` имеет несколько перегруженных версий. Так, она может удалить все элементы множества. Ее можно вызвать для итератора, возвращенного, скажем, в результате поиска, чтобы удалить один элемент с найденным значением, как показано ниже.

```
MSETINT::iterator elementFound = msetInts.
find(nNumberToErase);
if (elementFound != msetInts.end())
    msetInts.erase(elementFound);
else
    cout << "Элемент не найден!" << endl;
```

Точно так же вы можете использовать функцию `erase()` для удаления из мультимножества диапазона значений:

```
MSETINT::iterator elementFound = msetInts.
find(valueToErase);
if (elementFound != msetInts.end())
    msetInts.erase(msetInts.begin(), elementFound);
```

Приведенный выше фрагмент удаляет все элементы от первого до элемента со значением `valueToErase`, не включая его. И множество, и мультимножество могут быть полностью освобождены от своего содержимого с помощью функции-члена `clear()`.

Теперь, после краткого обзора базовых функций множества и мультимножества, пришло время рассмотреть пример практического применения этого контейнерного класса. Пример в листинге 19.5 является самой простой реализацией телефонного справочника, позволяющего пользователю вставлять имена и номера телефонов, находить их, удалять и отображать все номера.

ЛИСТИНГ 19.5. Телефонный справочник, демонстрирующий возможности класса `set`

```
0: #include <set>
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: template <typename T>
6: void DisplayContents(const T& container)
7: {
8:     for(auto iElement = container.cbegin();
9:         iElement != container.cend();
10:         ++iElement )
11:         cout << *iElement << endl;
12:
13:     cout << endl;
14: }
15:
16: struct ContactItem
```

```
17: {
18:     string name;
19:     string phoneNum;
20:     string displayAs;
21:
22:     ContactItem(const string& nameInit, const string & phone)
23:     {
24:         name = nameInit;
25:         phoneNum = phone;
26:         displayAs = (name + ": " + phoneNum);
27:     }
28:
29:     // Используется в set::find() для поиска
30:     bool operator ==(const ContactItem& itemToCompare) const
31:     {
32:         return (itemToCompare.name == this->name);
33:     }
34:
35:     // Используется для сортировки
36:     bool operator <(const ContactItem& itemToCompare) const
37:     {
38:         return (this->name < itemToCompare.name);
39:     }
40:
41:     // Используется в DisplayContents для вывода в cout
42:     operator const char*() const
43:     {
44:         return displayAs.c_str();
45:     }
46: };
47:
48: int main()
49: {
50:     set<ContactItem> setCs;
51:     setCs.insert(ContactItem("Jack Welsch", "+1 7889 879 879"));
52:     setCs.insert(ContactItem("Bill Gates", "+1 97 7897 8799 8"));
53:     setCs.insert(ContactItem("Angi Merkel", "+49 23456 5466"));
54:     setCs.insert(ContactItem("Dim Medvedev", "+7 6645 4564 797"));
55:     setCs.insert(ContactItem("John Travolta", "91 234 4564 789"));
56:     setCs.insert(ContactItem("Ben Affleck", "+1 745 641 314"));
57:     DisplayContents(setCs);
58:
59:     cout << "Введите имя для удаления: ";
60:     string inputName;
61:     getline(cin, inputName);
62:
63:     auto contactFound = setCs.find(ContactItem(inputName, ""));
64:     if(contactFound != setCs.end())
65:     {
66:         setCs.erase(contactFound);
```



```
67:         cout << "После удаления " << inputName << endl;
68:         DisplayContents(setCs);
69:     }
70:     else
71:         cout << "Контакт не найден" << endl;
72:
73:     return 0;
74: }
```

Результат

```
Angi Merkel: +49 23456 5466
Ben Affleck: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Dim Medvedev: +7 6645 4564 797
Jack Welsch: +1 7889 879 879
John Travolta: 91 234 4564 789
```

Введите имя для удаления: **Bill Gates**

```
После удаления Bill Gates
Angi Merkel: +49 23456 5466
Ben Affleck: +1 745 641 314
Dim Medvedev: +7 6645 4564 797
Jack Welsch: +1 7889 879 879
John Travolta: 91 234 4564 789
```

Анализ

Этот пример очень похож на листинг 18.7, в котором список был отсортирован в алфавитном порядке, но в данном случае сортировка множества осуществляется непосредственно при вставке. Как показывает вывод, чтобы обеспечить сортировку элементов в множестве, не нужно вызывать никаких функций, поскольку элементы вставляются в соответствующие упорядоченному расположению места при вставке в множество с использованием оператора `<`, реализованного в строках 36–39. Программа запрашивает пользователя о том, какая запись должна быть удалена, и в строке 63 вызов функции `find()` находит указанную пользователем запись (которая в строке 68 удаляется из множества с помощью метода `erase()`).

СОВЕТ

Эта реализация телефонного справочника основана на классе `set`, а потому не позволяет содержать несколько записей с одинаковыми значениями. Если необходима реализация справочника, позволяющая хранить две записи с одинаковым именем (скажем, Том), то выбирайте для нее класс `multiset`. Если контейнер `setCs` станет мультимножеством, весь приведенный выше код продолжит корректно работать. Чтобы выяснить количество элементов с определенным значением в мультимножестве, воспользуйтесь функцией-членом `count()`.

Преимущества и недостатки использования множеств и мультимножеств

Классы `set` и `multiset` STL предоставляют существенные преимущества для приложений, нуждающихся в частом проведении поиска. Поскольку их содержимое отсортировано, поиск осуществляется быстрее, чем в векторе или списке. Но для предоставления этого преимущества контейнер должен сортировать элементы во время вставки. Таким образом, вставка элементов приводит к дополнительным затратам на их сортировку, являющимся необходимой платой за возможность частого использования таких функций, как `find()`.

Функция `find()` использует внутреннюю структуру бинарного дерева. Эта древовидная структура является причиной другого неявного недостатка множества по сравнению с таким последовательным контейнером, как вектор. Элемент в векторе, на который указывает итератор (скажем, возвращенный функцией `std::find()`), может быть перезаписан новым значением. Но в случае множества элементы располагаются во внутренней структуре в определенном порядке, а потому никогда не следует допускать перезаписи элемента с помощью итератора, даже если бы программно это было возможно.

Реализация хеш-множеств `std::unordered_set` и `std::unordered_multiset`

Контейнеры `std::set` и `std::multiset` выполняют сортировку элементов (которые одновременно являются ключами) на основании предиката `std::less<T>` или предиката, предоставленного пользователем. Поиск в отсортированном контейнере выполняется гораздо быстрее, чем в не отсортированном (таком, как вектор). Упорядоченность обеспечивает логарифмическую сложность поиска. Это означает, что время, потраченное на поиск элемента в множестве, пропорционально не количеству элементов в нем, а логарифму этого количества. Таким образом, поиск среди 10 000 элементов множества осуществляется вдвое дольше, чем среди 100 элементов (так как $100^2 = 10000$, так что $\log(10000) = 2 \cdot \log(100)$).

Однако даже такого существенного увеличения производительности по сравнению с неотсортированным контейнером (продолжительность поиска в котором прямо пропорциональна количеству хранящихся в нем элементов) иногда недостаточно. Программисты и математики упорно ищут способ осуществления вставки и сортировки за константное время, и одним из таких способов является реализация на базе хеша, в котором хеш-функция используется для определения индекса элемента. Добавляемый в хеш-множество элемент сначала обрабатывается хеш-функцией, которая генерирует (по возможности уникальный) индекс ячейки, в которую он помещаются. Библиотека STL начиная со стандарта C++ предоставляет свой вариант хеш-множества в виде контейнерного класса `std::unordered_set`.

СОВЕТ

Чтобы использовать классы контейнеров `std::unordered_set` и `std::unordered_multiset`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include<unordered_set>
```

Применение этого класса не слишком отличается от использования класса `std::set`:

```
// Создание экземпляра:
unordered_set<int> usetInt;

// Вставка элемента
usetInt.insert(1000);

// Поиск find():
auto elementFound = usetInt.find(1000);

if (elementFound != usetInt.end())
    cout << *elementFound << endl;
```

Одной из важнейших особенностей контейнера `unordered_set` является доступность хеш-функции, отвечающей за генерацию индексов:

```
unordered_set<int>::hasher HFn = usetInt.hash_function();
```

Принимать решение о применении `std::unordered_set` или `std::set` лучше после выполнения хронометража работы соответствующих контейнеров в условиях, максимально приближенных к условиям реального использования приложения.

В листинге 19.6 показано применение некоторых из наиболее распространенных методов, предоставляемых классом `std::hash_set`.

ЛИСТИНГ 19.6. Применение методов класса `std::unordered_set`

```
0: #include<unordered_set>
1: #include <iostream>
2: using namespace std;
3:
4: template <typename T>
5: void DisplayContents(const T& cont)
6: {
7:     cout << "Неупорядоченное множество содержит: ";
8:     for(auto element = cont.cbegin();
9:         element != cont.cend();
10:        ++element )
11:         cout<< *element << ' ';
12:     cout << endl;
13:
14:     cout << "Число элементов = " << cont.size() << endl;
```

```
15:     cout << "Число ячеек = " << cont.bucket_count() << endl;
16:     cout << "Мак коэффициент загрузки = " << cont.max_load_factor();
17:     cout << endl << "Коэффициент загрузки: ";
18:     cout << cont.load_factor() << endl << endl;
19: }
20:
21: int main()
22: {
23:     unordered_set<int> usetInt{1,-3,2017,300,-1,989,-300,9};
24:     DisplayContents(usetInt);
25:     usetInt.insert(999);
26:     DisplayContents(usetInt);
27:
28:     cout << "Введите искомый элемент: ";
29:     int input = 0;
30:     cin >> input;
31:     auto elementFound = usetInt.find(input);
32:
33:     if (elementFound != usetInt.end())
34:         cout << *elementFound << " найден" << endl;
35:     else
36:         cout << input << " отсутствует" << endl;
37:
38:     return 0;
39: }
```

Результат

Неупорядоченное множество содержит: 9 1 -3 989 -1 2017 300 -300

Число элементов = 8

Число ячеек = 8

Мак коэффициент загрузки = 1

Коэффициент загрузки: 1

Неупорядоченное множество содержит: 9 1 -3 989 -1 2017 300 -300 999

Число элементов = 9

Число ячеек = 64

Мак коэффициент загрузки = 1

Коэффициент загрузки: 0.140625

Введите искомый элемент: **-300**

-300 найден

Анализ

В этом примере создается контейнер `unordered_set` для целых чисел; он инициализируется с использованием списка инициализации в строке 23, а затем на экране отображаются его содержимое, а также статистика, предоставляемая методами

`max_bucket_count()`, `load_factor()` и `max_load_factor()`, как показано в строках 14–18. Вывод свидетельствует о том, что начальное количество ячеек в множестве равно восьми, при этом в контейнере находится восемь элементов и коэффициент загрузки равен 1 (максимальное значение). Когда в контейнер `unordered_set` вставляется девятый элемент, он реорганизуется, создавая 64 ячейки и “перехешируя” таблицу; коэффициент загрузки при этом уменьшается. Остальная часть кода в `main()` демонстрирует, что синтаксис поиска элементов в контейнере `unordered_set` аналогичен таковому в контейнере `set`. Метод `find()` возвращает итератор, который следует сравнить со значением итератора, указывающего на конец контейнера, как показано в строке 33, прежде чем он будет использован.

ПРИМЕЧАНИЕ

Поскольку хеши обычно используются в хеш-таблице для поиска значения, заданного по ключу, обратитесь за подробной информацией к разделу о контейнере `std::unordered_map` занятия 20, “Классы отображений библиотеки STL”.

Контейнер `std::unordered_map` является реализацией хеш-таблицы, появившейся в стандарте C++11.

РЕКОМЕНДУЕТСЯ

Помните, что контейнеры `set` и `multiset` библиотеки STL оптимизированы для применения в программах с частым поиском.

Помните, что контейнер `std::multiset` допускает несколько одинаковых элементов (ключей), а контейнер `std::set` разрешает хранить только уникальные значения.

Используйте метод `multiset::count(значение)` для поиска количества элементов с определенным значением.

Помните, что методы `set::size()` и `multiset::size()` возвращают количество элементов в контейнере.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте определять операторы `<` и `==` для классов, объекты которых могут храниться в таких контейнерах, как `set` и `multiset`. Первый становится предикатом сортировки по умолчанию, а второй используется для таких функций, как `set::find()`.

Не используйте контейнеры `std::set` и `std::multiset` в сценариях с частыми вставками и нечастыми поисками. Для этого обычно лучше подходят такие контейнеры, как `std::vector` и `std::list`.

Резюме

На сегодняшнем занятии рассмотрены контейнеры STL `set` и `multiset`, их основные функции-члены и характеристики. Вы также увидели их применение для разработки простого телефонного справочника, реализующего функции поиска и удаления.

Вопросы и ответы

- **Как мне объявить множество целых чисел, отсортированных и хранящихся в порядке убывания величин?**

Шаблон класса `set<int>` определяет множество целых чисел. Он использует заданный по умолчанию предикат сортировки `std::less<T>`, обеспечивающий сортировку элементов в порядке возрастания величин, и может быть также выражен как `set<int, less<int>>`. Для сортировки в порядке убывания величин определите множество как `set<int, greater<int>>`.

- **Что будет, если вставить строку "Jack" в множество строк дважды?**

Множество не предназначено для хранения совпадающих значений. Реализация класса `std::set` не позволит вставить значение, которое уже есть в множестве.

- **Что нужно изменить в предыдущем примере, чтобы все-таки получить два экземпляра строки "Jack"?**

Реализация класса `set` позволяет хранить только уникальные значения. Смените выбранный контейнер на `multiset`.

- **Какая функция-член класса `multiset` возвращает количество элементов с определенным значением в контейнере?**

Это функция `count(значение)`.

- **Используя функцию `find()`, я нашел элемент в множестве, и теперь у меня есть указывающий на него итератор. Как мне использовать этот итератор для изменения значения, на которое он указывает?**

Никак. Некоторые устаревшие реализации STL могли бы позволить пользователю изменить значение элемента в множестве с помощью итератора, возвращенного, например, функцией `find()`. Но так поступать некорректно. Итератор, указывающий на элемент множества, должен использоваться как константный, даже если реализация STL не обеспечивает его константность.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, "Ответы".

Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Вы объявляете множество целых чисел как `set<int>`. Какая функция будет использована в качестве критерия сортировки?
2. Где вы обнаружите совпадающие элементы в контейнере `multiset`?
3. Какая функция контейнеров `set` и `multiset` возвращает количество элементов в них?

Упражнения

1. Дополните пример телефонного справочника из этого занятия поиском имени человека по заданному номеру телефона. (Указание: измените операторы `<` и `==` и обеспечьте сортировку и сравнение записей по номеру телефона.)
2. Определите мультимножество для хранения введенных слов и их значений, т.е. создайте мультимножество, работающее в качестве словаря. (Указание: мультимножество должно хранить объекты структуры, которая содержит две строки: слово и его значение.)
3. Напишите простую программу, демонстрирующую, что множество не может хранить совпадающие элементы, а мультимножество может.

ЗАНЯТИЕ 20

Классы отображений библиотеки STL

Стандартная библиотека шаблонов (STL) предоставляет разработчикам классы контейнеров для приложений, которым требуются частые и быстрые поиски.

На этом занятии...

- Как использовать классы `map`, `multimap`, `unordered_map` и `unordered_multimap` STL
- Вставка, удаление и поиск элементов
- Предоставление пользовательского предиката сортировки
- Основы работы хеш-таблиц

Введение в классы отображений библиотеки STL

Классы `map` и `multimap` являются контейнерами пар “ключ–значение”, допускающими поиск на основе ключа, как показано на рис. 20.1.

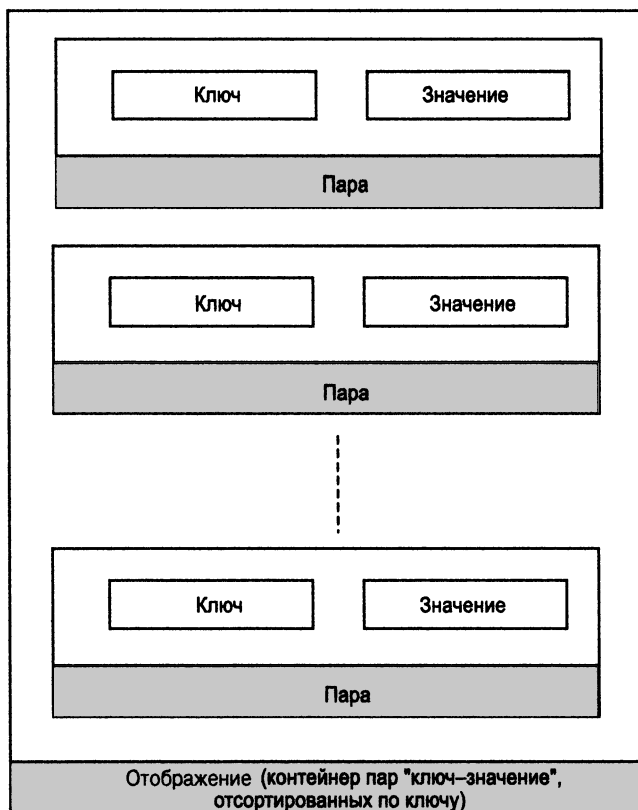


РИС. 20.1. Визуальное представление контейнеров, содержащих пары “ключ–значение”

Различие между отображением (`map`) и мультиотображением (`multimap`) в том, что последнее допускает наличие совпадающих ключей, в то время как отображение позволяет хранить только уникальные ключи.

Чтобы облегчить быстрый поиск, реализации классов `map` и `multimap` внутренне представляют собой сбалансированные бинарные деревья. Это означает, что элементы, вносимые в отображение или мультиотображение, сортируются во время вставки. Это также значит, что, в отличие от вектора, в котором элементы в любой позиции могут быть заменены другими, элементы отображения в каждой позиции не могут быть просто заменены новым элементом с другим значением ключа, так как это нарушит свойство отсортированности элементов в отображении в соответствии со значениями ключей.

СОВЕТ

Чтобы использовать класс `std::map` или `std::multimap`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include<map>
```

Фундаментальные операции с классами `std::map` и `std::multimap`

Прежде чем вы сможете использовать функции-члены классов `map` и `multimap`, необходимо их инстанцировать.

Инстанцирование классов `std::map` и `std::multimap`

Создание экземпляра отображения или мультиотображения с целыми числами в качестве ключа и строками в качестве значений требует специализации шаблона класса `std::map` или `std::multimap`. Разработчик должен указать тип ключа, значения и (необязательно) предиката, определяющего упорядочение элементов для сортировки при вставке. Поэтому типичный синтаксис инстанцирования отображения выглядит следующим образом:

```
#include <map>
using namespace std;
...
map<keyType, valueType, Predicate=std::less <keyType>> mapObject;
multimap<keyType, valueType, Predicate=std::less <keyType>> mmapObject;
```

Таким образом, третий параметр шаблона является необязательным. Когда предоставляются только типы ключа и значения, то в качестве третьего параметра шаблона — предиката, определяющего критерии сортировки — контейнеры `std::map` и `std::multimap` используют предикат по умолчанию `std::less<>`. Упомянутые отображение и мультиотображение с целыми числами в качестве ключа и строками в качестве значений имеют следующий вид:

```
std::map<int, string> mapIntToStr;
std::multimap<int, string> mmapIntToStr;
```

В листинге 20.1 подробно показаны способы создания их экземпляров.

ЛИСТИНГ 20.1. Инстанцирование `map` и `multimap`, отображающих целочисленные ключи на строковые значения

```
0: #include<map>
1: #include<string>
2:
3: template<typename KeyType>
4: struct ReverseSort
5: {
```

```
6:     bool operator()(const KeyType& key1, const KeyType& key2)
7:     {
8:         return (key1 > key2);
9:     }
10: };
11:
12: int main()
13: {
14:     using namespace std;
15:
16:     // map и multimap для ключей int со значениями string
17:     map<int, string> mapIntToStr1;
18:     multimap<int, string> mmapIntToStr1;
19:
20:     // map и multimap создаются как копия другого контейнера
21:     map<int, string> mapIntToStr2(mapIntToStr1);
22:     multimap<int, string> mmapIntToStr2(mmapIntToStr1);
23:
24:     // map и multimap создаются из части другого контейнера
25:     map<int, string> mapIntToStr3(mapIntToStr1.cbegin(),
26:                                 mapIntToStr1.cend());
27:
28:     multimap<int, string> mmapIntToStr3(mmapIntToStr1.cbegin(),
29:                                       mmapIntToStr1.cend());
30:
31:     // map и multimap с сортировкой в обратном порядке
32:     map<int, string, ReverseSort<int>> mapIntToStr4(
33:         mapIntToStr1.cbegin(), mapIntToStr1.cend());
34:
35:     multimap<int, string, ReverseSort<int>> mmapIntToStr4(
36:         mapIntToStr1.cbegin(), mapIntToStr1.cend());
37:
38:     return 0;
39: }
```

Анализ

Для начала сосредоточимся на строках 12–39 функции `main()`. Простейшее отображение и мультиотображение целочисленных ключей и строковых значений создаются в строках 17 и 18. В строках 25–29 демонстрируется создание отображения и мультиотображения, инициализированных диапазоном значений из других контейнеров. В строках 31–36 демонстрируются создание экземпляров отображения и мультиотображения с пользовательским критерием сортировки. Обратите внимание, что сортировка по умолчанию (использованная в предыдущих экземплярах) использует предикат `std::less<T>`, который сортирует элементы в порядке возрастания ключей. Чтобы изменить это поведение, необходимо предоставить предикат, который может представлять собой класс или структуру, реализующую оператор `operator()`. Такая структура `ReverseSort` находится в строках 3–10 и используется при создании экземпляра отображения в строке 32 и мультиотображения в строке 35.

Вставка элементов в `map` и `multimap`

Большинство функций `map` и `multimap` работают одинаково. Они получают подобные параметры и возвращают значения подобных типов. Для вставки элементов в контейнеры обоих видов используется функция-член `insert()`:

```
std::map<int, std::string> mapIntToStr1;
// Вставка пары, полученной с помощью функции make_pair()
mapIntToStr.insert(make_pair(-1, "Minus One"));
```

Поскольку элементы этих двух контейнеров содержат пары “ключ–значение”, вы можете непосредственно вставлять инициализированные пары `std::pair`:

```
mapIntToStr.insert(pair<int, string>(1000, "One Thousand"));
```

В качестве альтернативы можно использовать для вставки синтаксис массива, который привычен пользователю и поддерживается с помощью оператора индексации `operator[]` (только у `map`):

```
mapIntToStr[1000000] = "One Million";
```

Вы можете также создать экземпляр мультиотображения как копию отображения:

```
std::multimap<int, std::string> mmapIntToStr(mapIntToStr.cbegin(),
                                             mapIntToStr.cend());
```

В листинге 20.2 приведены различные методы вставки элементов в отображения.

ЛИСТИНГ 20.2. Вставка элементов в `map` и `multimap`

```
0: #include <map>
1: #include <iostream>
2: #include<string>
3:
4: using namespace std;
5:
6: // Определение синонимов типов map и multimap для удобочитаемости
7: typedef map <int, string> MAP_INT_STRING;
8: typedef multimap <int, string> MMAP_INT_STRING;
9:
10: template <typename T>
11: void DisplayContents(const T& cont)
12: {
13:     for(auto element = cont.cbegin();
14:         element != cont.cend();
15:         ++element)
16:         cout << element->first << " -> "
17:              << element->second << endl;
18:     cout << endl;
19: }
20:
21: int main()
```

```
22: {
23:     MAP_INT_STRING mapIntToStr;
24:
25:     // Вставка пары с использованием ключевого слова value_type
26:     mapIntToStr.insert(MAP_INT_STRING::value_type(3, "Three"));
27:
28:     // Вставка пары с использованием функции make_pair()
29:     mapIntToStr.insert(make_pair(-1, "Minus One"));
30:
31:     // Вставка объекта пары непосредственно
32:     mapIntToStr.insert(pair<int, string>(1000, "One Thousand"));
33:
34:     // Вставка пары с использованием синтаксиса массива
35:     mapIntToStr[1000000] = "One Million";
36:
37:     cout << "map содержит " << mapIntToStr.size();
38:     cout << " пары \"ключ-значение\". Это:" << endl;
39:     DisplayContents(mapIntToStr);
40:
41:     // Создание мультиотображения, являющегося копией отображения
42:     MMAP_INT_STRING mmapIntToStr(mapIntToStr.cbegin(),
43:                                   mapIntToStr.cend());
44:
45:     // insert() работает так же, как в мультиотображении
46:     // Мультиотображение может хранить дубликаты:
47:     mmapIntToStr.insert(make_pair(1000, "Thousand"));
48:
49:     cout << endl << "multimap содержит " << mmapIntToStr.size();
50:     cout << " пар \"ключ-значение\"." << endl;
51:     cout << "Элементы multimap: " << endl;
52:     DisplayContents(mmapIntToStr);
53:
54:     // Мультиотображение возвращает число пар с одинаковым ключом
55:     cout << "Пар в multimap с ключом 1000: "
56:           << mmapIntToStr.count(1000) << endl;
57:
58:     return 0;
59: }
```

Результат

```
map содержит 4 пары "ключ-значение". Это:
-1 -> Minus One
3 -> Three
1000 -> One Thousand
1000000 -> One Million
```

```
multimap содержит 5 пар "ключ-значение".  
Элементы multimap:  
-1 -> Minus One  
3 -> Three  
1000 -> One Thousand  
1000 -> Thousand  
1000000 -> One Million
```

Пар в `multimap` с ключом 1000: 2

Анализ

Обратите внимание на определение синонимов типов в строках 7 и 8. В результате код будет выглядеть немного проще (и уменьшит запутанность, вызываемую синтаксисом шаблонов). Строки 10–19 содержат функцию `DisplayContents()`, адаптированную для отображения и мультиотображения, в которой итератор использует поля `first` для доступа к ключу и `second` — для доступа к значению. В строках 26–32 показаны различные способы вставки пар “ключ–значение” в отображение с использованием перегруженных вариантов метода `insert()`. В строке 35 продемонстрирована возможность вставки элементов в отображение с использованием семантики массива с применением оператора `[]`. Обратите внимание на то, что указанные механизмы вставки (кроме оператора `[]`) работают и для мультиотображения, которое представлено в том же листинге. Интересно, что мультиотображение инициализировано как копия отображения (строки 42 и 43). Вывод показывает, что оба контейнера автоматически сортируют вставляемые пары “ключ–значение” в порядке возрастания ключей. Вывод также демонстрирует, что мультиотображение способно хранить пары с одинаковым ключом (в данном случае — 1000). Строка 56 демонстрирует применение метода `multimap::count()`, возвращающего количество элементов в контейнере с указанным ключом.

Поиск элементов в отображении

Ассоциативные контейнеры, такие как `map` и `multimap`, предоставляют функцию-член `find()`, позволяющую находить значения с заданным ключом. Результат операции поиска всегда представляет собой итератор:

```
multimap<int, string>::const_iterator pairFound = mapIntToStr.find(Key);
```

Прежде чем использовать этот итератор для доступа к найденному значению, необходимо проверить успешность поиска:

```
if (pairFound != mapIntToStr.end())  
{  
    cout << "Ключ " << pairFound->first << " указывает на значение: ";  
    cout << pairFound->second << endl;  
}  
else  
    cout << "Ключ " << Key << " в отображении не найден" << endl;
```

СОВЕТ

Если вы используете компилятор, совместимый со стандартом C++11, при объявлении итератора удобнее использовать ключевое слово `auto`:

```
auto pairFound = mapIntToStr.find(Key);
```

Компилятор автоматически выводит тип итератора из возвращаемого значения функции `map::find()`.

Пример в листинге 20.3 демонстрирует применение функции `multimap::find()`.

ЛИСТИНГ 20.3. Использование для поиска функции-члена `find()`

```
0: #include <map>
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: template <typename T>
6: void DisplayContents(const T& cont)
7: {
8:     for(auto element = cont.cbegin();
9:         element != cont.cend();
10:         ++element )
11:         cout << element->first << " -> "
12:             << element->second << endl;
13:     cout << endl;
14: }
15:
16: int main()
17: {
18:     map<int, string> mapIntToStr;
19:
20:     mapIntToStr.insert(make_pair(3, "Three"));
21:     mapIntToStr.insert(make_pair(45, "Forty Five"));
22:     mapIntToStr.insert(make_pair(-1, "Minus One"));
23:     mapIntToStr.insert(make_pair(1000, "Thousand"));
24:
25:     cout << "multimap содержит " << mapIntToStr.size();
26:     cout << " пары. Это:" << endl;
27:
28:     // Вывод содержимого отображения на экран
29:     DisplayContents(mapIntToStr);
30:
31:     cout << "Введите ключ для поиска: ";
32:     int Key = 0;
33:     cin >> Key;
34:
35:     auto pairFound = mapIntToStr.find(Key);
36:     if (pairFound != mapIntToStr.end())
```

```
37:     {
38:         cout << "Ключ " << pairFound->first << " указывает на ";
39:         cout << "значение " << pairFound->second << endl;
40:     }
41:     else
42:         cout << "Ключ " << Key << " не найден"
              << endl;
43:
44:     return 0;
45: }
```

Результат

`multimap` содержит 4 пары. Это:

```
-1 -> Minus One
3  -> Three
45 -> Forty Five
1000 -> Thousand
```

Введите ключ для поиска: **45**

Ключ 45 указывает на значение Forty Five

Следующий запуск (в котором функция `find()` не находит соответствующего значения):

`multimap` содержит 4 пары. Это:

```
-1 -> Minus One
3  -> Three
45 -> Forty Five
1000 -> Thousand
```

Введите ключ для поиска: 2011

Ключ 2011 не найден

Анализ

В строках 20–23 функции `main()` отображение заполняется примерами пар, каждая из которых отображает целочисленный ключ на строковое значение. Когда пользователь вводит ключ, функция `find()` в строке 35 выполняет его поиск в отображении. Функция `map::find()` всегда возвращает итератор, который необходимо сравнить с итератором, возвращаемым методом `end()`, как показано в строке 36. Если итератор корректен, для доступа к значению используется член `second` (строка 39). При втором запуске был введен ключ 2011, которого нет в отображении, так что пользователь получил сообщение об ошибке.

ВНИМАНИЕ!

Никогда не используйте результат функции `find()` непосредственно, не проверив полученный итератор.

Поиск элементов в мультиотображении STL

Если бы в листинге 20.3 использовалось мультиотображение, у нас была бы возможность хранить в контейнере несколько пар с одинаковым ключом, а следовательно, и необходимость поиска всех значений, ключи которых совпадают. Поэтому в случае мультимножества вам пришлось бы использовать метод `multiset::count()` для выяснения количества значений, соответствующих ключу, и прибегнуть к инкременту итератора, чтобы получить доступ ко всем значениям с одинаковым ключом:

```
auto pairFound = mmapIntToStr.find(Key);

// Проверка успешности поиска
if(pairFound != mmapIntToStr.end())
{
    // Количество пар с одним и тем же предоставленным ключом
    size_t numPairsInMap = mmapIntToStr.count(1000);

    for(size_t counter = 0;
        counter < numPairsInMap; // Оставаться в границах
        ++counter )
    {
        cout << "Ключ: " << pairFound->first; // Ключ
        cout << ", Значение[" << counter << "] = ";
        cout << pairFound->second << endl;    // Значение

        ++pairFound;
    }
}
else
    cout << "Элемент в multimap не найден";
```

Удаление элементов из map и multimap

И `map`, и `multimap` предоставляют функцию-член `erase()`, которая удаляет элементы из контейнера. Для удаления всех элементов с определенным ключом его следует передать функции `erase()` в качестве аргумента:

```
mapObject.erase(key);
```

Другая форма функции `erase()` позволяет удалить элемент, определенный указывающим на него итератором:

```
mapObject.erase(element);
```

Вы можете удалить из `map` и `multimap` диапазон элементов, для чего следует передать функции итераторы, указывающие границы диапазона:

```
mapObject.erase(lowerBound, upperBound);
```

В листинге 20.4 показано применение функции `erase()`.

ЛИСТИНГ 20.4. Удаление элементов из мультиотображения

```
0: #include<map>
1: #include<iostream>
2: #include<string>
3: using namespace std;
4:
5: template<typename T>
6: void DisplayContents(const T& cont)
7: {
8:     for(auto element = cont.cbegin();
9:         element != cont.cend();
10:         ++element )
11:         cout << element->first << " -> "
12:             << element->second << endl;
13:     cout<< endl;
14: }
15:
16: int main()
17: {
18:     multimap<int, string> mmapIntToStr;
19:
20:     // Вставка пар "ключ-значение" в мультиотображение
21:     mmapIntToStr.insert(make_pair(3, "Three"));
22:     mmapIntToStr.insert(make_pair(45, "Forty Five"));
23:     mmapIntToStr.insert(make_pair(-1, "Minus One"));
24:     mmapIntToStr.insert(make_pair(1000, "Thousand"));
25:
26:     // Вставка дубликатов в мультиотображение
27:     mmapIntToStr.insert(make_pair(-1, "Minus one"));
28:     mmapIntToStr.insert(make_pair(1000, "One thousand"));
29:
30:     cout<< "multimap содержит "<< mmapIntToStr.size();
31:     cout<< " пар. "<< "Это:"<< endl;
32:     DisplayContents(mmapIntToStr);
33:
34:     // Удаление элемента с ключом -1 из мультиотображения
35:     auto nPrErsd = mmapIntToStr.erase(-1);
36:     cout<< "Удалено " << nPrErsd << " пары с ключом -1." << endl;
37:
38:     // Удаление элемента, указанного итератором
39:     auto iPair = mmapIntToStr.find(45);
40:     if(iPair != mmapIntToStr.end())
41:     {
42:         mmapIntToStr.erase(iPair);
43:         cout<< "Удалена пара с ключом 45." << endl;
44:     }
45:
46:     // Удаление из мультиотображения диапазона...
```

```
47:     cout << "Удаление пар с ключом 1000." << endl;
48:     mmapIntToStr.erase(mmapIntToStr.lower_bound(1000),
49:                         mmapIntToStr.upper_bound(1000));
50:
51:     cout << "Теперь multimap содержит " << mmapIntToStr.size();
52:     cout << " пар. Это:" << endl;
53:     DisplayContents(mmapIntToStr);
54:
55:     return 0;
56: }
```

Результат

multimap содержит 6 пар. Это:

```
-1 -> Minus One
-1 -> Minus one
3 -> Three
45 -> Forty Five
1000 -> Thousand
1000 -> One thousand
```

```
Удалено 2 пары с ключом -1.
Удалена пара с ключом 45.
Удаление пар с ключом 1000.
Теперь multimap содержит 1 пару. Это:
3 -> Three
```

Анализ

Код в строках 21–28 вставляет в мультиотображение значения, некоторые из которых являются дубликатами (поскольку мультиотображение, в отличие от отображения, допускает вставку элементов с одинаковыми ключами). После того как все пары вставлены в мультиотображение, код удаляет элементы с помощью варианта функции `erase()`, которая получает ключ и удаляет все элементы с этим ключом, равным в данном случае `-1` (строка 35). Функция `multimap::erase(Key)` возвращает количество удаленных элементов, которое отображается на экране. В строке 42 итератор, возвращенный в строке 39 вызовом функции `find(45)`, использован для удаления из отображения элемента с ключом 45. Строки 48 и 49 демонстрируют удаление диапазона пар, определенного с помощью вызовов `lower_bound()` и `upper_bound()`.

Применение пользовательского предиката

Определения шаблонов `map` и `multimap` включают третий параметр, позволяющий передать предикат сортировки для настройки их функционирования. Если не

предоставлять этот третий параметр (как было показано в примерах выше), применяется критерий сортировки по умолчанию, обеспечиваемый предикатом `std::less<>` (который, по существу, сравнивает два объекта с помощью оператора `<`).

Для предоставления другого критерия сортировки необходимо предоставить бинарный предикат, например, в форме класса или структуры, реализующей оператор `operator()`:

```
template<typename Тип_Ключа>
struct Predicate
{
    bool operator()(const Тип_Ключа& key1, const Тип_Ключа& key2)
    {
        // Здесь находится логика отношения "меньше"
    }
};
```

Отображение, содержащее ключ типа `std::string`, по умолчанию использует для сортировки оператор `<`, определенный в классе `std::string` и применяемый предикатом сортировки по умолчанию `std::less<std::string>`, а потому сортировка чувствительна к регистру символов в строке. Это не годится для многих приложений, таких как телефонный справочник, поскольку им требуется сортировка, не чувствительная к регистру. Один из способов решения этой задачи заключается в предоставлении отображению предиката сортировки, который сравнивает ключи-строки без учета регистра символов:

```
map <Тип_Ключа, Тип_Значения, Предикат> Объект_Отображения;
```

В листинге 20.5 это продемонстрировано подробно.

ЛИСТИНГ 20.5. Предоставление предиката сортировки без учета регистра символов

```
0: #include<map>
1: #include<algorithm>
2: #include<string>
3: #include<iostream>
4: using namespace std;
5:
6: template <typename T>
7: void DisplayContents(const T& cont)
8: {
9:     for(auto element = cont.cbegin();
10:         element != cont.cend();
11:         ++element )
12:         cout << element->first << " -> " << element->second << endl;
13:
14:     cout << endl;
15: }
16:
17: struct PredIgnoreCase
18: {
```

```
19: bool operator()(const string& str1, const string& str2) const
20: {
21:     string str1NoCase(str1), str2NoCase(str2);
22:     transform(str1.begin(), str1.end(), str1NoCase.begin(), ::tolower);
23:     transform(str2.begin(), str2.end(), str2NoCase.begin(), ::tolower);
24:
25:     return(str1NoCase< str2NoCase);
26: };
27: };
28:
29: typedef map<string, string> DIR_WITH_CASE;
30: typedef map<string, string, PredIgnoreCase> DIR_NOCASE;
31:
32: int main()
33: {
34:     // Чувствительное к регистру отображение
35:     DIR_WITH_CASE dirWithCase;
36:
37:     dirWithCase.insert(make_pair("John", "2345764"));
38:     dirWithCase.insert(make_pair("JOHN", "2345764"));
39:     dirWithCase.insert(make_pair("Sara", "42367236"));
40:     dirWithCase.insert(make_pair("Jack", "32435348"));
41:
42:     cout << "Вывод чувствительного к регистру map:" << endl;
43:     DisplayContents(dirWithCase);
44:
45:     // Нечувствительное к регистру отображение
46:     DIR_NOCASE dirNoCase(dirWithCase.begin(), dirWithCase.end());
47:
48:     cout << "Вывод нечувствительного к регистру map:" << endl;
49:     DisplayContents(dirNoCase);
50:
51:     // Поиск имени в двух отображениях
52:     cout << "Введите имя для поиска:" << endl << "> ";
53:     string name;
54:     cin >> name;
55:
56:     auto pairWithCase = dirWithCase.find(name);
57:     if(pairWithCase != dirWithCase.end())
58:         cout << "Номер при чувствительности к регистру: "
59:             << pairWithCase->second << endl;
60:     else
61:         cout << "Номер не найден" << endl;
62:
63:     auto pairNoCase = dirNoCase.find(name);
64:     if (pairNoCase != dirNoCase.end())
65:         cout << "Номер при нечувствительности к регистру: "
66:             << pairNoCase->second << endl;
67:     else
```

```
68:         cout << "Номер не найден" << endl;
69:
70:     return 0;
71: }
```

Результат

Вывод чувствительного к регистру map:

```
JOHN -> 2345764
Jack  -> 32435348
John  -> 2345764
Sara  -> 42367236
```

Вывод нечувствительного к регистру map:

```
Jack -> 32435348
JOHN -> 2345764
Sara -> 42367236
```

Введите имя для поиска:

> **sara**

Номер не найден

Номер при нечувствительности к регистру: 42367236

Анализ

Рассматриваемый код содержит два каталога с одинаковым содержимым: один был создан с заданным по умолчанию предикатом сортировки `std::less<T>` и оператором `<`, работающим с учетом регистра символов, а другой — с предикатом `PredIgnoreCase` (строки 17–27), сравнивающим две строки после преобразования их символов в нижний регистр, так что для него слова "John" и "JOHN" будут одинаковы. Вывод показывает, что при поиске в двух отображениях слова 'sara' в независимом от регистра отображении будет найдена запись Sara, тогда как отображение с предикатом по умолчанию найти эту запись не в состоянии. Обратите также внимание, что в отображении, чувствительном к регистру символов, имеются две записи с именами "John" и "JOHN", в то время как отображение, нечувствительное к регистру, рассматривает их как дубликат и позволяет иметь только один элемент с данным ключом.

ПРИМЕЧАНИЕ

В листинге 20.5 структура `PredIgnoreCase` может быть классом; в этом случае оператор `operator()` должен быть описан как `public`. С точки зрения компилятора C++ структура представляет собой класс, члены которого по умолчанию являются открытыми. Наследование структур также является по умолчанию открытым.

Этот пример демонстрирует возможность использования предикатов для настройки поведения отображения, а также то, что потенциально ключ может иметь любой тип и что программист может предоставить предикат, определяющий требуемое

поведение отображения для этого типа. Обратите внимание на то, что предикат в нашем примере был структурой, которая реализует `operator()`. Однако это вполне может быть и класс. Такие объекты именуются также *функциональными объектами* (function object), или *функторами* (functor). Более подробно об этом речь идет на занятии 21, “Понятие о функциональных объектах”.

ПРИМЕЧАНИЕ

Контейнер `std::map` хорошо подходит для хранения пар “ключ–значение”, когда требуется искать значение для данного ключа. Отображение действительно гарантирует более высокую производительность, чем вектор или список, когда дело доходит до поиска. Но при увеличении количества элементов замедляется поиск даже в отображении. Производительность поиска в нем имеет логарифмическую сложность, т.е. время поиска пропорционально логарифму количества находящихся в контейнере элементов. Попросту говоря, логарифмическая сложность означает, что при 10 000 элементов поиск в таком контейнере, как `std::map` или `std::set`, осуществляется вдвое медленнее, чем при 100 элементах ($100^2=10000$). Неотсортированный вектор имеет линейную сложность поиска, т.е. время поиска среди 10 000 элементов в среднем в 100 раз больше, чем среди 100 элементов.

Контейнер для пар “ключ–значение” на базе хеш-таблиц

Начиная со стандарта C++11 STL предоставляет хеш-отображение в форме класса `std::unordered_map`. Для использования этого шаблонного контейнера в программу нужно включить соответствующий заголовочный файл:

```
#include<unordered_map>
```

Контейнер `unordered_map` обеспечивает константное время вставки, удаления и поиска произвольного элемента в контейнере.

Как работают хеш-таблицы

Хотя в рамках этой книги мы не будем обсуждать данную тему во всех подробностях (она была предметом слишком многих диссертаций), мы все же попытаемся разобраться в том, как работают *хеш-таблицы* (hash table).

Хеш-таблицу можно рассматривать как коллекцию пар “ключ–значение”, в которой таблица позволяет найти значение по заданному ключу. Различие между хеш-таблицей и простым отображением заключается в том, что первая хранит пары “ключ–значение” в индексированных ячейках, причем индекс определяет относительную позицию ячейки в таблице (как в массиве). Индекс определяется хеш-функцией, которой передается значение ключа:

Индекс = Хеш_функция(Ключ, Размер_Таблицы);

При выполнении поиска для заданного ключа *Хеш_функция()* используется еще раз, чтобы вычислить позицию элемента и вернуть из таблицы значение, находящееся в этой позиции, как если бы это был простой элемент массива. В тех случаях, когда *Хеш_функция()* запрограммирована не оптимально, один и тот же *Индекс* может быть у нескольких элементов, которые в результате располагаются в одной и той же ячейке, которая при этом вынуждена содержать список элементов. В таких случаях, называемых *коллизиями* (collision), поиск осуществляется медленнее и имеет неконстантную продолжительность.

Использование `unordered_map` и `unordered_multimap`

С точки зрения применения эти два контейнера мало отличаются от контейнеров `std::map` и `std::multimap` соответственно. Инстанцирование, вставка и поиск следуют общей схеме:

```
// Инстанцирование unordered_map для int и string:
unordered_map<int, string> umapIntToStr;

// insert()
umapIntToStr.insert(make_pair(1000, "Thousand"));

// find():
auto iPairThousand = umapIntToStr.find(1000);
cout << iPairThousand->first << " -> " << iPairThousand->second << endl;

// Поиск значения с использованием семантики массива:
cout << "umapIntToStr[1000] = " << umapIntToStr[1000] << endl;
```

Тем не менее контейнеру `unordered_map` присуща одна очень важная особенность — доступность хеш-функции, ответственной за решение о порядке сортировки:

```
unordered_map<int, string>::hasher HFn =
    umapIntToStr.hash_function();
```

С помощью вызова хеш-функции можно узнать индекс, который она присваивает конкретному ключу:

```
size_t HashingValue1000 = HFn(1000);
```

Контейнер `unordered_map` хранит пары “ключ–значение” в ячейках и обеспечивает автоматическую балансировку загрузки, когда количество элементов в отображении достигает (или приближается) количества ячеек в нем:

```
cout << "Коэффициент загрузки      = "
    << umapIntToStr.load_factor()    << endl;
cout << "Max коэффициент загрузки = "
    << umapIntToStr.max_load_factor() << endl;
cout << "Max количество ячеек      = "
    << umapIntToStr.max_bucket_count() << endl;
```


Функция-член `load_factor()` возвращает коэффициент загрузки, т.е. степень заполнения ячеек контейнера `unordered_map`. Когда в ходе вставки значение `load_factor()` достигает значения `max_load_factor()`, отображение реорганизуется, увеличивая количество доступных ячеек, и перестраивает хеш-таблицу (листинг 20.6).

СОВЕТ

Контейнер `std::unordered_multimap` подобен контейнеру `unordered_map`, но допускает наличие нескольких пар с одинаковым ключом.

ЛИСТИНГ 20.6. Работа с хеш-таблицей `unordered_map`

```

0: #include<iostream>
1: #include<string>
2: #include<unordered_map>
3: using namespace std;
4:
5: template <typename T1, typename T2>
6: void DisplayUnorderedMap(unordered_map<T1, T2>& cont)
7: {
8:     cout << "unordered_map содержит:" << endl;
9:     for(auto element = cont.cbegin();
10:         element != cont.cend();
11:         ++element )
12:         cout << element->first << " -> " << element->second << endl;
13:
14:     cout << "size()           = " << cont.size()           << endl;
15:     cout << "bucket_count()  = " << cont.bucket_count()  << endl;
16:     cout << "load_factor()   = " << cont.load_factor()   << endl;
17:     cout << "max_load_factor() = " << cont.max_load_factor() << endl;
18: }
19:
20: int main()
21: {
22:     unordered_map<int, string> umapIntToStr;
23:     umapIntToStr.insert(make_pair(1, "One"));
24:     umapIntToStr.insert(make_pair(45, "Forty Five"));
25:     umapIntToStr.insert(make_pair(1001, "Thousand One"));
26:     umapIntToStr.insert(make_pair(-2, "Minus Two"));
27:     umapIntToStr.insert(make_pair(-1000, "Minus One Thousand"));
28:     umapIntToStr.insert(make_pair(100, "One Hundred"));
29:     umapIntToStr.insert(make_pair(12, "Twelve"));
30:     umapIntToStr.insert(make_pair(-100, "Minus One Hundred"));
31:
32:     DisplayUnorderedMap<int, string>(umapIntToStr);
33:
34:     cout << "\nВставка еще одного элемента" << endl;
35:     umapIntToStr.insert(make_pair(300, "Three Hundred"));
36:     DisplayUnorderedMap<int, string>(umapIntToStr);
37:
38:     cout << "Введите ключ для поиска: ";
39:     int Key = 0;
40:     cin >> Key;

```

```
41:
42:     auto element = umapIntToStr.find(Key);
43:     if (element != umapIntToStr.end())
44:         cout << "Найден! Значение = " << element->second << endl;
45:     else
46:         cout << "Ключ не найден!" << endl;
47:
48:     return 0;
49: }
```

Результат

unordered_map содержит:

```
1 -> One
-2 -> Minus Two
45 -> Forty Five
1001 -> Thousand One
-1000 -> Minus One Thousand
12 -> Twelve
100 -> One Hundred
-100 -> Minus One Hundred
size()           = 8
bucket_count()   = 8
load_factor()    = 1
max_load_factor() = 1
```

Вставка еще одного элемента

unordered_map содержит:

```
1 -> One
-2 -> Minus Two
45 -> Forty Five
1001 -> Thousand One
-1000 -> Minus One Thousand
12 -> Twelve
100 -> One Hundred
-100 -> Minus One Hundred
300 -> Three Hundred
size()           = 9
bucket_count()   = 64
load_factor()    = 0.140625
max_load_factor() = 1
```

Введите ключ для поиска: **300**

Найден! Значение = Three Hundred

Анализ

Рассмотрите вывод и обратите внимание на то, как контейнер `unordered_map`, изначально насчитывающий восемь ячеек и заполненный восемью элементами, изменяет свои размеры при вставке девятого элемента. Количество ячеек при этом увеличивается до 64. Обратите внимание на значения, возвращаемые функциями-членами `max_bucket_count()`, `load_factor()` и `max_load_factor()` в строках 14–17, а также

на то, что остальная часть кода практически не отличается от кода для контейнера `std::map`. То же самое относится и к применению метода `find()` в строке 42, который возвращает итератор, который, как и у контейнера `std::map`, следует сравнить со значением, возвращаемым `end()`, чтобы удостовериться в успехе операции поиска.

ВНИМАНИЕ!

Не следует полагаться на порядок элементов в контейнере `unordered_map` — как свидетельствует его название, данные хранятся в нем в неупорядоченном по ключу виде. Порядок одних элементов относительно других в отображении зависит от многих факторов, включая значение ключа, порядок вставки, количество ячеек и т.д.

Данные контейнеры оптимизированы для повышения производительности поиска, поэтому полагаться на порядок элементов при их обходе нельзя.

ПРИМЕЧАНИЕ

Продолжительность вставки и поиска в контейнере `std::unordered_map` (при отсутствии коллизий) практически константна и не зависит от количества содержащихся в нем элементов. Но это не обязательно делает контейнер `std::unordered_map` предпочтительнее контейнера `std::map`, который обеспечивает логарифмическую сложность во всех случаях. Константная продолжительность поиска при не слишком большом количестве элементов может оказаться намного больше логарифмической.

При выборе типа контейнера имеет смысл провести испытания, моделирующие применение контейнера в реальном сценарии.

РЕКОМЕНДУЕТСЯ

Используйте отображение `map`, если вам необходимо хранить пары “ключ–значение” с уникальными ключами.

Используйте `multimap`, если вам необходимо хранить пары “ключ–значение”, ключи которых могут повторяться (например, в телефонном справочнике).

Помните, что и отображение, и мультиотображение, как и другие контейнеры STL, предоставляют метод `size()`, возвращающий количество хранимых в контейнере элементов.

Используйте контейнеры `unordered_map` и `unordered_multimap`, когда абсолютно необходима константная продолжительность вставки и поиска (обычно при очень большом количестве элементов в контейнере).

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте, что метод `multimap::count(Ключ)` сообщает количество имеющихся в контейнере пар с одним и тем же значением *Ключа*.

Не забывайте проверять успешность поиска путем сравнения результата `find()` со значением, возвращаемым функцией-членом `end()`.

Резюме

На сегодняшнем занятии рассматривались шаблонные классы контейнеров STL `map` и `multimap`, их важнейшие функции-члены и характеристики. Вы также узнали, что у этих контейнеров — логарифмическая сложность вставки и поиска и что STL предоставляет также хеш-таблицы в виде контейнеров `unordered_map` и `unordered_multimap`. Эти контейнеры демонстрируют высокую производительность операций вставки и поиска, которая не зависит от количества элементов в контейнере. Вы также узнали о важности возможности настройки критериев сортировки с помощью пользовательского предиката (листинг 20.5).

Вопросы и ответы

- **Как мне объявить контейнер `map` для хранения пар целых чисел, отсортированных в порядке убывания?**
`map<int, int>` определяет отображение целых чисел на целые числа. Оно имеет заданный по умолчанию предикат сортировки `std::less<int>`, который приводит к сортировке элементов в порядке возрастания. Для сортировки в порядке убывания следует определить отображение как `map<int, int, greater<int>>`.
- **Что будет при вставке в отображение с ключами-строками элемента с ключом "Jack" дважды?**
 Отображение не позволяет иметь элементы с одинаковыми ключами, поэтому реализация класса `std::map` не позволит вставить второй такой элемент.
- **Что нужно изменить в предыдущем примере, чтобы все-таки вставить два элемента с ключом "Jack"?**
 Реализация класса `map` позволяет хранить только уникальные значения ключей. Смените выбранный контейнер на `multimap`.
- **Какая функция-член класса `multimap` возвращает количество элементов с определенным значением в контейнере?**
 Функция `count(значение)`.
- **Используя функцию `find()`, я нашел в отображении нужный мне элемент, и у меня теперь есть итератор, указывающий на него. Как мне использовать этот итератор для изменения значения, на которое он указывает?**
 Никак. Некоторые реализации библиотеки STL могли бы позволить пользователю изменить значение элемента отображения с помощью итератора, возвращенного, например, функцией `find()`. Но так поступать неправильно. Итератор, указывающий на элемент отображения, должен использоваться как константный, даже когда реализация STL не препятствует этому. (Все сказанное относится к ключу элемента, но не к связанному с ключом значению, которое вы можете изменить, обратившись к нему с помощью синтаксиса `iterator->second`. — *Примеч. ред.*)
- **Я использую устаревший компилятор, который не поддерживает ключевое слово `auto`. Как мне объявить переменную, которая содержит возвращаемое значение метода `map::find()`?**

Итератор определяется с использованием синтаксиса

Контейнер<Тип>::iterator Имя_Переменной;

Таким образом, объявление итератора для отображения целых чисел на целые числа будет следующим:

```
std::map<int,int>::iterator pairFound = mapIntegers.find(1000);
if (pairFound != mapIntegers.end())
{    /* Выполнить определенные действия */ }
```

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Вы объявляете отображение как `map<int,int>`. Какая функция предоставляет критерий сортировки для него?
2. Как расположены элементы с ключами-дублями в мультиотображении?
3. Какая функция контейнеров `map` и `multimap` возвращает количество хранящихся в них элементов?
4. Как расположены элементы с ключами-дублями в отображении?

Упражнения

1. Необходимо написать приложение, работающее как телефонный справочник, в котором имена людей могут не быть уникальными. Какой контейнер вы выберете? Напишите его определение.
2. Вот определение шаблона `map` вашего приложения словаря:

```
map <wordProperty, string, fPredicate> mapWordDefinition;
```

Здесь `wordProperty` представляет собой следующую структуру:

```
struct wordProperty
{
    string strWord;
    bool bIsFromLatin;
};
```

Определите бинарный предикат `fPredicate`, который позволяет отображению сортировать ключи типа `wordProperty` в соответствии с содержащимся в ключе строковым атрибутом.

3. Продемонстрируйте на примере простой программы, что отображение не может хранить записи с совпадающими ключами, а мультиотображение — может.

Часть IV

Углубляемся в STL

В ЭТОЙ ЧАСТИ...

ЗАНЯТИЕ 21. Понятие о функциональных объектах

ЗАНЯТИЕ 22. Лямбда-выражения языка C++11

ЗАНЯТИЕ 23. Алгоритмы библиотеки STL

ЗАНЯТИЕ 24. Адаптивные контейнеры: стек и очередь

ЗАНЯТИЕ 25. Работа с битовыми флагами при использовании библиотеки STL

ЗАНЯТИЕ 21

Понятие о функциональных объектах

Функциональные объекты (function object), или функторы (functor), могут показаться чем-то экзотическим или пугающим, но, в сущности, вы уже должны были их видеть (а возможно, и использовали, еще не понимая этого).

На этом занятии...

- Концепция функциональных объектов
- Использование функциональных объектов как предикатов
- Реализация унарных и бинарных предикатов с использованием функциональных объектов

Концепция функциональных объектов и предикатов

На концептуальном уровне функциональные объекты — это объекты, работающие как функции. Однако на уровне реализации функциональные объекты — это объекты класса, реализующего оператор `operator()`. Хотя функции и указатели на функции также могут рассматриваться как функциональные объекты, возможность объекта класса, который реализует `operator()`, хранить свое *состояние* (т.е. значения в атрибутах класса) делает его очень полезным при работе с алгоритмами стандартной библиотеки шаблонов (STL).

Функциональные объекты, как правило, используются при работе с STL и подразделяются на следующие типы.

- *Унарная функция* (unary function). Такая функция вызывается с одним аргументом, например как $f(x)$. Когда унарная функция возвращает значение типа `bool`, она называется *предикатом* (predicate).
- *Бинарная функция* (binary function). Бинарная функция вызывается с двумя аргументами, например $f(x, y)$. Когда бинарная функция возвращает значение типа `bool`, она называется *бинарным предикатом* (binary predicate).

Функциональные объекты, возвращающие значение типа `bool`, обычно используются в алгоритмах при принятии решений. На предыдущих занятиях вы встречались с такими алгоритмами — `sort()` и `find()`. Функциональный объект, объединяющий два функциональных объекта, называется *адаптивным функциональным объектом* (adaptive function object).

Типичные приложения функциональных объектов

Для объяснения функциональных объектов можно исписать несколько страниц теорией, а можно рассмотреть и понять их работу на примере небольшого приложения. Давайте прибегнем к практическому подходу и перейдем сразу к применению функциональных объектов, или функторов, при программировании на C++!

Унарные функции

Функции с одним параметром называются унарными. Унарная функция может делать нечто очень простое, например отображать элемент на экране. Это можно реализовать, например, следующим образом:

```
// Унарная функция
template <typename elementType>
void FuncDisplayElement(const elementType & element)
{
    cout << element << ' ';
};
```

Функция `FuncDisplayElement()` получает один параметр шаблонного типа `elementType`, который выводит на консоль с помощью потока `std::cout`. Та же функция может иметь и другое представление, в котором реализация функции фактически содержится в операторе `operator()` класса или структуры:

```
// Структура, способная вести себя как унарная функция
template <typename elementType>
struct DisplayElement
{
    void operator ()(const elementType& element) const
    {
        cout << element << ' ';
    }
};
```

СОВЕТ

Обратите внимание, что `DisplayElement` является структурой. Если бы это был класс, то оператор `operator()` должен был бы быть описан с модификатором доступа `public`. Структура представляет собой класс, все члены которого открыты по умолчанию.

С алгоритмом STL `for_each` может применяться любая из этих реализаций, отображая, таким образом, содержимое коллекции на экране по одному элементу, как показано в листинге 21.1.

ЛИСТИНГ 21.1. Вывод содержимого коллекции с помощью унарной функции

```
0: #include <algorithm>
1: #include <iostream>
2: #include <vector>
3: #include <list>
4: using namespace std;
5:
6: // Структура, ведущая себя как унарная функция
7: template <typename elementType>
8: struct DisplayElement
9: {
10:     void operator ()(const elementType& element) const
11:     {
12:         cout << element << ' ';
13:     }
14: };
15:
16: int main()
17: {
18:     vector <int> numsInVec{ 0, 1, 2, 3, -1, -9, 0, -999 };
19:     cout << "Вектор содержит:" << endl;
20:
21:     for_each(numsInVec.begin(),          // Начало диапазона
```

```
22:         numsInVec.end(),           // Конец диапазона
23:         DisplayElement<int>()); // Унарный функтор
24:
25:     // Вывод списка символов
26:     list<char> charsInList{ 'a', 'z', 'k', 'd' };
27:     cout << endl << "Список содержит:" << endl;
28:
29:     for_each(charsInList.begin(),
30:             charsInList.end(),
31:             DisplayElement<char>());
32:
33:     return 0;
34: }
```

Результат

```
Вектор содержит:
0 1 2 3 -1 -9 0 -999
Список содержит:
a z k d
```

Анализ

В строках 7–14 содержится функциональный объект `DisplayElement`, реализующий оператор `operator()`. Этот функциональный объект используется алгоритмом `std::for_each()` в строках 21–23. Алгоритм `for_each()` получает три параметра: начало и конец диапазона и функцию, вызываемую для каждого элемента в этом диапазоне. Другими словами, код вызывает оператор `DisplayElement::operator()` для каждого элемента вектора `numsInVec`. В строках 29–31 демонстрируются те же возможности для списка символов.

СОВЕТ

В листинге 21.1 можно использовать вместо структуры `struct DisplayElement` обычную функцию `FuncDisplayElement`:

```
for_each(charsInList.begin(),
        charsInList.end(),
        FuncDisplayElement);
```

СОВЕТ

Стандарт C++11 вводит лямбда-выражения, являющиеся безымянными функциональными объектами.

Версия лямбда-выражения структуры `DisplayElement<T>` из листинга 21.1 делает весь код компактнее, включая определение структуры и ее применение в трех строках функции `main()`, заменяя строки 21–24 следующими:

```
// Вывод элементов с использованием лямбда-выражения
for_each(numsInVec.begin(), // Начало диапазона
         numsInVec.end(),   // Конец диапазона
         [](int& element){cout<<element<<' ';;});
// В предыдущей строке – лямбда-выражение
```

Лямбда-выражения – фантастическое усовершенствование C++, и вы непременно должны изучить их на занятии 22, “Лямбда-выражения языка C++11”. Листинг 22.1 демонстрирует использование лямбда-функции в алгоритме `for_each()` для отображения содержимого контейнера вместо функционального объекта, как в листинге 21.1.

Реальное преимущество использования функционального объекта, реализованного в структуре, становится очевидным, когда объект структуры используется для хранения информации. Это то, чего не может обеспечить функция `FuncDisplayElement()`. Дело в том, что структура способна не только реализовать `operator()`, но и иметь данные-члены. Вот несколько измененная версия кода, в которой используются атрибуты структуры:

```
template <typename elementType>
struct DisplayElementKeepCount
{
    int count;

    DisplayElementKeepCount() // Конструктор
    {
        count = 0;
    }

    void operator ()(const elementType& element)
    {
        ++count;
        cout << element << ' ';
    }
};
```

В приведенном фрагменте структура `DisplayElementKeepCount` немного модифицирована по сравнению с предыдущей версией. Оператор `operator()` больше не является константной функцией-членом, поскольку он выполняет инкремент значения переменной-члена `count` (а следовательно, изменяет ее), используемой для хранения количества вызовов для отображения данных. Такой подсчет возможен благодаря открытому атрибуту `count`. В листинге 21.2 показано применение функционального объекта, способного хранить состояние.

ЛИСТИНГ 21.2. Использование функционального объекта, хранящего состояние

```
0: #include<algorithm>
1: #include<iostream>
2: #include<vector>
```

```
3: using namespace std;
4:
5: template<typename elementType>
6: struct DisplayElementKeepCount
7: {
8:     int count;
9:
10:    DisplayElementKeepCount() : count(0) {} // Конструктор
11:
12:    void operator()(const elementType& element)
13:    {
14:        ++count;
15:        cout << element<< ' ';
16:    }
17: };
18:
19: int main()
20: {
21:     vector<int> numsInVec{ 22, 2017, -1, 999, 43, 901 };
22:     cout << "Вывод содержимого вектора: "<< endl;
23:
24:     DisplayElementKeepCount<int> result;
25:     result = for_each(numsInVec.begin(),
26:                       numsInVec.end(),
27:                       DisplayElementKeepCount<int>());
28:
29:     cout << endl << "Функтор вызван " << result.count << " раз.";
30:
31:     return 0;
32: }
```

Результат

```
Вывод содержимого вектора:
22 2017 -1 999 43 901
Функтор вызван 6 раз.
```

Анализ

Самое большое различие между этим примером и листингом 21.1 заключается в использовании структуры `DisplayElementKeepCount` в качестве возвращаемого значения алгоритма `for_each()`. Оператор `operator()`, реализованный в структуре `DisplayElementKeepCount`, вызывается алгоритмом `for_each()` для каждого элемента в контейнере. Этот оператор выводит на экран элемент и увеличивает внутренний счетчик, хранящийся в переменной `count`. После завершения работы алгоритма `for_each()` возвращенный им объект используется в строке 29 для вывода количества обработанных элементов. Обратите внимание, что использование в этом случае обычной функции вместо реализованного в структуре оператора не позволило бы использовать счетчик так просто.

Унарный предикат

Унарная функция, которая возвращает значение типа `bool`, является предикатом и помогает алгоритмам STL принимать решения. В листинге 21.3 приведен пример предиката, определяющий, является ли вводимый элемент кратным исходному значению.

ЛИСТИНГ 21.3. Унарный предикат, определяющий, является ли одно число кратным другому

```
0: // Структура, выступающая унарным предикатом
1: template <typename numberType>
2: struct IsMultiple
3: {
4:     numberType Divisor;
5:
6:     IsMultiple(const numberType& divisor)
7:     {
8:         Divisor = divisor;
9:     }
10:
11:     bool operator ()(const numberType& element) const
12:     {
13:         // Проверка, кратен ли аргумент делителю
14:         return ((element % Divisor) == 0);
15:     }
16: };
```

Анализ

Здесь оператор `operator()` возвращает тип `bool` и может работать в качестве унарного предиката. Структура имеет конструктор и инициализируется значением делителя. Это значение хранится в объекте, а затем используется для определения, делится ли на него переданный в качестве аргумента элемент. Как можно заметить, в реализации оператора `()` используется оператор деления по модулю `%`, который возвращает остаток от деления на значение `Divisor`. Предикат сравнивает этот остаток с нулем, чтобы проверить кратность чисел.

В листинге 21.4, как и в листинге 21.3, предикат используется для определения кратности чисел заданному пользователем делителю.

ЛИСТИНГ 21.4. Использование унарного предиката `IsMultiple` с алгоритмом `std::find_if()` для поиска элемента, кратного заданному пользователем делителю

```
0: #include <algorithm>
1: #include <vector>
2: #include <iostream>
3: using namespace std;
4: // Вставка кода из листинга 21.3
5:
```

```
6: int main()
7: {
8:     vector<int> numsInVec{ 25, 26, 27, 28, 29, 30, 31 };
9:     cout << "Вектор содержит: 25, 26, 27, 28, 29, 30, 31" << endl;
10:
11:     cout << "Введите делитель (> 0): ";
12:     int divisor = 2;
13:     cin >> divisor;
14:
15:     // Поиск первого кратного делителю
16:     auto element = find_if(numsInVec.begin(),
17:                             numsInVec.end(),
18:                             IsMultiple<int>(divisor));
19:
20:     if (element != numsInVec.end())
21:     {
22:         cout << "Первый кратный " << divisor;
23:         cout << " элемент - " << *element << endl;
24:     }
25:
26:     return 0;
27: }
```

Результат

Вектор содержит: 25, 26, 27, 28, 29, 30, 31

Введите делитель (> 0): 4

Первый кратный 4 элемент - 28

Анализ

Пример начинается с простого контейнера — вектора целых чисел. Применение унарного предиката осуществляется в алгоритме поиска `find_if()`, показанного в строках 16–18. Функциональный объект `IsMultiple<int>()` инициализируется предоставляемым пользователем значением делителя, которое сохраняется в переменной-члене `Divisor`. Алгоритм `find_if()` вызывает оператор `IsMultiple::operator()` унарного предиката для каждого элемента в указанном диапазоне. Когда `operator()` возвращает значение `true` (что происходит, когда элемент делится без остатка на 4), алгоритм `find_if()` возвращает итератор `element`, указывающий на этот элемент. Результат вызова `find_if()` сравнивается с результатом вызова метода `end()` контейнера, чтобы удостовериться в успешности поиска элемента (строка 20). Затем полученный итератор `element` используется для отображения значения (строка 23).

СОВЕТ

Чтобы увидеть, как применение лямбда-выражений повышает компактность программы, представленной в листинге 21.4, взгляните на листинг 22.3 занятия 22, “Лямбда-выражения языка C++11”.

Унарные предикаты применяются в большом количестве алгоритмов STL, таких как `std::partition()`, позволяющий разделить диапазон с помощью предиката, или `stable_partition` (который делает то же самое с сохранением относительного порядка разделяемых элементов). Еще одним примером могут служить функции поиска, такие как `std::find_if()`, и функции наподобие `std::remove_if()`, позволяющие удалять из определенного диапазона элементы, удовлетворяющие предикату.

Бинарные функции

Функции типа $f(x, y)$ полезны, в частности, когда они возвращают некоторое значение, вычисляемое на основании полученных аргументов. Такие бинарные функции применяются для вычисления арифметических действий с двумя операндами, например таких, как сложение, умножение, вычитание и т.д. Типичная бинарная функция, возвращающая произведение входных аргументов, может быть написана следующим образом:

```
template <typename elementType>
class Multiply
{
public:
    elementType operator ()(const elementType& elem1,
                           const elementType& elem2)
    {
        return (elem1 * elem2);
    }
};
```

Представляющий интерес код находится в `operator()`, который получает два аргумента и возвращает их произведение. Подобные бинарные функции используются в таких алгоритмах, как `std::transform()`, в которых их можно, например, использовать для перемножения содержимого двух контейнеров. В листинге 21.5 показано применение такого бинарного функтора в алгоритме `std::transform()`.

ЛИСТИНГ 21.5. Использование бинарного функтора для умножения двух диапазонов

```
0: #include <vector>
1: #include <iostream>
2: #include <algorithm>
3:
4: template <typename elementType>
5: class Multiply
6: {
7:     public:
8:         elementType operator()(const elementType& elem1,
9:                                const elementType& elem2)
10:        {
11:            return (elem1 * elem2);
12:        }
13: };
```



```
14:
15: int main()
16: {
17:     using namespace std;
18:
19:     vector <int> multiplicands{ 0, 1, 2, 3, 4 };
20:     vector <int> multipliers{ 100, 101, 102, 103, 104 };
21:
22:     // Третий контейнер хранит результат умножения
23:     vector <int> vecResult;
24:
25:     // Готовим место для результата умножения
26:     vecResult.resize(multipliers.size());
27:     transform(multiplicands.begin(), // Диапазон множимых
28:              multiplicands.end(),    // Конец диапазона
29:              multipliers.begin(),    // Диапазон множителей
30:              vecResult.begin(),      // Результаты
31:              Multiply <int>() );     // Операция умножения
32:
33:     cout << "Первый вектор:" << endl;
34:     for(size_t index = 0; index < multiplicands.size(); ++index)
35:         cout << multiplicands[index] << ' ';
36:     cout << endl;
37:
38:     cout << "Второй вектор:" << endl;
39:     for(size_t index = 0; index < multipliers.size(); ++index)
40:         cout << multipliers[index] << ' ';
41:     cout << endl;
42:
43:     cout << "Результат умножения:" << endl;
44:     for(size_t index = 0; index < vecResult.size(); ++index)
45:         cout << vecResult[index] << ' ';
46:
47:     return 0;
48: }
```

Результат

Первый вектор:

0 1 2 3 4

Второй вектор:

100 101 102 103 104

Результат умножения:

0 101 204 309 416

Анализ

В строках 4–13 содержится класс `Multiply`, показанный в приведенном выше фрагменте кода. В данном примере алгоритм `std::transform()` используется для поэлементного перемножения содержимого двух диапазонов и сохранения результата вычисления в третьем. Рассматриваемые диапазоны содержатся в объектах `multiplicands`, `multipliers` и `vecResult`, все из которых представляют собой объекты класса `std::vector`. Другими словами, функция `std::transform()` в строках 27–31 используется для умножения каждого элемента вектора `multiplicands` на соответствующий элемент вектора `multipliers` и сохраняет результат умножения в векторе `vecResult`. Само умножение осуществляется бинарной функцией `Multiply::operator()`, которая вызывается для каждого элемента исходных диапазонов векторов. Возвращаемое значение `operator()` сохраняется в векторе `vecResult`.

Таким образом, этот пример демонстрирует применение бинарных функций для выполнения арифметических операций с элементами в контейнерах STL. Пример, приведенный далее, также использует алгоритм `std::transform()`, но применяет его для преобразования строки в строку строчных символов с помощью функции `tolower()`.

Бинарный предикат

Бинарным предикатом обычно называется функция, которая получает два аргумента и возвращает значение типа `bool`. Эти функции находят применение в таких алгоритмах STL, как `std::sort()`. Листинг 21.6 демонстрирует применение бинарного предиката, который сравнивает две строки после их перевода в нижний регистр. Такой предикат может применяться, например, при выполнении не зависящей от регистра сортировки вектора строк.

ЛИСТИНГ 21.6. Бинарный предикат для сортировки строк, не зависящей от регистра

```
0: #include <algorithm>
1: #include <string>
2: using namespace std;
3:
4: class CompareStringNoCase
5: {
6: public:
7:     bool operator ()(const string& str1, const string& str2) const
8:     {
9:         string str1LowerCase;
10:
11:         // Выделение памяти
12:         str1LowerCase.resize(str1.size());
13:
14:         // Преобразование всех символов в нижний регистр
15:         transform(str1.begin(), str1.end(),
16:                 str1LowerCase.begin(), ::tolower);
17:
```

```
18:         string str2LowerCase;
19:         str2LowerCase.resize(str2.size());
20:         transform(str2.begin(),str2.end (),
21:                 str2LowerCase.begin(), ::tolower);
22:
23:         return (str1LowerCase < str2LowerCase);
24:     }
25: };
```

Анализ

Бинарный предикат, реализованный в `operator()`, сначала переводит введенные строки в нижний регистр, используя алгоритм `std::transform()`, как показано в строках 15 и 20, а затем использует оператор сравнения строк `<` для возврата результата сравнения.

Вы можете использовать этот бинарный предикат с алгоритмом `std::sort()` для сортировки динамического массива, содержащегося в векторе строк, как показано в листинге 21.7.

ЛИСТИНГ 21.7. Использование функционального объекта класса

`CompareStringNoCase` для не зависящей от регистра сортировки вектора строк

```
0: // Здесь вставьте код класса CompareStringNoCase из листинга 21.6
1: #include <vector>
2: #include <iostream>
3:
4: template <typename T>
5: void DisplayContents(const T& container)
6: {
7:     for(auto element = container.cbegin();
8:         element != container.cend();
9:         ++element )
10:         cout << *element << endl;
11: }
12:
13: int main()
14: {
15:     // Определение вектора строк
16:     vector <string> vecNames;
17:
18:     // Вставка в вектор нескольких имен
19:     vecNames.push_back("jim");
20:     vecNames.push_back("Jack");
21:     vecNames.push_back("Sam");
22:     vecNames.push_back("Anna");
23:
24:     cout << "Имена в порядке вставки:" << endl;
```

```
25:     DisplayContents(vecNames);
26:
27:     cout << "Имена после сортировки по умолчанию:" << endl;
28:     sort(vecNames.begin(), vecNames.end());
29:     DisplayContents(vecNames);
30:
31:     cout << "Имена после сортировки с предикатом:" << endl;
32:     sort(vecNames.begin(), vecNames.end(), CompareStringNoCase());
33:     DisplayContents(vecNames);
34:
35:     return 0;
36: }
```

Результат

Имена в порядке вставки:

jim
Jack
Sam
Anna

Имена после сортировки по умолчанию:

Anna
Jack
Sam
jim

Имена после сортировки с предикатом:

Anna
Jack
jim
Sam

Анализ

Вывод отображает содержимое вектора на трех этапах. На первом содержимое отображается в порядке вставки. На втором этапе, после сортировки в строке 28 с использованием заданного по умолчанию предиката сортировки `less<T>`, вывод демонстрирует, что `jim` располагается не после `Jack`, поскольку эта сортировка зависит от регистра благодаря оператору `string::operator<`. Последняя сортировка в строке 32 использует класс предиката `CompareStringNoCase<>` (реализованный в листинге 21.6), который гарантирует, что `jim` будет следовать после `Jack` несмотря на различие в регистре.

Бинарные предикаты применяются во множестве алгоритмов STL. Например, в алгоритме `std::unique()`, удаляющем совпадающие соседние элементы, в алгоритмах `std::sort()` и `std::stable_sort()`, выполняющих сортировку, в алгоритме `std::transform()`, позволяющем выполнить операцию над двумя диапазонами, и во многих других алгоритмах STL, нуждающихся в бинарном предикате.

Резюме

На этом занятии вы познакомились с функторами (или функциональными объектами). Вы узнали, что функциональные объекты, реализованные в виде структуры или класса, полезнее простых функций, поскольку могут использоваться и для хранения состояния. Вы также изучили предикаты, которые являются частным случаем функциональных объектов, и ознакомились с некоторыми практическими примерами, демонстрирующими удобство их применения.

Вопросы и ответы

- **Предикат — это специальная категория функциональных объектов. Что делает его таким особенным?**

Предикаты всегда возвращают логическое значение.

- **Какой функциональный объект следует использовать при вызове такой функции, как `remove_if()`?**

Вы должны использовать унарный предикат, который может получить в конструкторе дополнительную информацию, используемую при вычислении оператора `operator()`.

- **Какой функциональный объект я должен использовать для контейнера `map`?**
- Бинарный предикат.

- **Может ли простая функция без возвращаемого значения использоваться как функтор?**

Да. Функция без возвращаемых значений вполне может делать что-то полезное, например выводить на экран входные данные.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Как называется унарная функция, возвращающая значение типа `bool`?
2. Есть ли польза от функционального объекта, который не изменяет данные и не возвращает значения типа `bool`? Можете привести пример?
3. Каково определение термина *функциональный объект*?

Упражнения

1. Напишите унарную функцию, которая применяется в алгоритме `std::for_each()` для вывода удвоенного входного параметра.
2. Дополните этот предикат так, чтобы можно было вывести количество его вызовов.
3. Напишите бинарный предикат, обеспечивающий сортировку в порядке возрастания.

ЗАНЯТИЕ 22

Лямбда-выражения языка C++11

Лямбда-выражения (lambda expressions) — это компактное средство определения и создания функциональных объектов без имени, введенный в стандарте C++11.

На этом занятии...

- Как создать лямбда-выражение
- Использование лямбда-выражений в качестве предикатов
- Обобщенные лямбда-выражения C++14
- Как создать лямбда-выражение, способное хранить состояние, и работать с ним

Что такое лямбда-выражение

Лямбда-выражение (или просто лямбда) можно считать компактной версией безымянной структуры (или класса) с открытым оператором `operator()`. В этом смысле лямбда-выражение — это функциональный объект, подобный представленным на занятии 21, “Понятие о функциональных объектах”. Прежде чем переходить к анализу разработки лямбда-функций, рассмотрим функциональный объект из листинга 21.1:

```
// Структура, ведущая себя как унарная функция
template <typename elementType>
struct DisplayElement
{
    void operator ()(const elementType& element) const
    {
        cout << element << ' ';
    }
};
```

Этот функциональный объект отображает на экране с использованием потока `cout` элемент и обычно используется в таких алгоритмах, как `std::for_each()`:

```
// Отобразить массив целых чисел
for_each(vecIntegers.begin(),    // Начало диапазона
        vecIntegers.end(),      // Конец диапазона
        DisplayElement<int>()); // Унарный функциональный объект
```

Лямбда-выражение позволяет компактно записать код, включив в вызов определение функционального объекта:

```
// Отобразить массив целых чисел, используя лямбда-выражения
for_each(vecIntegers.begin(),    //Начало диапазона
        vecIntegers.end(),      //Конец диапазона
        [](const int&element){cout<<element<<' '}); //Лямбда-выражение
```

Когда компилятор встречает лямбда-выражение, в данном случае это

```
[](const int&element){cout<<element<<' '};
```

он автоматически разворачивает его в представление, подобное структуре `DisplayElement<int>`:

```
struct NoName
{
    void operator ()(const int& element) const
    {
        cout << element << ' ';
    }
};
```

Как определить лямбда-выражение

Определение лямбда-выражения должно начинаться с квадратных скобок ([]). Эти скобки, по существу, говорят компилятору, что началось лямбда-выражение. За ними следует список параметров, являющийся таким же списком параметров, как и тот, который вы предоставили бы своей реализации оператора `operator()`, если бы не использовали лямбда-выражение.

Лямбда-выражение для унарной функции

Лямбда-версия унарного оператора `operator (Type)`, получающего один параметр, имела бы следующий вид:

```
[] (Type paramName) { /* Код лямбда-выражения */ }
```

Обратите внимание, что при необходимости параметр можно передать по ссылке:

```
[] (Type& paramName) { /* Код лямбда-выражения */ }
```

Листинг 22.1 демонстрирует применение лямбда-функции для отображения содержимого контейнера стандартной библиотеки шаблонов (STL) с использованием алгоритма `for_each()`.

ЛИСТИНГ 22.1. Вывод элементов контейнера с помощью алгоритма `for_each()` с лямбда-выражением

```
0: #include <algorithm>
1: #include <iostream>
2: #include <vector>
3: #include <list>
4:
5: using namespace std;
6:
7: int main()
8: {
9:     vector<int> numsInVec{ 101, -4, 500, 21, 42, -1 };
10:
11:     list<char> charsInList{ 'a', 'h', 'z', 'k', 'l' };
12:     cout << "Вывод вектора с использованием лямбды:" << endl;
13:
14:     // Вывод массива целых чисел
15:     for_each(numsInVec.cbegin(),
16:             numsInVec.cend(),
17:             [](const int& element){cout << element << ' ';});
18:
19:     cout << endl;
20:     cout << "Вывод списка с использованием лямбды:" << endl;
21:
22:     // Вывод списка символов
```

```
23:     for_each(charsInList.cbegin(),
24:               charsInList.cend(),
25:               [](auto& element){cout << element << ' '; });
26:
27:     return 0;
28: }
```

Результат

Вывод вектора с использованием лямбды:

101 -4 500 21 42 -1

Вывод списка с использованием лямбды:

a h z k l

Анализ

Интерес представляют два лямбда-выражения в строках 17 и 25. Они очень похожи, если не учитывать тип входного параметра, поскольку они приспособлены к типу элементов контейнеров, с которыми работают. Первое лямбда-выражение получает один параметр типа `int`, поскольку оно используется для поэлементного вывода вектора целых чисел, тогда как второе получает параметр типа `char` (автоматически выводимого компилятором), поскольку предназначено для отображения элементов типа `char`, хранящихся в контейнере `std::list`.

СОВЕТ

Вы можете заметить, что второе лямбда-выражение в листинге 22.1 немного отличается от первого:

```
for_each(charsInList.cbegin(),
          charsInList.cend(),
          [](auto& element){cout << element << ' '; });
```

Это лямбда-выражение использует возможности автоматического вывода типа компилятором, которое вызывается с помощью ключевого слова `auto`. Это усовершенствование лямбда-выражений, поддерживаемое C++14-совместимыми компиляторами. Компилятор трактует это лямбда-выражение как

```
for_each(charsInList.cbegin(),
          charsInList.cend(),
          [](const char& element){cout << element << ' '; });
```

ПРИМЕЧАНИЕ

Вывод листинга 22.1 аналогичен выводу листинга 21.1. Фактически эта программа — просто лямбда-версия листинга 21.1, в котором был использован функциональный объект `DisplayElement<T>`.

Сравнивая эти два листинга, можно прийти к выводу, что лямбда-функции обладают серьезным потенциалом, позволяющим сделать код C++ проще и компактнее.

Лямбда-выражение для унарного предиката

Предикат позволяет принимать решения. Унарный предикат — это унарное выражение, которое возвращает значение типа `bool` (`true` или `false`). Лямбда-выражения также могут возвращать значения. Например, следующий код представляет собой лямбда-выражение, которое возвращает значение `true` для четных чисел:

```
[] (int& num) {return ((num % 2) == 0); }
```

Природа возвращаемого значения в данном случае указывает компилятору, что лямбда-выражение возвращает тип `bool`.

Вы можете использовать данное лямбда-выражение, являющееся унарным предикатом, в таких алгоритмах, как `std::find_if()`, для поиска четных чисел в коллекции. Соответствующий пример приведен в листинге 22.2.

ЛИСТИНГ 22.2. Поиск четных чисел в коллекции с использованием лямбда-выражения в качестве унарного предиката

```
0: #include<algorithm>
1: #include<vector>
2: #include<iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     vector<int> numsInVec{ 25, 101, 2017, -50 };
8:
9:     auto evenNum = find_if(numsInVec.cbegin(),
10:                          numsInVec.cend(),
11:                          [](const int& num){return ((num % 2) == 0); } );
12:
13:     if (evenNum != numsInVec.cend())
14:         cout << "Четное число найдено: " << *evenNum << endl;
15:
16:     return 0;
17: }
```

Результат

Четное число найдено: -50

Анализ

Лямбда-функция, работающая как унарный предикат, представлена в строке 11. Алгоритм `find_if()` вызывает унарный предикат для каждого элемента диапазона. Когда предикат возвращает значение `true`, алгоритм `find_if()` сообщает о найденном

элементе, возвращая итератор, указывающий на найденный элемент. В нашем случае предикат (лямбда-выражение) возвращает значение `true`, когда алгоритм `find_if()` встречает четное целое число (т.е. остаток от его деления на 2 равен нулю).

ПРИМЕЧАНИЕ

В листинге 22.2 демонстрируется не только лямбда-выражение, работающее как унарный предикат, но и использование ключевого слова `const` в лямбда-выражениях.

Не забывайте использовать его для входных параметров, особенно если это ссылки, чтобы избежать внесения непреднамеренных изменений в значения элементов в контейнере.

Лямбда-выражения с состоянием и списки захвата [. . .]

В листинге 22.2 был создан унарный предикат, который возвращал значение `true`, если целое число делилось на 2, т.е. было четным. Но что если нужна некоторая более обобщенная функция, которая возвращает значение `true`, если число делится на значение, предоставленное пользователем? Это значение необходимо хранить в лямбда-выражении как “состояние”:

```
int Divisor = 2; // Исходное значение
...
auto element = find_if(Начало_диапазона,
                       Конеч_диапазона,
                       [Divisor](int dividend){return (dividend % Divisor) == 0; } );
```

Список аргументов, переданный в виде *переменных состояния* (state variable) в квадратных скобках ([. . .]), называется также *списком захвата* (capture list) лямбда-выражения.

ПРИМЕЧАНИЕ

Такое лямбда-выражение — это однострочный эквивалент 16 строк кода в листинге 21.3, определяющем унарный предикат `struct IsMultiple<>`. Таким образом, введенные стандартом C++11 лямбда-выражения резко повышают эффективность программирования и выразительность программ на C++.

В листинге 22.3 демонстрируется применение унарного предиката с переменной состояния для поиска в коллекции числа, кратного предоставленному пользователем делителю.

ЛИСТИНГ 22.3. Использование лямбда-выражений с сохранением состояния

```
0: #include <algorithm>
1: #include <vector>
2: #include <iostream>
```

```
3: using namespace std;
4:
5: int main()
6: {
7:     vector<int> numsInVec{25, 26, 27, 28, 29, 30, 31};
8:     cout << "Вектор: {25, 26, 27, 28, 29, 30, 31}";
9:
10:    cout << endl << "Введите делитель (> 0): ";
11:    int divisor = 2;
12:    cin >> divisor;
13:
14:    // Поиск первого элемента, кратного divisor
15:    vector<int>::iterator element;
16:    element = find_if(numsInVec.begin(),
17:                     numsInVec.end(),
18:                     [divisor](int dividend){return (dividend % divisor) == 0;});
19:
20:    if (element != numsInVec.end())
21:    {
22:        cout << "Первый элемент, делящийся на " << divisor;
23:        cout << ": " << *element << endl;
24:    }
25:
26:    return 0;
27: }
```

Результат

```
Вектор: {25, 26, 27, 28, 29, 30, 31}
Введите делитель (> 0): 4
Первый элемент, делящийся на 4: 28
```

Анализ

Лямбда-выражение, хранящее состояние и работающее как предикат, находится в строке 18. Переменная состояния `divisor` аналогична переменной `IsMultiple::Divisor`, которую вы видели в листинге 21.3. Следовательно, переменные состояния сродни членам класса функционального объекта, который вы использовали до C++11. Таким образом, теперь вы можете передать состояние в лямбда-функцию и настроить ее применение с использованием этого состояния.

ПРИМЕЧАНИЕ

Листинг 22.3 использует лямбда-эквивалент функционального объекта из листинга 21.4, но без класса. Возможности лямбда-выражений C++11 сэкономили нам 16 строк кода.

Обобщенный синтаксис лямбда-выражений

Лямбда-выражение всегда начинается с квадратных скобок и может быть настроено так, чтобы получать несколько переменных состояния, разделенных запятыми в списке захвата [...]:

```
[Переменная1, Переменная2] (Тип& Параметр) { /* Код лямбда-выражения */ }
```

Если эти переменные состояния должны изменяться в пределах тела лямбда-выражения, добавьте ключевое слово `mutable`:

```
[Переменная1, Переменная2] (Тип& Параметр) mutable { /* Код */ }
```

Обратите внимание, что в этом случае передаваемые в списке захвата [] переменные могут изменяться в пределах лямбда-выражения, но вовне эти изменения не передаются. Если необходимо, чтобы внесенные в пределах лямбда-выражения изменения переменных распространялись также вовне, следует использовать ссылки:

```
[&Переменная1, &Переменная2] (Тип& Параметр) { /* Код */ }
```

Лямбда-выражения могут получать несколько входных параметров, отделяемых запятыми:

```
[Переменная] (Тип1& Параметр1, Тип2& Параметр2) { /* Код */ }
```

Если вы хотите указать тип возвращаемого значения и не создавать неоднозначности для компилятора, используйте оператор `->` так, как показано далее:

```
[Переменная1, Переменная2] (Тип& Параметр) -> Возвращаемый_Тип  
{ /* Код лямбда-выражения */ }
```

И наконец составная инструкция {} может содержать несколько инструкций, разделенных точкой с запятой (;), как показано ниже:

```
[Переменная1, Переменная2] (Тип& Параметр) -> Возвращаемый_Тип  
{ Инструкция1; Инструкция2; Инструкция3; return Значение; }
```

ПРИМЕЧАНИЕ

В случае, если лямбда-выражение распространяется на несколько строк, тип возвращаемого значения лучше указать явно — для большей удобочитаемости.

В листинге 22.5 показана лямбда-функция, определяющая тип возвращаемого значения и охватывающая несколько строк.

Таким образом, лямбда-функция — это компактная, полнофункциональная замена функционального объекта, например такого, как следующий:

```
template<typename Тип1, typename Тип2>  
struct IsNowTooLong  
{  
    // Переменные состояния
```

```

Тип1 var1;
Тип2 var2;

// Конструктор
IsNowTooLong(const Тип1& in1, Тип2& in2):var1(in1),var2(in2){};

// Фактическое предназначение
Возвращаемый_тип operator()
{
    Инструкция1;
    Инструкция2;
    return (Значение_или_выражение);
}
};

```

Лямбда-выражение для бинарной функции

Бинарная функция получает два параметра и (необязательно) возвращает значение. Эквивалентное лямбда-выражение выглядит следующим образом:

```
[...](Тип1& Параметр1, Тип2& Параметр2) { /* Код */ }
```

В листинге 22.4 показана лямбда-функция, поэлементно перемножающая два вектора равных размеров. Она использует алгоритм `std::transform()` и сохраняет результат в третьем векторе.

ЛИСТИНГ 22.4. Лямбда-выражение в качестве бинарной функции

```

0: #include <vector>
1: #include <iostream>
2: #include <algorithm>
3:
4: int main()
5: {
6:     using namespace std;
7:
8:     vector<int> vecMultiplicand{ 0, 1, 2, 3, 4 };
9:     vector<int> vecMultiplier{ 100, 101, 102, 103, 104 };
10:
11:     // Для хранения результата умножения
12:     vector<int> vecResult;
13:
14:     // Подготовка места для размещения результата
15:     vecResult.resize(vecMultiplier.size());
16:
17:     transform(vecMultiplicand.begin(), // Диапазон множимых
18:              vecMultiplicand.end(),   // Конец диапазона
19:              vecMultiplier.begin(),   // Множители

```



```
20:         vecResult.begin(),           // Результаты
21:         [](int a, int b){return a*b;});
22:
23:     cout << "Содержимое первого вектора:" << endl;
24:     for(size_t index=0; index<vecMultiplicand.size(); ++index)
25:         cout << vecMultiplicand[index] << ' ';
26:     cout << endl;
27:
28:     cout << "Содержимое второго вектора:" << endl;
29:     for(size_t index=0; index<vecMultiplier.size(); ++index)
30:         cout << vecMultiplier[index] << ' ';
31:     cout << endl;
32:
33:     cout << "Результат умножения:" << endl;
34:     for(size_t index=0; index<vecResult.size(); ++index)
35:         cout << vecResult[index] << ' ';
36:
37:     return 0;
38: }
```

Результат

Содержимое первого вектора:

0 1 2 3 4

Содержимое второго вектора:

100 101 102 103 104

Результат умножения:

0 101 204 309 416

Анализ

Лямбда-выражение используется в строке 21 в качестве параметра алгоритма `std::transform()`. Этот алгоритм принимает два диапазона и применяет преобразование, определяемое бинарной функцией. Возвращаемые значения бинарной функции сохраняются в выходном контейнере. В данном случае бинарная функция представляет собой лямбда-выражение, принимающее два целых числа и возвращающее результат их умножения в качестве возвращаемого значения. Это возвращаемое значение сохраняется алгоритмом `std::transform()` в векторе `vecResult`. Вывод демонстрирует содержимое двух контейнеров и результат перемножения их элементов.

ПРИМЕЧАНИЕ

В листинге 22.4 демонстрируется лямбда-эквивалент функционального объекта `Multiply<>` из листинга 21.5.

Лямбда-выражение для бинарного предиката

Бинарная функция, возвращающая значение `true` или `false` для принятия решения, называется *бинарным предикатом*. Такие предикаты применяются, например, в алгоритмах сортировки, таких как `std::sort()`, которые вызывают бинарный предикат для двух значений в контейнере, чтобы узнать, какой из них должен располагаться перед другим. Обобщенный синтаксис бинарного предиката имеет следующий вид:

```
[...](Тип1& Параметр1, Тип2& Параметр2)
{ /* Код */ return Логическое_значение; }
```

В листинге 22.5 продемонстрировано применение лямбда-выражения для сортировки, нечувствительной к регистру.

ЛИСТИНГ 22.5. Использование лямбда-выражения в качестве бинарного предиката

```
0: #include <algorithm>
1: #include <string>
2: #include <vector>
3: #include <iostream>
4: using namespace std;
5:
6: template <typename T>
7: void DisplayContents(const T& input)
8: {
9:     for(auto element = input.cbegin();
10:         element != input.cend();
11:         ++element )
12:         cout << *element << endl;
13: }
14:
15: int main()
16: {
17:     vector <string> namesInVec{ "jim", "Jack", "Sam", "Anna" };
18:
19:     cout << "Имена в порядке вставки:" << endl;
20:     DisplayContents(namesInVec);
21:
22:     cout << "Имена после сортировки по умолчанию: " << endl;
23:     sort(namesInVec.begin(), namesInVec.end());
24:     DisplayContents(namesInVec);
25:
26:     cout << "Имена после сортировки с предикатом:" << endl;
27:     sort(namesInVec.begin(), namesInVec.end(),
28:         [](const string& str1, const string& str2) -> bool
29:         {
30:             string str1LC; // LC = lowercase
31:
32:             // Выделение памяти
```

```
33:         str1LC.resize(str1.size());
34:
35:         // Преобразование в строчные
36:         transform(str1.begin(), str1.end(),
37:                 str1LC.begin(), ::tolower);
38:         string str2LC;
39:         str2LC.resize(str2.size());
40:         transform(str2.begin(), str2.end(),
41:                 str2LC.begin(), ::tolower);
42:         return (str1LC < str2LC);
43:     } // Конец лямбда-выражения
44: }; // Конец алгоритма сортировки
45:
46: DisplayContents(namesInVec);
47:
48: return 0;
49: }
```

Результат

Имена в порядке вставки:

jim
Jack
Sam
Anna

Имена после сортировки по умолчанию:

Anna
Jack
Sam
jim

Имена после сортировки с предикатом:

Anna
Jack
jim
Sam

Анализ

Здесь приведена большая лямбда-функция, охватывающая строки 28–43 и используемая в качестве третьего параметра алгоритма `std::sort()`. Этот код демонстрирует, что лямбда-функция может состоять из множества инструкций и использовать явное указание типа возвращаемого значения (`bool`), как показано в строке 28. Вывод демонстрирует содержимое вектора в порядке вставки значений, где `jim` следует перед `Jack`, содержимое вектора после сортировки по умолчанию без пользовательского предиката (строка 23), где `jim` следует после `Sam` благодаря чувствительности оператора `string::operator<` к регистру символов, и наконец версию, использующую лямбда-выражение (строки 28–42), преобразующее строки в нижний регистр перед их сравнением. При использовании такого предиката `jim` следует после `Jack`.

ПРИМЕЧАНИЕ

Необычно большое лямбда-выражение листинга 22.5 является лямбда-версией класса `CompareStringNoCase` из листинга 21.6, используемого в листинге 21.7.

Безусловно, это не оптимальное использование лямбда-выражения, поскольку функциональный объект из листинга 21.6 допускает повторное использование в других сортировках (а также в других алгоритмах, которые нуждаются в бинарном предикате).

Поэтому лямбда-выражения необходимо использовать тогда, когда они коротки, изящны и эффективны.

РЕКОМЕНДУЕТСЯ

Помните, что лямбда-выражения всегда начинаются с квадратных скобок — пустых или со списком захвата.

Помните, что, если не указано иное, переменные состояния, указанные в списке захвата, не допускают изменений. Для обеспечения возможности изменения используйте ключевое слово `mutable`.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте, что лямбда-выражения — это безымянные представления класса или структуры с оператором `operator()`.

Не забывайте использовать константные параметры при создании лямбда-выражений там, где это возможно.

Не забывайте явно указывать тип возвращаемого значения, если лямбда-выражение достаточно велико и включает множество инструкций в теле лямбда-выражения.

Не предпочитайте лямбда-выражения функциональным объектам, когда лямбда-выражения становятся слишком длинными, поскольку они переопределяются при каждом использовании и не позволяют повторно использовать код.

Резюме

На сегодняшнем занятии рассмотрено очень важное средство языка C++11 — лямбда-выражения. Вы узнали, что лямбда-выражения — это безымянные функциональные объекты, способные получать параметры, сохранять состояние, возвращать значения и включать многострочный код. Вы узнали, как использовать лямбда-выражения вместо функциональных объектов в таких алгоритмах STL, как `find()`, `sort()` или `transform()`. Лямбда-выражения делают программирование на C++ быстрым и эффективным, поэтому используйте их везде, где их применение оправдано.

Вопросы и ответы

- Всегда ли я должен предпочитать лямбда-выражения функциональным объектам?

Лямбда-выражения, состоящие из нескольких строк кода (как, например, в листинге 22.5), не в состоянии повысить эффективность программирования по сравнению с функциональным объектом, который легко можно использовать многократно.

■ **Как передаются параметры состояния лямбда-выражения — по значению или по ссылке?**

Когда лямбда-выражение создается со списком захвата следующим образом:

```
[Переменная1, .. ПеременнаяN] (Тип& Параметр, ... ) { /* Код */ }
```

переменные состояния копируются (а не передаются по ссылке). Если вы хотите использовать их как ссылочные параметры, используйте следующий синтаксис:

```
[&Переменная1, .. &ПеременнаяN] (Тип& Параметр, ... ) { /* Код */ }
```

В этом случае нужна особая осторожность, поскольку изменение переменных состояния, переданных в списке захвата, распространяется вне лямбда-выражения.

■ **Могу ли я использовать локальные переменные функции в лямбда-выражении?**

Вы можете передать локальные переменные в списке захвата. Если вы хотите захватить все переменные сразу, используйте следующий синтаксис:

```
[=] (Тип& Параметр, ... ) { /* Код */ }
```

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Как компилятор распознает начало лямбда-выражения?
2. Как передать переменные состояния лямбда-функции?
3. Как вернуть возвращаемое значение из лямбды?

Упражнения

1. Напишите бинарный предикат, который обеспечил бы сортировку в порядке убывания, в виде лямбда-выражения.
2. Напишите лямбда-функцию, которая, будучи использованной в алгоритме `for_each()`, добавляет заданное пользователем значение к значениям элементов в контейнере, например в векторе.

ЗАНЯТИЕ 23

Алгоритмы библиотеки STL

Важнейшей частью стандартной библиотеки шаблонов (STL) является набор обобщенных функций, предоставляемых заголовочным файлом `<algorithm>`, которые помогают в работе с содержимым контейнеров.

На этом занятии вы узнаете, как использование стандартных алгоритмов спасает от написания стереотипного кода, повышая вашу производительность и надежность ваших программ.

На этом занятии...

- Подсчет, поиск, копирование и удаление элементов контейнера
- Установка значений диапазонов элементов равными возвращаемым значениям генератора или предопределенному константному значению
- Сортировка или разделение элементов диапазона
- Вставка элементов в корректную позицию отсортированного диапазона

Что такое алгоритмы STL

Наиболее популярными обобщенными алгоритмами, которые находят применение в программах широкого диапазона применения, являются такие алгоритмы, как поиск, удаление, сортировка или подсчет количества элементов. STL решает эти и многие другие задачи в форме обобщенных шаблонов функций, которые взаимодействуют с контейнерами с помощью итераторов. Чтобы использовать алгоритмы STL, программист должен включить в код программы заголовочный файл `<algorithm>`.

ПРИМЕЧАНИЕ

Хотя большинство алгоритмов работают с контейнерами с помощью итераторов, не все они обязательно связаны с обработкой контейнеров и не все алгоритмы нуждаются в итераторах. Некоторые из них, такие как `swap()`, просто обменивают пару переданных им значений. Аналогично алгоритмы `min()` и `max()` также работают непосредственно со значениями.

Классификация алгоритмов STL

Алгоритмы STL можно подразделить на два обширных типа: изменяющие и не изменяющие данные, с которыми они работают.

Не изменяющие алгоритмы

Алгоритмы, которые не изменяют ни порядок, ни содержимое контейнера, называются *не изменяющими алгоритмами* (non-mutating algorithm). Некоторые из них представлены в табл. 23.1.

ТАБЛИЦА 23.1. Краткий перечень не изменяющих алгоритмов

Алгоритм	Описание
<i>Алгоритмы подсчета</i>	
<code>count()</code>	Находит в диапазоне все элементы, значения которых равны предоставленному значению
<code>count_if()</code>	Находит в диапазоне все элементы, значения которых удовлетворяют предоставленному условию
<i>Алгоритмы поиска</i>	
<code>search()</code>	Поиск в диапазоне первого вхождения заданной последовательности на основании равенства элементов (т.е. оператора <code>==</code>) или указанного бинарного предиката
<code>search_n()</code>	Поиск в диапазоне первого вхождения <code>n</code> элементов с заданным значением или удовлетворяющих заданному предикату
<code>find()</code>	Поиск в диапазоне первого элемента, равного заданному значению
<code>find_if()</code>	Поиск в диапазоне первого элемента, удовлетворяющего заданному условию

Алгоритм	Описание
<code>find_end()</code>	Поиск в диапазоне последнего вхождения элемента из заданного поддиапазона
<code>find_first_of()</code>	Поиск в диапазоне первого вхождения любого элемента целевого диапазона или (в перегруженной версии) первого вхождения элемента, удовлетворяющего заданному критерию поиска
<code>adjacent_find()</code>	Поиск в коллекции двух смежных элементов, которые равны или удовлетворяют заданному условию
<i>Алгоритмы сравнения</i>	
<code>equal()</code>	Сравнивает два элемента на равенство или использует заданный предикат для той же цели
<code>mismatch()</code>	Находит первую позицию различия в двух диапазонах элементов, используя заданный бинарный предикат
<code>lexicographical_compare()</code>	Сравнивает элементы двух последовательностей, чтобы определить, какая из них меньше

Изменяющие алгоритмы

Изменяющие алгоритмы (mutating algorithm) изменяют содержимое или порядок последовательности, с которой они работают. Некоторые из наиболее полезных изменяющих алгоритмов, предоставляемых библиотекой STL, приведены в табл. 23.2.

ТАБЛИЦА 23.2. Краткий перечень изменяющих алгоритмов

Алгоритм	Описание
<i>Алгоритмы инициализации</i>	
<code>fill()</code>	Присваивает заданное значение каждому элементу в указанном диапазоне
<code>fill_n()</code>	Присваивает заданное значение первым <i>n</i> элементам в указанном диапазоне
<code>generate()</code>	Присваивает возвращаемое значение заданного функционального объекта каждому элементу в указанном диапазоне
<code>generate_n()</code>	Присваивает сгенерированное функцией значение определенному количеству элементов в указанном диапазоне
<i>Алгоритмы изменения</i>	
<code>for_each()</code>	Выполняет операцию с каждым из элементов диапазона. Когда переданный функциональный аргумент изменяет элементы диапазона, алгоритм <code>for_each()</code> становится изменяющим
<code>transform()</code>	Применяет определенную унарную функцию к каждому элементу в указанном диапазоне
<i>Алгоритмы копирования</i>	
<code>copy()</code>	Копирует один диапазон в другой
<code>copy_backward()</code>	Копирует один диапазон в другой, упорядочивая его элементы в обратном порядке

Алгоритм	Описание
<i>Алгоритмы удаления</i>	
<code>remove()</code>	Удаляет элемент с заданным значением из указанного диапазона
<code>remove_if()</code>	Удаляет элемент, удовлетворяющий заданному унарному предикату, из указанного диапазона
<code>remove_copy()</code>	Копирует все элементы исходного диапазона в результирующий, кроме элементов с определенным значением
<code>remove_copy_if()</code>	Копирует все элементы исходного диапазона в результирующий, кроме элементов, удовлетворяющих заданному унарному предикату
<code>unique()</code>	Сравнивает смежные элементы в диапазоне и удаляет соседствующие дубликаты. Перегруженная версия использует бинарный предикат
<code>unique_copy()</code>	Копирует все элементы из исходного диапазона в результирующий, кроме соседствующих дубликатов
<i>Алгоритмы замены</i>	
<code>replace()</code>	Заменяет в диапазоне каждый элемент, соответствующий указанному значению, заданным значением
<code>replace_if()</code>	Заменяет в диапазоне каждый элемент, удовлетворяющий указанному условию, заданным значением
<i>Алгоритмы сортировки</i>	
<code>sort()</code>	Сортирует элементы диапазона, используя заданный критерий сортировки, являющийся бинарным предикатом. Может изменять относительные позиции эквивалентных элементов
<code>stable_sort()</code>	Устойчивая сортировка — подобна алгоритму <code>sort()</code> , но с сохранением относительного порядка эквивалентных элементов
<code>partial_sort()</code>	Сортирует указанное количество элементов диапазона
<code>partial_sort_copy()</code>	Копирует элементы исходного диапазона в результирующий с сортировкой
<i>Алгоритмы разделения</i>	
<code>partition()</code>	Разделяет заданный диапазон на два множества: значения элементов первого удовлетворяют унарному предикату, после чего следуют не удовлетворяющие ему. Может не поддерживать относительный порядок элементов в множестве
<code>stable_partition()</code>	Разделяет заданный диапазон на два множества, как и алгоритм <code>partition()</code> , но обеспечивает сохранение исходного относительного порядка элементов

Алгоритм	Описание
<i>Алгоритмы для работы с отсортированными контейнерами</i>	
<code>binary_search()</code>	Используется для определения наличия элемента в отсортированной коллекции
<code>lower_bound()</code>	Возвращает итератор, указывающий на первую позицию, куда потенциально может быть вставлен элемент отсортированной коллекции на основании его значения или заданного бинарного предиката
<code>upper_bound()</code>	Возвращает итератор, указывающий на последнюю позицию, куда потенциально может быть вставлен элемент отсортированной коллекции на основании его значения или заданного бинарного предиката

Использование алгоритмов STL

Применение алгоритмов STL, приведенных в табл. 23.1 и 23.2, лучше всего изучать непосредственно на практическом примере. Для этого изучим подробности использования алгоритмов на приведенных ниже примерах кода.

Поиск элементов по заданному значению или условию

Для заданного контейнера, например такого, как вектор, алгоритмы STL `find()` и `find_if()` позволяют найти элемент, который равен указанному значению или удовлетворяет некоторому условию соответственно. Применение алгоритма `find()` осуществляется по следующей схеме:

```
auto element = find(numsInVec.cbegin(), // Начало диапазона
                    numsInVec.cend(),   // Конец диапазона
                    numToFind );        // Искомое значение

// Проверка успешности поиска
if (element != numsInVec.cend())
    cout << "Значение найдено!" << endl;
```

Алгоритм `find_if()` похож на `find()`, но требует предоставления унарного предиката (унарной функции, возвращающей значение `true` или `false`) в качестве третьего параметра.

```
auto evenNumber = find_if(numsInVec.cbegin(), // Начало диапазона
                           numsInVec.cend(),   // Конец диапазона
                           [](int element) { return (element % 2) == 0; } );

if (evenNumber != numsInVec.cend())
    cout << "Значение найдено!" << endl;
```

Обе функции поиска возвращают итератор, который необходимо сравнить с результатом, возвращаемым функцией-членом контейнера `end()` или `cend()`, чтобы проверить успешность операции поиска. Если проверка показала успешность поиска, этот итератор можно использовать в программе. В листинге 23.1 показано применение функции `find()` для поиска значения в векторе и функции `find_if()` для поиска первого четного числа.

ЛИСТИНГ 23.1. Использование функций `find()` и `find if()`

```
0: #include <iostream>
1: #include <algorithm>
2: #include <vector>
3:
4: int main()
5: {
6:     using namespace std;
7:     vector<int> numsInVec{ 2017, 0, -1, 42, 10101, 25 };
8:
9:     cout << "Введите искомое число: ";
10:    int numToFind = 0;
11:    cin >> numToFind;
12:
13:    auto element = find(numsInVec.cbegin(), // Начало диапазона
14:                        numsInVec.cend(),   // Конец диапазона
15:                        numToFind);         // Искомый элемент
16:
17:    // Проверка успешности поиска
18:    if (element != numsInVec.cend())
19:        cout << "Значение " << *element << " найдено!\n";
20:    else
21:        cout << "Значение " << numToFind << " не найдено.\n";
22:
23:    cout << "Поиск первого четного числа с помощью find_if:\n";
24:
25:    auto evenNum = find_if(numsInVec.cbegin(), // Начало и конец
26:                           numsInVec.cend(),   // диапазона
27:                           [](int element){return (element%2) == 0; });
28:
29:    if (evenNum != numsInVec.cend())
30:    {
31:        cout << "Число '" << *evenNum << "' в позиции [";
32:        cout << distance(numsInVec.cbegin(), evenNum) << "]"<<endl;
33:    }
34:
35:    return 0;
36: }
```

Результат

Введите искомое число: **42**
 Значение 42 найдено!
 Поиск первого четного числа с помощью `find_if`:
 Число '0' в позиции [1]

Второе выполнение:

Введите искомое число: **2016**
 Значение 2016 не найдено.
 Поиск первого четного числа с помощью `find_if`:
 Число '0' в позиции [1]

Анализ

Функция `main()` начинается с создания вектора целых чисел, инициализируемого набором значений в строке 7. Алгоритм `find()` в строках 13–15 используется для поиска введенного пользователем числа. Использование алгоритма `find_if()` для поиска первого четного числа в заданном диапазоне представлено в строках 25–27. В строке 27 приведен унарный предикат, переданный алгоритму как лямбда-выражение. Это лямбда-выражение возвращает значение `true`, когда элемент делится на 2, тем самым указывая алгоритму, что данный элемент удовлетворяет проверяемому критерию. Обратите внимание на применение в строке 32 алгоритма `std::distance()` для поиска позиции элемента относительно начала контейнера.

ВНИМАНИЕ!

В листинге 23.1 всегда проверяется допустимость итератора, возвращенного функцией `find()` или `find_if()`. Эту проверку никогда не следует пропускать, поскольку успех операции `find()` никогда нельзя считать само собой разумеющимся.

Подсчет элементов с использованием значения или условия

Алгоритмы `std::count()` и `std::count_if()` обеспечивают подсчет элементов в заданном диапазоне. Алгоритм `std::count()` позволяет подсчитать количество элементов, которые равны заданному значению (для проверки используется оператор равенства `operator==`):

```
size_t numZeroes = count(numsInVec.begin(), numsInVec.end(), 0);
cout << "Количество нулевых элементов: " << numZeroes << endl;
```

Алгоритм `std::count_if()` позволяет подсчитать количество элементов, которые удовлетворяют унарному предикату, переданному в качестве параметра (это может быть функциональный объект или лямбда-выражение):

```
// Унарный предикат:
template <typename elementType>
```

```

bool IsEven(const elementType& number)
{
    return ((number % 2) == 0); // Истинно, если число четное
}

...
// Использование алгоритма count_if с унарным предикатом IsEven:
size_t numEvenNums = count_if(numsInVec.begin(),
                               numsInVec.end(), IsEven<int>);
cout << "Количество четных элементов " << numEvenNums << endl;

```

Применение этих функций продемонстрировано в листинге 23.2.

ЛИСТИНГ 23.2. Применение функций `count()` и `count if()`

```

0: #include <algorithm>
1: #include <vector>
2: #include <iostream>
3:
4:     // Унарный предикат
5:     template <typename elementType>
6:     bool IsEven(const elementType& number)
7:     {
8:         return ((number % 2) == 0); // Четное ли number?
9:     }
10:
11: int main()
12: {
13:     using namespace std;
14:     vector <int> numsInVec{ 2017, 0, -1, 42, 10101, 25 };
15:
16:     size_t numZeroes = count(numsInVec.cbegin(), numsInVec.cend(), 0);
17:     cout << "Нулевых элементов: " << numZeroes << endl << endl;
18:
19:     size_t numEvenNums = count_if(numsInVec.cbegin(),
20:                                   numsInVec.cend(), IsEven <int> );
21:
22:     cout << "Четных элементов : " << numEvenNums << endl;
23:     cout << "Нечетных элементов: ";
24:     cout << numsInVec.size() - numEvenNums << endl;
25:
26:     return 0;
27: }

```

Результат

Нулевых элементов: 1

Четных элементов : 2

Нечетных элементов: 4

Анализ

В строке 16 алгоритм `count()` использован для определения количества нулевых значений в векторе `vector<int>`. Аналогично в строке 19 алгоритм `count_if()` использован для определения количества четных чисел в векторе. Обратите внимание на третий параметр, которым является унарный предикат `IsEven()`, определенный в строках 5–9. Количество элементов с нечетными значениями в векторе вычисляется путем вычитания возвращаемого значения функции `count_if()` из общего количества содержащихся в векторе элементов, возвращаемого функцией `size()`.

ПРИМЕЧАНИЕ

В листинге 23.2 алгоритм `count_if()` использует в качестве предиката функцию `IsEven()`, тогда как в листинге 23.1 использована лямбда-функция, выполняющая работу функции `IsEven()` в алгоритме `find_if()`. Лямбда-версия экономит строки кода, но следует помнить, что если бы два примера были объединены, функция `IsEven()` была бы применима в алгоритме как `find_if()`, так и `count_if()`, обеспечивая возможность ее многократного использования.

Поиск элемента или диапазона в коллекции

В листинге 23.1 продемонстрирована возможность поиска элемента в контейнере. Но иногда необходимо найти диапазон значений. В таких ситуациях следует использовать алгоритм `search()` или `search_n()`. Алгоритм `search()` может быть применен для проверки наличия одного поддиапазона в другом диапазоне:

```
auto range = search(numsInVec.begin(), // Начало диапазона поиска
                    numsInVec.end(),   // Конец диапазона поиска
                    numsInList.begin(), // Начало искомого диапазона
                    numsInList.end()); // Конец искомого диапазона
```

Алгоритм `search_n()` применяется для проверки наличия в контейнере `n` последовательных элементов с одним и тем же значением:

```
auto partialRange = search_n(numsInVec.begin(), // Начало диапазона
                              numsInVec.end(),  // Конец диапазона
                              3,                 // Количество искомым элементов
                              9);                // Искомый элемент
```

Обе функции возвращают итератор, указывающий на первый элемент найденной последовательности, а перед применением этот итератор следует сравнить с результатом функции `end()`. В листинге 23.3 показано применение алгоритмов `search()` и `search_n()`.

ЛИСТИНГ 23.3. Поиск диапазона в коллекции с использованием алгоритмов `search()` и `search_n()`

```
0: #include <algorithm>
1: #include <vector>
```

```
2: #include <list>
3: #include <iostream>
4: using namespace std;
5:
6: template <typename T>
7: void DisplayContents(const T& container)
8: {
9:     for(auto element = container.cbegin();
10:         element != container.cend();
11:         ++element)
12:         cout << *element << ' ';
13:
14:     cout << endl;
15: }
16:
17: int main()
18: {
19:     vector<int> numsInVec{2017, 0, -1, 42, 10101, 25, 9, 9, 9};
20:     list<int> numsInList{ -1, 42, 10101 };
21:
22:     cout << "Содержимое вектора:" << endl;
23:     DisplayContents(numsInVec);
24:
25:     cout << "Содержимое списка:" << endl;
26:     DisplayContents(numsInList);
27:
28:     cout << "Поиск содержимого списка в векторе:" << endl;
29:     auto range = search(numsInVec.cbegin(),
30:                         numsInVec.cend(),
31:                         numsInList.cbegin(),
32:                         numsInList.cend());
33:
34:     // Проверка успешности поиска
35:     if (range != numsInVec.end())
36:     {
37:         cout << "Список входит в вектор в позиции: ";
38:         cout << distance(numsInVec.cbegin(), range) << endl;
39:     }
40:
41:     cout << "Поиск {9, 9, 9} в векторе:" << endl;
42:     auto partialRange = search_n(numsInVec.cbegin(),
43:                                  numsInVec.cend(),
44:                                  3,    // Количество элементов
45:                                  9 ); // Значение элементов
46:
47:     if (partialRange != numsInVec.end())
48:     {
49:         cout << "{9,9,9} найдены в векторе в позиции: ";
50:         cout << distance(numsInVec.cbegin(), partialRange) << endl;
```

```

51:     }
52:
53:     return 0;
54: }

```

Результат

```

Содержимое вектора:
2017 0 -1 42 10101 25 9 9 9
Содержимое списка:
-1 42 10101
Поиск содержимого списка в векторе:
Список входит в вектор в позиции: 2
Поиск {9, 9, 9} в векторе:
{9,9,9} найдены в векторе в позиции: 6

```

Анализ

Листинг начинается с определения двух контейнеров, вектора и списка, заполненных целочисленными значениями. Алгоритм `search()` используется для выявления наличия содержимого списка в векторе, как показано в строке 29. Поскольку мы хотим искать содержимое всего списка во всем векторе, диапазон задается итераторами, возвращаемыми методами `cbegin()` и `end()` классов этих двух контейнеров. Фактически это демонстрирует, как хорошо итераторы соединяют алгоритмы с контейнерами. Физические характеристики контейнеров, предоставляющих эти итераторы, не имеют значения для алгоритмов, осуществляющих поиск содержимого списка в векторе, поскольку они работают только с итераторами. Алгоритм `search_n()` используется в строке 42 для поиска первого вхождения подпоследовательности `{9, 9, 9}` в вектор.

Инициализация элементов в контейнере заданным значением

Алгоритмы STL `fill()` и `fill_n()` позволяют заполнить содержимое заданного диапазона определенным значением. Алгоритм `fill()` используется для перезаписи значений элементов в диапазоне, указанном его границами, некоторым значением:

```

vector<int> numsInVec(3);

// Заполнить все элементы контейнера значением 9
fill(numsInVec.begin(), numsInVec.end(), 9);

```

Как и предполагается из его названия, алгоритм `fill_n()` устанавливает значения для `n` элементов. Он нуждается в итераторе, указывающем начало диапазона, количестве и значении для заполнения:

```

fill_n(numsInVec.begin()+3, /*Количество*/ 3, /*Значение*/ -9);

```


В листинге 23.4 показано, как применение этих алгоритмов упрощает инициализацию элементов вектора `vector<int>`.

ЛИСТИНГ 23.4. Использование алгоритмов `fill()` и `fill_n()` для установки исходных значений контейнера

```
0: #include <algorithm>
1: #include <vector>
2: #include <iostream>
3:
4: int main()
5: {
6:     using namespace std;
7:
8:     // Инициализация вектора из 3 элементов
9:     vector<int> numsInVec(3);
10:
11:    // Заполняем все элементы контейнера значением 9
12:    fill(numsInVec.begin(), numsInVec.end(), 9);
13:
14:    // Увеличиваем размер вектора до 6 элементов
15:    numsInVec.resize(6);
16:
17:    // Заполняем новые (с позиции 3) три элемента значением -9
18:    fill_n(numsInVec.begin() + 3, 3, -9);
19:
20:    cout << "Содержимое вектора:" << endl;
21:    for(size_t index = 0; index < numsInVec.size(); ++index)
22:    {
23:        cout << "Элемент[" << index << "] = ";
24:        cout << numsInVec[index] << endl;
25:    }
26:
27:    return 0;
28: }
```

Результат

```
Содержимое вектора:
Элемент[0] = 9
Элемент[1] = 9
Элемент[2] = 9
Элемент[3] = -9
Элемент[4] = -9
Элемент[5] = -9
```

Анализ

В листинге 23.4 функции `fill()` и `fill_n()` используются для инициализации содержимого контейнера двумя отдельными наборами значений, как показано в строках 12 и 18. Обратите внимание на применение функции `resize()` перед заполнением диапазона значениями. По существу, этот вызов создает элементы, которые впоследствии будут заполнены значениями. Алгоритм `fill()` воздействует на весь диапазон, а алгоритм `fill_n()` может воздействовать только на часть диапазона.

СОВЕТ

Можно заметить, что код в листингах 23.1–23.3 использует константные версии итераторов, т.е. `cbegin()` и `cend()`, при определении границ диапазонов контейнера. Однако листинг 23.4 использует `begin()` и `end()`. Дело в том, что алгоритм `fill()` должен изменять элементы контейнера, а это не может быть сделано с помощью константных итераторов, которые не допускают изменения элементов, на которые они указывают.

Использование константных итераторов — хорошая практика, но при необходимости изменять элементы, на которые они указывают, следует использовать неконстантные итераторы.

Использование алгоритма `std::generate()` для инициализации значениями, генерируемыми во время выполнения

Подобно тому, как функции `fill()` и `fill_n()` заполняют коллекцию определенным значением, алгоритмы STL `generate()` и `generate_n()` инициализируют коллекции значениями, возвращаемыми унарной функцией.

Вы можете использовать функцию `generate()` для заполнения диапазона с использованием значений, возвращаемых функцией-генератором:

```
generate(numsInVec.begin(), numsInVec.end(), // Диапазон
        rand );                               // Функция-генератор
```

Алгоритм `generate_n()` подобен алгоритму `generate()`, с тем отличием, что ему необходимо указать количество элементов, которым будут присвоены значения, а не границы диапазона:

```
generate_n(numsInList.begin(), 5, rand);
```

Таким образом, вы можете использовать эти два алгоритма для инициализации содержимого контейнера, например, содержимым файла или просто случайными значениями, как показано в листинге 23.5.

ЛИСТИНГ 23.5. Использование алгоритмов `generate()` и `generate_n()` для инициализации коллекции случайными значениями

```
0: #include <algorithm>
1: #include <vector>
```

```
2: #include <list>
3: #include <iostream>
4: #include <ctime>
5:
6: int main()
7: {
8:     using namespace std;
9:     srand(time(nullptr)); // Инициализация ГСЧ
10:
11:     vector<int> numsInVec(5);
12:     generate(numsInVec.begin(), numsInVec.end(), // Диапазон
13:             rand);                               // Функция-генератор
14:
15:     cout << "Элементы вектора: ";
16:     for(size_t index = 0; index < numsInVec.size(); ++index)
17:         cout << numsInVec[index] << " ";
18:     cout << endl;
19:
20:     list<int> numsInList(5);
21:     generate_n(numsInList.begin(), 3, rand);
22:
23:     cout << "Элементы списка: ";
24:     for(auto element = numsInList.begin();
25:         element != numsInList.end();
26:         ++element )
27:         cout << *element << ' ';
28:
29:     return 0;
30: }
```

Результат

Элементы вектора: 29777 24660 23366 875 32318

Элементы списка: 14702 13460 28140 0 0

Анализ

При использовании генератора случайных чисел мы инициализировали его значением текущего времени, как показано в строке 9, чтобы при каждом новом запуске генерировалась новая последовательность случайных чисел. В строке 12 листинга 23.5 для заполнения всех элементов вектора случайными значениями, предоставляемыми функцией `rand()`, используется функция `generate()`. Обратите внимание, что функция `generate()` получает в качестве аргументов диапазон, а следовательно, вызывает определенный функциональный объект `rand()` для каждого элемента указанного диапазона. Функция `generate_n()`, напротив, получает только исходную позицию. Затем функциональный объект `rand()` вызывается столько раз, сколько задано параметром `count`. В результате содержимое заданного количества элементов будет перезаписано. Элементы контейнера вне заданных границ не затрагиваются.

Обработка элементов диапазона с использованием алгоритма `for_each()`

Алгоритм `for_each()` применяет заданный объект унарной функции к каждому элементу в указанном диапазоне. Он используется так:

```
fnObjType retVal = for_each(Начало_диапазона,
                             Конец_диапазона,
                             Унарный_функциональный_объект);
```

Унарный_функциональный_объект может быть также лямбда-выражением, которое получает один параметр. Возвращаемое значение `for_each()` возвращает функциональный объект (именуемый также функтором), который обрабатывал каждый элемент в заданном диапазоне. Преимущество такой конструкции в том, что при использовании для создания функционального объекта структуры или класса можно хранить информацию о состоянии, к которой можно обратиться впоследствии, по завершении функции `for_each()`. Этот подход показан в листинге 23.6, код которого использует функциональный объект для отображения элементов в диапазоне, а также подсчитывает их количество.

ЛИСТИНГ 23.6. Отображение содержимого последовательности с использованием алгоритма `for_each()`

```
0: #include <algorithm>
1: #include <iostream>
2: #include <vector>
3: #include <string>
4: using namespace std;
5:
6: template <typename elementType>
7: struct DisplayElementKeepcount
8: {
9:     int count;
10:     DisplayElementKeepcount(): count(0) {}
11:
12:     void operator ()(const elementType& element)
13:     {
14:         ++count;
15:         cout << element << ' ';
16:     }
17: };
18:
19: int main()
20: {
21:     vector <int> numsInVec{ 2017, 0, -1, 42, 10101, 25 };
22:
23:     cout << "Элементы вектора:" << endl;
24:     DisplayElementKeepcount<int> functor =
```

```
25:         for_each(numsInVec.cbegin(),    // Начало диапазона
26:                   numsInVec.cend(),      // Конец диапазона
27:                   DisplayElementKeepcount<int>()); // Функтор
28:     cout << endl;
29:
30:     // Используем сохраненное состояние возвращенного функтора
31:     cout<<"Выведено '"<<functor.count<<"' элементов."<<endl;
32:
33:     string str("for_each and strings!");
34:     cout << "Образец строки: " << str << endl;
35:
36:     cout << "Вывод символов лямбда-выражением:" << endl;
37:     int numChars = 0;
38:     for_each(str.cbegin(),
39:             str.cend(),
40:             [&numChars](char c){cout << c << ' '; ++numChars;});
41:
42:     cout << endl;
43:     cout << "Выведено '" << numChars << "' символов." << endl;
44:
45:     return 0;
46: }
```

Результат

```
Элементы вектора:
2017 0 -1 42 10101 25
Выведено '6' элементов.
Образец строки: for_each and strings!
Вывод символов лямбда-выражением:
f o r _ e a c h   a n d   s t r i n g s !
Выведено '21' символов.
```

Анализ

Данный код демонстрирует удобство алгоритма `for_each()` (использованного в строках 25 и 38), а также его способность возвратить функциональный объект `functor`, разработанный таким образом, чтобы содержать информацию о количестве его вызовов. В коде используются два примера диапазонов: один, содержащийся в векторе целых чисел `numsInVec`, а другой, `str`, — объект класса `std::string`. В первом вызове используется унарный предикат `DisplayElementKeepcount`, а во втором — лямбда-выражение. В первом случае для каждого элемента в заданном диапазоне алгоритм `for_each()` вызывает оператор `operator()`, который, в свою очередь, выводит элемент на экран и увеличивает значение внутреннего счетчика. Завершив работу, функция `for_each()` возвращает функциональный объект, член которого `count` хранит количество вызовов оператора `operator()` объекта. Такой способ хранения информации (или состояния) в объекте, который возвращается алгоритмом,

может оказаться весьма полезным в практических ситуациях. Алгоритм `for_each()` в строке 38 делает для объекта класса `std::string` то же самое, что и его аналог в строке 25, но с использованием лямбда-выражения вместо функционального объекта.

Выполнение преобразований с помощью алгоритма `std::transform()`

Алгоритмы `std::for_each()` и `std::transform()` очень похожи в том, что оба вызывают функциональный объект для каждого элемента в исходном диапазоне. Однако у алгоритма `std::transform()` есть две версии. Первая версия получает унарную функцию и обычно используется для преобразований наподобие символов строки в верхний или нижний регистр с использованием функции `toupper()` или `tolower()`:

```
string str("THIS is a TEst string!");
transform(str.begin(),           // Начало исходного диапазона
          str.end(),             // Конец исходного диапазона
          strLowCCopy.begin(),    // Начало целевого диапазона
          tolower );             // Унарная функция
```

Вторая версия получает в качестве аргумента бинарную функцию, позволяющую алгоритму `transform()` обработать пару элементов, взятых из двух разных диапазонов:

```
// Сложить элементы из двух диапазонов и сохранить результат в третьем
transform(numsInVec1.begin(),    // Начало исходного диапазона 1
          numsInVec1.end(),      // Конец исходного диапазона 1
          numsInVec2.begin(),    // Начало исходного диапазона 2
          numInDeque.begin(),    // Сохранить результат в очереди
          plus<int>() );         // Бинарная функция plus
```

Обе версии алгоритма `transform()` всегда присваивают результат указанной функции преобразования элементам предоставленного целевого диапазона, в отличие от алгоритма `for_each()`, который работает только с одним диапазоном. Использование алгоритма `std::transform()` показано в листинге 23.7.

ЛИСТИНГ 23.7. Использование алгоритма `std::transform()` с унарными и бинарными функциями

```
0: #include <algorithm>
1: #include <string>
2: #include <vector>
3: #include <deque>
4: #include <iostream>
5: #include <functional>
6:
7: int main()
8: {
9:     using namespace std;
10:
```

```

11:     string str("THIS is a TEst string!");
12:     cout << "Исходная строка: " << str << endl;
13:
14:     string strLowCCopy;
15:     strLowCCopy.resize(str.size());
16:
17:     transform(str.cbegin(),          // Начало исходного диапазона
18:              str.cend(),             // Конец исходного диапазона
19:              strLowCCopy.begin(),    // Начало целевого диапазона
20:              ::tolower);            // Унарная функция
21:
22:     cout << "Результат применения 'tolower':" << endl;
23:     cout << "\"" << strLowCCopy << "\"" << endl << endl;
24:
25:     // Два вектора целых чисел...
26:     vector<int> numsInVec1{ 2017, 0, -1, 42, 10101, 25 };
27:     vector<int> numsInVec2(numsInVec1.size(), -1);
28:
29:     // Целевой диапазон для хранения результата
30:     deque<int> sumInDeque(numsInVec1.size());
31:
32:     transform(numsInVec1.cbegin(), // Начало исходного диапазона 1
33:              numsInVec1.cend(),    // Конец исходного диапазона 1
34:              numsInVec2.cbegin(),  // Начало исходного диапазона 2
35:              sumInDeque.begin(),    // Начало целевого диапазона
36:              plus<int>());          // Бинарная функция
37:
38:     cout << "Результат суммирования:" << endl;
39:     cout << "Index   Vector1 + Vector2\t= Result (Deque)" << endl;
40:     for(size_t index = 0; index < numsInVec1.size(); ++index)
41:     {
42:         cout << index << " \t " << numsInVec1[index] << "\t+   ";
43:         cout << numsInVec2[index] << "\t\t= ";
44:         cout << sumInDeque[index] << endl;
45:     }
46:
47:     return 0;
48: }

```

Результат

Исходная строка: THIS is a TEst string!

Результат применения 'tolower':

"this is a test string!"

Результат суммирования:

Index	Vector1 + Vector2	= Result (Deque)
-------	-------------------	------------------

0	2017 + -1	= 2016
---	-----------	--------

1	0	+	-1	= -1
2	-1	+	-1	= -2
3	42	+	-1	= 41
4	10101	+	-1	= 10100
5	25	+	-1	= 24

Анализ

Пример демонстрирует две версии алгоритма `std::transform()`: ту, которая воздействует на один диапазон с использованием унарной функции `tolower()`, как показано в строке 20, и ту, которая воздействует на два диапазона с использованием бинарной функции `plus()`, как показано в строке 36. Первая версия посимвольно изменяет регистр символов строки на нижний. Если вместо функции `tolower()` использовать функцию `toupper()`, строка будет переведена в верхний регистр. Другая версия алгоритма `std::transform()`, представленная в строках 32–36, воздействует на элементы, взятые из двух исходных диапазонов (в данном случае из двух векторов), и использует бинарный предикат в форме функции STL `plus()` (определена в заголовочном файле `<functional>`) для их суммирования. Функция `std::transform()` получает за один раз одну пару значений, передает их бинарной функции `plus()` и присваивает результат элементу в целевом диапазоне, который в данном случае принадлежит контейнеру класса `std::deque`. Обратите внимание, что результат записывается в контейнер иного типа просто в демонстрационных целях, демонстрируя, насколько хорошо итераторы абстрагируют контейнеры и их реализацию от алгоритмов STL; функция `transform()`, будучи алгоритмом, работает с диапазонами и не обязана знать подробности о контейнере, который реализует эти диапазоны. Так, исходные диапазоны могут быть в векторе, а целевой диапазон — в деке, и все будет работать прекрасно, пока допустимы определяющие диапазон границы (предоставляемые функции `transform()` в качестве входных аргументов).

Операции копирования и удаления

Библиотека STL предоставляет три очевидные функции копирования: `copy()`, `copy_if()` и `copy_backward()`. Функция `copy()` присваивает содержимое исходного диапазона целевому диапазону в прямом порядке:

```
auto lastElement =
    copy(numsInList.cbegin(), // Начало исходного диапазона
        numsInList.cend(),   // Конец исходного диапазона
        numsInVec.begin());  // Начало целевого диапазона
```

Функция `copy_if()` копирует элемент, только если предоставленный унарный предикат возвращает для него значение `true`:

```
// Скопировать только нечетные числа из списка в вектор
copy_if(numsInList.cbegin(), numsInList.cend(),
        lastElement, // Позиция копирования в целевом диапазоне
        [](int element){return ((element % 2) == 1);});
```


Функция `copy_backward()` присваивает содержимое исходного диапазона целевому в обратном порядке:

```
copy_backward(numsInList.cbegin(),
             numsInList.cend(),
             numsInVec.end());
```

Функция `remove()` удаляет из контейнера элементы, равные определенному значению:

```
// Удалить все нули и изменить размер вектора с помощью erase()
auto newEnd = remove(numsInVec.begin(), numsInVec.end(), 0);
numsInVec.erase(newEnd, numsInVec.end());
```

Функция `remove_if()` использует унарный предикат и удаляет из контейнера те элементы, для которых этот предикат возвращает значение `true`:

```
// Удалить из вектора все нечетные числа, используя remove_if()
newEnd = remove_if(numsInVec.begin(), numsInVec.end(),
                  [](int element) {return ((element % 2) == 1);});

numsInVec.erase(newEnd, numsInVec.end()); // Изменение размера
```

Применение функций удаления и копирования показано в листинге 23.8.

ЛИСТИНГ 23.8. Применение функций `copy()`, `copy if()`, `remove()` и `remove if()`

```
0: #include <algorithm>
1: #include <vector>
2: #include <list>
3: #include <iostream>
4: using namespace std;
5:
6: template <typename T>
7: void DisplayContents(const T& container)
8: {
9:     for(auto element = container.cbegin();
10:         element != container.cend();
11:         ++element)
12:         cout << *element << ' ';
13:
14:     cout << "| Количество элементов: "<<container.size()<<endl;
15: }
16:
17: int main()
18: {
19:     list <int> numsInList{ 2017, 0, -1, 42, 10101, 25 };
20:
21:     cout << "Исходный список:" << endl;
22:     DisplayContents(numsInList);
23:
```

```

24:     // Инициализация вектора двойного размера
25:     vector<int> numsInVec(numsInList.size() * 2);
26:
27:     auto lastElement = copy(numsInList.cbegin(),
28:                             numsInList.cend(),
29:                             numsInVec.begin());
30:
31:     // Копирование нечетных чисел
32:     copy_if(numsInList.cbegin(), numsInList.cend(),
33:            lastElement,
34:            [](int element){return ((element % 2) != 0);});
35:
36:     cout << "Целевой вектор после копирований:" << endl;
37:     DisplayContents(numsInVec);
38:
39:     // Удаление нулей и изменение размера с помощью erase()
40:     auto newEnd = remove(numsInVec.begin(), numsInVec.end(), 0);
41:     numsInVec.erase(newEnd, numsInVec.end());
42:
43:     // Удаление нечетных чисел с помощью remove_if
44:     newEnd = remove_if(numsInVec.begin(), numsInVec.end(),
45:                       [](int element) {return ((element % 2) != 0);});
46:     numsInVec.erase(newEnd, numsInVec.end());
47:
48:     cout << "Целевой вектор после удалений:" << endl;
49:     DisplayContents(numsInVec);
50:
51:     return 0;
52: }

```

Результат

Исходный список:

2017 0 -1 42 10101 25 | Количество элементов: 6

Целевой вектор после копирований:

2017 0 -1 42 10101 25 2017 -1 10101 25 0 0 | Количество элементов: 12

Целевой вектор после удалений:

42 | Количество элементов: 1

Анализ

Применение функции `copy()` представлено в строке 27, где содержимое списка копируется в вектор. Функция `copy_if()` используется в строке 32, где она копирует все четные числа из исходного диапазона списка `numsInList` в диапазон назначения вектора `numsInVec` начиная с позиции, указанной итератором `lastElement`, возвращенным функцией `copy()`. Функция `remove()` в строке 40 используется для удаления из вектора `numsInVec` всех элементов со значением 0. Функция `remove_if()` используется в строке 44 для удаления всех нечетных чисел.

ВНИМАНИЕ!

В листинге 23.8 показано, что функции `remove()` и `remove_if()` возвращают итератор, указывающий на новый конец контейнера. Однако размер контейнера `numsInVec` при удалении не изменяется. Элементы алгоритмами удаления переставляются так, что “остающиеся” элементы сдвигаются в начало, а “удаляемые” — в конец диапазона, но размер вектора при этом остается неизменным. Чтобы изменить размеры контейнера (что очень важно, иначе в конце останутся нежелательные значения, и чего не в состоянии сделать алгоритм удаления, работающий с итераторами, а не с контейнерами), необходимо использовать итератор, возвращенный функцией `remove()` или `remove_if()`, в последующем вызове метода `erase()`, как показано в строках 41 и 46.

Замена значений и элементов с использованием условия

Алгоритмы STL `replace()` и `replace_if()` позволяют заменить в коллекции элементы, которые соответственно равны определенному значению или удовлетворяют некоторому заданному условию. Функция `replace()` заменяет элементы на основании сравнения с помощью оператора `==`:

```
cout << "replace заменяет значения 5 значением 8" << endl;
replace(numsInVec.begin(), numsInVec.end(), 5, 8);
```

Функция `replace_if()` получает определенный пользователем унарный предикат, который возвращает значение `true` для каждого значения, подлежащего замене:

```
cout << "replace_if заменяет четные значения значением -1" << endl;
replace_if(numsInVec.begin(), numsInVec.end(),
    [](int element) {return ((element % 2) == 0); }, -1);
```

Применение этих функций показано в листинге 23.9.

ЛИСТИНГ 23.9. Использование функций `replace()` и `replace_if()` для замены значений в определенном диапазоне

```
0: #include <iostream>
1: #include <algorithm>
2: #include <vector>
3: using namespace std;
4:
5: template <typename T>
6: void DisplayContents(const T& container)
7: {
8:     for(auto element = container.cbegin();
9:         element != container.cend();
10:         ++element)
11:         cout << *element << ' ';
12:
13:     cout << "| Количество элементов: " << container.size() << endl;
```

```
14: }
15:
16: int main()
17: {
18:     vector<int> numsInVec(6);
19:
20:     // Заполнение элементами 8 8 8 5 5 5
21:     fill(numsInVec.begin(), numsInVec.begin() + 3, 8);
22:     fill_n(numsInVec.begin() + 3, 3, 5);
23:
24:     // Перемешивание содержимого контейнера
25:     random_shuffle(numsInVec.begin(), numsInVec.end());
26:
27:     cout << "Исходное содержимое вектора:" << endl;
28:     DisplayContents(numsInVec);
29:
30:     cout << endl << "Замена 5 значением 8" << endl;
31:     replace(numsInVec.begin(), numsInVec.end(), 5, 8);
32:
33:     cout << "Замена четных значений значением -1" << endl;
34:     replace_if(numsInVec.begin(), numsInVec.end(),
35:         [](int element) {return ((element % 2) == 0); }, -1);
36:
37:     cout << endl << "Вектор после замен:" << endl;
38:     DisplayContents(numsInVec);
39:
40:     return 0;
41: }
```

Результат

Исходное содержимое вектора:
5 8 5 8 8 5 | Количество элементов: 6

Замена 5 значением 8
Замена четных значений значением -1

Вектор после замен:
-1 -1 -1 -1 -1 -1 | Количество элементов: 6

Анализ

Код заполняет вектор `numsInVec` типа `vector<int>` несколькими значениями 5 и 8, затем перемешивает их, используя алгоритм STL `std::random_shuffle()`, как показано в строке 25. Строка 31 демонстрирует применение функции `replace()` для замены всех значений 5 значениями 8. Поэтому, когда в строке 34 функция `replace_if()` заменяет все четные числа значением -1, в результате получается, что все элементы коллекции становятся равными -1, как и показывает вывод.

Сортировка, поиск в отсортированной коллекции и удаление дубликатов

Сортировка и поиск в отсортированном диапазоне (для повышения производительности) встречаются в практических приложениях очень часто. Как правило, имеется массив информации, которая должна быть отсортирована. Для сортировки контейнера можно воспользоваться алгоритмом `STL sort()`:

```
sort(numsInVec.begin(), numsInVec.end()); // В порядке возрастания
```

Эта версия функции `sort()` применяет бинарный предикат `std::less<>`, использующий оператор `operator<`, реализованный типом содержащихся в векторе элементов. Вы можете предоставить собственный предикат, чтобы изменить порядок сортировки, используя следующую перегруженную версию алгоритма:

```
sort(numsInVec.begin(), numsInVec.end(),
      [](int lhs, int rhs){return (lhs > rhs);}); // В порядке убывания
```

Кроме того, зачастую требуется удалить дубликаты из коллекции. Для удаления расположенных рядом повторяющихся значений используется алгоритм `unique()`:

```
auto newEnd = unique(numsInVec.begin(), numsInVec.end());
numsInVec.erase(newEnd, numsInVec.end()); // Изменить размер
```

Для быстрого поиска STL предоставляет алгоритм `binary_search()`, который работает только в отсортированном контейнере:

```
bool elementFound = binary_search(numsInVec.begin(),
                                   numsInVec.end(), 2011);
```

```
if (elementFound)
    cout << "Элемент найден!" << endl;
```

В листинге 23.10 показаны упомянутые выше алгоритмы STL — `std::sort()`, который способен отсортировать диапазон, `std::binary_search()`, обеспечивающий поиск в отсортированном диапазоне, и `std::unique()`, удаляющий расположенные рядом совпадающие элементы (они становятся смежными после сортировки).

ЛИСТИНГ 23.10. Использование функций `sort()`, `binary_search()` и `unique()`

```
0: #include <algorithm>
1: #include <vector>
2: #include <string>
3: #include <iostream>
4: using namespace std;
5:
6: template <typename T>
7: void DisplayContents(const T& container)
8: {
9:     for(auto element = container.cbegin();
10:        element != container.cend();
```

```

11:         ++element)
12:     cout << *element << endl;
13: }
14:
15: int main()
16: {
17:     vector<string> vecNames{"John", "jack", "sean", "Anna"};
18:
19:     // Вставка дубликата
20:     vecNames.push_back("jack");
21:
22:     cout << "Исходный вектор:" << endl;
23:     DisplayContents(vecNames);
24:
25:     cout << "Отсортированный вектор:" << endl;
26:     sort(vecNames.begin(), vecNames.end());
27:     DisplayContents(vecNames);
28:
29:     cout << "Поиск \"John\" с помощью 'binary_search':" << endl;
30:     bool elementFound = binary_search(vecNames.begin(),
31:                                       vecNames.end(), "John");
32:
33:     if (elementFound)
34:         cout << "\"John\" найден в векторе!" << endl;
35:     else
36:         cout << "Элемент не найден." << endl;
37:
38:     // Удаление смежных дубликатов
39:     auto newEnd = unique(vecNames.begin(), vecNames.end());
40:     vecNames.erase(newEnd, vecNames.end());
41:
42:     cout << "Вектор после применения 'unique':" << endl;
43:     DisplayContents(vecNames);
44:
45:     return 0;
46: }

```

Результат

Исходный вектор:

```

John
jack
sean
Anna
jack

```

Отсортированный вектор:

```

Anna
John

```

```
jack
jack
sean
Поиск "John" с помощью 'binary_search':
"John" найден в векторе!
Вектор после применения 'unique':
Anna
John
jack
sean
```

Анализ

Приведенный выше код сначала сортирует вектор `vecNames` (строка 26), а затем (строка 30) использует алгоритм `binary_search()` для поиска в нем элемента `John Doe`. Точно так же в строке 39 используется алгоритм `std::unique()` для удаления смежных дубликатов. Обратите внимание, что функция `unique()`, как и функция `remove()`, не изменяет размер контейнера. Это приводит к переносу значений, но не к сокращению общего количества элементов. Чтобы избавиться от нежелательных или неизвестных значений в конце контейнера, после функции `unique()` всегда следует вызывать функцию `vector::erase()`, используя итератор, возвращенный функцией `unique()`, как показано в строке 40.

ВНИМАНИЕ!

Такие алгоритмы, как `binary_search()`, работают только в отсортированных контейнерах. При использовании этого алгоритма с неотсортированным вектором могут возникнуть нежелательные последствия.

ПРИМЕЧАНИЕ

Функция `stable_sort()` используется точно так же, как представленная ранее функция `sort()`. Функция `stable_sort()` сохраняет относительный порядок отсортированных элементов. Это сохранение порядка обеспечивается ценой некоторого снижения производительности, что следует учитывать, если относительный порядок смежных элементов не является важным.

Разделение диапазона

Функция `std::partition()` позволяет разделить исходный диапазон на две части: в одной элементы удовлетворяют унарному предикату, а в другой — не удовлетворяют:

```
bool IsEven(const int& num) // Унарный предикат
{
    return ((num % 2) == 0);
}
...
partition(numsInVec.begin(), numsInVec.end(), IsEven);
```

Однако функция `std::partition()` не гарантирует сохранение относительного порядка элементов в пределах каждого раздела. Когда это важно, следует использовать функцию `std::stable_partition()`:

```
stable_partition(numsInVec.begin(), numsInVec.end(), IsEven);
```

В листинге 23.11 показано применение этих алгоритмов.

ЛИСТИНГ 23.11. Использование алгоритмов `partition()` и `stable_partition()` для разделения диапазона целых чисел на четные и нечетные значения

```
0: #include <algorithm>
1: #include <vector>
2: #include <iostream>
3: using namespace std;
4:
5: bool IsEven(const int& num) // Унарный предикат
6: {
7:     return ((num % 2) == 0);
8: }
9:
10: template <typename T>
11: void DisplayContents(const T& container)
12: {
13:     for(auto element = container.cbegin();
14:         element != container.cend();
15:         ++element)
16:         cout << *element << ' ';
17:
18:     cout << "| Количество элементов: " << container.size() << endl;
19: }
20:
21: int main()
22: {
23:     vector<int> numsInVec{2017, 0, -1, 42, 10101, 25, 34, 19};
24:
25:     cout << "Исходный вектор: " << endl;
26:     DisplayContents(numsInVec);
27:
28:     vector<int> vecCopy(numsInVec);
29:
30:     cout << "Результат partition(): " << endl;
31:     partition(numsInVec.begin(), numsInVec.end(), IsEven);
32:     DisplayContents(numsInVec);
33:
34:     cout << "Результат stable_partition(): " << endl;
35:     stable_partition(vecCopy.begin(), vecCopy.end(), IsEven);
36:     DisplayContents(vecCopy);
37:
38:     return 0;
39: }
```

Результат

```

Исходный вектор:
2017 0 -1 42 10101 25 34 19 | Количество элементов: 8
Результат partition():
34 0 42 -1 10101 25 2017 19 | Количество элементов: 8
Результат stable_partition():
0 42 34 2017 -1 10101 25 19 | Количество элементов: 8

```

Анализ

Код делит диапазон целых чисел, содержащийся в векторе `numsInVec`, на четные и нечетные значения. Сначала разделение осуществляется с использованием функции `std::partition()`, как показано в строке 31, а затем с использованием функции `stable_partition()` в строке 35. Для сравнения результатов вектор `numsInVec` копируется в вектор `vecCopy`; первый разделяется с использованием функции `partition()`, а последний — с использованием `stable_partition()`. Различие в результатах использования функций `stable_partition()` и `partition()` вполне очевидно в приведенном выводе программы. Алгоритм `stable_partition()` обеспечивает относительный порядок элементов в каждом разделе. Обратите внимание, что поддержка этого порядка может сказываться на производительности, и это влияние может быть как незначительным, так и существенным в зависимости от типа содержащихся в диапазоне объектов.

ПРИМЕЧАНИЕ

Функция `stable_partition()` работает медленнее, чем `partition()`, а потому ее следует использовать только тогда, когда важен относительный порядок элементов в контейнере.

Вставка элементов в отсортированную коллекцию

Для отсортированной коллекции важно, чтобы элементы вставлялись в правильную позицию. Библиотека STL предоставляет такие функции, как `lower_bound()` и `upper_bound()`, позволяющие решить эту задачу:

```

auto minInsertPos = lower_bound(listNames.begin(), listNames.end(),
                                "Brad Pitt" );
auto maxInsertPos = upper_bound(listNames.begin(), listNames.end(),
                                "Brad Pitt" );

```

Функции `lower_bound()` и `upper_bound()` возвращают итераторы, указывающие минимальную и максимальную позиции в отсортированном диапазоне, в который элемент может быть вставлен без нарушения порядка сортировки.

В листинге 23.12 показано применение функции `lower_bound()` для вставки элемента в минимальную позицию отсортированного списка имен.

ЛИСТИНГ 23.12. Использование функций `lower_bound()` и `upper_bound()` для вставки в отсортированную коллекцию

```

0: #include <algorithm>
1: #include <list>
2: #include <string>
3: #include <iostream>
4: using namespace std;
5:
6: template <typename T>
7: void DisplayContents(const T& container)
8: {
9:     for(auto element = container.cbegin();
10:        element != container.cend();
11:        ++element)
12:         cout << *element << endl;
13: }
14:
15: int main()
16: {
17:     list<string> names{ "John", "Brad", "jack", "sean", "Anna" };
18:
19:     cout << "Отсортированный список:" << endl;
20:     names.sort();
21:     DisplayContents(names);
22:
23:     cout << "Нижний индекс, в который может быть вставлен \"Brad\": ";
24:     auto minPos = lower_bound(names.begin(), names.end(), "Brad");
25:     cout << distance(names.begin(), minPos) << endl;
26:
27:     cout << "Верхний индекс, в который может быть вставлен \"Brad\": ";
28:     auto maxPos = upper_bound(names.begin(), names.end(), "Brad");
29:     cout << distance(names.begin(), maxPos) << endl;
30:
31:     cout << endl;
32:
33:     cout << "Список после вставки \"Brad\": " << endl;
34:     names.insert(minPos, "Brad");
35:     DisplayContents(names);
36:
37:     return 0;
38: }

```

Результат

Отсортированный список:

Anna
Brad
John
jack

```
sean
```

```
Нижний индекс, в который может быть вставлен "Brad": 1
```

```
Верхний индекс, в который может быть вставлен "Brad": 2
```

```
Список после вставки "Brad":
```

```
Anna
```

```
Brad
```

```
Brad
```

```
John
```

```
jack
```

```
sean
```

Анализ

Новый элемент может быть вставлен в отсортированную коллекцию в нескольких потенциальных позициях, причем самую близкую к началу коллекции указывает итератор, возвращенный функцией `lower_bound()`, а наиболее близкую к концу коллекции указывает итератор, возвращенный функцией `upper_bound()`. В листинге 23.12 в отсортированную коллекцию вставляется строка "Brad", уже имеющаяся в ней. Верхняя и нижняя границы различны (они совпадали бы, если бы в коллекции не было такого элемента). Применение этих функций показано в строках 24 и 29 соответственно. Как показывает вывод программы, при использовании для вставки строки в список итератора, возвращенного функцией `lower_bound()` (строка 35), список сохраняет отсортированное состояние. Таким образом, данные алгоритмы позволяют осуществлять вставку в коллекцию, не нарушая порядок отсортированного содержимого. Итератор, возвращенный функцией `upper_bound()`, сработал бы не менее корректно.

РЕКОМЕНДУЕТСЯ

Используйте метод `erase()` класса контейнера после алгоритмов `remove()`, `remove_if()` и `unique()` для изменения размера контейнера.

Проверяйте корректность итератора, возвращаемого функциями `find()`, `find_if()`, `search()` и `search_n()`, сравнивая его с возвращаемым значением метода `end()` контейнера, прежде чем использовать его для обращения к элементу.

Предпочитайте функцию `stable_partition()` функции `partition()`, а функцию `stable_sort()` функции `sort()` только тогда, когда важно сохранение относительного порядка отсортированных или разделенных элементов, поскольку версии `stable_*` могут снизить производительность приложения.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте сортировать содержимое контейнера, прежде чем вызывать метод `unique()` для удаления повторяющихся смежных значений. Функция `sort()` гарантирует, что все элементы с совпадающими значениями будут смежными, обеспечивая возможность работы функции `unique()`.

Не забывайте, что функцию `binary_search()` следует вызывать только для отсортированного контейнера.

Резюме

На сегодняшнем занятии рассматривался один из самых важных аспектов STL: алгоритмы. Вы изучили различные типы алгоритмов, а примеры помогли вам лучше понять их применение.

Вопросы и ответы

- Могу ли я применить изменяющий алгоритм, такой как `std::transform()`, к ассоциативному контейнеру, такому как `std::set`?

Даже если бы это было возможно, поступать так не нужно. Элементы в ассоциативном контейнере `set` следует рассматривать как константные. Дело в том, что ассоциативные контейнеры сортируют свои элементы при вставке, и относительные позиции элементов играют важную роль в таких функциях, как `find()`, а также для эффективности работы контейнера. Поэтому изменяющие алгоритмы, такие как `std::transform()`, не должны использоваться с множествами STL.

- Я должен присвоить определенное значение каждому элементу последовательного контейнера. Могу ли я использовать для этого алгоритм `std::transform()`?

Хотя алгоритм `std::transform()` для этого вполне применим, лучше использовать алгоритм `fill()` или `fill_n()`.

- Изменяет ли алгоритм `copy_backward()` расположение элементов контейнера на обратное?

Нет, он этого не делает. Алгоритм STL `copy_backward()` изменяет на обратный порядок элементов при копировании, но не порядок хранимых элементов, т.е. копирование начинается с конца диапазона и продолжается к началу. Чтобы обратить содержимое коллекции, используйте алгоритм `std::reverse()`.

- Могу ли я использовать алгоритм `std::sort()` для списка?

Алгоритм `std::sort()` применяется к списку так же, как и к любому другому последовательному контейнеру. Однако у списка есть особое свойство: существующие итераторы остаются корректными при операциях со списком, в то время как функция `std::sort()` не может этого гарантировать. Поэтому список STL предоставляет собственный алгоритм `sort()` в форме функции-члена `list::sort()`, который и следует использовать, поскольку он гарантирует, что итераторы на элементы списка останутся допустимыми, даже если их относительные позиции в списке изменятся.

- Почему так важно использовать такие функции, как `lower_bound()` или `upper_bound()`, при вставке в отсортированный диапазон?

Эти функции предоставляют соответственно первую и последнюю позиции в отсортированной коллекции, в которую может быть вставлен элемент без нарушения порядка сортировки.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Необходимо удалить из списка элементы, удовлетворяющие некоторому заданному условию. Какую функцию вы используете: `std::remove_if()` или `list::remove_if()`?
2. У вас есть список элементов типа `ContactItem`. Как функция `list::sort()` отсортирует элементы списка этого типа в отсутствии явно определенного бинарного предиката?
3. Как часто алгоритм STL `generate()` вызывает функцию `generator()`?
4. Чем функция `std::transform()` отличается от функции `std::for_each()`?

Упражнения

1. Напишите бинарный предикат, получающий в качестве аргументов строки и возвращающий значение, представляющее результат сравнения строк, не зависящего от регистра символов.
2. Продемонстрируйте, как алгоритмы STL, такие как `std::copy()`, используют итераторы для выполнения своих задач, не нуждаясь в знании природы целевой коллекции при копировании двух последовательностей, содержащихся в двух разных по типу контейнерах.
3. Вы пишете приложение, которое записывает характеристики звезд, видимых на горизонте в порядке их восхождения. В астрономии важна информация о размере звезды и о ее относительной высоте и порядке. Сортируя эту коллекцию звезд по размерам, вы использовали бы функцию `std::sort()` или `std::stable_sort()`?

ЗАНЯТИЕ 24

Адаптивные контейнеры: стек и очередь

Стандартная библиотека шаблонов (STL) предоставляет шаблонные классы, способные адаптировать другие контейнеры для моделирования поведения очереди и стека. Контейнеры, которые внутренне используют другой контейнер и обеспечивают иное поведение, называются *адаптивными контейнерами* (adaptive container) или просто *адаптерами*.

На этом занятии...

- Поведенческие характеристики стеков и очередей
- Использование адаптера STL `stack`
- Использование адаптера STL `queue`
- Использование адаптера STL `priority_queue`

Поведенческие характеристики стеков и очередей

Стеки и очереди очень похожи на массивы и списки, но с ограничениями, касающимися вставки элементов, доступа к ним и удаления. Их поведенческие характеристики определяются расположением элементов при вставке и позицией элемента, который может быть извлечен из контейнера.

Стеки

Стек (stack) — это структура в памяти, действующая по принципу *последним вошел, первым вышел* (Last-In-First-Out — LIFO), элементы которой могут быть вставлены или извлечены из вершины контейнера. Стек можно представить как стопку тарелок. Последняя положенная в стопку тарелка будет взята первой. К тарелкам в середине и в основании доступа нет. Этот способ организации элементов, подразумевающий “добавление и извлечение из вершины”, представлен на рис. 24.1.

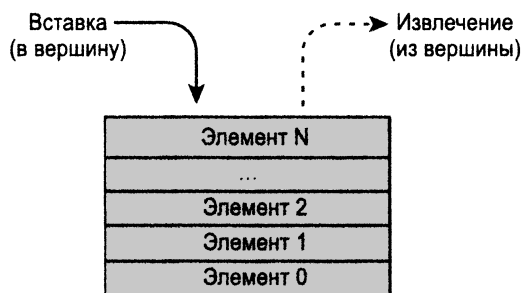


РИС. 24.1. Работа со стеком

Такое поведение стопки тарелок моделирует обобщенный контейнер `std::stack` STL.

СОВЕТ

Чтобы использовать класс `std::stack`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <stack>
```

Очереди

Очередь (queue) — это структура в памяти, действующая по принципу *первым вошел, первым вышел* (First-In-First-Out — FIFO), элементы которой могут вставляться после уже находящихся в ней и извлекаться в том же порядке, в котором были вставлены. Очередь можно представить как очередь ожидающих людей, в которой, кто первым встал в очередь, тот раньше всех ее покинет (в такой очереди никто не пробирается без очереди). Этот способ организации элементов, подразумевающий вставку в конец и извлечение из начала, представлен на рис. 24.2.

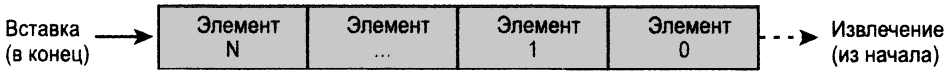


РИС. 24.2. Работа с очередью

Такое поведение очереди моделирует обобщенный контейнер STL `std::queue`.

СОВЕТ

Чтобы использовать класс `std::queue`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <queue>
```

Использование класса STL stack

Стек STL представляет собой шаблон класса `stack`, для использования которого необходимо включить в код заголовочный файл `<stack>`. Это обобщенный класс, обеспечивающий вставку элементов в вершину контейнера и извлечение их оттуда и не разрешающий доступ к элементам в середине стека или их просмотр. В некотором смысле поведение класса `std::stack` очень похоже на стопку тарелок.

Создание экземпляра стека

Некоторые реализации STL определяют шаблон класса `stack` так:

```
template <
    class Тип_Элемента,
    class Контейнер = deque<Тип>
> class stack;
```

Параметр `Тип_Элемента` указывает тип элементов, которые будут храниться в стеке. Второй параметр шаблона, `Контейнер`, — это класс контейнера, на основе которого реализован стек. По умолчанию для внутреннего хранения данных стека используется класс `std::deque`, но он может быть заменен классом `std::vector` или `std::list`. Таким образом, создание экземпляра стека целых чисел имеет следующий вид:

```
std::stack<int> numsInStack;
```

Если необходимо создать стек объектов какого-нибудь иного типа, например класса `Tuna`, то можно использовать следующий синтаксис:

```
std::stack<Tuna> tunasInStack;
```

Для создания стека на базе другого контейнера используйте следующий синтаксис:

```
std::stack<double, vector<double>> doublesStackedInVec;
```

В листинге 24.1 демонстрируются способы инстанцирования стека.

ЛИСТИНГ 24.1. Инстанцирование стека STL

```
0: #include <stack>
1: #include <vector>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     // Стек целых чисел
8:     stack<int> numsInStack;
9:
10:    // Стек чисел типа double
11:    stack<double> dblsInStack;
12:
13:    // Стек чисел типа double, содержащихся в векторе
14:    stack<double, vector<double>> doublesStackedInVec;
15:
16:    // Инициализация стека копией другого стека
17:    stack<int> numsInStackCopy(numsInStack);
18:
19:    return 0;
20: }
```

Анализ

Этот пример ничего не выводит, он лишь демонстрирует создание экземпляра шаблона стека STL. В строках 8 и 11 создаются два экземпляра объекта класса `stack` для хранения элементов типов `int` и `double` соответственно. В строке 14 также создается экземпляр стека, но с использованием второго параметра шаблона — явного указания класса контейнера (`vector`), который стек должен использовать внутренне. Если этот второй параметр шаблона не предоставлен, по умолчанию вместо него используется класс `std::deque`. И наконец в строке 17 показано, что один объект стека может быть создан как копия другого.

Функции-члены класса `stack`

Стек, который адаптирует другой контейнер, такой как `deque`, `list` или `vector`, реализует свои функциональные возможности, ограничивая способы вставки и извлечения элементов для обеспечения поведения, которое ожидается от механизма стека. В табл. 24.1 приведены открытые функции-члены класса `stack` и способы их применения на примере стека целых чисел.

ТАБЛИЦА 24.1. Функции-члены класса `stack`

Функция	Описание
<code>push()</code>	Вставляет элемент в вершину стека <code>numsInStack.push(25);</code>

Функция	Описание
<code>pop()</code>	Извлекает элемент из вершины стека <code>numsInStack.pop();</code>
<code>empty()</code>	Проверяет, не пуст ли стек; возвращает значение типа <code>bool</code> <code>if (numsInStack.empty()) DoSomething();</code>
<code>size()</code>	Возвращает количество элементов в стеке <code>size_t nNumElements = numsInStack.size();</code>
<code>top()</code>	Возвращает ссылку на верхний элемент в стеке <code>cout << "Верхний элемент = " << numsInStack.top();</code>

Как свидетельствует таблица, к открытым функциям-членам стека относятся только те методы, которые обеспечивают вставку и извлечение, согласующиеся с поведением стека. Таким образом, при том, что базовый контейнер может быть деком, вектором или списком, функциональные возможности этого контейнера, чтобы имитировать поведенческие характеристики стека, доступны не будут.

Вставка и извлечение с помощью методов `push()` и `pop()`

Для вставки элементов в стек используется метод `stack<T>::push()`:

```
numsInStack.push(25); // Вставить 25 в вершину стека
```

По определению стек разрешает доступ к элементу в вершине с помощью метода `top()`:

```
cout << numsInStack.top() << endl;
```

Если необходимо извлечь верхний элемент, можно использовать функцию `pop()`:

```
numsInStack.pop(); // Извлечение верхнего элемента из стека
```

В листинге 24.2 показаны вставка элементов в стек с помощью метода `push()` и их извлечение с помощью метода `pop()`.

ЛИСТИНГ 24.2. Работа со стеком целых чисел

```
0: #include <stack>
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:     stack<int> numsInStack;
7:
8:     // push: вставка значений в вершину стека
9:     cout << "Внесение в стек {25, 10, -1, 5}:" << endl;
```

```
10:     numsInStack.push(25);
11:     numsInStack.push(10);
12:     numsInStack.push(-1);
13:     numsInStack.push(5);
14:
15:     cout << "Всего " << numsInStack.size() << " элемента" << endl;
16:     while(numsInStack.size() != 0)
17:     {
18:         cout << "Элемент на вершине: " << numsInStack.top() << endl;
19:         numsInStack.pop(); // pop: извлечение верхнего элемента
20:     }
21:
22:     if (numsInStack.empty())
23:         cout << "Извлечены все элементы - стек пуст!" << endl;
24:
25:     return 0;
26: }
```

Результат

```
Внесение в стек {25, 10, -1, 5}:
Всего 4 элемента
Элемент на вершине: 5
Элемент на вершине: -1
Элемент на вершине: 10
Элемент на вершине: 25
Извлечены все элементы - стек пуст!
```

Анализ

Сначала в стек целых чисел `numsInStack` с использованием метода `stack::push()`, как показано в строках 9–13, вставляются значения; затем они извлекаются с использованием метода `stack::pop()`. Стек разрешает доступ только к верхнему элементу, к которому можно обратиться, используя метод `stack::top()`, как показано в строке 18. Элементы могут быть извлечены из стека по одному с помощью метода `stack::pop()`, как показано в строке 19. Цикл `while` перебирает стек, гарантируя, что операция `pop()` будет повторяться до тех пор, пока стек не окажется пустым. Как свидетельствует порядок элементов при выводе, элементы, вставленные последними, извлекаются первыми, демонстрируя поведение стека.

В листинге 24.2 показаны все пять функций-членов стека. Обратите внимание на то, что методы `push_back()` и `insert()`, доступные для всех последовательных контейнеров STL, используемых как базовые контейнеры классом `stack`, недоступны в качестве открытых функций-членов класса `stack`. То же самое относится к итераторам, позволяющим просмотреть все элементы, включая не расположенные в вершине контейнера. Все, что предоставляет стек, — это только верхний элемент, и ничего кроме.

Использование класса STL queue

Для применения шаблона класса STL queue требуется включить в исходный текст программы его заголовочный файл `<queue>`. Это обобщенный класс, обеспечивающий вставку элементов только в конец контейнера, а извлечение — только с начала и не разрешающий доступ или просмотр элементов в середине. В некотором смысле поведение класса `std::queue` очень похоже на поведение очереди людей к кассе в супермаркете.

Создание экземпляра очереди

Шаблон класса `std::queue` определен следующим образом:

```
template <
    class Тип_Элемента,
    class Контейнер = deque<Тип>
> class queue;
```

Параметр *Тип_Элемента* задает тип объектов, которые будут храниться в очереди. Второй параметр шаблона, *Контейнер*, — это класс контейнера, используемого классом `std::queue` для хранения данных. Возможными кандидатами на этот параметр шаблона являются список, вектор и дек. По умолчанию используется дек — класс `deque`.

Самый простой экземпляр очереди целых чисел создается следующим образом:

```
std::queue<int> numsInQ;
```

Если необходимо создать очередь, содержащую элементы типа `double` в контейнере `std::list` (вместо заданной по умолчанию двухсторонней очереди), используйте следующий код:

```
std::queue<double, list<double>> dblsInQList;
```

Точно так же, как стек, очередь может быть создана, как копия другой очереди:

```
std::queue<int> qCopy(numsInQ);
```

В листинге 24.3 показаны некоторые способы создания экземпляра класса `std::queue`.

ЛИСТИНГ 24.3. Инстанцирование очереди STL

```
0: #include <queue>
1: #include <list>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     // Очередь целых чисел
8:     queue<int> numsInQ;
```

```

9:
10: // Очередь чисел типа double
11: queue<double> qDoubles;
12:
13: // Очередь чисел типа double, хранящаяся в списке
14: queue<double, list<double>> dblsInQList;
15:
16: // Очередь создана как копия другой очереди
17: queue<int> copyQ(numsInQ);
18:
19: return 0;
20: }

```

Анализ

Пример демонстрирует, как может быть создан экземпляр обобщенного класса STL `queue`, чтобы получить очередь целых чисел (строка 8) и чисел типа `double` (строка 11). При создании экземпляра очереди `dblsInQList` в строке 14 во втором параметре шаблона было явно указано, что базовым внутренним контейнером очереди будет класс `std::list`. При отсутствии второго параметра шаблона, как и в первых двух очередях, для базового контейнера содержимого очереди по умолчанию используется класс `std::deque`.

Функции-члены класса `queue`

Реализация контейнера `std::queue`, как и `std::stack`, базируется на таких контейнерах STL, как `vector`, `list` или `deque`. Класс `queue` предоставляет только те функции-члены, которые реализуют поведенческие характеристики очереди. В табл. 24.2 приведены функции-члены класса `queue`, используемые очередью целых чисел `numsInQ` в листинге 24.3.

ТАБЛИЦА 24.2. Функции-члены класса `std::queue`

Функция	Описание
<code>push()</code>	Вставляет элемент в конец очереди, т.е. в ее последнюю позицию <code>numsInQ.push(10);</code>
<code>pop()</code>	Извлекает элемент из начала очереди, т.е. из ее первой позиции <code>numsInQ.pop();</code>
<code>front()</code>	Возвращает ссылку на элемент в начале очереди <code>cout << "Первый элемент: " << numsInQ.front();</code>
<code>back()</code>	Возвращает ссылку на элемент в конце очереди, т.е. на последний вставленный элемент <code>cout << "Последний элемент: " << numsInQ.back();</code>
<code>empty()</code>	Проверяет, не пуста ли очередь; возвращает значение типа <code>bool</code> <code>if (numsInQ.empty()) cout << "Очередь пуста!";</code>
<code>size()</code>	Возвращает количество элементов в очереди <code>size_t nNumElements = numsInQ.size();</code>

Класс STL queue не предоставляет такие функции, как `begin()` и `end()`, хотя они доступны в большинстве контейнеров STL, включая базовые классы `deque`, `vector` и `list`, лежащие в основе класса очереди. Это сделано преднамеренно, чтобы единственными допустимыми операциями очереди были те, которые согласуются с ее поведенческими характеристиками.

Вставка в конец и извлечение из начала очереди с использованием методов `push()` и `pop()`

Для вставки элементов в очередь используется метод `push()`:

```
numsInQ.push(5); // Элемент вставляется в конец очереди
```

Извлечение, напротив, осуществляется из начала очереди с помощью метода `pop()`:

```
numsInQ.pop(); // Извлечь элемент из начала очереди
```

В отличие от стека, в очереди для просмотра доступны элементы с обоих концов контейнера, для чего используются методы `front()` и `back()`:

```
cout << "Элемент в начале: " << numsInQ.front() << endl;
cout << "Элемент в конце: " << numsInQ.back() << endl;
```

Вставка, извлечение и просмотр элементов очереди показаны в листинге 24.4.

ЛИСТИНГ 24.4. Вставка, извлечение и просмотр элементов очереди целых чисел

```
0: #include <queue>
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:     queue<int> numsInQ;
7:
8:     cout << "Вставка {10, 5, -1, 20}" << endl;
9:     numsInQ.push(10);
10:    numsInQ.push(5);
11:    numsInQ.push(-1);
12:    numsInQ.push(20);
13:
14:    cout << "Всего " << numsInQ.size() << " элемента" << endl;
15:    cout << "Первый элемент: " << numsInQ.front() << endl;
16:    cout << "Последний элемент: " << numsInQ.back() << endl;
17:
18:    while(numsInQ.size() != 0)
19:    {
20:        cout << "Удаление " << numsInQ.front() << endl;
21:        numsInQ.pop(); // Извлечение элемента из начала
22:    }
```

```
23:
24:     if (numsInQ.empty())
25:         cout << "Очередь пуста!" << endl;
26:
27:     return 0;
28: }
```

Результат

Вставка {10, 5, -1, 20}
Всего 4 элемента
Первый элемент: 10
Последний элемент: 20
Удаление 10
Удаление 5
Удаление -1
Удаление 20
Очередь пуста!

Анализ

В строках 9–12 элементы добавляются в конец очереди `numsInQ` с использованием метода `push()`. Методы `front()` и `back()` используются для обращения к элементам в начальной и конечной позициях очереди, как показано в строках 15 и 16. Цикл `while` в строках 18–22 отображает элементы из начала очереди, прежде чем извлечь их с использованием метода `pop()` в строке 21. Это продолжается до тех пор, пока очередь не опустеет. Вывод демонстрирует, что элементы были извлечены из очереди в том же порядке, в каком они были вставлены, поскольку вставляются они в конец очереди, а извлекаются с начала.

Использование класса `STL priority_queue`

Для применения шаблона класса `STL priority_queue` требуется включить в исходный текст программы его заголовочный файл `<queue>`. *Очередь с приоритетами* (`priority queue`) отличается от обычной очереди тем, что элемент с наивысшим значением (или значением, считающимся наивысшим согласно некоторому бинарному предикату) идет без очереди, т.е. оказывается доступным в начале очереди, а работа с такими очередями ограничивается только их началом.

Создание экземпляра очереди с приоритетами

Шаблон класса `std::priority_queue` определен следующим образом:

```
template <
    class Тип_Элемента,
    class Контейнер = vector<Тип_Элемента>,
    class Сравнение = less<typename Контейнер::value_type>
> class priority_queue
```

Параметр *Тип Элемента* задает тип объектов, которые будут храниться в очереди с приоритетами. Второй параметр шаблона, *Контейнер*, — это класс контейнера, используемого классом `std::priority_queue` для хранения данных, а третий параметр позволяет программисту определить бинарный предикат для выявления элемента, располагающегося в очереди первым. Если бинарный предикат не определен, класс `priority_queue` использует заданный по умолчанию предикат `std::less<>`, который сравнивает объекты, используя `operator<`.

Самый простой экземпляр очереди с приоритетами целых чисел создается следующим образом:

```
std::priority_queue<int> numsInPrioQ;
```

Если необходимо создать очередь с приоритетами, содержащую элементы типа `int` в контейнере `std::deque` и использующую для определения приоритета элементов предикат `greater<int>`, используется следующий код:

```
priority_queue<int, deque<int>, greater<int>> numsInPrioQ_Inverse;
```

Подобно стеку, экземпляр очереди может быть создан как копия другой очереди:

```
std::priority_queue<int> copyQ(numsInPrioQ);
```

Инстанцирование класса `priority_queue` представлено в листинге 24.5.

ЛИСТИНГ 24.5. Инстанцирование класса `priority_queue`

```
0: #include <queue>
1: #include <functional>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     // Очередь с приоритетами с элементами int
8:     priority_queue<int> numsInPrioQ;
9:
10:    // Очередь с приоритетами с элементами double
11:    priority_queue<double> dblsInPrioQ;
12:
13:    // Очередь int с приоритетами с сортировкой std::greater <>
14:    priority_queue<int, deque<int>, greater<int>> numsInDescendingQ;
15:
16:    // Очередь с приоритетами, созданная как копия
17:    priority_queue<int> copyQ(numsInPrioQ);
18:
19:    return 0;
20: }
```

Анализ

Строки 8 и 11 демонстрируют создание двух экземпляров класса `priority_queue` для объектов типа `int` и `double` соответственно. Отсутствие всех остальных параметров шаблона заставляет принять значения по умолчанию и использовать класс `std::vector` как внутренний контейнер данных, а `std::less<>` — как критерий сравнения. Поэтому приоритетным в данной очереди будет целое число с наивысшим значением; оно будет находиться в начале очереди с приоритетами. Для экземпляра `numsInDescendingQ` в качестве внутреннего контейнера задан дек, а в качестве предиката сравнения — `std::greater<>`. Этот предикат создает очередь, в начале которой будет находиться наименьшее число.

Результат использования предиката `std::greater<T>` объясняется в листинге 24.7 далее на этом занятии.

СОВЕТ

Чтобы использовать предикат `std::greater<>`, в исходный текст программы следует включить заголовочный файл `<functional>`.

Функции-члены класса `priority_queue`

Функции-члены `front()` и `back()`, доступные в классе `queue`, недоступны в классе `priority_queue`. Функции-члены класса `priority_queue` перечислены в табл. 24.3.

ТАБЛИЦА 24.3. Функции-члены класса `std::priority_queue`

Функция	Описание
<code>push()</code>	Вставляет элемент в очередь с приоритетами <code>numsInPrioQ.push(10);</code>
<code>pop()</code>	Извлекает элемент из начала очереди, т.е. элемент с наивысшим приоритетом <code>numsInPrioQ.pop();</code>
<code>top()</code>	Возвращает ссылку на элемент с наивысшим приоритетом в очереди, занимающий первую позицию <code>cout << "Первый элемент очереди: " << numsInPrioQ.top();</code>
<code>empty()</code>	Проверяет, не пуста ли очередь с приоритетами; возвращает значение типа <code>bool</code> <code>if (numsInPrioQ.empty()) cout << "Очередь пуста!";</code>
<code>size()</code>	Возвращает количество элементов в очереди с приоритетами <code>size_t nNumElements = numsInPrioQ.size();</code>

Как свидетельствует таблица, к членам очереди с приоритетами можно обратиться, только используя метод `top()`, возвращающий элемент с самым высоким значением приоритета, согласно заданному пользователем предикату (или предикату `std::less<>` при его отсутствии).

Вставка в конец и извлечение из начала очереди с приоритетами с использованием методов push() и pop()

Для вставки элементов в очередь с приоритетами используется метод push():

```
numsInPrioQ.push(5); // Элементы размещаются в порядке убывания
```

Извлечение же осуществляется с начала с помощью метода pop():

```
numsInPrioQ.pop(); // Извлечение элемента из начала очереди
```

Использование методов класса priority_queue показано в листинге 24.6.

ЛИСТИНГ 24.6. Работа с очередью с приоритетами с помощью методов push(), top() и pop()

```

0: #include <queue>
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     priority_queue<int> numsInPrioQ;
8:     cout << "Вставка {10, 5, -1, 20} в priority_queue" << endl;
9:     numsInPrioQ.push(10);
10:    numsInPrioQ.push(5);
11:    numsInPrioQ.push(-1);
12:    numsInPrioQ.push(20);
13:
14:    cout << numsInPrioQ.size() << " элемента в очереди" << endl;
15:    while(!numsInPrioQ.empty())
16:    {
17:        cout << "Удаление элемента " << numsInPrioQ.top() << endl;
18:        numsInPrioQ.pop();
19:    }
20:
21:    return 0;
22: }
```

Результат

```

Вставка {10, 5, -1, 20} в priority_queue
4 элемента в очереди
Удаление элемента 20
Удаление элемента 10
Удаление элемента 5
Удаление элемента -1
```

Анализ

Сначала в коде листинга 24.6 выполняется вставка целых чисел в очередь с приоритетами (строки 9–12), а затем — извлечение элементов из вершины, используя метод `pop()`, как показано в строке 18. Вывод демонстрирует, что в начале очереди всегда находится элемент с самым большим приоритетом. Применение метода `priority_queue::pop()` удаляет элемент, значение приоритета которого наибольшее среди всех элементов контейнера. Ссылку на этот элемент можно получить с помощью метода `top()` (строка 17). Если предикат сравнения приоритетов не задан, очередь по умолчанию сортирует элементы в порядке убывания.

Следующий пример в листинге 24.7 демонстрирует создание экземпляра класса `priority_queue` с предикатом `std::greater<int>`. При этом в очереди наивысший приоритет имеет наименьшее число, которое и будет доступно с помощью метода `top()`.

ЛИСТИНГ 24.7. Создание экземпляра очереди с приоритетами

```
0: #include <queue>
1: #include <iostream>
2: #include <functional>
3: int main()
4: {
5:     using namespace std;
6:
7:     // Очередь priority_queue с предикатом greater<int>
8:     priority_queue<int, vector<int>, greater<int>>> numsInPrioQ;
9:
10:    cout << "Вставка {10, 5, -1, 20} в priority queue" << endl;
11:    numsInPrioQ.push(10);
12:    numsInPrioQ.push(5);
13:    numsInPrioQ.push(-1);
14:    numsInPrioQ.push(20);
15:
16:    cout << numsInPrioQ.size() << " элемента в очереди" << endl;
17:    while(!numsInPrioQ.empty())
18:    {
19:        cout << "Удаление элемента " << numsInPrioQ.top() << endl;
20:        numsInPrioQ.pop();
21:    }
22:
23:    return 0;
24: }
```

Результат

```
Вставка {10, 5, -1, 20} в priority queue
4 элемента в очереди
Удаление элемента -1
```

Удаление элемента 5
 Удаление элемента 10
 Удаление элемента 20

Анализ

Большая часть кода и все значения, предоставляемые объекту `priority_queue` в этом примере, преднамеренно оставлены такими же, как и в предыдущем листинге 24.6. Однако вывод в данном примере демонстрирует, что эти две очереди ведут себя по-разному. Используемая в листинге 24.7 очередь с приоритетами использует для сравнения элементов предикат `greater<int>`, показанный в строке 8. В результате целое число с наименьшим значением считается имеющим наивысший приоритет, а потому помещается в первую позицию очереди. Поэтому функция `top()`, использованная в строке 19, всегда выводит наименьшее целое число перед его извлечением в строке 20 из очереди с помощью метода `pop()`.

Таким образом, когда элементы извлекаются из рассмотренной очереди с приоритетами, целые числа извлекаются из нее в порядке увеличения их значений.

Резюме

На этом занятии рассматривалось применение трех основных адаптивных контейнеров STL: стека, очереди и очереди с приоритетами. Они адаптируют последовательные контейнеры, хранящие данные, к требованиям, предъявляемым к соответствующим структурам данных, предоставляя лишь часть функций-членов, необходимую для моделирования соответствующего поведения.

Вопросы и ответы

■ Может ли быть изменен элемент в середине стека?

Нет, это противоречило бы самому предназначению стека.

■ Могу ли я итерировать все элементы очереди?

Очередь не предоставляет итераторы на свои элементы; обратиться можно только к конечным элементам очереди.

■ Могут ли алгоритмы STL работать с адаптивными контейнерами?

Алгоритмы STL используют итераторы. Поскольку ни класс `stack`, ни класс `queue` не предоставляют итераторы, использование алгоритмов STL с этими контейнерами невозможно.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить

задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Можно ли изменить поведение контейнера `priority_queue` так, чтобы последним извлекался элемент с самым большим значением?
2. Имеется очередь с приоритетами, элементами которой являются объекты класса `Coin` (монета). Какой оператор-член этого класса необходимо определить, чтобы в первой позиции очереди с приоритетами оказывались монеты с наибольшим номиналом?
3. Имеется стек элементов класса `Coin`, содержащий шесть объектов. Можно ли обратиться к элементу, вставленному первым, или извлечь его из стека?

Упражнения

1. Очередь людей (класс `Person`) выстроилась к почтовому отделению. Класс `Person` имеет атрибуты, хранящие возраст и пол, и определен следующим образом:

```
class Person
{
    public:
        int  age;
        bool isFemale;
};
```

Напишите бинарный предикат для использования в `priority_queue`, который позволит сотруднику обслужить сначала стариков и женщин (в указанном порядке).

2. Напишите программу, которая, используя класс `stack`, меняет порядок символов во введенной пользователем строке на обратный.

ЗАНЯТИЕ 25

Работа с битовыми флагами при использовании библиотеки STL

Биты могут быть очень эффективным средством хранения параметров и флагов. Стандартная библиотека шаблонов (STL) предоставляет классы, способные организовать информацию на уровне отдельных битов и работать с ней.

На этом занятии...

- Класс `bitset`
- Класс `vector<bool>`

Класс `bitset`

Класс STL `std::bitset` (множество битов) предназначен для обработки информации в виде битов и битовых флагов. Класс `std::bitset` не является контейнерным классом библиотеки STL, поскольку не способен изменять свои размеры. Это вспомогательный класс, оптимизированный для работы с последовательностью битов, длина которой известна на момент компиляции.

СОВЕТ

Чтобы использовать класс `std::bitset`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <bitset>
```

Инстанцирование класса `std::bitset`

Данному шаблону класса требуется только один параметр, содержащий количество битов, которое должно храниться в объекте этого класса:

```
bitset<4> fourBits; // 4 бита, инициализированных 0000
```

Множество битов можно инициализировать последовательностью битов, представленной в виде строки `char*`:

```
bitset<5> fiveBits("10101"); // 5 битов 10101
```

При инстанцировании можно копировать одно множество битов в другое:

```
bitset<8> eightBitsCopy(eightbits);
```

В листинге 25.1 представлены некоторые из способов создания экземпляра класса `bitset`.

ЛИСТИНГ 25.1. Создание экземпляра класса `std::bitset`

```
0: #include <bitset>
1: #include <iostream>
2: #include <string>
3:
4: int main()
5: {
6:     using namespace std;
7:
8:     bitset<4> fourBits; // 4 бита, инициализированные 0000
9:     cout << "fourBits: " << fourBits << endl;
10:
11:    bitset<5> fiveBits("10101"); // 5 битов – 10101
12:    cout << "fiveBits: " << fiveBits << endl;
13:
14:    bitset<6> sixBits(0b100001); // Бинарный литерал C++14
15:    cout << "sixBits: " << sixBits << endl;
```

```
16:
17:     bitset<8> eightBits(255); // Инициализация значением long int
18:     cout << "eightBits: " << eightBits << endl;
19:
20:     // Инстанцирование как копия другого экземпляра
21:     bitset<8> eightBitsCopy(eightBits);
21:
23:     return 0;
24: }
```

Результат

```
fourBits: 0000
fiveBits: 10101
sixBits: 100001
eightBits: 11111111
```

Анализ

В этом листинге продемонстрированы четыре способа создания объекта класса `bitset`. Конструктор по умолчанию инициализирует битовую последовательность нулями, как показано в строке 9. Строка в стиле C, содержащая строковое представление битовой последовательности, используется для инициализации в строке 11. Тип `unsigned long`, который содержит десятичное значение двоичной последовательности, использован в строке 14, а копирующий конструктор использован в строке 21. Обратите внимание на то, что в каждом из этих случаев вы обязаны указать в качестве параметра шаблона количество битов, которые будет содержать это множество. Данное значение фиксируется во время компиляции и динамически во время выполнения не изменяется. В отличие от вектора, во множество нельзя вставить битов больше, чем было определено во время компиляции.

СОВЕТ

Обратите внимание на бинарный литерал `0b100001` в строке 14. Префикс `0b` или `0B` указывает компилятору, что последующие цифры представляют собой бинарное представление целого числа. Эта новинка появилась в C++, начиная со стандарта C++14.

Использование класса `std::bitset` и его членов

Класс `bitset` предоставляет функции-члены, позволяющие осуществить установку и сброс битов, их чтение и запись в поток. Он предоставляет также операторы, позволяющие выводить содержимое множества битов и выполнять побитовые логические операции.

Полезные операторы, предоставляемые классом `std::bitset`

Операторы рассматривались на занятии 12, “Типы операторов и их перегрузка”, на котором вы узнали, что важнейшая задача операторов заключается в обеспечении удобства и простоты использования класса. Класс `std::bitset` предоставляет несколько весьма полезных операторов, представленных в табл. 25.1 и существенно облегчающих его использование. Примеры, объясняющие использование операторов, подразумевают множество битов `fourBits` из листинга 25.1.

ТАБЛИЦА 25.1. Операторы, поддерживаемые классом `std::bitset`

Оператор	Описание
<code>operator<<</code>	Передаёт текстовое представление битовой последовательности в поток вывода <code>cout << fourBits;</code>
<code>operator>></code>	Передаёт строку в объект класса <code>bitset</code> <code>"0101" >> fourBits;</code>
<code>operator&</code>	Выполняет побитовую операцию И <code>bitset<4> result(fourBits1 & fourBits2);</code>
<code>operator </code>	Выполняет побитовую операцию ИЛИ <code>bitwise<4> result(fourBits1 fourBits2);</code>
<code>operator^</code>	Выполняет побитовую операцию ИСКЛЮЧАЮЩЕГО ИЛИ <code>bitwise<4> result(fourBits1 ^ fourBits2);</code>
<code>operator~</code>	Выполняет побитовую операцию НЕ <code>bitwise<4> result(~fourBits1);</code>
<code>operator>>=</code>	Выполняет оператор битового сдвига вправо <code>fourBits >>= (2); // Сдвиг на два бита вправо</code>
<code>operator<<=</code>	Выполняет оператор битового сдвига влево <code>fourBits <<= (2); // Сдвиг на два бита влево</code>
<code>operator[N]</code>	Возвращает ссылку на бит номер <i>N</i> в последовательности <code>fourBits[2] = 0; // Установить третий бит в 0</code> <code>bool bNum = fourBits[2]; // Читать третий бит</code>

В дополнение к ним класс `std::bitset` предоставляет такие присваивающие операторы, как `|=`, `&=`, `^=` и `~=`.

Методы класса `std::bitset`

Биты могут находиться в двух состояниях: они либо установлены (1), либо сброшены (0). Для работы с содержимым множества битов можно воспользоваться функциями-членами класса `bitset` (табл. 25.2), позволяющими работать с отдельными (или со всеми) битами множества.

ТАБЛИЦА 25.2. Методы класса `std::bitset`

Функция	Описание
<code>set()</code>	Устанавливает все биты последовательности равными 1 <code>fourBits.set();</code> // Теперь множество содержит 1111
<code>set(N, val=1)</code>	Присваивает биту номер <i>N</i> значение <i>val</i> (по умолчанию — 1) <code>fourBits.set(2,0);</code> // Установить третий бит равным 0
<code>reset()</code>	Сбрасывает все биты последовательности в 0 <code>fourBits.reset();</code> // Теперь множество содержит 0000
<code>reset(N)</code>	Сбрасывает бит номер <i>N</i> <code>fourBits.reset(2);</code> // Теперь третий бит равен 0
<code>flip()</code>	Инвертирует все биты последовательности <code>fourBits.flip();</code> // 0101 изменилось на 1010
<code>size()</code>	Возвращает количество битов последовательности <code>size_t NumBits = fourBits.size();</code> // Возвращает 4
<code>count()</code>	Возвращает количество установленных битов <code>size_t NumBitsSet = fourBits.count();</code> <code>size_t NumBitsReset = fourBits.size() - fourBits.count();</code>

Применение приведенных выше операторов и методов продемонстрировано в листинге 25.2.

ЛИСТИНГ 25.2. Логические операции с множеством битов

```

0: #include <bitset>
1: #include <string>
2: #include <iostream>
3:
4: int main()
5: {
6:     using namespace std;
7:     bitset <8> inputBits;
8:     cout << "Введите последовательность из 8 битов: ";
9:
10:    cin >> inputBits; // Пользовательский ввод множества
11:
12:    cout << "Вы ввели: единиц " << inputBits.count() << endl;
13:    cout << "           : нулей  ";
14:    cout << inputBits.size() - inputBits.count() << endl;
15:
16:    bitset <8> inputFlipped(inputBits); // Копирование
17:    inputFlipped.flip();                // Инверсия битов
18:
19:    cout << "Инвертированное множество: " << inputFlipped << endl;
20:
21:    cout << "Выполнение логических операций:" << endl;
22:    cout << inputBits << " & " << inputFlipped << " = ";
23:    cout << (inputBits & inputFlipped) << endl; // Побитовое И

```

```
24:
25:     cout << inputBits << " | " << inputFlipped << " = ";
26:     cout << (inputBits | inputFlipped) << endl; // Побитовое ИЛИ
27:
28:     cout << inputBits << " ^ " << inputFlipped << " = ";
29:     cout << (inputBits ^ inputFlipped) << endl; // Побитовое XOR
30:
31:     return 0;
32: }
```

Результат

Введите последовательность из 8 битов: **10110110**

Вы ввели: единиц 5

: нулей 3

Инвертированное множество: 01001001

Выполнение логических операций:

10110110 & 01001001 = 00000000

10110110 | 01001001 = 11111111

10110110 ^ 01001001 = 11111111

Анализ

Эта интерактивная программа демонстрирует не только простые бинарные операции между двумя последовательностями битов с использованием класса `std::bitset`, но и удобство его потоковых операторов. Операторы сдвига (`>>` и `<<`), реализованные классом `std::bitset`, позволяют выводить последовательности битов на экран и читать небольшие последовательности, вводимые пользователем. Множество битов `inputBits` получает введенную пользователем последовательность (строка 10). Используемый в строке 12 метод `count()` сообщает количество единиц в последовательности, а количество нулей вычисляется как разность между возвращаемыми значениями метода `size()`, возвращающего количество битов в множестве, и метода `count()`, как показано в строке 14. Множество битов `inputFlipped`, которое изначально представляет собой копии множества `inputBits`, далее инвертируется с использованием метода `flip()` в строке 17. Теперь оно содержит последовательность инвертированных битов, в котором биты, имевшие нулевые значения, стали единичными, а единичные — нулевыми. Остальная часть программы демонстрирует результат выполнения побитовых операций И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ между этими двумя множествами битов.

ПРИМЕЧАНИЕ

Одним из недостатков шаблона класса `bitset<N>` библиотеки STL является его неспособность динамически изменять свои размеры. Вы можете использовать класс `bitset` только там, где количество хранимых битов известно во время компиляции.

Библиотека STL снабжает программиста классом `vector<bool>` (в некоторых реализациях STL именуемым также `bit_vector`), который преодолевает этот недостаток.

Класс `vector<bool>`

Класс `vector<bool>` является частичной специализацией класса `std::vector`, предназначенной для хранения логических данных. Этот класс в состоянии динамически измерить свой размер, поэтому программист может не знать заранее количества логических флагов во время компиляции.

СОВЕТ

Чтобы использовать класс `std::vector<bool>`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <vector>
```

Создание экземпляра класса `vector<bool>`

Экземпляр класса `vector<bool>` создается подобно вектору:

```
vector<bool> boolFlags1;
```

Например, можно создать вектор с 10 логическими флагами, инициализированными значением 1 (т.е. `true`):

```
vector<bool> boolFlags2(10, true);
```

Вы можете также создать объект как копию другого объекта:

```
vector<bool> boolFlags2Copy(boolFlags2);
```

Некоторые из способов создания экземпляра класса `vector<bool>` представлены в листинге 25.3.

ЛИСТИНГ 25.3. Создание экземпляра класса `vector<bool>`

```
0: #include <vector>
1:
2: int main()
3: {
4:     using namespace std;
5:
6:     // Создание экземпляра объекта конструктором по умолчанию
7:     vector<bool> boolFlags1;
8:
9:     // Инициализация вектора 10 элементами true
10:    vector<bool> boolFlags2(10, true);
11:
12:    // Создание объекта как копии другого объекта
13:    vector<bool> vecBool2Copy(vecBool2);
14:
15:    return 0;
16: }
```

Анализ

Здесь продемонстрированы некоторые из способов создания объекта класса `vector<bool>`. В строке 7 используется конструктор по умолчанию. В строке 10 показано создание объекта, который изначально содержит 10 логических флагов, инициализированных значением `true`. Строка 13 демонстрирует, как один объект класса `vector<bool>` может быть создан как копия другого.

Функции и операторы класса `vector<bool>`

Класс `vector<bool>` предоставляет функцию `flip()`, которая инвертирует состояние логических значений в последовательности подобно функции `bitset<>::flip()`.

В остальном этот класс очень похож на класс `std::vector` в том смысле, что можно применить функцию `push_back()` к флагам последовательности. Пример в листинге 25.4 демонстрирует применение этого класса в подробностях.

ЛИСТИНГ 25.4. Использование класса `vector<bool>`

```
0: #include <vector>
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     vector<bool> boolFlags(3); // 3 логических флага
7:     boolFlags[0] = true;
8:     boolFlags[1] = true;
9:     boolFlags[2] = false;
10:
11:     boolFlags.push_back(true); // Добавить четвертый флаг
12:
13:     cout << "Содержимое вектора: " << endl;
14:     for(size_t index = 0; index < boolFlags.size(); ++index)
15:         cout << boolFlags[index] << ' ';
16:
17:     cout << endl;
18:     boolFlags.flip();
19:
20:     cout << "Содержимое вектора: " << endl;
21:     for(size_t index = 0; index < boolFlags.size(); ++index)
22:         cout << boolFlags[index] << ' ';
23:
24:     cout << endl;
25:
26:     return 0;
27: }
```

Результат

```
Содержимое вектора:
1 1 0 1
Содержимое вектора:
0 0 1 0
```

Анализ

Здесь для обращения к логическим флагам в векторе используется оператор `operator[]` (строки 7–9), как и в обычном векторе. Функция `flip()` используется в строке 18 для инверсии индивидуальных битовых флагов, по существу преобразовывая все 0 в 1 и обратно. Обратите внимание на применение функции `push_back()` в строке 11. Хотя изначально вектор `boolFlags` был создан для хранения трех флагов (строка 6), в строке 11 к нему добавляется еще один. Добавление большего количества флагов, чем было определено вначале, а также возможность выбрать их количество динамически уже после компиляции отличают `vector<bool>` от класса `std::bitset`.

СОВЕТ

Начиная с C++11 вы можете инстанцировать `boolFlags` из листинга 25.4 с помощью инициализации списком:

```
vector<bool> boolFlags {true, true, false};
```

Резюме

На сегодняшнем занятии рассмотрен весьма эффективный инструмент для работы с битовыми последовательностями и флагами: класс `std::bitset`. Вы также узнали о классе `vector<bool>`, который позволяет хранить логические флаги, количество которых необязательно знать во время компиляции.

Вопросы и ответы

- В ситуации, в которой применимы оба класса, `std::bitset` и `vector<bool>`, какой из них вы предпочли бы для хранения бинарных флагов?

Класс `std::bitset`, поскольку он лучше всего подходит для этого требования.

- У меня есть объект `myBitSeq` класса `std::bitset`, в котором содержится определенное количество битов. Как мне определить количество битов со значением 0 (или `false`)?

Метод `bitset::count()` возвращает количество битов со значением 1. Вычтя это значение из значения, возвращенного методом `bitset::size()` (общее количество хранимых битов), мы получим количество 0 в последовательности.

- Могу ли я использовать итераторы для доступа к индивидуальным элементам в объекте класса `vector<bool>`?

Да. Поскольку класс `vector<bool>` — это частичная специализация класса `std::vector`, а этот класс поддерживает итераторы.

- Могу ли я задать количество элементов, которые будут храниться в объекте класса `vector<bool>`, во время компиляции?

Да, либо указав их количество в перегруженном конструкторе, либо используя функцию `vector<bool>::resize()` позднее.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приставайте к изучению материала следующего занятия.

Контрольные вопросы

1. Может ли множество битов расширять свой внутренний буфер для хранения переменного количества элементов?
2. Почему класс `bitset` не считается контейнерным классом STL?
3. Стоит ли использовать класс `std::vector` для хранения фиксированного количества битов, известного на момент компиляции?

Упражнения

1. Создайте объект класса `bitset`, содержащий четыре бита. Инициализируйте его числом, отобразите результат и сложите его с другим множеством битов. (Предостережение: множества битов не допускают синтаксис `bitsetA = bitsetX + bitsetY`.)
2. Покажите, как бы вы инвертировали биты в множестве битов.

Часть V

Сложные концепции C++

В ЭТОЙ ЧАСТИ...

ЗАНЯТИЕ 26. Понятие интеллектуальных указателей

ЗАНЯТИЕ 27. Применение потоков для ввода и вывода

ЗАНЯТИЕ 28. Обработка исключений

ЗАНЯТИЕ 29. Что дальше

ЗАНЯТИЕ 26

Понятие интеллектуальных указателей

Программисты C++ не обязаны применять простые типы указателей при работе с динамической памятью. Вместо этого они могут использовать интеллектуальные указатели.

На этом занятии...

- Что такое интеллектуальный указатель и зачем он нужен
- Как реализуются интеллектуальные указатели
- Типы интеллектуальных указателей
- Почему не следует использовать устаревший тип `std::auto_ptr`
- Интеллектуальный указатель STL `std::unique_ptr`
- Популярные библиотеки интеллектуальных указателей

Что такое интеллектуальный указатель

Попросту говоря, *интеллектуальный указатель* (smart pointer) C++ — это класс с перегруженными операторами, который ведет себя, как обычный указатель. В то же время он предоставляет дополнительные возможности в обеспечении надлежащего и своевременного освобождения динамически создаваемых данных и облегчает управление жизненным циклом объекта.

Проблемы обычных указателей

В отличие от других современных языков программирования, C++ предоставляет разработчику полную свободу в выделении, освобождении и управлении памятью. К сожалению, эта свобода — палка о двух концах. С одной стороны, она придает языку C++ его мощь, но с другой — обеспечивает возможность таких связанных с памятью проблем, как утечка памяти, когда динамически создаваемые объекты оказываются не освобожденными.

Например:

```
CData *pData = mObject.GetData();  
/*
```

Вопрос: Был ли объект, на который указывает указатель pData, динамически выделен с использованием оператора new?
Кто его освобождает: вызывающая сторона или вызываемая?

```
*/  
Ответ: Неизвестно!  
*/  
pData->Display();
```

В приведенном выше примере кода нет никакого очевидного способа выяснить информацию об объекте, на который указывает указатель pData.

- Создан ли он в динамически выделенной для него памяти (а поэтому в конечном счете должен быть освобожден)?
- Несет ли ответственность за его освобождение вызывающая сторона?
- Будет ли он автоматически освобожден деструктором объекта?

Хотя частично такие двусмысленности могут быть решены за счет вставки комментариев и соблюдения общепринятых правил написания кода, эти механизмы слишком свободны, чтобы эффективно предотвратить все ошибки, причиной которых является неправильное обращение с динамически выделенными данными и указателями.

Чем могут помочь интеллектуальные указатели

С учетом описанных выше проблем использования обычных указателей и способов управления памятью следует заметить, что программист C++ не обязан использовать именно их, когда дело доходит до управления данными в динамической памяти. Программист может выбрать более безопасный и надежный способ управления выделением памяти и динамическими данными с помощью интеллектуальных указателей:

```
smart_pointer<CData> spData = mObject.GetData();

// Используем интеллектуальный указатель, как обычный!
spData->Display();
(*spData).Display();

// Можно не заботиться об освобождении памяти (деструктор
// интеллектуального указателя сделает это самостоятельно)
```

Таким образом, интеллектуальные указатели ведут себя, как обычные указатели (будем называть их *простыми указателями* (raw pointer)), но предоставляют полезные возможности с помощью *перегруженных операторов* и *деструкторов*, гарантируя своевременное освобождение динамически выделенных данных.

Как реализованы интеллектуальные указатели

Пока что этот вопрос может быть упрощен до вопроса “Почему интеллектуальный указатель `spData` способен функционировать, как обычный указатель?” Ответ таков: чтобы позволить программисту использовать их, как обычные указатели, классы интеллектуальных указателей перегружают оператор разыменования (*) и оператор обращения к члену (->). Перегрузка операторов обсуждалась ранее, на занятии 12, “Типы операторов и их перегрузка”.

Кроме того, чтобы позволить вам управлять объектами в динамической памяти, тип которых вы выбираете сами, почти все хорошие классы интеллектуальных указателей являются шаблонными и содержат обобщенную реализацию своих функциональных возможностей. Будучи шаблонами, они обладают универсальностью и могут быть специализированы для управления объектами с выбранным вами типом.

В листинге 26.1 содержится типичная реализация простого класса интеллектуального указателя.

ЛИСТИНГ 26.1. Минимально необходимые компоненты класса интеллектуального указателя

```
0: template <typename T>
1: class smart_pointer
2: {
3: private:
4:     T* rawPtr;
5: public:
6:     smart_pointer(T*pData):rawPtr(pData){} // Конструктор
7:     ~smart_pointer(){ delete pData; };      // Деструктор
8:
9:     // Копирующий конструктор
10:    smart_pointer(const smart_pointer & anotherSP);
11:    // Оператор копирующего присваивания
12:    smart_pointer& operator=(const smart_pointer& anotherSP);
```

```
13:
14:     T& operator*() const    // Оператор разыменования
15:     {
16:         return *(rawPtr);
17:     }
18:
19:     T* operator->() const // Оператор обращения к члену
20:     {
21:         return rawPtr;
22:     }
23: };
```

Анализ

Показанный выше класс интеллектуального указателя демонстрирует реализацию операторов `*` и `->`, объявленных в строках 14–17 и 19–22. Они позволяют этому классу функционировать как “указатель” в обычном смысле. Например, чтобы использовать интеллектуальный указатель на объект класса `Tuna`, вы создаете его экземпляр следующим образом:

```
smart_pointer<Tuna> smartTuna(new Tuna);
smartTuna->Swim();
// Альтернативный вариант:
(*pSmartDog).Swim();
```

Данный класс `smart_pointer` пока еще не имеет и не реализует функциональных возможностей, которые сделали бы его достаточно интеллектуальным классом и обеспечили бы его преимущество перед обычным указателем. Конструктор (строка 6) получает указатель, который сохраняется в классе интеллектуального указателя как внутренний объект. Деструктор освобождает этот указатель, обеспечивая автоматическое освобождение памяти.

ПРИМЕЧАНИЕ

Реализация, которая делает интеллектуальный указатель действительно интеллектуальным, — это реализация копирующего конструктора, оператора присваивания и деструктора. Именно они определяют поведение объекта интеллектуального указателя, когда он передается в функции, присваивается или выходит из области видимости (т.е. уничтожается). Поэтому перед переходом к изучению полной реализации интеллектуального указателя следует рассмотреть некоторые возможные типы интеллектуальных указателей.

Типы интеллектуальных указателей

Управление памятью (т.е. реализация модели владения) представляет собой то, чем отличаются классы интеллектуальных указателей. Интеллектуальные указатели решают, что делать с ресурсом при копировании и присваивании. Самые простые

реализации зачастую приводят к проблемам производительности, тогда как самые быстрые могут не удовлетворять требованиям всех приложений. В конце концов, разработчик должен понимать, как функционирует тот или иной интеллектуальный указатель, прежде чем решит использовать его в своем приложении.

Классификация интеллектуальных указателей фактически основана на их стратегии управления ресурсами памяти.

- Глубокое копирование
- Копирование при записи
- Подсчет ссылок
- Список ссылок
- Деструктивное копирование

Давайте бегло рассмотрим каждую из этих стратегий, прежде чем переходить к изучению интеллектуального указателя `std::unique_ptr`, предоставляемого стандартной библиотекой C++.

Глубокое копирование

Каждый экземпляр интеллектуального указателя, реализующего глубокое копирование, содержит полную копию объекта, которым он управляет. Всякий раз, когда копируется интеллектуальный указатель, копируется и объект, на который он указывает (т.е. осуществляется глубокое копирование). Интеллектуальный указатель, выходя из области видимости, освобождает память, на которую указывает (с помощью деструктора).

Преимущество такого интеллектуального указателя становится очевидным при работе с полиморфными объектами, как демонстрирует приведенный далее код, в котором интеллектуальный указатель позволяет избежать *срезки* (slicing):

```
// Пример срезки при передаче полиморфных объектов по значению
// Fish - базовый класс для классов Tuna и Carp, а
// Fish::Swim() - виртуальная функция
void MakeFishSwim(Fish aFish) // Обратите внимание на тип параметра
{
    aFish.Swim();             // Виртуальная функция
}

// ... Некая функция
Carp freshWaterFish;
MakeFishSwim(freshWaterFish); // Срезка: функции MakeFishSwim() пере-
                              // дается только часть Fish объекта Carp

Tuna marineFish;
MakeFishSwim(marineFish);    // Снова срезка
```

Проблема срезки решается, когда разработчик использует интеллектуальный указатель на основе глубокого копирования, как показано в листинге 26.2.

ЛИСТИНГ 26.2. Использование интеллектуального указателя на основе глубокого копирования для передачи полиморфных объектов через их базовые типы

```
0: template <typename T>
1: class deepcopy_smart_ptr
2: {
3:     private:
4:         T* object;
5:     public:
6:         //... прочие функции
7:
8:         // копирующий конструктор указателя
9:         deepcopy_smart_ptr(const deepcopy_smart_ptr& source)
10:        {
11:            // Clone() - виртуальная функция , гарантирующая глубокое
12:            object = source->Clone();           // копирование
13:        }
14:
15:        // Оператор копирующего присваивания
16:        deepcopy_smart_ptr& operator=(const deepcopy_smart_ptr& source)
17:        {
18:            if (object)
19:                delete object;
20:
21:            object = source->Clone();
22:        }
23: };
```

Анализ

Как можно видеть, класс `deepcopy_smart_pointer` в строках 9–13 реализует копирующий конструктор, который обеспечивает глубокое копирование полиморфного объекта с помощью функции `Clone()`, которую должен реализовать класс. Точно так же реализуется оператор копирующего присваивания в строках 16–22. Для простоты в этом примере подразумевается, что виртуальная функция `Clone()` реализована базовым классом `Fish`. Как правило, интеллектуальные указатели, реализующие модель глубокого копирования, получают данную функцию либо как параметр шаблона, либо как функциональный объект.

Таким образом, когда интеллектуальный указатель передается как указатель на базовый класс `Fish`, часть `Carp` не срезается:

```
deepcopy_smart_ptr<Carp> freshWaterFish(new Carp);
MakeFishSwim(freshWaterFish); // Carp не будет срезан
```

Глубокое копирование, реализованное в конструкторе интеллектуального указателя, обеспечивает передачу объекта без срезки, даже при том что синтаксически функции `MakeFishSwim()` требуется только его базовая часть.

Недостаток механизма глубокого копирования — в низкой производительности. Для некоторых приложений это не проблема, но в ряде случаев потеря производительности может не позволить программисту использовать такой интеллектуальный указатель в своем приложении. В такой ситуации программист может просто передать функции наподобие `MakeFishSwim()` указатель на базовый класс (обычный указатель `Fish*`). Другие интеллектуальные указатели пытаются решить проблему снижения производительности иными методами.

Механизм копирования при записи

Идиома *копирования при записи* (Copy on Write — COW) пытается оптимизировать производительность интеллектуальных указателей с глубоким копированием за счет совместного использования указателей до первой попытки записи объекта. При первой попытке вызова неконстантной функции такой указатель обычно создает копию объекта, для которого вызвана эта неконстантная функция, в то время как другие экземпляры указателя продолжают совместно использовать исходный объект.

Такие указатели с копированием при записи имеют множество приверженцев. Поклонники указателей COW полагают реализацию константных и неконстантных версий операторов `*` и `->` ключевым моментом, обеспечивающим функциональность таких указателей. Неконстантные версии операторов создают копии.

Главное при выборе для своего проекта указателя, реализующего идиому копирования при записи, — это ясное понимание подробностей его реализации до его применения в своих программах. В противном случае вы рискуете оказаться в ситуации, когда копий окажется или слишком мало, или слишком много.

Интеллектуальные указатели со счетчиком ссылок

Идиома *счетчика ссылок* (reference counting) в общем случае представляет собой механизм подсчета количества пользователей объекта. Когда их количество сокращается до нуля, объект освобождается. Счетчик ссылок — это очень хороший механизм для совместного использования объектов без необходимости их копирования. Если вы когда-либо работали с технологией Microsoft под названием “COM”, то концепция подсчета ссылок, определенно, должна быть вам знакома.

У таких интеллектуальных указателей должен быть счетчик ссылок, увеличивающийся при копировании объекта указателя. Есть по крайней мере два популярных способа хранения этого счетчика.

- Счетчик ссылок содержится в объекте, на который указывает указатель.
- Счетчик ссылок поддерживается классом указателя и представляет собой совместно используемый объект.

Первый вариант, когда счетчик ссылок содержится в объекте, называется *внедренным счетчиком ссылок* (intrusive reference counting), поскольку при этом должен быть изменен сам объект. В этом случае объект содержит счетчик ссылок, осуществляет его инкремент и предоставляет его значение любому классу интеллектуального указателя, управляющему им. Кстати, именно этот подход используется в технологии COM.

Второй вариант, когда счетчик ссылок содержится в совместно используемом объекте, представляет собой механизм хранения в динамически выделенной памяти счетчика ссылок (например, в виде целочисленного значения), который при копировании увеличивается копирующим конструктором, а при уничтожении интеллектуального указателя уменьшается его деструктором.

Механизм счетчика ссылок удобен при работе интеллектуальных указателей только при использовании объектов. Управление объектами с помощью интеллектуальных указателей при одновременном наличии простых указателей оказывается не лучшей идеей, поскольку интеллектуальный указатель (интеллектуально) освобождает объект, когда счетчик ссылок доходит до нуля, но простой указатель при этом продолжает указывать на область памяти, которая вашему приложению больше не принадлежит. Кроме того, подсчет ссылок может вызывать специфические проблемы в некоторых ситуациях, например при наличии двух объектов, которые содержат указатели один на другой. Наличие такой *циклической зависимости* (cyclic dependency) удерживает значение счетчиков ссылок отличным от нуля.

Интеллектуальный указатель со списком ссылок

Интеллектуальный указатель *со списком ссылок* (reference-linked) не подсчитывает количество ссылок; ему надо только знать, когда количество ссылок достигнет нуля, чтобы можно было освободить объект.

Такие интеллектуальные указатели называются указателями со списком ссылок потому, что их реализация основана на двухсвязном списке. Когда новый интеллектуальный указатель создается как копия существующего, он добавляется в список. Когда интеллектуальный указатель выходит из области видимости или удаляется, деструктор удаляет этот указатель из этого списка. Такой интеллектуальный указатель, как и интеллектуальный указатель со счетчиком ссылок, страдает от проблем, вызванных циклической зависимостью.

Деструктивное копирование

Деструктивное копирование (destructive copy) — это механизм, который при копировании интеллектуального указателя передает получателю полное владение хранимым объектом, а сам сбрасывается:

```
destructive_copy_smartptr<SampleClass> smartPtr(new SampleClass());
```

```
SomeFunc(smartPtr); // Владение передается в SomeFunc  
// Не используйте больше smartPtr в вызывающей функции!
```

Хотя принцип деструктивного копирования не назовешь интуитивно понятным, преимущество использования интеллектуального указателя с деструктивным копированием заключается в том, что он гарантирует существование в любой момент времени только одного активного указателя на объект. Это очень хорошо подходит для возвращения указателя из функций, а также полезно в случаях, когда деструктивное поведение можно использовать в ваших интересах.

Реализация указателей с деструктивным копированием отличается от рекомендованных стандартных подходов программирования на языке C++ (листинг 26.3).

ВНИМАНИЕ!

Указатель `std::auto_ptr` является, безусловно, наиболее популярным (или известным, в зависимости от вашей точки зрения) указателем деструктивного копирования. Такой интеллектуальный указатель бесполезен после того, как он был передан функции или скопирован в другой.

Использование указателя `std::auto_ptr` в языке C++11 не рекомендовано. Вместо него следует использовать указатель `std::weak_ptr`, который не может быть передан по значению благодаря закрытым копирующему конструктору и оператору копирующего присваивания; он может быть передан как аргумент только по ссылке.

ЛИСТИНГ 26.3. Типичный интеллектуальный указатель деструктивного копирования

```

0: template <typename T>
1: class destructivecopy_ptr
2: {
3: private:
4:     T* object;
5: public:
6:     destructivecopy_ptr(T* input):object(input) {}
7:     ~destructivecopy_ptr() { delete object; }
8:
9:     // Копирующий конструктор
10:    destructivecopy_ptr(destructivecopy_ptr& source)
11:    {
12:        // Получение копии во владение
13:        object = source.object;
14:
15:        // Удаление источника
16:        source.object = 0;
17:    }
18:
19:    // Оператор копирующего присваивания
20:    destructivecopy_ptr& operator=(destructivecopy_ptr& rhs)
21:    {
22:        if (object != source.object)
23:        {
24:            delete object;
25:            object = source.object;
26:            source.object = 0;
27:        }
28:    }
29: };
30:
31: int main()

```

```
32: {  
33:     destructivecopy_ptr<int> num(new int);  
34:     destructivecopy_ptr<int> copy = num;  
35:  
36:     // num теперь является некорректным указателем  
37:     return 0;  
38: }
```

Анализ

В листинге 26.3 показана самая важная часть реализации интеллектуального указателя с деструктивным копированием. Строки 10–17 и 20–28 содержат копирующий конструктор и оператор копирующего присваивания соответственно. Эти функции делают копируемый указатель недействительным, т.е. после копирования исходный указатель получает значение `nullptr`, оправдывая тем самым название *деструктивное копирование*. Оператор присваивания делает то же самое. Таким образом, указатель `num` фактически становится некорректным в строке 34 после присваивания другому указателю. Такое поведение для операции присваивания контринтуитивно.

ВНИМАНИЕ!

Копирующий конструктор и оператор копирующего присваивания, которые критически важны для реализации интеллектуальных указателей с деструктивным копированием, продемонстрированные в листинге 26.3, вызывают массу критических замечаний. В отличие от большинства классов C++, у этого класса не может быть копирующего конструктора и оператора присваивания, получающих константные ссылки, поскольку они должны изменять исходный объект, делая его недействительным после копирования. Это не только является отклонением от традиционной семантики копирующего конструктора и оператора присваивания, но и делает использование класса интеллектуального указателя непонятным интуитивно. Мало кто может ожидать, что оригинал после копирования окажется недействительным. Тот факт, что такие интеллектуальные указатели уничтожают исходный объект копирования, делает их неподходящими для использования в контейнерах STL, таких как `std::vector` или любой другой коллекции, которую вы могли бы использовать. Эти контейнеры должны уметь внутренне копировать свое содержимое, а это приводит к тому, что указатели оказываются недействительными.

В силу всех этих причин многие разработчики боятся интеллектуальных указателей с деструктивным копированием как чумы.

СОВЕТ

Начиная со стандарта C++11 применение указателя `std::auto_ptr` не рекомендуется; вместо него следует использовать интеллектуальный указатель `std::weak_ptr`.

Использование интеллектуального указателя `std::unique_ptr`

Класс `std::unique_ptr` — нововведение стандарта C++11; он несколько отличается от класса `auto_ptr` тем, что не допускает копирование и присваивание.

СОВЕТ

Чтобы использовать класс `std::unique_ptr`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <memory>
```

Класс `unique_ptr` — это простой интеллектуальный указатель, подобный представленному в листинге 26.1, но с закрытыми копирующим конструктором и оператором присваивания, чтобы запретить копирование при передаче его в функции по значению или при присваивании. Его применение демонстрируется в листинге 26.4.

ЛИСТИНГ 26.4. Использование класса `std::unique_ptr`

```
0: #include <iostream>
1: #include <memory>    // Для использования std::unique_ptr
2: using namespace std;
3:
4: class Fish
5: {
6:     public:
7:         Fish() {cout << "Fish: создан!" << endl;}
8:         ~Fish() {cout << "Fish: уничтожен!" << endl;}
9:
10:        void Swim() const {cout << "Плავает в воде" << endl;}
11: };
12:
13: void MakeFishSwim(const unique_ptr<Fish>& inFish)
14: {
15:     inFish->Swim();
16: }
17:
18: int main()
19: {
20:     unique_ptr<Fish> smartFish(new Fish);
21:
22:     smartFish->Swim();
23:     MakeFishSwim(smartFish); // OK, MakeFishSwim использует ссылку
24:
25:     unique_ptr<Fish> copySmartFish;
26:     // copySmartFish = smartFish; // Ошибка: operator= закрытый
27:
28:     return 0;
29: }
```

Результат

```
Fish: создан!  
Плавает в воде  
Плавает в воде  
Fish: уничтожен!
```

Анализ

Рассмотрим последовательность вывода. Обратите внимание: при том что объект, на который указывает указатель `smartFish`, был создан в функции `main()`, он был, как и ожидалось, освобожден (автоматически) без явного вызова оператора `delete`. Такое поведение класса `unique_ptr`: выходящий из области видимости указатель освобождает объект, которым владеет, с помощью деструктора. Обратите внимание, как в строке 23 можно передать указатель `smartFish` в качестве аргумента функции `MakeFishSwim()`. Здесь нет копирования, поскольку функция `MakeFishSwim()` получает параметр по ссылке (см. строку 13). Если вы удалите символ ссылки `&` из строки 13, то немедленно получите ошибку компиляции, поскольку копирующий конструктор закрыт и недоступен. Аналогично присваивание одного объекта класса `unique_ptr` другому, как показано в строке 26, также не разрешено благодаря закрытому оператору копирующего присваивания.

Таким образом, указатель класса `unique_ptr` безопаснее, чем указатель класса `auto_ptr` (который ныне не рекомендован к употреблению), поскольку не делает недействительным исходный объект интеллектуального указателя во время копирования или присваивания. Тем не менее он же обеспечивает простое управление памятью, освобождая объект во время уничтожения интеллектуального указателя.

СОВЕТ

В листинге 26.4 показано, что `unique_ptr` не поддерживает копирование:

```
copySmartFish = smartFish; // Ошибка: оператор = закрытый
```

Однако семантика перемещения поддерживается. Таким образом, следующий вариант кода работает:

```
unique_ptr<Fish> sameFish(std::move(smartFish));  
// Теперь указатель smartFish пуст
```

Если вы хотите написать лямбда-выражение с захватом `unique_ptr`, используйте в захвате `std::move()`, что поддерживается начиная со стандарта C++14:

```
std::unique_ptr<char> alphabet(new char);  
*alphabet = 's';  
auto lambda = [capture = std::move(alphabet)]()  
{ std::cout << *capture << endl; };  
// Теперь alphabet пуст, так как его содержимое перемещено  
lambda();
```

Не огорчайтесь, если приведенный выше код кажется вам слишком экзотическим; он и в самом деле сложен, а использованная в нем конструкция, скорее всего, большинству профессиональных программистов в их практике никогда не встретится.

ПРИМЕЧАНИЕ

При написании многопоточного приложения используйте классы `std::shared_ptr` и `std::weak_ptr`, предоставляемые C++11-совместимыми библиотеками. Они облегчают безопасное с точки зрения многопоточности совместное использование объектов на основе счетчиков ссылок.

Популярные библиотеки интеллектуальных указателей

Очевидно, что версия интеллектуального указателя, предоставляемого стандартной библиотекой C++, не в состоянии удовлетворить требования каждого программиста. Поэтому имеется множество библиотек интеллектуальных указателей.

Библиотека Boost (www.boost.org) предоставляет набор хорошо проверенных и документированных классов интеллектуальных указателей, а также многих других полезных вспомогательных классов. Получить подробную информацию об интеллектуальных указателях Boost и загрузить их можно по адресу http://www.boost.org/libs/smart_ptr/smart_ptr.htm.

Резюме

На этом занятии рассмотрено, как использование подходящих интеллектуальных указателей способно сократить количество выделений памяти и помочь в решении вопросов, связанных с владением объектами. Вы изучили различные типы интеллектуальных указателей и, что важнее всего, их поведение в конкретных приложениях. Теперь вы знаете, что не должны использовать указатель `std::auto_ptr`, поскольку он делает недействительным исходный объект при копировании и присваивании. Вы также узнали о новом классе интеллектуального указателя `std::unique_ptr`, совместимом со стандартом C++11.

Вопросы и ответы

- Мне нужен вектор указателей. Могу ли я выбрать класс `auto_ptr` в качестве типа объекта, который будет содержаться в векторе?

Вы вообще не должны использовать класс `std::auto_ptr`. Это не рекомендуется. Единственной операцией копирования или присваивания достаточно, чтобы сделать исходный объект недействительным.

- Какие два оператора должен реализовать класс, чтобы называться классом интеллектуального указателя?

Это операторы разыменования `*` и выбора члена `->`. Они позволяют использовать объекты класса интеллектуального указателя с использованием семантики обычного указателя.

- У меня есть приложение, в котором классы `Class1` и `Class2` содержат атрибуты, указывающие один на другой. Должен ли я использовать в этом случае указатель со счетчиком ссылок?

Вероятно, нет — из-за циклической зависимости, которая не позволит обнулить счетчик ссылок, а следовательно, оставит объекты двух классов в выделенной памяти, не освобождая их.

- Класс `string` также управляет символьным массивом, расположенным в динамической памяти. Является ли класс `string` интеллектуальным указателем? Нет. Такие классы обычно не реализуют операторы `*` и `->`, а поэтому не могут считаться интеллектуальными указателями.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Где стоит поискать готовый интеллектуальный указатель, прежде чем приступить к написанию собственного класса интеллектуального указателя для своего приложения?
2. Может ли интеллектуальный указатель существенно замедлить ваше приложение?
3. Где интеллектуальные указатели со счетчиками ссылок могут хранить эти счетчики?
4. Должен ли связанный список, применяемый в интеллектуальных указателях со списком ссылок, быть односвязным или двухсвязным?

Упражнения

1. **Отладка.** Найдите ошибку в следующем коде:

```
std::auto_ptr<SampleClass> object(new SampleClass());
std::auto_ptr<SampleClass> anotherObject(object);
object->DoSomething();
anotherObject->DoSomething();
```
2. Используйте класс `unique_ptr` для создания экземпляра класса `Carp`, наследника класса `Fish`. Передайте объект как указатель на `Fish` и отметьте комментарием срезку, если таковая имеет место.
3. **Отладка.** Укажите ошибку в следующем коде:

```
std::unique_ptr<Tuna> myTuna(new Tuna);
unique_ptr<Tuna> copyTuna;
copyTuna = myTuna;
```

ЗАНЯТИЕ 27

Применение потоков для ввода и вывода

Фактически вы использовали потоки на всем протяжении этой книги начиная с первого же занятия, где на экран выводилась строка “Hello World” с помощью потока `std::cout`. Пришло время обратить внимание на эту часть языка C++ и изучить потоки с практической точки зрения.

На этом занятии...

- Что такое потоки и как они используются
- Как записывать и читать файлы, используя потоки
- Вспомогательные операции с потоками C++

Концепция потоков

Предположим, вы разрабатываете программу, которая читает данные с диска, выводит их на экран, читает пользовательский ввод с клавиатуры и сохраняет данные на диске. Разве не было бы хорошо, если бы вы могли выполнять все действия чтения и записи с использованием одинаковой схемы независимо от устройства или местоположения, откуда поступают данные? Именно это и обеспечивают потоки C++.

Потоки (stream) C++ — это обобщенная реализация логики чтения и записи (другими словами, ввода и вывода), позволяющая использовать единообразные схемы чтения и записи данных. Эти схемы одинаковы независимо от того, читаете ли вы данные с диска, с клавиатуры или записываете их на диск или на экран. Нужно только использовать соответствующий потоковый класс, а уж его реализация позаботится о подробностях, специфических для устройства или операционной системы.

Давайте обратимся к соответствующей строке из вашей первой программы на C++ (см. листинг 1.1):

```
std::cout << "Hello World!" << std::endl;
```

Здесь `std::cout` — это потоковый объект класса `ostream`, предназначенный для вывода информации на консоль. Чтобы использовать класс `std::cout`, необходимо включить в исходный текст заголовочный файл `<iostream>`, который предоставляет эту и другие функциональные возможности, такие как объект `std::cin`, позволяющий читать из потока.

Что я подразумеваю, когда говорю, что потоки обеспечивают единообразный и специфический для устройств доступ? Например, если бы необходимо было записать текст “Hello World” в текстовый файл, то можно было бы использовать такой синтаксис объекта файлового потока `fsHello`:

```
fsHello << "Hello World!" << endl; // Запись в файловый поток
```

Как можно заметить, после выбора соответствующего потокового класса запись текста “Hello World” в файл практически не отличается от вывода на экран.

СОВЕТ

Оператор `operator<<`, используемый при записи в поток, называется *оператором вывода в поток* (stream insertion operator). Он используется при выводе на экран, в файл и т.д.

Оператор `operator>>`, используемый при записи потока в переменную, называется *оператором извлечения из потока* (stream extraction operator). Он используется при чтении данных, вводимых с клавиатуры, из файла и т.д.

Заметим, что на этом занятии потоки рассматриваются с практической точки зрения.

Важнейшие классы и объекты потоков C++

Язык C++ предоставляет набор стандартных классов и заголовочных файлов, позволяющих выполнять ряд наиболее важных и часто используемых операций ввода и вывода. В табл. 27.1 содержится список наиболее часто используемых классов.

ТАБЛИЦА 27.1. Наиболее часто используемые классы потоков C++ в пространстве имен std

Класс или объект	Описание
cout	Стандартный поток вывода, как правило, переадресуемый на консоль
cin	Стандартный поток ввода, как правило, используемый для чтения данных в переменные
cerr	Стандартный поток вывода для сообщений об ошибках
fstream	Класс потока ввода и вывода для файловых операций; производный от классов ofstream и ifstream
ofstream	Класс потока вывода для файловых операций, обычно используется для создания файлов
ifstream	Класс потока ввода для файловых операций, обычно используется для чтения из файла
stringstream	Класс потока ввода и вывода для строковых операций; производный от классов istream и ostream; обычно используется для выполнения преобразования в строки (или из строк) других типов

ПРИМЕЧАНИЕ

Объекты cout, cin и cerr являются глобальными объектами потоковых классов ostream, istream и ostream соответственно. Будучи глобальными объектами, они инициализируются до выполнения функции main().

При использовании потокового класса есть возможность использовать *манипуляторы* (manipulator), которые выполняют определенные действия по настройке потоков. Одним из них является манипулятор std::endl, который использовался нами для вывода символа новой строки:

```
std::cout << "Строка заканчивается здесь ->" << std::endl;
```

Некоторые другие манипуляторы и флаги приведены в табл. 27.2.

ТАБЛИЦА 27.2. Наиболее часто используемые манипуляторы для работы с потоками

Манипуляторы вывода	Задача
<code>endl</code>	Вставляет символ новой строки
<code>ends</code>	Вставляет нулевой символ
Манипуляторы систем счисления	Задача
<code>dec</code>	Вынуждает поток интерпретировать ввод или отображать вывод как десятичное число
<code>hex</code>	Вынуждает поток интерпретировать ввод или отображать вывод как шестнадцатеричное число
<code>oct</code>	Вынуждает поток интерпретировать ввод или отображать вывод как восьмеричное число
Манипуляторы представления значений с плавающей запятой	Задача
<code>fixed</code>	Вынуждает поток отображать значения в форме с фиксированной точкой
<code>scientific</code>	Вынуждает поток отображать значения в научном (экспоненциальном) представлении
<code>setprecision</code>	Устанавливает точность десятичного представления, переданную как параметр
<code>setw</code>	Устанавливает ширину поля, переданную как параметр
<code>setfill</code>	Устанавливает символ заполнения, переданный как параметр
<code>setbase</code>	Устанавливает основание системы счисления, используя <code>dec</code> , <code>hex</code> или <code>oct</code> в качестве параметра
<code>setiosflag</code>	Устанавливает флаги с использованием входного параметра-маски с типом <code>std::ios_base::fmtflags</code>
<code>resetiosflag</code>	Восстанавливает значения по умолчанию для определенного типа, указываемого содержимым <code>std::ios_base::fmtflags</code>

Использование `std::cout` для вывода форматированных данных на консоль

Объект `std::cout`, используемый для записи в поток стандартного устройства вывода, является, вероятно, самым используемым потоком в этой книге (и не только). Сейчас пришло время вернуться к потоку `cout` и использовать некоторые из манипуляторов для изменения способа выравнивания и отображения данных.

Изменение формата представления чисел

Поток `std::cout` можно заставить отображать целые числа в шестнадцатеричной или восьмеричной записи. В листинге 27.1 демонстрируется использование потока `cout` для отображения введенных чисел в различных системах счисления.

ЛИСТИНГ 27.1. Отображение целого числа с использованием потока `cout` и флагов `<iomanip>`

```
0: #include <iostream>
1: #include <iomanip>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Введите целое число: ";
7:     int input = 0;
8:     cin >> input;
9:
10:    cout << "Восьмеричная запись      : " << oct << input << endl;
11:    cout << "Шестнадцатеричная запись: " << hex << input << endl;
12:
13:    cout << "Шестнадцатеричная запись с указанием основания: ";
14:    cout << setiosflags(ios_base::hex|ios_base::showbase|
15:                        ios_base::uppercase) << input << endl;
16:
17:    cout << "После сброса флагов ввода-вывода          : ";
18:    cout << resetiosflags(ios_base::hex|ios_base::showbase|
19:                         ios_base::uppercase) << input << endl;
20:
21:    return 0;
22: }
```

Результат

```
Введите целое число: 253
Восьмеричная запись      : 375
Шестнадцатеричная запись: fd
Шестнадцатеричная запись с указанием основания: 0XFD
После сброса флагов ввода-вывода          : 253
```

Анализ

В примере использованы представленные в табл. 27.2 манипуляторы для изменения способа отображения потоком `cout` введенного пользователем целого числа. Обратите внимание на использование манипуляторов `oct` и `hex` в строках 10 и 11. В строках 14 и 15 функция `setiosflags()` используется для задания отображения числа прописными буквами в шестнадцатеричном формате. В результате поток `cout` отображает

введенное целое число 253 как 0XFD. В результате использования функции `resetio-flags()` в строках 18 и 19 поток `cout` снова отображает целое число в десятичном виде. Вот другой способ смены отображения целых чисел в десятичном виде:

```
cout << dec << input << endl; // Отображать в десятичном формате
```

При отображении потоком `cout` таких чисел, как π , можно также задавать точность, т.е. определять количество знаков десятичного числа после запятой, которое будет представлено, либо задать отображение числа в экспоненциальном представлении (листинг 27.2).

ЛИСТИНГ 27.2. Использование `cout` для отображения числа π и площади круга

```
0: #include <iostream>
1: #include <iomanip>
2: using namespace std;
3:
4: int main()
5: {
6:     const double Pi = 3.1415926535898;
7:     cout << "Pi = " << Pi << endl;
8:
9:     cout << endl << "Точность = 7: " << endl;
10:    cout << setprecision(7);
11:    cout << "Pi = " << Pi << endl;
12:    cout << fixed << "Фиксированная запись Pi = " << Pi << endl;
13:    cout << scientific << "Научная запись Pi = " << Pi << endl;
14:
15:    cout << endl << "Точность = 10: " << endl;
16:    cout << setprecision(10);
17:    cout << "Pi = " << Pi << endl;
18:    cout << fixed << "Фиксированная запись Pi = " << Pi << endl;
19:    cout << scientific << "Научная запись Pi = " << Pi << endl;
20:
21:    cout << endl << "Введите радиус: ";
22:    double Radius = 0.0;
23:    cin >> Radius;
24:    cout << "Площадь круга: " << 2*Pi*Radius*Radius << endl;
25:
26:    return 0;
27: }
```

Результат

```
Pi = 3.14159
```

```
Точность = 7:
```

```
Pi = 3.141593
```

Фиксированная запись $\text{Pi} = 3.1415927$

Научная запись $\text{Pi} = 3.1415927\text{e}+00$

Точность = 10:

$\text{Pi} = 3.1415926536\text{e}+00$

Фиксированная запись $\text{Pi} = 3.1415926536$

Научная запись $\text{Pi} = 3.1415926536\text{e}+00$

Введите радиус: 9.99

Площадь круга: $6.2706252198\text{e}+02$

Анализ

Вывод демонстрирует, что при увеличении точности до 7 в строке 10 и до 10 в строке 16 представление значения числа Pi изменяется. Обратите также внимание на то, что после применения манипулятора `scientific` результат вычисления площади круга отображается как $6.2706252198\text{e}+02$.

Выравнивание текста и установка ширины поля

Для установки ширины поля в символах можно использовать такой манипулятор, как `setw()`. В результате любая вставка в поток осуществляется с выравниванием по правому краю с указанной шириной. Аналогично манипулятор `setfill()` применяется для определения символа, заполняющего пустое пространство в ситуации, показанной в листинге 27.3.

ЛИСТИНГ 27.3. Установка ширины поля и символа заполнения

```
0: #include <iostream>
1: #include <iomanip>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "По умолчанию!" << endl;
7:
8:     cout << setw(35); // установка поля шириной 35 символов
9:     cout << "Выравнивание вправо!" << endl;
10:
11:     cout << setw(35) << setfill('*');
12:     cout << "Выравнивание вправо!" << endl;
13:
14:     cout << "Опять по умолчанию!" << endl;
15:
16:     return 0;
17: }
```

Результат

```
По умолчанию!  
                                Выравнивание вправо!  
*****Выравнивание вправо!  
Опять по умолчанию!
```

Анализ

Вывод демонстрирует результат применения манипулятора `setw(35)` в строке 8 и манипулятора `setfill('*')` в строке 11 для объекта `cout`. Как можно видеть, в предпоследней строке вывода свободное пространство, предшествующее тексту, заполнено звездочками, как и определено манипулятором `setfill()`.

Использование `std::cin` для ввода

Поток `std::cin` универсален — он позволяет читать данные простых типов, таких как `int`, `double` или `char*`, а также читать с экрана строки и символы, используя такие методы, как `getline()`.

Использование `std::cin` для ввода простых старых типов данных

С помощью потока `cin` можно читать целые числа, числа с плавающей точкой или символы непосредственно из стандартного устройства ввода. Листинг 27.4 демонстрирует применение потока `cin` для чтения простых типов данных, введенных пользователем.

ЛИСТИНГ 27.4. Использование потока `cin` для чтения простых типов данных

```
0: #include<iostream>  
1: using namespace std;  
2:  
3: int main()  
4: {  
5:     cout << "Введите целое число: ";  
6:     int inputNum = 0;  
7:     cin >> inputNum;  
8:  
9:     cout << "Введите число Pi: ";  
10:    double Pi = 0.0;  
11:    cin >> Pi;  
12:  
13:    cout << "Введите три символа, разделенных пробелами: "<<endl;  
14:    char char1 = '\0', char2 = '\0', char3 = '\0';  
15:    cin >> char1 >> char2 >> char3;  
16:
```

```
17:     cout << "Введены следующие переменные: " << endl;
18:     cout << "inputNum: " << inputNum << endl;
19:     cout << "Pi: " << Pi << endl;
20:     cout << "Три символа: " << char1 << char2 << char3 << endl;
21:
22:     return 0;
23: }
```

Результат

```
Введите целое число: 1234
Введите число Pi: 0.31415926e1
Введите три символа, разделенных пробелами: C + +
Введены следующие переменные:
inputNum: 1234
Pi: 3.14159
Три символа: C++
```

Анализ

Самая интересная часть листинга 27.4 заключается в вводе значения `Pi` с использованием экспоненциальной формы записи и сохранении этих данных в переменной `Pi` типа `double`. Обратите внимание, как три символьные переменные заполняются в пределах одной строки (строка 15).

Использование метода `std::cin::get()` для ввода в буфер `char*`

Поток `cin` позволяет записывать данные непосредственно в переменную типа `int`; подобное можно сделать и с массивом `char` в стиле `C`:

```
cout << "Введите строку: " << endl;
char charBuf[10] = {0}; // Может содержать максимум 10 символов
cin >> charBuf; // Опасно: пользователь может ввести больше 10 символов
```

При записи в символьный буфер очень важно не выйти за его границы, чтобы избежать аварийного завершения программы или нарушения системы безопасности. Поэтому лучше читать данные в символьный буфер следующим образом:

```
cout << "Введите строку: " << endl;
char charBuf[10] = {0};
cin.get(charBuf, 9); // Прекращение вставки на 9-м символе
```

Этот более безопасный способ вставки текста в буфер стиля `C` использован в листинге 27.5.

ЛИСТИНГ 27.5. Вставка в символьный буфер без выхода за его границы

```
0: #include<iostream>
1: #include<string>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Введите строку: " << endl;
7:     char charBuf[10] = {0};
8:     cin.get(charBuf, 9);
9:     cout << "charBuf: " << charBuf << endl;
10:
11:     return 0;
12: }
```

Результат

Введите строку:

Длинная строка, выходящая за границы

charBuf: Длинная с

Анализ

Как показывает вывод, благодаря использованию в строке 8 метода `cin::get()` в буфер `char` записаны только первые девять символов. Это самый безопасный способ работы с буферами фиксированной длины.

СОВЕТ

По возможности не используйте массивы типа `char`. Используйте вместо них тип `std::string`.

Использование `std::cin` для ввода в переменную типа `std::string`

Поток `cin` — весьма универсальный инструмент, позволяющий поместить введенную пользователем строку непосредственно в переменную типа `std::string`:

```
std::string input;
cin >> input; // Прекращение вставки при первом пробеле
```

В листинге 27.6 показан ввод с помощью потока `cin` в переменную типа `std::string`.

ЛИСТИНГ 27.6. Вставка текста в строку `std::string` с помощью `std::cin`

```
0: #include<iostream>
1: #include<string>
2: using namespace std;
```

```
3:
4: int main()
5: {
6:     cout << "Введите ваше имя: ";
7:     string name;
8:     cin >> name;
9:     cout << "Привет, " << name << endl;
10:
11:     return 0;
12: }
```

Результат

Введите ваше имя: **Siddhartha Rao**
Привет, Siddhartha

Анализ

Вывод отобразил мое имя не полностью, поскольку так была реализована программа. Я ожидал, что переменная name, заполняемая из потока cin в строке 8, будет содержать введенные мной имя и фамилию, а не только одно имя. Что же произошло? Поток cin остановил вставку, когда встретился с первым пробелом.

Чтобы позволить пользователю ввести строку полностью, включая пробелы, необходимо использовать функцию `getline()`:

```
string name;
getline(cin, name);
```

Применение функции `getline()` с потоком cin показано в листинге 27.7.

ЛИСТИНГ 27.7. Чтение введенной строки с помощью функции `getline()`

```
0: #include<iostream>
1: #include<string>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Введите ваше имя: ";
7:     string name;
8:     getline(cin, name);
9:     cout << "Привет, " << name << endl;
10:
11:     return 0;
12: }
```

Результат

Введите ваше имя: **Siddhartha Rao**

Привет, Siddhartha Rao

Анализ

Функция `getline()`, показанная в строке 8, решила проблему ввода символа пробела. Теперь вывод содержит введенную пользователем строку полностью.

Использование потока `std::fstream` для работы с файлом

Класс `std::fstream` языка C++ обеспечивает (относительно) независимый от платформы доступ к файлу. Класс `std::fstream` наследует класс `std::ofstream` для записи в файл и класс `std::ifstream` для чтения из него.

Другими словами, класс `std::fstream` обеспечивает возможность как чтения, так и записи.

СОВЕТ

Чтобы использовать класс `std::fstream`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <fstream>
```

Открытие и закрытие файла с помощью методов `open()` и `close()`

Прежде чем использовать объект класса `fstream`, `ofstream` или `ifstream`, необходимо открыть файл с помощью метода `open()`:

```
fstream myFile;
myFile.open("HelloFile.txt", ios_base::in|ios_base::out|ios_base::trunc);

if (myFile.is_open()) // Проверка успешности открытия файла
{
    // Чтение или запись

    myFile.close();
}
```

Метод `open()` получает два аргумента: первый — путь и имя открываемого файла (если не указать путь, то подразумевается текущий каталог приложения), а второй — режим открытия файла. Выбранный выше режим открытия позволяет создать файл заново, даже если он уже существует (`ios_base::trunc`), а также читать и записывать в файл (`in|out`).

Обратите внимание на применение метода `is_open()` для проверки успешности выполнения метода `open()`.

ВНИМАНИЕ!

Заккрытие файлового потока с помощью метода `close()` важно для сохранения его содержимого.

Альтернативный способ открытия файлового потока — с использованием конструктора:

```
fstream myFile("HelloFile.txt",  
               ios_base::in|ios_base::out|ios_base::trunc);
```

Если необходимо открыть файл только для записи, используйте следующий режим открытия:

```
ofstream myFile("HelloFile.txt", ios_base::out);
```

Если же необходимо открыть файл только для чтения, смените режим его открытия, как показано далее:

```
ifstream myFile("HelloFile.txt", ios_base::in);
```

СОВЕТ

Независимо от того, что вы используете, конструктор или метод `open()`, рекомендуется проверять успешность открытия файла с помощью метода `is_open()`, прежде чем использовать соответствующий объект файлового потока.

Файловый поток может быть открыт в нескольких режимах.

- `ios_base::app`. Дозапись в конец существующего файла без его усечения.
- `ios_base::ate`. Переносит файловый указатель в конец файла, но запись данных возможна в любое место файла.
- `ios_base::trunc`. Усекает существующий файл (принят по умолчанию).
- `ios_base::binary`. Создает бинарный файл (по умолчанию создается текстовый файл).
- `ios_base::in`. Открывает файл для чтения.
- `ios_base::out`. Открывает файл для записи.

Создание и запись текстового файла с использованием метода `open()` и оператора `<<`

После открытия файлового потока вы можете писать в него, используя оператор `<<` (листинг 27.8).

ЛИСТИНГ 27.8. Создание нового текстового файла и запись в него

```
0: #include<fstream>
1: #include<iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     ofstream myFile;
7:     myFile.open("HelloFile.txt", ios_base::out);
8:
9:     if (myFile.is_open())
10:    {
11:        cout << "Файл успешно открыт" << endl;
12:
13:        myFile << "Мой первый текстовый файл!" << endl;
14:        myFile << "Привет, файл!";
15:
16:        cout << "Закрытие файла" << endl;
17:        myFile.close();
18:    }
19:
20:    return 0;
21: }
```

Результат

Файл успешно открыт
Закрытие файла

Содержимое файла HelloFile.txt:

Мой первый текстовый файл!
Привет, файл!

Анализ

В строке 7 файл открывается в режиме `ios_base::out`, т.е. только для записи. В строке 9 проверяется успешность выполнения метода `open()`, а затем осуществляется запись в файловый поток с использованием оператора вывода `operator<<`, как показано в строках 13 и 14. И наконец в строке 17 файл закрывается.

ПРИМЕЧАНИЕ

В листинге 27.8 демонстрируется, что запись в файловый поток выполняется точно так же, как на стандартное устройство вывода (т.е. на консоль) с использованием объекта `cout`.

Это означает, что потоки C++ обеспечивают единообразный способ работы с различными устройствами, будь то запись текста на экран с помощью объекта `cout` или запись в файл с помощью объекта типа `ofstream`.

Чтение текстового файла с использованием метода `open()` и оператора `>>`

Для чтения файла можно воспользоваться объектом `fstream`, если открыть его с помощью флага `ios_base::in`, или использовать объект `ifstream`. В листинге 27.9 демонстрируется чтение из файла `HelloFile.txt`, созданного в листинге 27.8.

ЛИСТИНГ 27.9. Чтение текста из файла `HelloFile.txt`, созданного в листинге 27.8

```
0: #include<fstream>
1: #include<iostream>
2: #include<string>
3: using namespace std;
4:
5: int main()
6: {
7:     ifstream myFile;
8:     myFile.open("HelloFile.txt", ios_base::in);
9:
10:    if (myFile.is_open())
11:    {
12:        cout << "Файл успешно открыт. Он содержит:" << endl;
13:        string fileContents;
14:
15:        while(myFile.good())
16:        {
17:            getline(myFile, fileContents);
18:            cout << fileContents << endl;
19:        }
20:
21:        cout << "Закрытие файла." << endl;
22:        myFile.close();
23:    }
24:    else
25:        cout << "open(): ошибка открытия файла" << endl;
26:
27:    return 0;
28: }
```

Результат

```
Файл успешно открыт. Он содержит:
Мой первый текстовый файл!
Привет, файл!
Закрытие файла.
```

ПРИМЕЧАНИЕ

Чтобы код листинга 27.9 прочитал текстовый файл `HelloFile.txt`, созданный в листинге 27.8, следует либо переместить его в рабочий каталог этого проекта, либо объединить этот код с предыдущим.

Анализ

Как обычно, вызов метода `is_open()` в строке 8 проверяет успешность вызова метода `open()`. Обратите внимание на применение оператора `>>` при чтении содержимого файла в переменную типа `string`, которая затем отображается с помощью потока `cout` в строке 18. В данном примере функция `getline()` используется для чтения из файлового потока тем же способом, что и в листинге 27.7 при чтении ввода пользователя, по одной строке за раз.

Запись и чтение из бинарного файла

Процесс записи в бинарный файл по сути не слишком отличается от процесса, который вам уже известен в настоящий момент. При открытии файла следует использовать флаг `ios_base::binary`. Обычно для записи и чтения используются методы `ofstream::write()` и `ifstream::read()`, показанные в листинге 27.10.

ЛИСТИНГ 27.10. Запись структуры в бинарный файл и ее восстановление оттуда

```
0: #include<fstream>
1: #include<iomanip>
2: #include<string>
3: #include<iostream>
4: using namespace std;
5:
6: struct Human
7: {
8:     Human() {} ;
9:     Human(const char* iName, int iAge, const char*iDOB):age(iAge)
10:    {
11:        strcpy(name, iName);
12:        strcpy(DOB, iDOB);
13:    }
14:
15:    char name[30];
16:    int age;
17:    char DOB[20];
18: };
19:
20: int main()
21: {
22:     Human input("Siddhartha Rao", 101, "Май 1916");
23:
24:     ofstream fsOut("MyBinary.bin",ios_base::out|ios_base::binary);
25:
26:     if (fsOut.is_open())
```

```
27:     {
28:         cout << "Запись объекта в бинарный файл" << endl;
29:         fsOut.write((const char*)&input, sizeof(input));
30:         fsOut.close();
31:     }
32:
33:     ifstream fsIn("MyBinary.bin", ios_base::in|ios_base::binary);
34:
35:     if(fsIn.is_open())
36:     {
37:         Human somePerson;
38:         fsIn.read((char*)&somePerson, sizeof(somePerson));
39:
40:         cout << "Чтение объекта из бинарного файла: " << endl;
41:         cout << "Имя      = " << somePerson.name << endl;
42:         cout << "Возраст = " << somePerson.age << endl;
43:         cout << "Родился = " << somePerson.DOB << endl;
44:     }
45:
46:     return 0;
47: }
```

Результат

```
Запись объекта в бинарный файл
Чтение объекта из бинарного файла:
Имя      = Siddhartha Rao
Возраст = 101
Родился = Май 1916
```

Анализ

В строках 22–31 создается экземпляр структуры `Human`, содержащей атрибуты `name`, `age` и `DOB`. Она сохраняется на диске в бинарном файле `MyBinary.bin` с использованием объекта класса `ofstream`. Затем, в строках 33–44, эта информация считывается из файла с использованием другого потокового объекта класса `ifstream`. Информация для вывода атрибутов, таких как `name` и другие, считывается из бинарного файла. Этот пример демонстрирует применение объектов `ifstream` и `ofstream` для чтения и записи файлов с использованием методов `ifstream::read()` и `ofstream::write()` соответственно. Обратите внимание на приведение типов в строках 29 и 38, заставляющее трактовать указатели на структуру как указатели на `char`.¹

¹ Примененный здесь способ записи и чтения с передачей адреса объекта класса работает только в очень редких случаях “старых простых данных” (POD), когда у класса нет виртуальных функций, а все данные содержатся в классе — например, если бы то же поле `name` имело тип `char*` и указывало на имя в динамической памяти, при записи файла в одной программе и чтении в другой произошла бы ошибка (так как в файле была бы сохранена не строка имени, а ее адрес). — *Примеч. ред.*

ПРИМЕЧАНИЕ

Если бы это был не демонстрационный код, я записывал бы структуру `Human` со всеми ее атрибутами в файл XML. Формат XML обеспечивает гибкость и масштабируемость при хранении информации.

Если после сохранения такой структуры, как `Human`, в нее необходимо добавить новые атрибуты (например, `numChildren`), то вам придется позаботиться о том, чтобы метод `ifstream::read()` мог корректно читать бинарные данные, созданные более старой версией.

Использование `std::stringstream` для преобразования строк

Предположим, у нас имеется строка, содержащая строковое значение "45". Как преобразовать это строковое значение в целое число со значением 45 и обратно? Одной из весьма полезных утилит, предоставляемых языком C++, является класс `stringstream`, обеспечивающий выполнение множества преобразований такого рода.

СОВЕТ

Чтобы использовать класс `std::stringstream`, в исходный текст программы необходимо включить соответствующий заголовочный файл:

```
#include <sstream>
```

Некоторые из основных операций класса `stringstream` продемонстрированы в листинге 27.11.

ЛИСТИНГ 27.11. Преобразование целочисленного значения в строковое и обратно с помощью `std::stringstream`

```
0: #include<fstream>
1: #include<sstream>
2: #include<iostream>
3: using namespace std;
4:
5: int main()
6: {
7:     cout << "Введите целое число:";
8:     int input = 0;
9:     cin >> input;
10:
11:     stringstream converterStream;
12:     converterStream << input;
13:     string strInput;
14:     converterStream >> strInput;
15:
```

```
16:     cout << "Введенное число = " << input << endl;
17:     cout << "Преобразовано в строку: " << strInput << endl;
18:
19:     stringstream anotherStream;
20:     anotherStream << strInput;
21:     int copy = 0;
22:     anotherStream >> copy;
23:
24:     cout << "Преобразовано в целое число: " << copy << endl;
25:
26:     return 0;
27: }
```

Результат

Введите целое число: 45
Введенное число = 45
Преобразовано в строку: 45
Преобразовано в целое число: 45

Анализ

Здесь пользователя просят ввести целочисленное значение. Сначала это целое число вносится в объект класса `stringstream` (строка 12) с помощью оператора `<<`. Затем, в строке 14, оператор `>>` используется для преобразования этого целого числа в строку. Далее эта строка используется в качестве исходной для получения очередного целочисленного значения, на этот раз в переменной `copy`.

РЕКОМЕНДУЕТСЯ

Используйте класс `ifstream` тогда, когда намереваетесь только читать из файла.

Используйте класс `ofstream` тогда, когда намереваетесь только писать в файл.

Помните о проверке успешности открытия файлового потока с помощью метода `is_open()`. Используйте ее до того, как использовать данный файловый поток.

НЕ РЕКОМЕНДУЕТСЯ

Не забывайте закрывать файловый поток с помощью метода `close()` после его использования.

Не забывайте, что в результате чтения с помощью оператора `>>`, как в случае

```
cin >> strData;
```

переменная `strData` содержит текст лишь до первого пробела, встреченного во вводимой строке.

Не забывайте, что функция `getline(cin, strData);` извлекает из входного потока всю строку, включая пробелы.

Резюме

На этом занятии рассмотрены потоки C++ с практической точки зрения. Вы с самого начала книги учились использовать такие потоки ввода и вывода, как `cout` и `cin`. Теперь вы знаете, как создавать простые текстовые файлы и выполнять их запись и чтение. Вы узнали, как класс `stringstream` может помочь в преобразовании простых типов, таких как `int`, в строки и обратно.

Вопросы и ответы

- Если я могу использовать класс `fstream` и для записи, и для чтения из файла, то зачем мне классы `ofstream` и `ifstream`?

Если ваш код или модуль должен только читать из файла, то лучше использовать класс `ifstream`. Точно так же, если вам нужна только запись в файл, используйте класс `ofstream`. В обоих случаях можно было бы использовать и класс `fstream`, но для обеспечения целостности данных лучше иметь ограничительную политику, подобную использованию `const` (которое также не является обязательным, но крайне рекомендуется).

- Когда мне использовать метод `cin.get()`, а когда метод `cin.getline()`?

Метод `cin.getline()` гарантирует чтение всей строки, включая введенные пользователем пробелы. Метод `cin.get()` обеспечивает чтение пользовательского ввода по одному символу.

- Когда следует использовать класс `stringstream`?

Класс `stringstream` обеспечивает удобный способ преобразования целых чисел и других простых типов в строки и обратно (см. листинг 27.11).

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приставайте к изучению материала следующего занятия.

Контрольные вопросы

1. В программе необходима только запись в файл. Какой поток следует использовать в этом случае?
2. Как использовать объект `cin` для получения полной строки из входного потока?
3. Необходима запись объекта `std::string` в файл. Следует ли выбрать режим открытия файла `ios_base::binary`?

4. Вы открыли поток с помощью метода `open()`. Почему следует воспользоваться методом `is_open()`?

Упражнения

1. **Отладка.** Найдите ошибку в следующем коде:

```
fstream myFile;  
myFile.open("HelloFile.txt", ios_base::out);  
myFile << "Некоторый текст";  
myFile.close();
```

2. **Отладка.** Найдите ошибку в следующем коде:

```
ifstream myFile("SomeFile.txt");  
if(myFile.is_open())  
{  
    myFile << "Некоторый текст" << endl;  
    myFile.close();  
}
```


ЗАНЯТИЕ 28

Обработка исключений

Как следует из названия главы, речь пойдет об экстраординарных ситуациях, нарушающих выполнение вашей программы. До сих пор на занятиях мы проявляли чрезвычайный оптимизм, подразумевая, что выделение памяти всегда успешно, файлы всегда открываются и т.д. Однако действительность зачастую совершенно иная.

На этом занятии...

- Что такое исключение
- Как обрабатываются исключения
- Как обработка исключений помогает создавать надежные приложения C++

Что такое исключение

Ваша программа запрашивает память, читает и записывает данные, сохраняет файлы — все работает. В вашей великолепной среде разработки все выполняется безупречно, и вы гордитесь тем фактом, что ваше приложение не пропускает ни байта, хотя и управляет гигабайтами! Вы распространяете свое приложение, и клиент устанавливает его на тысячи рабочих станций. Некоторым из его компьютеров по десять лет. Жесткие диски на некоторых из них еле крутятся. Проходит совсем немного времени, и в вашей папке входящих писем появляются первые жалобы. В одних из них будет упоминание о нарушении прав доступа, а в других — о сообщении “Необработанное исключение”.

Вот тебе и на: “необработанное” и “исключение”. В вашей системе приложение работало отлично, так откуда все это взялось?

Все дело в том, что мир очень разнообразен. Не существует двух одинаковых компьютеров даже при одной и той же аппаратной конфигурации. Дело в том, что на каждом компьютере выполняется разное программное обеспечение, а состояние, в котором находится машина, влияет на объем ресурсов, доступных в определенный момент времени. Поэтому вполне вероятно, что отлично работавший в ваших условиях диспетчер памяти отказывает в выделении блока нужного размера в другой среде.

Такие отказы редки, но все же случаются. Эти отказы и приводят к *исключениям* (exception).

Исключения прерывают нормальный поток выполнения вашего приложения. В конце концов, если доступной памяти нет, нет никакой возможности заставить ваше приложение делать то, что оно должно делать. Тем не менее ваше приложение способно обработать исключение и отобразить пользователю сообщение об ошибке, выполнить по мере необходимости операции по сохранению данных и максимально корректно завершить работу.

Обработка исключений поможет избежать таких сообщений, как “Access Violation” или “Unhandled Exception”, а также соответствующих жалоб по электронной почте. Давайте рассмотрим, какие инструменты предоставляет язык C++, чтобы справиться с непредвиденным.

Что вызывает исключения

Исключения могут быть вызваны внешними факторами, например недостатком ресурсов в системе, или внутренними причинами вашего приложения, такими как использование недопустимого указателя или деление на нуль. Некоторые модули разрабатываются так, чтобы сообщать о происшедших ошибках, генерируя исключения, передаваемые вызывающей функции.

ПРИМЕЧАНИЕ

Чтобы защитить свой код от исключений, вы *обрабатываете* (handle) их, делая свой код безопасным в отношении исключений (exception safe).

Реализация безопасности в отношении исключений с помощью блоков try и catch

Когда дело доходит до реализации безопасности в отношении исключений, самыми важными оказываются ключевые слова C++ try и catch. Чтобы сделать инструкции безопасными в отношении исключений, следует поместить их в блок try и в блоке catch обработать исключения, которые будут сгенерированы в блоке try:

```
void SomeFunc()
{
    try
    {
        int* numPtr = new int;
        *numPtr = 999;
        delete numPtr;
    }
    catch(...) // ... Обработывает все исключения
    {
        cout << "Исключение в SomeFunc(), завершение работы" << endl;
    }
}
```

Использование блока catch (...) для обработки всех исключений

Как вы помните, на занятии 8, “Указатели и ссылки”, я упоминал о том, что стандартная форма оператора new возвращает допустимый указатель на область в памяти, если память выделена успешно; в противном случае оператор new генерирует исключение. В листинге 28.1 показано, как выделение памяти с использованием оператора new можно сделать безопасным по отношению к исключениям и как обрабатывать ситуацию, когда компьютер не в состоянии выделить запрошенную память.

ЛИСТИНГ 28.1. Использование блоков try и catch для обеспечения безопасности в отношении исключений при выделении памяти

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     cout << "Количество чисел, для которых нужна память: ";
6:     try
7:     {
8:         int input = 0;
9:         cin >> input;
```



```
10:
11:     // Запрос области в памяти и ее последующее освобождение
12:     int* numArray = new int[input];
13:     delete[] numArray;
14: }
15: catch(...)
16: {
17:     cout << "Извините, перехвачено исключение." << endl;
18: }
19:     return 0;
20: }
```

Результат

Количество чисел, для которых нужна память: -1
Извините, перехвачено исключение.

Анализ

Для этого примера я указал в качестве размера запрашиваемого массива **-1**. Это абсурдное значение, но пользователи иногда делают такие вещи. В отсутствие обработчика исключений работа программы закончилась бы некрасиво. Но благодаря обработчику исключения мы видим более осмысленное сообщение о перехваченном исключении.

В листинге 28.1 демонстрируется применение блоков `catch` и `try`. Блок `catch()` получает параметры, как обычная функция, а “...” означает, что данный блок `catch` принимает исключения всех типов. Но в данном случае мы могли бы захотеть принимать исключения только одного типа, `std::bad_alloc`, поскольку именно они генерируются при неудачном выполнении оператора `new`. Обработка исключений конкретного типа позволяет разделять обрабатывать различные проблемы, в том числе с получением дополнительной информации о проблеме.

Обработка исключения конкретного типа

Исключение в листинге 28.1 генерировалось стандартной библиотекой C++. Типы таких исключений известны заранее, и их обработка проще, поскольку вы точно знаете причину исключения, можете лучше организовать восстановление или по крайней мере отобразить для пользователя конкретное сообщение, как показано в листинге 28.2.

ЛИСТИНГ 28.2. Обработка исключения `std::bad_alloc`

```
0: #include <iostream>
1: #include<exception> // Для обработки исключения bad_alloc
2: using namespace std;
3:
```

```
4: int main()
5: {
6:     cout << "Количество чисел, для которых нужна память: ";
7:     try
8:     {
9:         int input = 0;
10:        cin >> input;
11:
12:        // Запрос памяти и ее последующее освобождение
13:        int* numArray = new int[input];
14:        delete[] numArray;
15:    }
16:    catch(std::bad_alloc& exp)
17:    {
18:        cout << "Перехвачено исключение: " << exp.what() << endl;
19:        cout << "Программа завершается." << endl;
20:    }
21:    catch(...)
22:    {
23:        cout << "Извините, перехвачено исключение." << endl;
24:    }
25:    return 0;
26: }
```

Результат

```
Количество чисел, для которых нужна память: -1
Перехвачено исключение: bad array new length
Программа завершается.
```

Анализ

Сравните вывод листинга 28.2 с выводом листинга 28.1. Как видите, теперь вы в состоянии указать причину внезапного окончания работы приложения точнее, а именно — “bad array new length” (ошибка длины запрашиваемого массива). Это связано с тем, что теперь в программе есть дополнительный блок catch (да, в ней теперь два блока catch), который обрабатывает исключения конкретного типа — catch(bad_alloc&), как показано в строках 16–20.

СОВЕТ

В общем случае вы можете вставить в программу столько блоков catch(), сколько вам нужно, располагая их один за другим в зависимости от типа ожидаемых исключений.

Блок catch(...), показанный в листинге 28.2, обрабатывает исключения всех типов, которые не были обработаны явно другими блоками catch (и поэтому должен располагаться последним).

Генерация исключения с помощью оператора throw

Когда вы обрабатывали исключение `std::bad_alloc` в листинге 28.2, речь шла об объекте класса `std::bad_alloc`, который сгенерировал в качестве исключения оператор `new`. Но вы вполне можете самостоятельно сгенерировать исключение требуемого вам типа. Для этого необходимо воспользоваться ключевым словом `throw`:

```
void DoSomething()
{
    if (что_то_не_так)
        throw Значение;
}
```

Давайте рассмотрим применение оператора `throw` на примере пользовательского типа исключения, генерируемого при попытке деления на ноль, как показано в листинге 28.3.

ЛИСТИНГ 28.3. Генерация специального исключения при попытке деления на ноль

```
0: #include<iostream>
1: using namespace std;
2:
3: double Divide(double dividend, double divisor)
4: {
5:     if(divisor == 0)
6:         throw "Делить на 0 нельзя";
7:
8:     return (dividend / divisor);
9: }
10:
11: int main()
12: {
13:     cout << "Введите делимое: ";
14:     double dividend = 0;
15:     cin >> dividend;
16:     cout << "Введите делитель: ";
17:     double divisor = 0;
18:     cin >> divisor;
19:
20:     try
21:     {
22:         cout << "Результат деления: "
23:             << Divide(dividend, divisor);
24:     }
25:     catch(char* exp)
26:     {
27:         cout << "Исключение: " << exp << endl;
28:         cout << "Программа завершена." << endl;
29:     }
30:     return 0;
31: }
```

Результат

```
Введите делимое: 2011
Введите делитель: 0
Исключение: Делить на 0 нельзя
Программа завершена.
```

Анализ

Этот код не только демонстрирует возможность обработки исключения типа `char*`, как показано в строке 24, но и то, что вы можете перехватывать исключение, сгенерированное в вызываемой функции `Divide()` в строке 6. Обратите также внимание на то, что в блок `try{}` заключена не вся функция `main()`, а только та ее часть, где ожидается генерация исключения. Это хорошая практика, поскольку обработка исключений может отрицательно влиять на производительность вашего кода.

Как работает обработка исключений

В функции `Divide()` листинга 28.3 генерируется исключение типа `char*`, которое затем обрабатывается обработчиком `catch(char*)` в вызывающей функции `main()`.

Когда исключение генерируется с помощью оператора `throw`, компилятор вставляет в код динамический поиск соответствующего блока `catch`, способного обработать это исключение. Логика обработки исключений сначала проверяет, находится ли строка, сгенерировавшая исключение, в пределах блока `try`. Если это так, то начинается поиск блока `catch`, который способен обработать исключение этого типа. Если же оператор `throw` находится вне блока `try` или если нет блока `catch`, соответствующего типу сгенерированного исключения, логика обработки исключения выполняет поиск в вызывающей функции. Так логика обработки исключений движется вверх по стеку вызовов, рассматривая одну вызывающую функцию за другой, пока не отыщет подходящий блок `catch`, способный обработать исключение данного типа. На каждом этапе разворачивания стека локальные переменные текущей функции уничтожаются в порядке, обратном порядку их создания (что продемонстрировано в листинге 28.4).

ЛИСТИНГ 28.4. Порядок уничтожения локальных объектов в случае исключения

```
0: #include <iostream>
1: using namespace std;
2:
3: struct StructA
4: {
5:     StructA() {cout << "StructA::StructA()" << endl; }
6:     ~StructA() {cout << "StructA::~~StructA()" << endl; }
7: };
8:
9: struct StructB
10: {
11:     StructB() {cout << "StructB::StructB()" << endl; }
```

```
12:     ~StructB() {cout << "StructB::~~StructB()" << endl; }
13: };
14:
15: void FuncB() // Генерация исключения
16: {
17:     cout << "B FuncB():" << endl;
18:     StructA objA;
19:     StructB objB;
20:     cout << "Генерируем исключение!" << endl;
21:     throw "Ловите меня!";
22: }
23:
24: void FuncA()
25: {
26:     try
27:     {
28:         cout << "B FuncA()" << endl;
29:         StructA objA;
30:         StructB objB;
31:         FuncB();
32:         cout << "FuncA: возврат в вызывающую функцию." << endl;
33:     }
34:     catch(const char* exp)
35:     {
36:         cout << "FuncA(): Перехвачено исключение: " << exp<<endl;
37:         cout << "FuncA(): Обработано, далее не передается"<<endl;
38:         // throw; // СНИМИТЕ комментарий для передачи в main()
39:     }
40: }
41:
42: int main()
43: {
44:     cout << "main(): Начало выполнения" << endl;
45:     try
46:     {
47:         FuncA();
48:     }
49:     catch(const char* exp)
50:     {
51:         cout << "Исключение: " << exp << endl;
52:     }
53:     cout << "main(): завершение работы." << endl;
54:     return 0;
55: }
```

Результат

```
main(): Начало выполнения
B FuncA()
StructA::StructA()
```

```
StructB::StructB()
B FuncB():
StructA::StructA()
StructB::StructB()
Генерируем исключение!
StructB::~StructB()
StructA::~StructA()
StructB::~StructB()
StructA::~StructA()
FuncA(): Перехвачено исключение: Ловите меня!
FuncA(): Обработано, далее не передается
main(): завершение работы.
```

Анализ

В листинге 28.4 функция `main()` вызывает функцию `FuncA()`, которая вызывает функцию `FuncB()`, генерирующую исключение в строке 21. Обе вызывающие функции, `FuncA()` и `main()`, являются безопасными в отношении исключений, поскольку у обеих реализован блок `catch(const char*)`. У функции `FuncB()`, генерирующей исключение, нет блоков `catch()`, а следовательно, первым обработчиком сгенерированного исключения будет блок `catch` в функции `FuncA()` (строки 34–39), поскольку функция `FuncA()` является вызывающей для функции `FuncB()`. Обратите внимание: функция `FuncA()` решает, что это исключение не имеет серьезного характера, полностью обработано и его не нужно передавать дальше функции `main()`. Так что функция `main()` продолжит свою работу, как будто никакой проблемы нет. Если же снять комментарий в строке 38, исключение будет передано вызывающей `FuncA` функции, так что функция `main()` также его получит.

Вывод программы демонстрирует также порядок создания объектов (это тот же порядок, в котором они расположены в коде) и их уничтожения при генерации исключения (обратный порядку создания объектов). Уничтожение объектов происходит не только в функции `FuncB()`, где было сгенерировано исключение, но и в функции `FuncA()`, которая вызвала функцию `FuncB()` и обработала сгенерированное в ней исключение.

ВНИМАНИЕ!

В листинге 28.4 показан порядок вызова деструкторов локальных объектов при генерации исключения.

Если деструктор объекта, вызванный в связи с исключением, также сгенерировал исключение, то это приведет к аварийному завершению вашего приложения.

Класс `std::exception`

При обработке исключения `std::bad_alloc` в листинге 28.2 вы фактически получали ссылку на объект класса исключения `std::bad_alloc`, сгенерированного оператором `new`. Класс `std::bad_alloc` — производный от стандартного класса `C++ std::exception`, объявленного в заголовочном файле `<exception>`.

Класс `std::exception` является базовым для следующих классов важных исключений.

- `bad_alloc`. Генерируется при неудачном выделении памяти оператором `new`.
- `bad_cast`. Генерируется оператором `dynamic_cast` при попытке приведения неправильного типа (без отношения наследования).
- `ios_base::failure`. Генерируется функциями и методами библиотеки `iostream`.

Класс `std::exception` является базовым классом, предоставляющим очень полезный и важный виртуальный метод, `what()`, возвращающий описательную информацию причины проблемы, вызвавшей исключение. Функция `exp.what()` в строке 18 листинга 28.2 предоставляет строку `"bad array new length"`, сообщая о том, что выделение памяти потерпело неудачу. Вы можете использовать класс `std::exception`, являющийся базовым классом для многих типов исключений, и создать один блок `catch(const exception&)`, способный обрабатывать все исключения, для которых класс `std::exception` является базовым:

```
void SomeFunc()
{
    try
    {
        // Код, безопасный в отношении исключений
    }
    catch(const std::exception& exp) // Обработка bad_alloc,
                                    // bad_cast и т.д.
    {
        cout << "Перехвачено исключение: " << exp.what() << endl;
    }
}
```

Пользовательский класс исключения, производный от `std::exception`

Вы можете сгенерировать исключение любого типа, какого пожелаете. Однако наследование от класса `std::exception` обладает тем преимуществом, что все существующие обработчики для исключений `catch(const std::exception&)`, которые перехватывают такие исключения, как `bad_alloc`, `bad_cast` и другие будут работать и автоматически обрабатывать и ваши пользовательские исключения, поскольку они имеют один и тот же базовый класс (листинг 28.5).

ЛИСТИНГ 28.5. Класс `CustomException`, происходящий от класса `std::exception`

```
0: #include <exception>
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: class CustomException: public std::exception
```

```
6: {
7:     string reason;
8: public:
9:     // Конструктор с указанием причины
10:    CustomException(const char* why):reason(why) {}
11:
12:    // Перекрытие виртуальной функции what()
13:    virtual const char* what() const throw()
14:    {
15:        return reason.c_str();
16:    }
17: };
18:
19: double Divide(double dividend, double divisor)
20: {
21:     if(divisor == 0)
22:         throw CustomException("CustomException: деление на 0");
23:
24:     return (dividend / divisor);
25: }
26:
27: int main()
28: {
29:     cout << "Введите делимое: ";
30:     double dividend = 0;
31:     cin >> dividend;
32:     cout << "Введите делитель: ";
33:     double divisor = 0;
34:     cin >> divisor;
35:     try
36:     {
37:         cout << "Результат деления: " << Divide(dividend, divisor);
38:     }
39:     catch(exception& exp) // Обработка в том числе CustomException
40:     {
41:         cout << exp.what() << endl;
42:         cout << "Программа завершена" << endl;
43:     }
44:
45:     return 0;
46: }
```

Результат

Введите делимое: 2011

Введите делитель: 0

CustomException: деление на 0

Программа завершена

Анализ

Это версия листинга 28.3, в котором при делении на нуль генерировалось простое исключение типа `char*`. Здесь мы создаем экземпляр класса `CustomException`, определенного в строках 5–17 как производного от класса `std::exception`. Обратите внимание на то, что наш класс исключения реализует виртуальную функцию `what()` в строках 13–16, которая возвращает описание причины генерации исключения. Логика обработчика `catch(exception&)` в функции `main()` (строки 39–43) обрабатывает исключения не только класса `CustomException`, но и других типов исключений (например, `bad_alloc`), для которых базовым является класс `exception`.

ПРИМЕЧАНИЕ

Обратите внимание на объявление виртуального метода `CustomException::what()` в строке 13 листинга 28.5:

```
virtual const char* what() const throw()
```

Оно заканчивается спецификатором `throw()`, который означает, что данная функция сама по себе не генерирует исключения — это очень важное и уместное ограничение для класса, объект которого используется как исключение. Если вы все же используете в этом методе оператор `throw`, то можете ожидать предупреждения от компилятора.

Если объявление функции заканчивается, например, спецификатором `throw(int)`, то это значит, что данная функция может генерировать исключение типа `int`.

РЕКОМЕНДУЕТСЯ

Помните о перехвате исключений типа `std::exception`.

Помните о возможности наследования пользовательского класса исключения (как и любого другого) от класса `std::exception`.

Генерируйте исключения осмотрительно. Они не являются заменой возврата из функций таких значений-флагов, как `true` или `false`.

НЕ РЕКОМЕНДУЕТСЯ

Не генерируйте исключения в деструкторах.

Не считайте успешность выделения памяти само собой разумеющейся; следует всегда заключать в блок `try` код, использующий оператор `new`, и создавать соответствующий обработчик `catch(std::exception&)`.

Не используйте сложную логику или выделение ресурсов в блоке `catch()`. Нельзя также генерировать исключения во время обработки других исключений.

Резюме

На этом занятии вы познакомились с очень важной частью практического программирования C++. Создание приложений, стабильно работающих вне собственной среды разработки, и интуитивно понятные пользователям сообщения важны для удовлетворения потребностей клиента. Всего этого позволяют добиться исключения. Вы

узнали, что код, выделяющий память или иные ресурсы, может столкнуться с проблемами, а следовательно, в нем должна присутствовать обработка исключений. Вы узнали, что язык C++ предоставляет стандартный класс исключения `std::exception`, который имеет смысл наследовать при необходимости создать собственный пользовательский класс исключения.

Вопросы и ответы

■ Почему нужно генерировать исключение, а не возвращать код ошибки?

Не всегда можно вернуть код ошибки. При неудаче оператора `new` необходимо обработать сгенерированное им исключение и воспрепятствовать отказу вашего приложения. Кроме того, если ошибка серьезная и делает невозможным последующее функционирование вашего приложения, лучше воспользоваться генерацией исключения.

■ Почему мой класс исключения должен наследовать класс `std::exception`?

Это, конечно же, необязательно, но позволит вам воспользоваться блоками `catch()`, которые обрабатывают исключения типа `std::exception`. Вы можете написать собственный класс исключения, который не является потомком других классов, но тогда вам придется добавлять собственные обработчики `catch(MyNewExceptionType&)` во все соответствующие места программы.

■ У меня есть функция, которая генерирует исключение. Должно ли оно обрабатываться в той же функции?

Нет. Следует только убедиться, что исключение данного типа обработает одна из вызывающих функций выше в стеке вызовов.

■ Может ли генерация исключения осуществляться из конструктора?

У конструкторов фактически нет иного выбора! У них нет возвращаемых значений, и генерация исключения — наилучший способ оповещения о проблеме.

■ Может ли генерация исключения осуществляться в деструкторе?

Технически — да. Но это *очень* плохая идея, поскольку деструкторы вызываются при разворачивании стека при обработке исключения. Если деструктор, вызванный в связи с обработкой исключения, сам сгенерирует исключение, может возникнуть весьма неприятная ситуация.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Что такое `std::exception`?
2. Какого типа исключение генерируется при неудаче выделения памяти оператором `new`?
3. Хороша ли идея выделить в обработчике исключения (блок `catch`) место под, скажем, миллион целых чисел для резервного копирования существующих данных?
4. Как бы вы обработали объект исключения класса `MyException`, производного от класса `std::exception`?

Упражнения

1. **Отладка.** Что не так в следующем коде?

```
class SomeIntelligentStuff
{
    bool StuffGoneBad;
public:
    ~SomeIntelligentStuff()
    {
        if(StuffGoneBad)
            throw "Проблема в данном классе";
    }
};
```

2. **Отладка.** Что не так в следующем коде?

```
int main()
{
    int* pMillionIntegers = new int[1000000];
    // Сделать нечто с миллионом целых чисел

    delete[]pMillionIntegers;
}
```

3. **Отладка.** Что не так в следующем коде?

```
int main()
{
    try
    {
        int* pMillionIntegers = new int[1000000];
        // Сделать нечто с миллионом целых чисел

        delete[]pMillionIntegers;
    }
    catch(exception& exp)
    {
        int* pAnotherMillionIntegers = new int[1000000];
        // Создать резервную копию pMillionIntegers
        // и сохранить ее на диске
    }
}
```

ЗАНЯТИЕ 29

Что дальше

Вы изучили основы программирования на языке C++. Фактически мы вышли за теоретические границы понимания, изучив стандартную библиотеку шаблонов (STL), шаблоны и то, как STL способна помочь вам писать эффективный и компактный код. Теперь пришло время обратить внимание на производительность и получить несколько полезных советов по программированию.

На этом занятии...

- Как приложение C++ может лучше использовать возможности процессора
- Потоки и многопоточность
- Полезные практические советы по программированию на C++
- Новые возможности, ожидаемые в C++17
- Как повысить свою квалификацию программиста на C++

Чем различаются современные процессоры

За последнее время процессоры компьютеров стали быстрее, их скорости обработки измеряются уже не в килогерцах (кГц) и мегагерцах (МГц), а в гигагерцах (ГГц). Например, процессор Intel 8086 (рис. 29.1), выпущенный в 1978 году, был 16-разрядным и работал с тактовой частотой примерно 10 МГц.

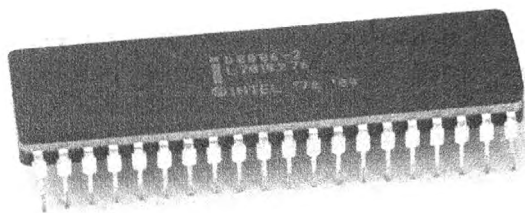


РИС. 29.1. Микропроцессор Intel 8086

Процессоры в наши дни стали значительно быстрее, и то же самое можно сказать о приложениях C++. Проще всего было бы положиться на постоянно улучшающиеся аппаратные средства и рост производительности программного обеспечения за счет повышения их быстродействия. Но хотя современные процессоры все еще становятся быстрее, истинное новаторство кроется в количестве ядер, которыми они обладают. На момент написания этой книги даже массовые смартфоны уже оснащены 64-разрядными микропроцессорами с четырьмя ядрами и большими вычислительными возможностями, чем настольные компьютеры десятилетие назад.

Многоядерный процессор можно рассматривать как одну микросхему с несколькими процессорами, работающими параллельно. Каждый процессор имеет собственный кеш L1 и способен работать независимо от других.

Чем быстрее процессор, тем выше скорость выполнения вашего приложения, что вполне логично. Но чем поможет наличие нескольких ядер процессора? Вполне очевидно, что каждое ядро способно выполнять приложения параллельно, но это необязательно делает ваше приложение быстрее, если вы не программируете его так, чтобы оно могло воспользоваться новыми возможностями. Однопоточные приложения C++, которые мы рассматривали до сих пор, вероятно, не извлекут никакой выгоды из работы в многоядерной системе. Такие приложения выполняются в одном потоке, а следовательно, используют только одно ядро, как показано на рис. 29.2.

Если ваше приложение выполняет все действия последовательно, то операционная система, скорее всего, предоставит ему столько же времени, сколько и другим приложениям, и оно будет использовать только одно ядро процессора. Другими словами, ваше приложение будет выполняться на многоядерном процессоре точно так же, как и многие годы назад (разве что благодаря наличию большого количества ядер оно не будет постоянно прерываться, уступая свой процессор другим приложениям и операционной системе).

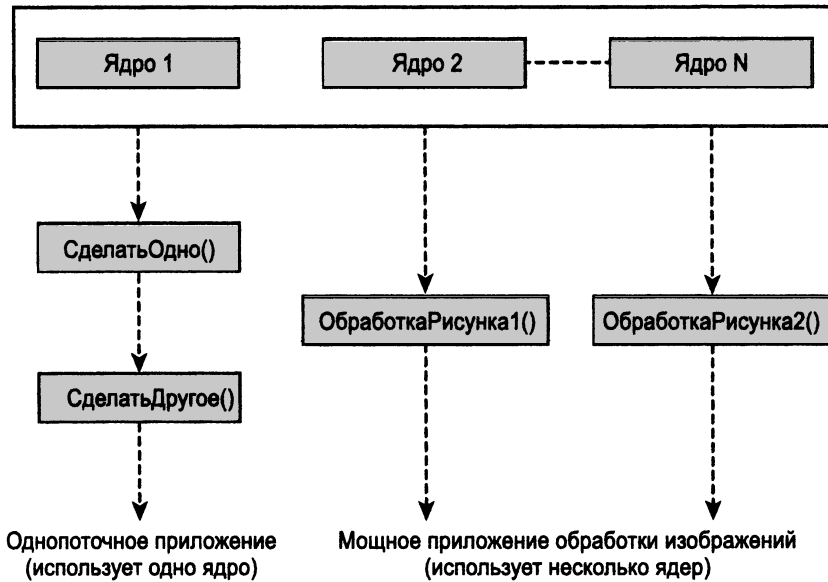


РИС. 29.2. Однопоточное приложение на многоядерном процессоре

Как лучше использовать несколько ядер

Ключевым решением в повышении эффективности приложений является создание многопоточных приложений, каждый поток которых выполняется параллельно, позволяя операционной системе распределять эти потоки среди нескольких ядер. Хотя обсуждение потоков и многопоточности выходит за рамки этой книги, я могу бегло затронуть эту тему и дать вам начальное представление о высокопроизводительных приложениях.

Что такое поток

Код приложения всегда работает в потоке выполнения. *Поток* (thread) — это синхронно выполняемый объект, операторы которого выполняются один за другим. Код функции `main()` рассматривается как выполняемый основным потоком приложения. В этом основном потоке можно создавать новые потоки, способные выполняться параллельно. Такие приложения, состоящие из одного или нескольких потоков, выполняющихся параллельно основному, называются *многопоточными приложениями* (multithreaded application).

Как именно должны создаваться потоки, определяется операционной системой. Вы можете создавать их непосредственно, с помощью вызова соответствующих функций API, предоставляемых операционной системой.

СОВЕТ

Язык C++ начиная со стандарта C++11 определяет функции для создания многопоточных приложений, которые сами заботятся о вызове функций API операционной системы. Это делает ваше многопоточное приложение намного более переносимым.

Если вы планируете разработку приложения только для одной операционной системы, можете ограничиться использованием API этой операционной системы.

ПРИМЕЧАНИЕ

Фактические действия по созданию потока специфичны для конкретной операционной системы. Язык C++11 попытается снабдить вас независимой от конкретной платформы абстракцией в виде класса `std::thread`, описанного в заголовочном файле `<thread>`. Однако если вы пишете программу для одной конкретной платформы, то можете использовать потоковые функции, специфичные для данной операционной системы.

При написании переносимого многопоточного приложения C++ можно подумать о применении Boost Thread Libraries (см. www.boost.org).

Зачем создавать многопоточные приложения

Многопоточность используется в приложениях, которые должны осуществлять множество действий параллельно. Предположим, что вы — один из 10 000 пользователей, пытающихся осуществить покупку на Amazon.com. Конечно, веб-сервер Amazon.com не может заставить 9 999 пользователей ожидать, пока один пользователь завершит свою покупку. Веб-сервер создает множество потоков, обслуживая всех пользователей одновременно. Если веб-сервер работает на многоядерном процессоре (готов спорить, что это так и есть), то такие потоки в состоянии извлечь преимущество из наличия доступных ядер и обеспечить оптимальную производительность для каждого пользователя.

Еще одним общеизвестным примером многопоточности является, например, приложение с графическим интерфейсом пользователя, которое взаимодействует с пользователем и в то же время выполняет некую полезную работу. Такие приложения обычно имеют поток пользовательского интерфейса (User Interface Thread), который отображает пользовательский интерфейс, изменяет его вид по мере надобности и принимает пользовательский ввод, а также рабочий поток (Worker Thread), который в фоновом режиме выполняет основную работу. К таким приложениям относятся, например, инструменты дефрагментации диска. После запуска такого приложения создается рабочий поток, начинающий просмотр и дефрагментацию диска. Одновременно поток пользовательского интерфейса отображает прогресс процесса дефрагментации, предоставляя при этом пользователю возможность отменить дефрагментацию. Чтобы поток пользовательского интерфейса мог отображать на экране прогресс процесса дефрагментации, рабочий поток, занимающийся ею, должен регулярно передавать сообщения потоку пользовательского интерфейса. Точно так же, чтобы рабочий поток узнал о необходимости прекратить работу, поток пользовательского интерфейса должен сообщить ему об этом.

ПРИМЕЧАНИЕ

Чтобы приложение могло функционировать как единое целое, а не коллекция бесконтрольных потоков, выполняющих свою работу независимо один от другого, многопоточные приложения нуждаются в средстве “общения” потоков между собой.

Последовательность также важна. Вы ведь не хотите, чтобы поток пользовательского интерфейса закончил работу раньше, чем рабочий поток закончит дефрагментацию? Нередки ситуации, когда один поток должен ожидать другой. Например, поток чтения из базы данных должен ожидать, пока завершит текущую операцию поток записи в базу данных.

Действия по организации ожидания потоками один другого называются *синхронизацией потоков* (thread synchronization).

Как потоки осуществляют транзакцию данных

Потоки способны совместно использовать переменные. У потока может быть доступ к глобальным данным. Потоки могут быть созданы с указателем на совместно используемый объект (структуры или класса) с данными, как показано на рис. 29.3.

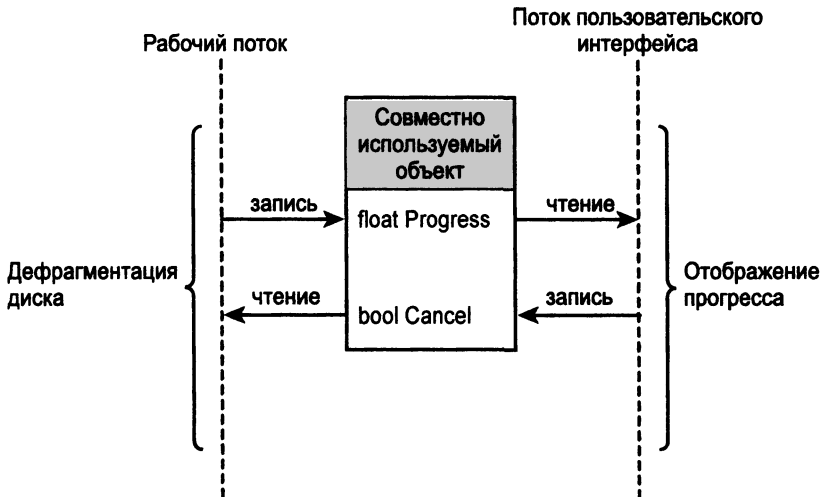


РИС. 29.3. Рабочий поток и поток пользовательского интерфейса совместно используют общие данные

Потоки могут общаться путем записи и чтения данных, хранящихся в некой области памяти, к которой они способны обращаться, а следовательно, совместно их использовать. В примере программы дефрагментации, в котором рабочему потоку известна доля выполненной работы, которую необходимо отображать потоку пользовательского интерфейса, рабочий поток может регулярно сохранять необходимое значение в переменной, используемой потоком пользовательского интерфейса для отображения.

Это довольно простой случай: один поток создает информацию, а другой использует ее. Но что будет, если выполнять запись и чтение одной и той же области памяти

будут несколько потоков? Одни потоки могут начать читать данные в тот момент, когда другие еще не закончили запись. Целостность данных оказывается под угрозой.

Вот почему потоки следует синхронизировать.

Использование мьютексов и семафоров для синхронизации потоков

Потоки — сущности уровня операционной системы, и объекты, используемые для их синхронизации, также предоставляются операционной системой. Большинство операционных систем предоставляет для синхронизации потоков *семафоры* (semaphore) и *мьютексы* (mutex).

Мьютекс (объект синхронизации путем взаимного исключения (mutual exclusion)) гарантирует доступ к части кода только одного потока. Другими словами, мьютекс используется для организации раздела кода, перед выполнением которого поток должен подождать, пока другой поток закончит его выполнение и освободит мьютекс. Когда следующий поток захватывает мьютекс, он может выполнять этот фрагмент кода; все прочие потоки будут вынуждены перейти в состояние ожидания, пока он освободит мьютекс. Начиная с C++11 стандартная библиотека предоставляет мьютексы в виде класса `std::mutex` в заголовочном файле `<mutex>`.

Используя семафоры, можно контролировать количество потоков, которые выполняют некоторый раздел кода. Семафор, разрешающий доступ только одному потоку, называется *бинарным семафором* (binary semaphore).

Проблемы, вызываемые многопоточностью

Многопоточность с ее необходимостью в хорошей синхронизации потоков способна стать причиной множества бессонных ночей, когда эффективность синхронизации оказывается неэффективной (читай: с ошибками). Вот две наиболее распространенные проблемы, с которыми сталкиваются многопоточные приложения.

- *Состояние гонки* (race condition). Два или более потоков пытаются записывать одни и те же данные. Кто победит? Каким окажется состояние этого объекта?
- *Взаимоблокировка* (deadlock). Два потока ожидают завершения один другого, и оба находятся в состоянии ожидания. В результате приложение “зависает”.

При хорошей синхронизации состояния гонки можно избежать. Обычно, когда потокам позволено писать в совместно используемый объект, необходимо дополнительно позаботиться о следующем:

- одновременно записывать данные может только один поток;
- никакому потоку не позволено читать, пока не завершится текущая запись объекта.

Избежать взаимоблокировки можно, устраняя ситуации, когда два потока вынуждены ожидать один другого. У вас, например, может быть главный поток, синхронизирующий рабочие потоки. Поток А может ожидать поток В, но поток В никогда не должен при этом ожидать поток А.

Разработка многопоточных приложений сама по себе является специализацией. Поэтому подробное рассмотрение этой интересной и захватывающей темы выходит за рамки данной книги. Для изучения практик многопоточного программирования следует обратиться к другой литературе или доступной в сети документации. Овладев этими технологиями, вы сможете создавать приложения C++, оптимально подготовленные для использования возможностей будущих многоядерных процессоров.

Как писать отличный код C++

Язык C++ значительно развился со времени первого выпуска благодаря усилиям по его развитию и стандартизации, предпринятые главными изготовителями компиляторов. В C++ к услугам программиста — множество утилит и функций, которые помогают писать компактный и понятный код на C++. Писать понятные и надежные приложения на C++ в действительности просто.

Ниже приведены полезные советы, которые помогут вам в создании хороших приложений C++.

- Присваивайте переменным осмысленные имена (понятные не только вам).
- Всегда инициализируйте такие переменные, как `int`, `float` и подобные им.
- Всегда инициализируйте указатели либо значением `nullptr`, либо допустимым адресом (например, возвращенным оператором `new`).
- При использовании массивов никогда не пересекайте их границы. Это вызывает переполнение буфера и может использоваться как брешь в системе безопасности.
- Не используйте строковые буфера `char*` в стиле C и такие функции, как `strlen()` и `strcpy()`. Тип `std::string` более безопасен и предоставляет много полезных вспомогательных методов, включая позволяющие находить длину, выполнять копирование и конкатенацию.
- Используйте статические массивы только тогда, когда абсолютно уверены в количестве элементов, которые они будут содержать. Если вы не уверены в этом значении, используйте динамический массив, такой как `std::vector`.
- При объявлении и определении функций, получающих параметры типов, отличных от POD (простых старых данных), старайтесь избегать ненужного этапа копирования, передавая их при вызове функции по ссылке.
- Если ваш класс содержит член (или члены) в виде простого указателя, обдумайте владение ресурсом и управление им в случае копирования и присваивания. Рассмотрите возможность создания копирующего конструктора и копирующего оператора присваивания.
- При написании вспомогательного класса, управляющего динамическим массивом или чем-то подобным, для повышения производительности не забудьте добавить перемещающий конструктор и перемещающий оператор присваивания.
- Не забывайте об использовании констант. В идеале функция `get()` не должна изменять члены класса, а следовательно, должна быть объявлена как константная. Точно так же параметры функций должны быть константными ссылками, если вы не планируете изменять значения, которые они содержат.

- Избегайте использования простых указателей. Используйте, где только можно, подходящие интеллектуальные указатели.
- При создании вспомогательного класса не жалейте усилий для поддержки всех операторов, которые сделают использование вашего класса более простым.
- По возможности отдавайте предпочтение шаблонам, а не макросам. Шаблоны безопасны с точки зрения типов.
- При разработке класса, объекты которого будут храниться в контейнере, таком как вектор или список, или использоваться как ключ в отображении, не забывайте предоставлять оператор `operator<`, определяющий заданный по умолчанию критерий сортировки.
- Если ваша лямбда-функция становится слишком большой, возможно, имеет смысл создать функциональный объект с оператором `operator()`, поскольку такой функтор обеспечивает повторное применение, а также единую точку поддержки.
- Никогда не считайте безоговорочно, что оператор `new` завершается успешно. Код выделения ресурса всегда может сгенерировать исключение, поэтому заключайте его в блок `try` с соответствующим блоком `catch()`.
- Никогда не используйте оператор `throw` в деструкторе класса.

Это отнюдь не исчерпывающий список, но он охватывает ряд наиболее важных вопросов и, несомненно, поможет вам в написании качественных и легко сопровождаемых программ на C++.

C++17: что новенького

Одним из больших преимуществ C++ является то, что комитет по стандартизации активно и постоянно совершенствует язык. Ожидается, что, как и его предшественник C++11, стандарт C++17 внесет в язык множество новых возможностей. Давайте рассмотрим некоторые особенности, которые, скорее всего, войдут в стандарт C++17 после его официальной ратификации.

ПРИМЕЧАНИЕ

Рассматриваемые на следующих страницах возможности войдут в стандарт с очень большой степенью вероятности, но в настоящее время не являются его частью. Скорее всего, ваш любимый компилятор поддерживает некоторые возможности лишь частично и не поддерживает их все. Кроме того, хотя это и маловероятно, в окончательный вариант C++17 могут не войти все описанные здесь возможности, несмотря на то что это ожидается на момент написания данной книги.

Инициализация в `if` и `switch`

Это небольшое, но важное усовершенствование синтаксиса `if` и `switch`, которое можно выразить следующим образом:

```
if (Инициализатор; условие)
{
    // Инструкции, выполняемые при истинности условия
}
```

Или

```
switch(Инициализатор; условие)
{
    // Блоки case
}
```

Переменная, объявленная в инструкции *Инициализатора*, уничтожается при выходе из инструкции `if`. Рассмотрим следующий код из листинга 20.3:

```
auto pairFound = mapIntToStr.find(key);
if (pairFound != mapIntToStr.end())
{
    cout << "Ключ " << pairFound->first << " указывает на: ";
    cout << pairFound->second << endl;
}
```

Теперь этот фрагмент может быть записан как

```
if (auto pairFound = mapIntToStr.find(key);
    pairFound != mapIntToStr.end())
{
    cout << "Ключ " << pairFound->first << " указывает на: ";
    cout << pairFound->second << endl;
}
```

Это не просто уменьшение кода (которого в данном фрагменте, по сути, нет). Это изменение гарантирует, что переменная `pairFound`, которая необходима только в блоке `if`, недоступна вне его. Ее область видимости ограничена до необходимого минимума. Кроме того, при копировании и вставке этого улучшенного блока вы перенесете всю необходимую логику в полном объеме.

Гарантия устранения копирования

При инициализации переменной возвращаемым значением функции может случиться так, что компилятор создаст временную копию целого числа, возвращаемого функцией `ReturnInt()`, перед тем как инициализировать им переменную `num`:

```
int num = ReturnInt();
```

C++17 требует от компилятора полностью обойтись без создания такой временной копии, т.е. устранить копирование.

Устранение накладных расходов выделения памяти с помощью `std::string_view`

Рассмотрим функцию, принимающую в качестве параметра `std::string`:

```
void DisplayString(const std::string& strIn)
{
    cout << strIn << endl;
}
```

При ее вызове с передачей строкового литерала последний сначала преобразуется во временный объект `std::string`, который и используется функцией `DisplayString()`:

```
DisplayString("Hello World!");
```

Этого преобразования можно избежать, используя вместо строки `std::string_view`:

```
void DisplayString(std::string_view& strIn)
{
    cout << strIn << endl;
}
```

Передача функции строкового литерала не повлечет за собой никаких накладных расходов, связанных с распределением памяти, если функция принимает в качестве аргумента `std::string_view`.

`std::variant` как безопасная с точки зрения типов альтернатива объединению

Объединения рассматривались на занятии 9, “Классы и объекты”. Одна из проблем, связанных с объединениями, заключается в том, что оно позволяет рассматривать содержимое одного типа как имеющее другой тип данных, поддерживаемый объединением, например

```
union SimpleUnion
{
    int num;
    double preciseNum;
};
```

Вы можете инстанцировать это объединение значением `double`, а затем использовать его как `int`:

```
SimpleUnion ul;
ul.preciseNum = 3.14; // Сохранение значения типа double
int num2 = ul.num;    // Работает; но ul содержит double!
```

C++17 предоставляет программисту тип `std::variant` — безопасную с точки зрения типов альтернативу для `union`:

```

variant<int, double> varSafe;
varSafe = 3.14;           // Сохраняем double
double pi = get<double>(varSafe); // 3.14
double pi2 = get<1>(varSafe);    // 3.14
get<char>(varSafe); // Ошибка компиляции: типа char нет
get<2>(varSafe);    // Ошибка компиляции: есть только два типа, а не три
try
{
    get<int>(varSafe);    // Генерация исключения при сохранении double
}
catch(bad_variant_access&) { /* Обработчик исключения */ }

```

Условная компиляция с использованием `if constexpr`

Эта конструкция аналогична конструкции `if-else` с тем отличием, что условие вычисляется во время компиляции и код в блоке `if` (или `else`) компилируется только тогда, когда условие выполняется во время компиляции.

```

#include <type_traits>
#include <iostream>
#include <iomanip>
using namespace std;
template <typename T>
void DisplayData(const T& data)
{
    if constexpr(is_integral<T>::value)
        cout << "Целочисленные данные: " << data << endl;
    else if constexpr(is_floating_point<T>::value)
        cout << setprecision(15) << "Данные с плавающей точкой: "
            << data << endl;
    else
        cout << "Неопределенные данные: " << data << endl;
}

```

В случае вызова `DisplayData(15)` компилятор, совместимый с C++17, будет компилировать только следующую строку:

```
cout << "Целочисленные данные: " << data << endl;
```

В случае вызова `DisplayData("Hello World!")`, поскольку тип `const char*` ведет к выполнению блока `else`, будет скомпилирована следующая строка:

```
cout << "Неопределенные данные: " << data << endl;
```

В сочетании с автоматическим выводом возвращаемого типа, описанного на занятии 7, “Организация кода с помощью функций”, эта мощная возможность потенциально позволяет функции возвращать значения разных типов в зависимости от пути выполнения.

Усовершенствованные лямбда-выражения

Ожидаются следующие усовершенствования лямбда-функций.

- Они должны поддерживаться внутри `constexpr`-функций.
- Они должны иметь возможность захвата `*this` с помощью синтаксиса `[*this]`.

Автоматический вывод типа для конструкторов

В C++14 для сочетания целочисленного значения и значения с плавающей точкой необходимо явно объявить тип пары:

```
std::pair<int, double> pairIntToDb(3, 3.14159265359);
```

C++17 позволяет упростить эту строку до

```
std::pair pairIntToDb(3, 3.14159265359);
```

Вывод типа аргументов шаблона конструктора выполняется автоматически.

`template<auto>`

Это расширение менее используемой возможности, заключающейся в том, что аргумент шаблона может содержать значение, которое используется во время компиляции. Например, `std::array` — это контейнер, который моделирует массивы с фиксированными размерами, известными во время компиляции. Для моделирования массива из 10 целых чисел требуется следующий код:

```
std::array<int, 10> myTenNums;
```

Объявление шаблона класса наподобие `std::array` имеет следующий вид:

```
template <class T, std::size_t N> struct array;
```

C++17 позволяет упростить тип параметра шаблона до `auto`, так что следующее объявление будет вполне корректным:

```
template <class T, auto N> struct array;
```

Изучение C++ на этом не заканчивается

Поздравляю, вы достигли больших успехов в изучении языка C++! Наилучший способ постоянно повышать квалификацию — это программировать и еще раз программировать! Язык C++ довольно сложен. И чем больше вы программируете, тем выше ваш уровень понимания того, как он внутренне работает.

Документация в вебе

Если необходимо узнать больше о сигнатурах контейнеров STL, их методах или алгоритмах, а также иные подробности, обратитесь к вебу. Одним из наиболее популярных ресурсов является сайт <http://www.cppreference.com/>.

Сетевые сообщества и помощь

У языка C++ богатые и яркие сетевые сообщества. Зарегистрируйтесь на таких сайтах, как StackOverflow (www.StackOverflow.com), CodeGuru (www.CodeGuru.com) и CodeProject (www.CodeProject.com), чтобы задавать технические вопросы и получать ответы от сообщества. (Если у вас проблемы с английским языком, вам будут рады на сайте StackOverflow на русском языке, <https://ru.stackoverflow.com/>. — *Примеч. ред.*)

Когда почувствуете себя уверенно, не стесняйтесь внести свой вклад в сообщество. Вы сможете найти ответы на сложные вопросы и многому научитесь в процессе общения.

Резюме

Это заключительное занятие — фактически начало самостоятельного дальнейшего изучения C++! Зайдя так далеко, вы изучили основные и дополнительные концепции языка. На этом занятии рассматривались теоретические основы многопоточного программирования. Вы узнали, что единственный способ извлечь пользу из наличия многоядерных процессоров заключается в организации вашей логики в потоки и обеспечении их параллельной обработки. Вы узнали о проблемах многопоточных приложений и способах их решения. Вы изучили также ряд основных полезных советов по программированию на языке C++. Вы знаете, что для написания хорошего кода на C++ нужно не только использовать передовые концепции, но и присваивать переменным имена, которые понятны другим, что, обрабатывая исключения, нужно позаботиться о возможности исключения непредвиденного типа, что вместо простых указателей можно использовать такие классы, как интеллектуальные указатели. Теперь вы готовы перейти к профессиональному программированию на C++.

Вопросы и ответы

- **Я вполне доволен производительностью моего приложения. Должен ли я все же реализовать многопоточность?**

Нет. Не все приложения должны быть многопоточными. Многопоточность нужна только при необходимости выполнения нескольких задач одновременно или для обслуживания большого количества пользователей.

- **Почему бы мне не использовать старый стиль программирования и не озадачиваться всеми этими C++11 и C++14?**

Эти стандарты существенно упрощают программирование. Такие ключевые слова, как `auto`, избавляют от долгих и утомительных объявлений итераторов, а лямбда-функции делают конструкции `for_each()` компактнее и без объектов функций. Поэтому преимущества использования современных стандартов в программировании на C++ позволяют создавать более короткие, более простые и более эффективные программы, чем старые версии стандарта.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить на практике полученные навыки. Попробуйте самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Д, “Ответы”. Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Мое приложение обработки изображений перестает отвечать в процессе исправления контраста. Что мне делать?
2. Мое многопоточное приложение обеспечивает чрезвычайно быстрый доступ к базе данных. Но иногда я замечаю, что выбранные данные неверны. Что я делаю неправильно?

Часть VI

Приложения

В ЭТОЙ ЧАСТИ...

ПРИЛОЖЕНИЕ А. Двоичные и шестнадцатеричные числа

ПРИЛОЖЕНИЕ Б. Ключевые слова языка C++

ПРИЛОЖЕНИЕ В. Приоритет операторов

ПРИЛОЖЕНИЕ Г. Коды ASCII

ПРИЛОЖЕНИЕ Д. Ответы

ПРИЛОЖЕНИЕ А

Двоичные и шестнадцате- ричные числа

Понимание работы двоичной и шестнадцатеричной систем счисления не является критически важным для разработки приложений на C++, но помогает лучше понимать происходящее.

Десятичная система счисления

Цифры, которые мы используем ежедневно, находятся в диапазоне 0–9. Этот набор цифр называется десятичной системой счисления. Поскольку система состоит из 10 отдельных цифр, она называется также системой с основанием 10.

Следовательно, поскольку основанием является 10, отсчитываемая от нуля позиция каждой цифры означает степень числа 10, умноженную на цифру.

$$957 = 9 \times 10^2 + 5 \times 10^1 + 7 \times 10^0 = 9 \times 100 + 5 \times 10 + 7$$

В числе 957 отсчитываемая от нуля позиция цифры 7 — 0, цифры 5 — 1, а цифры 9 — 2. Эта позиция индексирует степени основания 10, как показано в примере. Помните, что любое число в степени 0 дает 1 (таким образом, 10^0 — то же самое, что и 1000^0 , поскольку оба значения равны 1).

ПРИМЕЧАНИЕ

В десятичной системе счисления самыми важными являются степени числа 10. Цифры в числе умножаются на 10, 100, 1000 и так далее, чтобы определить размерность числа.

Двоичная система счисления

Система с основанием 2 называется двоичной системой счисления. Поскольку система допускает только два состояния, она представляется числами 0 и 1. В C++ эти числа обычно рассматриваются как false и true (true — не нуль).

Подобно тому как числа в десятичной системе счисления являются степенями основания 10, в двоичной системе они являются степенями основания 2:

$$101 \text{ (двоичное)} = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5 \text{ (десятичное)}.$$

Так, десятичным эквивалентом двоичного числа 101 будет 5.

ПРИМЕЧАНИЕ

Цифры в двоичном числе являются степенями числа 2, такими как 4, 8, 16, 32 и так далее, в зависимости от их разряда. Степень является отсчитываемым от нуля местом, которое занимает рассматриваемая цифра.

Чтобы понять систему двоичных цифр, рассмотрим табл. А.1, в которой приведены степени числа два.

ТАБЛИЦА А.1. Степени числа 2

Степень	Значение	Двоичное представление
0	$2^0 = 1$	1
1	$2^1 = 2$	10

Окончание табл. А.1

Степень	Значение	Двоичное представление
2	$2^2 = 4$	100
3	$2^3 = 8$	1000
4	$2^4 = 16$	10000
5	$2^5 = 32$	100000
6	$2^6 = 64$	1000000
7	$2^7 = 128$	10000000

Почему компьютеры используют двоичные числа

Широкое распространение двоичная система счисления получила относительно недавно (по сравнению со временем использования систем счисления вообще), после появления электроники и компьютеров. Развитие электроники и электронных компонентов привело к появлению систем, которые различали состояния компонентов как включенное (при наличии разницы потенциалов или напряжений) или как выключенное (при отсутствии разницы потенциалов или напряжений).

Эти состояния ВКЛ и ВЫКЛ очень удобно интерпретировать как 1 и 0, а также полностью представлять ими набор двоичных чисел и выполнять арифметические вычисления. Такие логические операции, как НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ, рассматривавшиеся на занятии 5, “Выражения, инструкции и операторы”, (см. табл. 5.2–5.5), легко реализуются электронными средствами, в результате чего двоичная система счисления стала простой и популярной в электронике.

Что такое биты и байты

Бит — основная единица в вычислительной системе, которая содержит двоичное состояние. Таким образом, о бите говорят, что он “установлен”, если он содержит состояние 1, или “сброшен”, если содержит состояние 0. Коллекция битов — это *байт*. Количество битов в байте теоретически не определено и зависит от используемых аппаратных средств.

Однако большинство вычислительных систем предполагает, что в байте находится 8 битов, по той простой причине, что 8 является степенью 2. Кроме того, восемь битов в байте позволяют передать до 2^8 (256) различных значений. Этих 256 отдельных значений более чем достаточно для представления всех символов в наборе символов ASCII.

Сколько байтов в килобайте

1 килобайт — это 1024 байта (2^{10} байтов). Точно так же 1024 килобайта составляют 1 мегабайт, а 1024 мегабайта — 1 гигабайт. 1024 гигабайта составляют 1 терабайт.

Шестнадцатеричная система счисления

Шестнадцатеричная система счисления имеет основание 16. Цифра в шестнадцатеричной системе может находиться в диапазоне 0–9 и A–F. Так, десятичное 10 — это шестнадцатеричное A, а десятичное 15 — шестнадцатеричное F.

Десятичное число	Шестнадцатеричное	Десятичное число (продолжение)	Шестнадцатеричное (продолжение)
0	0	8	8
1	1	9	9
2	2	10	A
3	3	11	B
4	4	12	C
5	5	13	D
6	6	14	E
7	7	15	F

Подобно тому как числа в десятичной системе счисления являются степеням основания 10, в двоичной системе — степенями основания 2, в шестнадцатеричной они являются степенями основания 16:

$$0x31F = 3 \times 16^2 + 1 \times 16^1 + F \times 16^0 = 3 \times 256 + 16 + 15 \text{ (десятичное)} = 799.$$

ПРИМЕЧАНИЕ

По соглашению в C++ шестнадцатеричные числа представляют с префиксом "0x".

Зачем нужна шестнадцатеричная система

Компьютеры работают с двоичными числами. Состояние каждого блока памяти в компьютере — 0 или 1. Однако, поскольку мы, люди, должны взаимодействовать с компьютером и специфической для программ информацией в виде нулей и единиц, мы нуждаемся в более компактном представлении небольших частей информации. Так, вместо того чтобы писать 1111 в двоичном виде, нам намного проще написать F в шестнадцатеричном.

Так, шестнадцатеричное представление может очень эффективно отобразить состояние 4 битов, а используя максимум две шестнадцатеричные цифры, можно представить состояние байта.

ПРИМЕЧАНИЕ

Менее популярна восьмеричная система счисления. Это система с основанием 8, включающая цифры от 0 до 7.

Преобразование в различные системы счисления

При работе с числами может возникнуть необходимость в просмотре одного и того же числа в представлении с разными основаниями, например двоичного числа в десятичном виде или десятичного числа в шестнадцатеричном.

В предыдущих примерах было продемонстрировано, как числа могут быть преобразованы из двоичного или шестнадцатеричного в десятичное число. Рассмотрим преобразование двоичных и шестнадцатеричных чисел в десятичное число.

Обобщенный процесс преобразования

При преобразовании числа из одной системы в другую вы последовательно делите преобразуемое число на основание целевой системы счисления. Полученный остаток заполняет разряды в целевой системе счисления, начиная с самого младшего разряда. Следующее деление использует частное предыдущей операции.

Так продолжается до тех пор, пока остаток в пределах целевой системы счисления и частное не достигнут 0.

Этот процесс также называется *методом разбиения* (breakdown method).

Преобразование десятичного числа в двоичное

Чтобы преобразовать десятичные 33 в двоичное, вычитайте из него самую высокую из возможных степеней числа 2 (32)

Знакоместо 1:	$35 / 2$	= частное 17, остаток 1
Знакоместо 2:	$17 / 2$	= частное 8, остаток 1
Знакоместо 3:	$8 / 2$	= частное 4, остаток 0
Знакоместо 4:	$4 / 2$	= частное 2, остаток 0
Знакоместо 5:	$2 / 2$	= частное 1, остаток 0
Знакоместо 6:	$1 / 2$	= частное 0, остаток 1

Двоичный эквивалент числа 33 (по знакоместам): 100011

Двоичный эквивалент числа 156

Знакоместо 1:	$156 / 2$	= частное 78, остаток 0
Знакоместо 2:	$78 / 2$	= частное 39, остаток 0
Знакоместо 3:	$39 / 2$	= частное 19, остаток 1
Знакоместо 4:	$19 / 2$	= частное 9, остаток 1
Знакоместо 5:	$9 / 2$	= частное 4, остаток 1
Знакоместо 6:	$4 / 2$	= частное 2, остаток 0
Знакоместо 7:	$2 / 2$	= частное 1, остаток 0
Знакоместо 9:	$1 / 0$	= частное 0, остаток 1

Двоичный эквивалент числа 156: 10011100

Преобразование десятичного числа в шестнадцатеричное

Процесс тот же, что и при преобразовании в двоичное число, но деление осуществляется на основание 16, а не 2.

Преобразование десятичного числа 5211 в шестнадцатеричное

Знакоместо 1: $5211 / 16$ = частное 325, остаток B_{16} (1110 — это B_{16})

Знакоместо 2: $325 / 16$ = частное 20, остаток 5

Знакоместо 3: $20 / 16$ = частное 1, остаток 4

Знакоместо 4: $1 / 16$ = частное 0, остаток 1

 $5205_{10} = 145B_{16}$

СОВЕТ

Чтобы лучше разобраться в работе различных систем счисления, напишите простую программу, подобную листингу 27.1 из занятия 27, “Применение потоков для ввода и вывода”. Она использует объект `std::cout` с манипуляторами для отображения целых чисел в шестнадцатеричной, десятичной и восьмеричной формах записи.

Чтобы отобразить целое число в двоичном формате, используйте класс `std::bitset`, который был описан на занятии 25, “Работа с битовыми флагами при использовании библиотеки STL”. Черпайте вдохновение из листинга 25.1.

ПРИЛОЖЕНИЕ Б

Ключевые слова языка C++

Ключевые слова зарезервированы компилятором для использования языком C++. Вы не можете определять классы, переменные или функции с ключевыми словами в качестве имен.

alignas	enum	return
alignof	explicit	short
and	export	signed
and_eq	extern	sizeof
asm	false	static
auto	float	static_assert
bitand	for	static_cast
bitor	friend	struct
bool	goto	switch
break	if	template
case	inline	this
catch	int	thread_local
char	long	throw
char16_t	mutable	true
char32_t	namespace	try
class	new	typedef
compl	noexcept	typeid
const	not	typename
constexpr	not_eq	union
const_cast	nullptr	unsigned
continue	operator	using
decltype	or	virtual
default	or_eq	void
delete	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
else	reinterpret_cast	xor_eq

ПРИМЕЧАНИЕ

На занятии 10, “Реализация наследования”, представлены два интересных ключевых слова — `final` и `override`. Они не являются зарезервированными ключевыми словами C++, т.е. вы можете использовать их для именования объектов и функций. Однако они имеют специальное значение в некоторых конструкциях, о чем рассказано на упомянутом занятии.

ПРИЛОЖЕНИЕ В

Приоритет операторов

Хорошей практикой программирования является использование круглых скобок, которые явно разграничивают операции. В отсутствие круглых скобок компилятор прибегает к предопределенному порядку очередности, в котором используются операторы. Это приоритет операторов, приведенный в таблице, которого придерживается компилятор C++ во избежание двусмысленности.

Приоритет операторов

Ранг	Название	Оператор
1	Разрешение области видимости	::
2	Прямое и косвенное обращение к члену класса, вызов функции, постфиксный инкремент и декремент	. -> [] () ++ --
3	Префиксный инкремент и декремент, дополнение и отрицание, унарные минус и плюс, получение адреса и ссылки, а также операторы new, new [], delete, delete [], sizeof() и приведения типов	++ -- ^ ! - + & * sizeof new new [] delete delete [] ()
4	Обращение к элементу по указателю	.* ->*
5	Умножение, деление, деление по модулю	* / %
6	Сложение, вычитание	+ -
7	Сдвиг влево, сдвиг вправо	<< >>
8	Меньше, меньше или равно, больше, больше или равно	< <= > >=
9	Равно, не равно	== !=
10	Побитовое И	&
11	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ	^
12	Побитовое ИЛИ	
13	Логическое И	&&
14	Логическое ИЛИ	
15	Тернарный условный оператор, генерация исключения, присваивание, составное присваивание	?: throw = *= /= %=
		+= -= <<= >>=
		&= = ^=
16	Запятая	,

ПРИЛОЖЕНИЕ Г

Коды ASCII

Работая с битами и байтами, компьютеры, по существу, работают с числами. Чтобы представить символьные данные в такой числовой системе, в свое время был принят стандарт ASCII (American Standard Code for Information Interchange). Стандарт ASCII назначает 7-битовые числовые коды латинским символам A–Z, a–z, цифрам 0–9, некоторым специальным клавишам (например,) и специальным символам (таким, как возврат на один символ).

7 битов обеспечивают 128 уникальных комбинаций, из которых первые 32 (0–31) зарезервированы, поскольку управляющие символы обычно используются для взаимодействия с периферийными устройствами, такими как принтеры.

Таблица ASCII отображаемых символов

Коды ASCII 32–127 используются для отображаемых символов, таких как 0–9, A–Z и a–z и некоторых других, таких как пробел. В приведенной ниже таблице перечислены десятичные и шестнадцатеричные значения, зарезервированные для этих символов.

Символ	DEC	HEX	Описание
	32	20	Пробел
!	33	21	Восклицательный знак
"	34	22	Двойные кавычки
#	35	23	Номер
\$	36	24	Доллар
%	37	25	Знак процента
&	38	26	Амперсанд
'	39	27	Одинарная кавычка
(40	28	Открывающая скобка
)	41	29	Закрывающая скобка
*	42	2A	Звездочка
+	43	2B	Плюс
,	44	2C	Запятая
-	45	2D	Дефис
.	46	2E	Точка
/	47	2F	Косая черта (или деление)
0	48	30	Ноль
1	49	31	Один
2	50	32	Два
3	51	33	Три
4	52	34	Четыре
5	53	35	Пять
6	54	36	Шесть
7	55	37	Семь
8	56	38	Восемь
9	57	39	Девять
:	58	3A	Двоеточие
;	59	3B	Точка с запятой
<	60	3C	Меньше (или открывающая угловая скобка)
=	61	3D	Равно
>	62	3E	Больше (или закрывающая угловая скобка)
?	63	3F	Вопросительный знак
@	64	40	Символ @
A	65	41	Прописная буква A
B	66	42	Прописная буква B

Продолжение таблицы

Символ	DEC	HEX	Описание
C	67	43	Прописная буква C
D	68	44	Прописная буква D
E	69	45	Прописная буква E
F	70	46	Прописная буква F
G	71	47	Прописная буква G
H	72	48	Прописная буква H
I	73	49	Прописная буква I
J	74	4A	Прописная буква J
K	75	4B	Прописная буква K
L	76	4C	Прописная буква L
M	77	4D	Прописная буква M
N	78	4E	Прописная буква N
O	79	4F	Прописная буква O
P	80	50	Прописная буква P
Q	81	51	Прописная буква Q
R	82	52	Прописная буква R
S	83	53	Прописная буква S
T	84	54	Прописная буква T
U	85	55	Прописная буква U
V	86	56	Прописная буква V
W	87	57	Прописная буква W
X	88	58	Прописная буква X
Y	89	59	Прописная буква Y
Z	90	5A	Прописная буква Z
[91	5B	Открывающая квадратная скобка
\	92	5C	Косая черта влево
]	93	5D	Закрывающая квадратная скобка
^	94	5E	Символ ^ (циркумфлекс)
_	95	5F	Символ подчеркивания
`	96	60	Символ ` (гравис)
a	97	61	Строчная буква a
b	98	62	Строчная буква b
c	99	63	Строчная буква c
d	100	64	Строчная буква d
e	101	65	Строчная буква e
f	102	66	Строчная буква f
g	103	67	Строчная буква g
h	104	68	Строчная буква h
i	105	69	Строчная буква i
j	106	6A	Строчная буква j
k	107	6B	Строчная буква k

Окончание таблицы

Символ	DEC	HEX	Описание
l	108	6C	Строчная буква l
m	109	6D	Строчная буква m
n	110	6E	Строчная буква n
o	111	6F	Строчная буква o
p	112	70	Строчная буква p
q	113	71	Строчная буква q
r	114	72	Строчная буква r
s	115	73	Строчная буква s
t	116	74	Строчная буква t
u	117	75	Строчная буква u
v	118	76	Строчная буква v
w	119	77	Строчная буква w
x	120	78	Строчная буква x
y	121	79	Строчная буква y
z	122	7A	Строчная буква z
{	123	7B	Открывающая фигурная скобка
	124	7C	Вертикальная линия
}	125	7D	Закрывающая фигурная скобка
~	126	7E	Символ ~ (тильда)
	127	7F	

ПРИЛОЖЕНИЕ Д

Ответы

Ответы к занятию 1

Контрольные вопросы

1. Интерпретатор — это инструмент, который интерпретирует исходный код (или промежуточный байт-код) и выполняет определенные действия. Компилятор получает на вход исходный текст программы и создает объектный файл. В языке C++ после компиляции и компоновки получается выполнимый файл, который может выполняться процессором непосредственно, без необходимости в дальнейшей интерпретации.
2. Компилятор получает на входе файл исходного кода C++ и создает объектный файл на машинном языке. Зачастую у вашего кода есть зависимости от библиотек и функций в других файлах кода. Создание этих связей и получение выполнимого файла, который интегрирует все явные и неявные зависимости, является задачей компоновщика.
3. Кодирование. Компиляция для создания объектного файла. Компоновка для создания выполнимого файла. Выполнение для тестирования. Отладка. Устранение ошибок в исходном тексте и повторение предыдущих этапов. В большинстве случаев компиляция и компоновка представляют собой один этап.

Упражнения

1. Отображает результат вычитания y из x , а также их умножения и сложения.
2. Результат: 2 48 14
3. Инструкция препроцессора `iostream`, находящаяся в строке 1, должна начинаться с `#`.
4. Отображает строку Hello Buggy World

Ответы к занятию 2

Контрольные вопросы

1. Код языка C++ чувствителен к регистру. `Int` не является для компилятора указанием целочисленного типа `int`.
2. Да.
/* Комментарий, использующий синтаксис в стиле C,
может располагаться в нескольких строках */

Упражнения

1. Причина неудачи в чувствительности к регистру компилятора C++. Ему неизвестно, что такое `std::Cout` и почему строка после этого не начинается с кавычки. Кроме того, функция `main()` всегда должна объявляться как возвращающая тип `int`.

2. Вот исправленная версия:

```
#include <iostream>
int main()
{
    std::cout << "Is there a bug here?"; // Теперь без ошибок
    return 0;
}
```

3. Эта программа основана на листинге 2.4 и демонстрирует вычитание и умножение:

```
#include <iostream>
using namespace std;

// Объявление функции
int DemoConsoleOutput();

int main()
{
    // Вызов функции
    DemoConsoleOutput();
    return 0;
}

// Определение функции
int DemoConsoleOutput()
{
    cout << "Вычитание 10 - 5 = " << 10 - 5 << endl;
    cout << "Умножение 10 * 5 = " << 10 * 5 << endl;

    return 0;
}
```

Результат

Вычитание $10 - 5 = 5$
 Умножение $10 * 5 = 50$

Ответы к занятию 3

Контрольные вопросы

1. В знаковом целом числе самый старший разряд означает знак числа (плюс или минус). Беззнаковое же целое число используется только для положительных значений.
2. Директива препроцессора `#define` инструктирует компилятор осуществить глобальную текстовую замену указанного значения. Однако эта замена не учитывает безопасности типов и является примитивным способом определения констант. Поэтому ее следует избегать.
3. Для гарантии, что она содержит определенное, а не случайное значение.
4. 2.
5. Имя не несет смысловой нагрузки и повторяет название типа. Хотя такой код компилируется нормально, людям его трудно читать и поддерживать. Такого желательно избегать. Для переменных лучше использовать описательные имена, которые отражают их цель, например

```
int Age = 0;
```

Упражнения

1. Это можно сделать несколькими способами:

```
enum YourCards {Ace = 43, Jack, Queen, King};
// Ace == 43, Jack == 44, Queen == 45, King == 46
// Альтернативный способ:
enum YourCards {Ace, Jack, Queen = 45, King};
// Ace == 0, Jack == 1, Queen == 45, King == 46
```
2. Просмотрите код листинга 3.5 и адаптируйте его для получения ответа на этот вопрос.
3. Вот программа, которая запрашивает радиус круга, а затем вычисляет его площадь и периметр:

```
#include <iostream>
using namespace std;

int main()
{
    const double Pi = 3.1416;
```

```

    cout << "Введите радиус: ";
    double radius = 0;
    cin >> radius;

    cout << "Площадь = " << Pi * radius * radius << endl;
    cout << "Длина   = " << 2 * Pi * radius << endl;

    return 0;
}

```

Результат

```

Введите радиус: 4
Площадь = 50.2656
Длина   = 25.1328

```

4. Если вы сохраните результат вычисления площади и периметра в целочисленной переменной, то при компиляции получите предупреждение (а не ошибку), и вывод будет выглядеть следующим образом:

Результат

```

Введите радиус: 4
Площадь = 50
Длина   = 25

```

5. Ключевое слово `auto` требует от компилятора автоматически выбрать тип переменной в зависимости от инициализирующего ее значения. В приведенном коде нет инициализации, и оператор приведет к ошибке при компиляции.

Ответы к занятию 4

Контрольные вопросы

1. 0 и 4 — это отсчитываемые от нуля индексы первого и последнего элементов массива с пятью элементами.
2. Нет, так как известна их небезопасность, особенно при обработке пользовательского ввода, поскольку они позволяют ввести строку длиннее массива.
3. Один нулевой завершающий символ.
4. Все зависит от того, как она используется. Если она используется в операторе `cout`, например, то механизм отображения будет читать последовательность символов, пока не найдет завершающий нулевой символ. При его отсутствии он пересечет границы массива и, возможно, приведет к краху приложения.
5. Достаточно заменить в объявлении вектора часть `int` частью `char`.

```
vector<char> dynArrChars(3);
```

Упражнения

1. Вот что получилось. Приложение инициализируется значением Rook (ладья), но оно достаточно простое, чтобы вы поняли все сами.

```
int main()
{
    enum Square
    {
        Empty = 0,
        Pawn,
        Rook,
        Knight,
        Bishop,
        King,
        Queen
    };
    Square chessBoard[8][8];
    // Инициализация клеток с ладьями
    chessBoard[0][0] = chessBoard[0][7] = Rook;
    chessBoard[7][0] = chessBoard[7][7] = Rook;
    return 0;
}
```

2. Чтобы присвоить значение пятому элементу массива, необходим доступ к элементу `myNumbers[4]`, поскольку индекс отсчитывается от нуля.
3. Обращение к четвертому элементу массива осуществляется до его инициализации или присваивания значения. Вывод будет непредсказуемым. Всегда инициализируйте переменные и массивы; в противном случае они будут содержать последнее значение, хранившееся в выделенной для них области памяти.

Ответы к занятию 5

Контрольные вопросы

1. Целочисленные типы не могут содержать десятичных значений, которые вполне возможны при делении двух чисел. Используйте тип `float`.
2. Поскольку компилятор интерпретирует числа как целые, результат равен 4.
3. Поскольку числитель указан как `32.0`, а не `32`, компилятор интерпретирует его как число с плавающей запятой, создав результат типа `float`, который составит 4,571.
4. Нет, `sizeof` — это оператор, который не может быть перегружен.
5. Это работает не так, как предполагалось, поскольку приоритет оператора суммы превосходит таковой для оператора сдвига, что приводит к сдвигу на $1 + 5 = 6$ битов, а не на 1 бит.
6. Результатом операции ИСКЛЮЧАЮЩЕЕ ИЛИ будет `false` согласно табл. 5.5.

Упражнения

1. Вот правильное решение:

```
int Result = ((number << 1) + 5) << 1; // Теперь очевидно
```

2. Результат содержит значение переменной `number`, сдвинутое на 7 битов влево, поскольку приоритет оператора `+` выше, чем оператора `<<`.
3. Ниже приведена программа, которая получает два логических значения, введенных пользователем, и демонстрирует результат использования побитовых операторов на них.

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Введите значение true(1) или false(0): ";
    bool value1 = false;
    cin >> value1;

    cout << "Введите второе значение true(1) или false(0): ";
    bool value2 = false;
    cin >> value2;

    cout << "Результаты логических операций: " << endl;
    cout << "И: " << (value1 & value2) << endl;
    cout << "ИЛИ: " << (value1 | value2) << endl;
    cout << "ИСКЛЮЧАЮЩЕЕ ИЛИ: " << (value1 ^ value2) << endl;

    return 0;
}
```

Результат

```
Введите значение true(1) или false(0): 1
Введите второе значение true(1) или false(0): 0
Результаты логических операций:
И: 0
ИЛИ: 1
ИСКЛЮЧАЮЩЕЕ ИЛИ: 1
```

Ответы к занятию 6

Контрольные вопросы

1. Отступы используются не для компилятора, а ради других программистов (людей), которые впоследствии будут читать или поддерживать ваш код.

2. Его следует избегать, чтобы ваш код не стал запутанным и дорогим в обслуживании.
3. См. код в ответе к упражнению 1, где используется оператор декремента.
4. Поскольку условие продолжения цикла `for` не удовлетворяется, цикл завершается, не выполнившись ни разу, поэтому оператор `cout` также ни разу не будет выполнен.

Упражнения

1. Необходимо помнить, что индексы массива отсчитываются от нуля, а индекс последнего элемента на единицу меньше его длины:

```
#include <iostream>
using namespace std;

int main()
{
    const int ARRAY_LEN = 5;
    int myNumbers[ARRAY_LEN] = {-55, 45, 9889, 0, 45};

    for(int nIndex = ARRAY_LEN - 1; nIndex >= 0; --nIndex)
        cout<<"myNumbers[" << nIndex
            << "] = "<<myNumbers[nIndex]<<endl;

    return 0;
}
```

Результат

```
myNumbers[4] = 45
myNumbers[3] = 0
myNumbers[2] = 9889
myNumbers[1] = 45
myNumbers[0] = -55
```

2. Вложенный цикл, эквивалентный использованному в листинге 6.14, но добавляющий элементы в два массива в обратном порядке, выглядит так:

```
#include <iostream>
using namespace std;

int main()
{
    const int ARRAY1_LEN = 3;
    const int ARRAY2_LEN = 2;

    int myNums1[ARRAY1_LEN] = {35, -3, 0};
    int MyInts2[ARRAY2_LEN] = {20, -1};

    cout << "Суммирование всех элементов myNums1 "
        << "со всеми элементами MyInts2:" << endl;
```



```

    for(int index1 = ARRAY1_LEN - 1; index1 >= 0; --index1)
        for(int index2 = ARRAY2_LEN - 1; index2 >= 0; --index2)
            cout << myNums1[index1] << " + " << MyInts2[index2]
                << " = " << myNums1[index1] + MyInts2[index2]
                << endl;

    return 0;
}

```

Результат

Суммирование всех элементов myNums1 со всеми элементами MyInts2:

```

0 + -1 = -1
0 + 20 = 20
-3 + -1 = -4
-3 + 20 = 17
35 + -1 = 34
35 + 20 = 55

```

3. Необходимо заменить фиксированное число 5 кодом, который запрашивает у пользователя следующее:

```

cout << "Сколько чисел Фибоначчи нужно вычислить: ";
int numsToCalculate = 0;
cin >> numsToCalculate;

```

4. Конструкция switch-case с использованием перечисляемой константы, указывающая, принадлежит ли цвет радуге, выглядит так:

```

#include <iostream>
using namespace std;
int main() {
    enum Colors {
        Violet = 0,
        Indigo,
        Blue,
        Green,
        Yellow,
        Orange,
        Red,
        Crimson,
        Beige,
        Brown,
        Peach,
        Pink,
        White,
    };

    cout << "Доступные цвета: " << endl;
    cout << "Violet: " << Violet << endl;
    cout << "Indigo: " << Indigo << endl;
    cout << "Blue: " << Blue << endl;
    cout << "Green: " << Green << endl;
    cout << "Yellow: " << Yellow << endl;
}

```

```

cout << "Orange: " << Orange << endl;
cout << "Red: " << Red << endl;
cout << "Crimson: " << Crimson << endl;
cout << "Beige: " << Beige << endl;
cout << "Brown: " << Brown << endl;
cout << "Peach: " << Peach << endl;
cout << "Pink: " << Pink << endl;
cout << "White: " << White << endl;
cout << "Введите выбранный код: ";
int YourChoice = Blue;
cin >> YourChoice;

switch(YourChoice) {
    case Violet:
    case Indigo:
    case Blue:
    case Green:
    case Yellow:
    case Orange:
    case Red:
        cout << "Выбранный цвет есть в радуге!" << endl;
        break;

    default:
        cout << "Этого цвета в радуге нет." << endl;
        break;
}

return 0;
}

```

Результат

```

Доступные цвета:
Violet: 0
Indigo: 1
Blue: 2
Green: 3
Yellow: 4
Orange: 5
RED: 6
Crimson: 7
Beige: 8
Brown: 9
Peach: 10
Pink: 11
White: 12
Введите выбранный код: 4
Выбранный цвет есть в радуге!

```

5. В выражении условия выхода из цикла `for` программист по невнимательности осуществил не сравнение, а присваивание счетчику значения 10.
6. Оператор `while` сопровождается пустым оператором `;` в той же строке. Поэтому следующий за ним код увеличения значения переменной `LoopCounter` никогда не будет достигнут, а следовательно, условие выхода никогда не будет выполнено, цикл никогда не закончится и операторы после него никогда не выполнятся.
7. Отсутствует оператор `break` (т.е. часть `default` будет выполняться всегда, вне зависимости от сработавшей ранее части `case`, что явно неправильно).

Ответы к занятию 7

Контрольные вопросы

1. Область видимости этих переменных — реализация функции.
2. `someNumber` — это ссылка на переменную в вызывающей функции. Она не содержит копию значения.
3. Рекурсивная функция.
4. Перегруженные функции.
5. На вершину! Стек похож на стопку тарелок; ту, которая находится сверху, можно взять, и именно на нее указывает указатель вершины стека.

Упражнения

1. Прототипы функций выглядели бы следующим образом:

```
double Area(double Radius);           // Сфера
double Area(double Radius, double Height); // Цилиндр
```

Реализации (определения) функций используют соответствующие формулы, предоставленные в вопросе, и возвращают вызывающей стороне объем как значение.
2. Аналог — в листинге 7.8. Прототип функции был бы следующим:

```
void ProcessArray(double numbers[], int length);
```
3. Чтобы это сработало, параметр `Result` функции `Area` должен быть ссылкой:

```
void Area(double radius, double &result)
```
4. Либо параметр со значением по умолчанию должен располагаться в конце, либо значения по умолчанию нужно определить для всех параметров.
5. Функция должна вернуть данные вызывающей стороне по ссылке:

```
void Area(double radius, double &area, double &circumference)
{
    area = 3.14 * radius * radius;
    circumference = 2 * 3.14 * radius;
}
```

Ответы к занятию 8

Контрольные вопросы

1. Если бы компилятор позволял такое, то это был бы очень простой способ нарушить то, для чего предназначены константные ссылки: защита данных от изменения.
2. Это операторы.
3. Адрес области памяти.
4. Оператор `*`.

Упражнения

1. 40.
2. В первом варианте аргументы копируются в вызываемую функцию. Во втором они не копируются, поскольку это ссылки на переменные вызывающей стороны, и функция может их изменять. Третий вариант использует указатели, которые в отличие от ссылок могут быть пусты или недопустимы. В этом случае следует обеспечить их допустимость.
3. Используйте ключевое слово `const`:

```
1: const int* pNum1 = &number;
```
4. Вы присваиваете целое число непосредственно указателю (т.е. перезаписываете содержавшийся в нем адрес целочисленного значения в памяти):

```
*pNumber = 9; // было: pNumber = 9;
```
5. Двойное освобождение одного и того же адреса области памяти, возвращенного оператором `new` указателю `pNumber` и скопированного в указатель `pNumberCopy`. Удалите один из операторов `delete`.
6. 30.

Ответы к занятию 9

Контрольные вопросы

1. В динамической памяти. Это то же самое, что и выделение памяти для типа `int` с использованием оператора `new`.
2. Оператор `sizeof()` вычисляет размер класса на основе заявленных переменных-членов. Поскольку размер указателя является постоянным и не зависит от размера данных, на которые он указывает, размер класса, содержащего один такой указатель-член, также остается постоянным.
3. Никто, кроме методов этого же класса.

4. Да, может.
5. Конструктор обычно используется для инициализации переменных-членов и ресурсов.
6. Деструкторы обычно используются для освобождения памяти и ресурсов.

Упражнения

1. Язык C++ чувствителен к регистру. Объявление класса должно начинаться со слова `class`, а не `Class`. Оно должно закончиться точкой с запятой (;), как показано ниже.

```
class Human
{
    int age;
    string name;

public:
    Human() {}
};
```

2. Поскольку переменная-член `Human::age` закрытая (вспомните, что, в отличие от структуры, члены класса являются по умолчанию закрытыми) и нет никакой открытой функции доступа, то нет и никакого способа, которым пользователь этого класса может обратиться к переменной `age`.
3. Вот версия класса `Human` со списком инициализации в конструкторе:

```
class Human
{
    int age;
    string name;

public:
    Human(string inputName, int inputAge)
        : name(inputName), age(inputAge) {}
};
```

4. Обратите внимание: число π является невидимым извне класса, как и требовалось:

```
#include <iostream>
using namespace std;
class Circle {
    const double Pi;
    double radius;
public:
    Circle(double InputRadius) : radius(InputRadius), Pi(3.1416) {}
    double GetCircumference() {
        return 2 * Pi * radius;
    }
};
```

```
double GetArea() {
    return Pi * radius * radius;
}
};
int main() {
    cout << "Введите радиус: ";
    double radius = 0;
    cin >> radius;
    Circle MyCircle(radius);
    cout << "Окружность = " << MyCircle.GetCircumference() << endl;
    cout << "Площадь = " << MyCircle.GetArea() << endl;
    return 0;
}
```

Ответы к занятию 10

Контрольные вопросы

1. Используйте модификатор доступа `protected`. Он обеспечит видимость члена базового класса для производного класса, но не таковому вне его.
2. Часть объекта, соответствующая производному классу, срезается, а по значению передается только часть, соответствующая базовому классу. Результат может быть непредсказуемым.
3. Композиция делает проект гибче.
4. Позволяет раскрыть методы базового класса.
5. Нет, поскольку у первого класса, который специализирует класс `Base`, т.е. класса `Derived`, есть отношения закрытого наследования с классом `Base`. Таким образом, открытые члены класса `Base` являются закрытыми для класса `Sub-Derived`, а следовательно, они недоступны.

Упражнения

1. Конструкторы вызываются в порядке объявления:
`Mammal-Bird-Reptile-Platypus`.
Удаление осуществляется в обратном порядке.
2. Например, так:

```
class Shape
{
    // ... Члены класса Shape
};

class Polygon: public Shape
{
    // ... Члены класса Polygon
}
```

```
class Triangle: public Polygon
{
    // ... Члены класса Triangle
}
```

3. Отношения наследования между классами D1 и Base должны быть закрытыми.
4. По умолчанию классы наследуются закрыто. Если бы Derived был структурой, то наследование было бы открытым.
5. Функция SomeFunc() ожидает передачи параметра типа Base по значению. Это означает, что вызов с указанным производным типом приведет к срезке, результатом которой непредсказуем:

```
Derived objectDerived;
SomeFunc(objectDerived); // Срезка
```

Ответы к занятию 11

Контрольные вопросы

1. Объявите абстрактный класс Shape с чисто виртуальными функциями Area() и Print(), и это заставит классы Circle и Triangle их реализовать.
2. Нет. Таблица виртуальных функций создается только для тех классов, которые содержат виртуальные функции.
3. Да, поскольку его экземпляр все еще не может быть создан. Пока у класса есть по крайней мере одна чистая виртуальная функция, он остается абстрактным, независимо от наличия или отсутствия других полностью определенных функций или переменных.

Упражнения

1. Ниже показана иерархия наследования для абстрактного класса Shape и производных от него классов Circle и Triangle.

```
#include<iostream>
using namespace std;

class Shape
{
public:
    virtual double Area() = 0;
    virtual void Print() = 0;
};

class Circle
{
    double Radius;
public:
```

```

Circle(double inputRadius) : Radius(inputRadius) {}

double Area()
{
    return 3.1415 * Radius * Radius;
}

void Print()
{
    cout << "Circle::Print()" << endl;
}
};

class Triangle
{
    double Base, Height;
public:
    Triangle(double inputBase, double inputHeight)
        : Base(inputBase), Height(inputHeight) {}

    double Area()
    {
        return 0.5 * Base * Height;
    }

    void Print()
    {
        cout << "Triangle::Print()" << endl;
    }
};

int main()
{
    Circle myRing(5);
    Triangle myWarningTriangle(6.6, 2);

    cout << "Площадь круга: " << myRing.Area() << endl;
    cout << "Площадь треугольника: " << myWarningTriangle.Area()
        << endl;

    myRing.Print();
    myWarningTriangle.Print();

    return 0;
}

```

2. Отсутствует виртуальный деструктор!
3. Без виртуального деструктора последовательность выполнения конструкторов была бы такой: `Vehicle()`, затем — `Car()`, а не виртуальный деструктор будет вызван только один — `~Vehicle()`.

Ответы к занятию 12

Контрольные вопросы

1. Нет, язык C++ не позволяет двум функциям с одним и тем же именем иметь разные возвращаемые значения. Вы можете создать две реализации оператора [] с разными типами возвращаемого значения, но при этом один оператор может быть определен как константная функция, а другой нет. В таком случае компилятор C++ выбирает неконстантную версию для действий, связанных с присваиванием, и константную версию в противном случае:

```
const Type& operator[](int Index) const;
Type& operator[](int Index);
```

2. Да, но только если я не хочу, чтобы мой класс позволил копировать или присваивать себя. Такое ограничение имеет смысл при программировании синглтона — класса, который разрешает иметь только один его экземпляр (см. листинг 9.10).
3. Поскольку у него нет никаких динамически выделенных ресурсов, содержащихся в пределах класса Date, способных вызвать ненужные циклы выделения и освобождения памяти в пределах копирующего конструктора или копирующего оператора присваивания, этот класс не является хорошим кандидатом на наличие конструктора перемещения или оператора присваивания при перемещении.

Упражнения

1. Оператор преобразования int().

```
class Date
{
    int day, month, year;
public:
    operator int()
    {
        return ((year * 10000) + (month * 100) + day);
    }

    // Конструктор и т.д.
};
```

2. Конструктор перемещения и оператор присваивания при перемещении приведены ниже.

```
class DynIntegers
{
private:
    int* arrayNums;
```

```
public:
    // Перемещающий конструктор
    DynIntegers(DynIntegers&& moveSrc)
    {
        arrayNums = moveSrc.arrayNums; // Получение владения
        moveSrc.arrayNums = nullptr;    // Освобождение от владения
                                         // исходного объекта
    }
    // Перемещающий оператор присваивания
    DynIntegers& operator=(DynIntegers&& moveSrc)
    {
        if(this != &moveSrc)
        {
            delete[] arrayNums;          //Освобождение своего ресурса
            arrayNums = moveSrc.arrayNums;
            moveSrc.arrayNums = nullptr;
        }
        return *this;
    }
    ~DynIntegers() {delete[] arrayNums;} // Деструктор
    // Реализация конструктора по умолчанию, копирующего конструктора,
    // оператора присваивания
};
```

Ответы к занятию 13

Контрольные вопросы

1. Оператор `dynamic_cast`.
2. Исправьте функцию, конечно. Оператор `const_cast` и операторы приведения вообще должны быть последним средством.
3. Правда.
4. Да, правда.

Упражнения

1. Результат динамической операции приведения всегда должен проверяться на корректность:

```
void DoSomething(Base* objBase)
{
    Derived* objDer = dynamic_cast <Derived*>(objBase);

    if(objDer) // Проверка корректности
        objDer->DerivedClassMethod();
}
```

2. Используйте оператор `static_cast`, поскольку известно, что указываемый объект имеет тип `Tuna`. Взяв за основу листинг 13.1, можно получить такую функцию `main()`:

```
int main()
{
    Fish* pFish = new Tuna;
    Tuna* pTuna = static_cast<Tuna*>(pFish);

    // Tuna::BecomeDinner работает только при
    // использовании корректного Tuna*
    pTuna->BecomeDinner();

    // Виртуальный деструктор в Fish гарантирует вызов ~Tuna()
    delete pFish;

    return 0;
}
```

Ответы к занятию 14

Контрольные вопросы

1. Конструкция препроцессора, препятствующая множественному или рекурсивному включению файлов заголовка.
2. 4.
3. $10 + 10 / 5 = 10 + 2 = 12$.
4. Использовать скобки: `#define SPLIT(x) ((x)/5)`

Упражнения

1. `#define MULTIPLY(a,b) ((a)*(b))`
2. Вот шаблон, аналогичный макросу из контрольного вопроса 4:

```
template<typename T> T Split(const T& input)
{
    return (input / 5);
}
```

3. Шаблонная функция `swap()` будет такой:

```
template <typename T>
void Swap(T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

4. `#define QUARTER(x) ((x) / 4)`

5. Определение шаблона класса выглядело бы так:

```
template <typename Array1Type, typename Array2Type>
class TwoArrays
{
private:
    Array1Type Array1[10];
    Array2Type Array2[10];
public:
    Array1Type& GetArray1Element(int Index){return Array1[Index];}
    Array2Type& GetArray2Element(int Index){return Array2[Index];}
};
```

6. Вот как может выглядеть шаблонная функция Display():

```
#include <iostream>
using namespace std;
void Display() {
}
template <typename First, typename ...Last>
void Display(First a, Last... U) {
    cout << a << endl;
    Display(U...);
}
int main() {
    Display('a');
    Display(3.14);
    Display('a', 3.14);
    Display('z', 3.14567, "Вариадический шаблон");
    return 0;
}
```

Результат

```
a
3.14
a
3.14
z
3.14567
Вариадический шаблон
```

Ответы к занятию 15

Контрольные вопросы

1. Контейнер deque. Только он обеспечивает вставку в начало и в конец контейнера за константное время.

2. Контейнер `std::set` или `std::map`, если у вас пары “ключ–значение”. Если элементы могут дублироваться, выберите контейнер `std::multiset` или `std::multimap`.
3. Да. Создавая экземпляр шаблона `std::set`, можете также задать второй параметр шаблона, являющийся двоичным предикатом, который класс `set` использует как критерий сортировки. Задайте в этом предикате критерии соответственно вашим требованиям.
4. Мост между алгоритмами и контейнерами образуют итераторы, чтобы первые (являющиеся обобщением) могли взаимодействовать с последними без необходимости знать конкретный тип контейнера.
5. Контейнер `hash_set` не является стандартным для C++. Вы не должны использовать его в переносимом приложении; применяйте в таких случаях контейнер `std::map`.

Ответы к занятию 16

Контрольные вопросы

1. Шаблон `std::basic_string <T>`.
2. Скопируйте эти две строки в два строковых объекта. Преобразуйте каждую скопированную строку в нижний или в верхний регистр. Получите результат сравнения преобразованных копий строк.
3. Нет, они не подобны. Строки в стиле C — это фактически простые указатели, родственные символьному массиву, тогда как строка библиотеки STL — это класс `string`, реализующий различные операторы и функции-члены для обработки строк, что делает их применение простым настолько, насколько это возможно.

Упражнения

1. Программа должна использовать функцию `std::reverse()`:

```
#include <string>
#include <iostream>
#include <algorithm>

int main()
{
    using namespace std;

    cout << "Введите слово для проверки:" << endl;
    string strInput;
    cin >> strInput;

    string strCopy(strInput);
```

```
reverse(strCopy.begin(), strCopy.end());

if (strCopy == strInput)
    cout << strInput << " - это палиндром!" << endl;
else
    cout << strInput << " - это не палиндром." << endl;

return 0;
}
```

2. Используйте функцию std::find():

```
#include <string>
#include <iostream>

using namespace std;

// Количество символов 'chToFind' в строке "strInput"
int GetNumCharacters(string& strInput, char chToFind)
{
    int nNumCharactersFound = 0;

    size_t nCharOffset = strInput.find(chToFind);
    while (nCharOffset != string::npos)
    {
        ++nNumCharactersFound;

        nCharOffset = strInput.find(chToFind, nCharOffset + 1);
    }
    return nNumCharactersFound;
}

int main()
{
    cout << "Введите строку:" << endl << "> ";
    string strInput;
    getline(cin, strInput);

    int nNumVowels = GetNumCharacters(strInput, 'a');
    nNumVowels += GetNumCharacters(strInput, 'e');
    nNumVowels += GetNumCharacters(strInput, 'i');
    nNumVowels += GetNumCharacters(strInput, 'o');
    nNumVowels += GetNumCharacters(strInput, 'u');

    // Обработка прописных букв

    cout << "Количество гласных в предложении = " << nNumVowels;

    return 0;
}
```

3. Используйте функцию `toupper()`:

```
#include <string>
#include <iostream>
#include <algorithm>

int main()
{
    using namespace std;

    cout << "Введите строку:" << endl;
    cout << "> ";

    string strInput;
    getline(cin, strInput);
    cout << endl;

    for(size_t nCharIndex = 0;
        nCharIndex < strInput.length();
        nCharIndex += 2)
        strInput[nCharIndex] = toupper(strInput[nCharIndex]);

    cout << "Преобразованная строка: " << endl;
    cout << strInput << endl << endl;

    return 0;
}
```

4. Это может быть очень просто реализовано так:

```
#include <string>
#include <iostream>

int main()
{
    using namespace std;

    const string str1 = "I";
    const string str2 = "Love";
    const string str3 = "STL";
    const string str4 = "String.";

    string strResult = str1+" "+str2+" "+str3+" "+str4;

    cout << "Предложение:" << endl;
    cout << strResult;

    return 0;
}
```

5. Воспользуйтесь `std::string::find()`:

```
#include <iostream>
#include <string>
int main() {
    using namespace std;
    string sampleStr("Good day String! Today is beautiful!");
    cout << "Строка: " << sampleStr << endl;
    cout << "Позиции буквы 'a'" << endl;
    auto charPos = sampleStr.find('a', 0);

    while(charPos != string::npos) {
        cout << "'" << 'a' << "' found";
        cout << " найдена в позиции: " << charPos << endl;
        // 'find' продолжает поиск со следующего символа
        onwards
        size_t charSearchPos = charPos + 1;
        charPos = sampleStr.find('a', charSearchPos);
    }

    return 0;
}
```

Результат

```
Строка: Good day String! Today is beautiful!
Позиции буквы 'a'
'a' найдена в позиции: 6
'a' найдена в позиции: 20
'a' найдена в позиции: 28
```

Ответы к занятию 17

Контрольные вопросы

1. Нет, не могут. За константное время элементы могут быть только добавлены в конец вектора.
2. Еще 10. При 11-й вставке произойдет повторное выделение.
3. Извлекает последний элемент, т.е. удаляет элемент с конца.
4. Типа `Matmal`.
5. С помощью оператора индексации (`[]`) или функции `at()`.
6. Итератор прямого доступа.

Упражнения

1. Одно из решений таково:

```
#include <vector>
#include <iostream>
```



```

using namespace std;

char DisplayOptions()
{
    cout << "Выберите действие:" << endl;
    cout << "1: Ввести целое число" << endl;
    cout << "2: Запрос значения по индексу" << endl;
    cout << "3: Вывод вектора" << endl << "> ";
    cout << "4: Выход!" << endl << "> ";

    char ch;
    cin >> ch;

    return ch;
}

int main()
{
    vector<int> vecData;

    char chUserChoice = '\0';
    while((chUserChoice = DisplayOptions()) != '4')
    {
        if (chUserChoice == '1')
        {
            cout << "Введите вставляемое целое число: ";
            int nDataInput = 0;
            cin >> nDataInput;

            vecData.push_back(nDataInput);
        }
        else if (chUserChoice == '2')
        {
            cout << "Введите индекс от 0 до ";
            cout << (vecData.size() - 1) << ": ";
            int nIndex = 0;
            cin >> nIndex;

            if (nIndex < (vecData.size()))
            {
                cout<<"Element["<<nIndex<<"] = "<<vecData[nIndex];
                cout << endl;
            }
        }
        else if (chUserChoice == '3')
        {
            cout << "Содержимое вектора: ";
            for(size_t nIndex = 0;
                nIndex < vecData.size(); ++nIndex)
                cout << vecData[nIndex] << ' ';
            cout << endl;
        }
    }
}

```

```
    }
}
return 0;
}
```

2. Используйте алгоритм `std::find()`:

```
vector<int>::iterator elementFound = std::find(vecData.begin(),
                                              vecData.end(), value);
```

3. Вот возможное решение. Обратите внимание, что класс `Dimensions` реализует оператор `const char*`, позволяющий работать с ним потоку `cout`.

```
#include <vector>
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
char DisplayOptions() {
    cout << "Выберите действие:" << endl;
    cout << "1: Ввод длины и ширины " << endl;
    cout << "2: Запрос значения по индексу" << endl;
    cout << "3: Вывод размеров всех упаковок" << endl;
    cout << "4: Выход!" << endl << "> ";
    char ch;
    cin >> ch;
    return ch;
}
class Dimensions {
    int length, breadth;
    string strOut;
public:
    Dimensions(int inL, int inB) : length(inL), breadth(inB) {}
    operator const char * () {
        stringstream os;
        os << "Длина "s << length << ", ширина: "s
        << breadth << endl;
        strOut = os.str();
        return strOut.c_str();
    }
};
int main() {
    vector<Dimensions> vecData;
    char chUserChoice = '\0';

    while((chUserChoice = DisplayOptions()) != '4') {
        if (chUserChoice == '1') {
            cout << "Введите длину и ширину: " << endl;
            int length = 0, breadth = 0;
            cin >> length;
            cin >> breadth;
            vecData.push_back(Dimensions(length, breadth));
```

```

    } else if (chUserChoice == '2') {
        cout << "Введите индекс от 0 до ";
        cout << (vecData.size() - 1) << ": ";
        size_t index = 0;
        cin >> index;

        if (index < (vecData.size())) {
            cout << "Element[" << index << "] = "
                 << vecData[index];
            cout << endl;
        }
    } else if (chUserChoice == '3') {
        cout << "Содержимое вектора: ";

        for(size_t index = 0; index < vecData.size(); ++index)
            cout << vecData[index] << ' ';

        cout << endl;
    }
}

return 0;
}

```

4. Инициализация списком делает код компактнее:

```

#include <deque>
#include <string>
#include <iostream>
using namespace std;
template<typename T>
void DisplayDeque(deque<T> inDQ) {
    for(auto element = inDQ.cbegin();
        element != inDQ.cend();
        ++element)
        cout << * element << endl;
}

int main() {
    deque<string> strDq { "Hello"s, "Containers are cool"s,
                        "C++ is evolving!"s
    };
    DisplayDeque(strDq);
    return 0;
}

```

Ответы к занятию 18

Контрольные вопросы

1. Элементы вполне могут быть вставлены в середину списка, равно как и в его конец или начало. Никакого выигрыша или потери производительности позиция вставки не обеспечит.

2. Особенность списка в том, что такие операции не влияют на допустимость существующих итераторов.
3. `mList.clear();` или `mList.erase(mList.begin(), mList.end());`;
4. Да, перегруженная версия функции `insert()` позволяет вставить диапазон элементов из исходной коллекции.

Упражнения

1. Решение, как в упражнении 1 занятия 17, “Классы динамических массивов библиотеки STL”, для вектора. Единственное отличие — в использовании функции вставки для списка: `mList.insert(mList.begin(), nDataInput);`
2. Сохраните итераторы для двух элементов в списке. Вставьте элемент между ними, используя функцию вставки. Используйте итераторы для демонстрации того, что они все еще в состоянии обратиться к значениям, на которые указывали прежде.
3. Вот возможное решение:

```
#include <vector>
#include <list>
#include <iostream>

using namespace std;

int main() {
    vector<int> vecData(4);
    vecData[0] = 0;
    vecData[1] = 10;
    vecData[2] = 20;
    vecData[3] = 30;
    list<int> listIntegers;
    // Вставка содержимого вектора в начало списка
    listIntegers.insert(listIntegers.begin(),
                       vecData.begin(), vecData.end());
    cout << "Содержимое списка: ";
    list<int>::const_iterator iElement;

    for(iElement = listIntegers.begin();
        iElement != listIntegers.end();
        ++iElement)
        cout << * iElement << " ";

    return 0;
};
```

4. Возможное решение приведено ниже.

```
#include <list>
#include <string>
#include <iostream>
```

```

using namespace std;

int main() {
    list<string> listNames;
    listNames.push_back("Jack");
    listNames.push_back("John");
    listNames.push_back("Anna");
    listNames.push_back("Skate");
    cout << "Содержимое списка: ";
    list<string>::const_iterator iElement;

    for(iElement=listNames.begin(); iElement!=listNames.end();
        ++iElement)
        cout << * iElement << " ";

    cout << endl;
    cout << "Содержимое после реверса: ";
    listNames.reverse();

    for(iElement=listNames.begin(); iElement!=listNames.end();
        ++iElement)
        cout << * iElement << " ";

    cout << endl;
    cout << "Содержимое после сортировки: ";
    listNames.sort();

    for(iElement=listNames.begin(); iElement!=listNames.end();
        ++iElement)
        cout << * iElement << " ";

    cout << endl;
    return 0;
}

```

Ответы к занятию 19

Контрольные вопросы

1. Критерий сортировки по умолчанию определяется как `std::less<>`, что фактически задействует оператор `operator<` для сравнения двух целых чисел, и возвращает значение `true`, если первое число меньше второго.
2. Рядом, один за другим.
3. Для всех контейнеров библиотеки STL это функция `size()`.

Упражнения

1. Одно из возможных решений:

```
#include <set>
#include <iostream>
#include <string>
using namespace std;
template <typename T>
void DisplayContents(const T & container) {
    for(auto iElement = container.cbegin();
        iElement != container.cend();
        ++iElement)
        cout << * iElement << endl;

    cout << endl;
}

struct ContactItem {
    string name;
    string phoneNum;
    string displayAs;
    ContactItem(const string & nameInit, const string & phone) {
        name = nameInit;
        phoneNum = phone;
        displayAs = (name + ": " + phoneNum);
    }
    // Используется в set::find()
    bool operator == (const ContactItem & itemToCompare) const
    {
        return (itemToCompare.phoneNum == this->phoneNum);
    }
    // Используется для сортировки
    bool operator < (const ContactItem & itemToCompare) const
    {
        return (this->phoneNum < itemToCompare.phoneNum);
    }
    // Используется для вывода DisplayContents в cout
    operator const char * () const
    {
        return displayAs.c_str();
    }
};

int main() {
    set<ContactItem> setContacts;
    setContacts.insert(
        ContactItem("Jack Welsch", "+1 7889 879 879"));
    setContacts.insert(
        ContactItem("Bill Gates", "+1 97 7897 8799 8"));
    setContacts.insert(
        ContactItem("Angi Merkel", "+49 23456 5466"));
    setContacts.insert(
        ContactItem("Vlad Putin", "+7 6645 4564 797"));
    setContacts.insert(
```

```

        ContactItem("John Travolta", "+1 234 4564 789"));
setContacts.insert(
    ContactItem("Ben Affleck", "+1 745 641 314"));
DisplayContents(setContacts);
cout << "Введите искомый номер: ";
string input;
getline(cin, input);
auto contactFound = setContacts.find(ContactItem("", input));

if (contactFound != setContacts.end()) {
    cout << "Номер принадлежит "
        << ( * contactFound).name << endl;
    DisplayContents(setContacts);
} else
    cout << "Контакт не найден" << endl;

return 0;
}

```

2. Структура и определение мультимножества могут быть такими:

```

#include <set>
#include <iostream>
#include <string>
using namespace std;
struct PAIR_WORD {
    string word;
    string meaning;
    PAIR_WORD(const string&sWord, const string&sMeaning)
    : word(sWord), meaning(sMeaning) {}
    bool operator<(const PAIR_WORD& pairAnotherWord)const
    {
        return (word < pairAnotherWord.word);
    }
    bool operator==(const string & key) {
        return (key == this->word);
    }
};

int main() {
    multiset <PAIR_WORD> msetDictionary;
    PAIR_WORD word1("C++", "A programming language");
    PAIR_WORD word2("Programmer", "A geek!");
    msetDictionary.insert(word1);
    msetDictionary.insert(word2);
    cout << "Введите слово, которое вы хотите найти"<< endl;
    string input;
    getline(cin, input);
    auto element = msetDictionary.find(PAIR_WORD(input, ""));
}

```

```

        if (element != msetDictionary.end())
            cout << "Значение: " << (*element).meaning << endl;

        return 0;
    }

```

3. Одно из решений приведено ниже.

```

#include <set>
#include <iostream>
using namespace std;
template <typename T>
void DisplayContent(const T & cont) {
    T::const_iterator element;

    for(element = cont.begin();
        element != cont.end(); ++element)
        cout << * element << " ";
}

int main() {
    multiset <int> msetIntegers;
    msetIntegers.insert(5);
    msetIntegers.insert(5);
    msetIntegers.insert(5);
    set <int> setIntegers;
    setIntegers.insert(5);
    setIntegers.insert(5);
    setIntegers.insert(5);
    cout << "Вывод содержимого multiset: ";
    DisplayContent(msetIntegers);
    cout << endl;
    cout << "Вывод содержимого set: ";
    DisplayContent(setIntegers);
    cout << endl;
    return 0;
}

```

Ответы к занятию 20

Контрольные вопросы

1. Критерий сортировки по умолчанию определяется как `std::less<>`.
2. Рядом, один за другим.
3. Функция `size()`.
4. В отображении нет двойных элементов!

Упражнения

1. Ассоциативный контейнер, который допускает двойные записи, например

```
std::multimap:
std::multimap<string, string> multimapPeopleNamesToNumbers;
```

2. Определение предиката таково:

```
struct fPredicate
{
    bool operator<(const wordProperty& lsh,
                  const wordProperty& rsh) const
    {
        return (lsh.strWord < rsh.strWord);
    }
};
```

3. Подобный пример приведен в упражнении 3 занятия 19, “Классы множеств STL”.

Ответы к занятию 21

Контрольные вопросы

1. Унарный предикат (unary predicate).
2. Например, он может отображать данные или просто подсчитывать элементы. См. применение `std::transform()` в листинге 21.6 с предикатом `tolower()`.
3. В языке C++ все сущности, которые существуют во время выполнения приложения, являются объектами. В данном случае даже структуры и классы могут работать как функции, отсюда и термин — *функциональный объект*. Обратите внимание на то, что функции могут быть доступны и через указатели на функции — они тоже являются функциональными объектами.

Упражнения

1. Решение имеет следующий вид:

```
template <typename elementType=int>
struct Double
{
    void operator()(const elementType element) const
    {
        cout << element * 2 << ' ';
    }
};
```

Вот как может быть использован этот унарный предикат:

```
#include<vector>
#include<iostream>
#include<algorithm>
```

```
using namespace std;
int main() {
    vector<int> vecIntegers;

    for(int nCount = 0; nCount < 10; ++nCount)
        vecIntegers.push_back(nCount);

    cout << "Вывод вектора целых чисел: " << endl;
    // Выводим массив целых чисел
    for_each(vecIntegers.begin(), // Начало диапазона
             vecIntegers.end(),   // Конец диапазона
             Double<>());         // Унарный функтор
    return 0;
}
```

2. Добавьте целочисленный член, значение которого увеличивается при каждом вызове оператора `operator()`:

```
template <typename elementType = int>
struct Double {
    int m_nUsageCount;

    // Конструктор
    Double() : m_nUsageCount(0) {};

    void operator()(const elementType element) const
    {
        ++m_nUsageCount;
        cout << element * 2 << ' ';
    }
};
```

3. Бинарный предикат имеет следующий вид:

```
template <typename elementType>
class SortAscending {
public:
    bool operator()(const elementType & num1,
                    const elementType & num2) const {
        return (num1 < num2);
    }
};
```

Вот как может быть использован этот предикат:

```
#include<iostream>
#include<vector>
#include<algorithm>
int main() {
    std::vector<int> vecIntegers;

    // Вставка чисел: 100, 90... 20, 10
    for(int nSample = 10; nSample > 0; --nSample)
        vecIntegers.push_back(nSample * 10);
```

```

std::sort(vecIntegers.begin(), vecIntegers.end(),
          SortAscending<int>());

for(size_t nElementIndex = 0;
    nElementIndex < vecIntegers.size();
    ++nElementIndex)
    cout << vecIntegers[nElementIndex] << ' ';

return 0;
}

```

Ответы к занятию 22

Контрольные вопросы

1. Лямбда всегда начинается с квадратных скобок ([]).
2. С помощью списка захвата:

```
[Var1, Var2, ...](Type& param) { ...; }
```

3. Следующим образом:

```
[Var1, Var2, ...](Type& param) -> ReturnType { ...; }
```

Упражнения

1. Вот одно из возможных решений:

```

sort(vecNumbers.begin(), vecNumbers.end(),
     [](int num1, int num2) {return (num1 > num2); });

```

2. Вот как может выглядеть это лямбда-выражение:

```

cout << "Число, добавляемое ко всем элементам: ";
int numInput = 0;
cin >> numInput;

for_each(vecNumbers.begin(), vecNumbers.end(),
         [=](int & element) { element += numInput;});

```

Пример, демонстрирующий решения упражнений 1 и 2:

```

#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
template <typename T>
void DisplayContents(const T & container) {
    for(auto element = container.cbegin();
        element != container.cend();
        ++element)
        cout << * element << ' ';
}

```

```

        cout << endl;
    }
    int main() {
        vector<int> vecNumbers { 25, -5, 122, 2011, -10001 };
        DisplayContents(vecNumbers);
        sort(vecNumbers.begin(), vecNumbers.end());
        DisplayContents(vecNumbers);
        sort(vecNumbers.begin(), vecNumbers.end(),
            [](int Num1, int Num2) {
                return (Num1 > Num2);
            });
        DisplayContents(vecNumbers);
        cout << "Число, добавляемое ко всем элементам: ";
        int numcontainer = 0;
        cin >> numcontainer;
        for_each(vecNumbers.begin(), vecNumbers.end(),
            [=](int & element) {
                element += numcontainer;
            });
        DisplayContents(vecNumbers);
        return 0;
    }

```

Результат

```

25 -5 122 2011 -10001
-10001 -5 25 122 2011
2011 122 25 -5 -10001
Число, добавляемое ко всем элементам: 5
2016 127 30 0 -9996

```

Ответы к занятию 23

Контрольные вопросы

1. Используйте функцию `std::list::remove_if()`, поскольку она гарантирует, что существующие итераторы на элементы в списке (которые не были удалены) останутся действительными.
2. Функция `list::sort()` (и даже `std::sort()`) в отсутствие явно заданного предиката прибегает к сортировке с использованием предиката `std::less<>`, который, в свою очередь, использует для сортировки объектов коллекции оператор `operator<`.
3. По одному разу для каждого элемента в диапазоне.
4. Функция `for_each()` работает с унарным предикатом и возвращает функциональный объект, который может содержать информацию состояния. Функция `transform()` может работать с унарным или бинарным предикатом и предоставляет перегруженную версию, которая в состоянии работать с двумя диапазонами.

Упражнения

1. Одно из возможных решений приведено ниже.

```
struct CaseInsensitiveCompare {
    bool operator()(const string & str1, const string & str2) const
    {
        string str1Copy(str1), str2Copy(str2);

        transform(str1Copy.begin(),
            str1Copy.end(), str1Copy.begin(), tolower);
        transform(str2Copy.begin(),
            str2Copy.end(), str2Copy.begin(), tolower);

        return (str1Copy < str2Copy);
    }
};
```

2. Вот пример такой демонстрации. Обратите внимание, что функция `std::copy()` работает, не зная характера коллекций. Она использует только классы итератора:

```
#include <vector>
#include <algorithm>
#include <list>
#include <string>
#include <iostream>

using namespace std;

int main() {
    list<string> listNames;
    listNames.push_back("Jack");
    listNames.push_back("John");
    listNames.push_back("Anna");
    listNames.push_back("Skate");
    vector<string> vecNames(4);
    copy(listNames.begin(), listNames.end(), vecNames.begin());
    vector<string> ::const_iterator iNames;

    for(iNames = vecNames.begin();
        iNames != vecNames.end(); ++iNames)
        cout << * iNames << ' ';

    return 0;
}
```

3. Различие между функциями `std::sort()` и `std::stable_sort()` в том, что последняя при сортировке сохраняет относительные положения одинаковых объектов. Поскольку приложение должно хранить данные в порядке событий, следует выбрать функцию `stable_sort()`, чтобы сохранить этот порядок.

Ответы к занятию 24

Контрольные вопросы

1. Да, предоставив соответствующий бинарный предикат.
2. Класс `Coin` должен реализовать оператор `operator<`.
3. Нет, воздействовать можно только на вершину стека. Поэтому вы не можете обратиться к элементу внизу стека.

Упражнения

1. Бинарный предикат может быть оператором `operator<`:

```
class Person {
public:
    int age;
    bool isFemale;

    bool operator<(const Person & anotherPerson) const {
        bool bRet = false;

        if (age > anotherPerson.age)
            bRet = true;
        else if (isFemale && anotherPerson.isFemale)
            bRet = true;

        return bRet;
    }
};
```

2. Просто поместите их в стек. При извлечении данных вы фактически меняете порядок содержимого на обратный, поскольку стек — это контейнер типа LIFO.

Ответы к занятию 25

Контрольные вопросы

1. Нет. Количество битов, которые может хранить множество битов, фиксируется во время компиляции.
2. Поскольку он им не является. Класс `bitset` не может менять свой размер динамически, как другие контейнеры; он не поддерживает итераторы, как контейнеры, и не нуждается в них.
3. Нет. Для этого лучше подходит класс `std::bitset`.

Упражнения

1. Вот пример кода, в котором объект класса `std::bitset` создается, инициализируется, отображается и добавляется:

```
#include <bitset>
#include <iostream>

int main() {
    // Инициализация значением 1001
    std::bitset<4> fourBits(9);
    std::cout << "fourBits: " << fourBits << std::endl;
    // Инициализация другого множества значением 0010
    std::bitset<4> fourMoreBits(2);
    std::cout << "fourMoreBits: " << fourMoreBits << std::endl;
    std::bitset<4> addResult(fourBits.to_ulong() +
                             fourMoreBits.to_ulong());
    std::cout << "Результат сложения равен " << addResult;
    return 0;
}
```

2. Вызовите функцию `flip()` для любого из объектов класса `bitset` в приведенном выше примере:

```
addResult.flip();
std::cout << "Результат применения flip(): "
          << addResult << std::endl;
```

Ответы к занятию 26

Контрольные вопросы

1. Лично я искал бы на www.boost.org. Надеюсь, вы тоже!
2. Нет. Как правило, хорошо разработанный (и правильно выбранный) интеллектуальный указатель не замедляет приложение.
3. При внедрении его содержат принадлежащие объекты; в противном случае эту информацию может содержать совместно используемый объект в динамической памяти.
4. Список следует перебирать в обоих направлениях, поэтому он должен быть двунаправленным.

Упражнения

1. Ошибка в строке `object->DoSomething()`; так как указатель потерял владение объектом во время предыдущего копирования. Эта строка приведет к сбою (или к какой-то иной неприятности).

2. Код может выглядеть следующим образом:

```
#include <memory>
#include <iostream>
using namespace std;

class Fish {
public:
    Fish() {
        cout << "Конструктор Fish" << endl;
    }
    ~Fish() {
        cout << "Деструктор Fish" << endl;
    }

    void Swim() const {cout << "Fish плавает в воде" << endl;}
};

class Carp: public Fish {
};

void MakeFishSwim(const unique_ptr<Fish> & inFish) {
    inFish->Swim();
}

int main() {
    unique_ptr<Fish> myCarp(new Carp);
    MakeFishSwim(myCarp);
    return 0;
}
```

Поскольку здесь нет никакого копирования, при условии, что функция `MakeFishSwim()` получает аргумент как ссылку, никакой срезки не происходит. Кроме того, обратите внимание на синтаксис создания экземпляра переменной `myCarp`.

3. Класс `unique_ptr` не допускает копирования и присваивания, поскольку и копирующий конструктор, и копирующий оператор присваивания являются закрытыми.

Ответы к занятию 27

Контрольные вопросы

1. Для только записи в файл используйте класс `ofstream`.
2. Используйте метод `cin.getline()`, как в листинге 27.7.
3. Нет, поскольку класс `std::string` содержит текстовую информацию, вы можете остаться в режиме по умолчанию, которым является текстовый режим (переход в бинарный режим не нужен).
4. Чтобы проверить успех выполнения метода `open()`.

Упражнения

1. Вы открыли файл, но не проверили успех этой операции с помощью метода `is_open()`, прежде чем использовать поток или закрыть его.
2. Вы не можете писать в поток `ifstream`, который предназначен только для чтения, но не для записи, и следовательно, не поддерживает оператор вывода в поток `<<`.

Ответы к занятию 28

Контрольные вопросы

1. Класс такой же, как и любой другой, но созданный специально как базовый для других классов исключений, таких как `bad_alloc`.
2. Исключение `std::bad_alloc`.
3. Нет, это плохая идея, так как возможна повторная генерация исключения из-за нехватки памяти.
4. С помощью того же обработчика `catch (std::exception&exp)`, что и для класса `bad_alloc`.

Упражнения

1. Никогда не генерируйте исключения в деструкторах.
2. Вы забыли обработать исключения (пропустили блок `try...catch`).
3. Не выделяйте память в блоке `catch`! Тем более такое большое ее количество. Если учесть, что вам уже не удалось выделить память в блоке `try`, продолжать попытки выделения не имеет никакого смысла.

Ответы к занятию 29

Контрольные вопросы

1. Похоже, ваше приложение выполняет все действия в пределах одного потока. Если обработка самого изображения (исправление контраста) интенсивно действует процессор, пользовательский интерфейс бездействует. Необходимо разделить эти два действия на два потока, чтобы операционная система выполняла их параллельно, деля процессорное время между потоком пользовательского интерфейса и рабочим потоком, вносящим исправления.
2. Возможно, ваши потоки плохо синхронизированы. Вы одновременно выполняете и запись в объект, и чтение из него, что приводит к несогласованным или искаженным возвращаемым данным. Используйте бинарный семафор — он гарантирует, что во время внесения изменений в таблицу ее будет невозможно читать.

Предметный указатель

- A**
 - ASCII 65
 - assert 403
 - auto 38, 72
- B**
 - bitset 120
- C**
 - C++
 - преимущества 32
 - C++11 25, 38, 71, 74, 151, 186, 275, 450, 639, 680
 - C++14 33, 38, 69, 75, 418, 453, 556, 617
 - C++17 39, 142, 275, 684
 - class 230
 - const 75, 368
 - constexpr 74, 76, 246, 278
- D**
 - dynamic_cast 327
- E**
 - exception 671
 - explicit 256, 264, 350
- F**
 - final 311, 336
 - friend 270
- L**
 - L-значение 107
- M**
 - multiset 496
 - mutable 560
 - mutex 682
- N**
 - nothrow 217
- O**
 - operator 344
 - override 335
- P**
 - POD 69
 - private 234, 236
 - public 234
- R**
 - R-значение 107
 - RAM 56
 - return 168, 175
 - RTTI 327, 386
- S**
 - set 496
 - sizeof 69
 - sizeof... 418
 - static 260
 - static_assert 420
 - STL 421, 427, 439
 - string 441
 - struct 269
- T**
 - template 406
 - this 266
 - thread 680
 - throw 668
 - typedef 73
 - typename 406
- U**
 - union 272
 - unique_ptr 637
- V**
 - vector 95
 - virtual 318, 334
- A**
 - Абстракция 237, 283
 - Агрегация 308
 - Адаптер 599
 - priority_queue 431
 - queue 431
 - stack 431
 - контейнера 428, 431
 - Алгоритм
 - adjacent_find 569
 - binary_search 571, 590
 - copy 569, 585
 - copy_backward 569, 586
 - copy_if 585
 - count 568, 573
 - count_if 568, 573
 - equal 569
 - fill 569, 577
 - fill_n 569, 577
 - find 432, 568, 571
 - find_end 569
 - find_first_of 569
 - find_if 432, 568, 571
 - for_each 540, 569, 581
 - generate 569, 579
 - generate_n 569, 579
 - lexicographical_compare 569
 - lower_bound 571, 594
 - mismatch 569
 - partial_sort 570
 - partial_sort_copy 570
 - partition 570, 592
 - random_shuffle 589
 - remove 570, 586
 - remove_copy 570
 - remove_copy_if 570
 - remove_if 432, 570, 586
 - replace 570, 588
 - replace_if 570, 588
 - reverse 432, 450
 - search 568, 575
 - search_n 568, 575
 - sort 549, 570, 590
 - stable_partition 570, 593
 - stable_sort 549, 570, 592
 - transform 432, 451, 549, 569, 583
 - unique 549, 570, 590
 - unique_copy 570
 - upper_bound 571, 594
 - изменяющий 569
 - не изменяющий 568
- Аргумент 43
- Б**
 - Базовый класс 284
 - Байт 695
 - Безопасность типов 383
 - Бинарный
 - оператор 353
 - предикат 487
 - Битовое множество 120
 - Бит 695
 - Блок 107

- В**
 Вектор 428, 458
 вставка 460, 461
 доступ 464
 емкость 468
 инициализация списком 461
 очистка 472
 размер 468
 создание экземпляра 458
 удаление 466
 Венгерская нотация 62
 Взаимоблокировка 682
 Виртуальная функция 318
 Виртуальное наследование 332
 Выполнимый файл 33
- Г**
 Глубокое копирование 631
- Д**
 Двухсторонняя очередь 470
 Декремент 109
 Дек 470
 очистка 472
 Деструктор 246
 базового класса 323
 виртуальный 320
 порядок вызовов 300
 Директива препроцессора 42, 396
 #define 80, 396
 #endif 399
 #ifndef 399
 #include 42, 399
 Друг 270
- Е**
 Емкость 468
- З**
 Закрытое наследование 303
 Защищенное наследование 305
 Знак 65
- И**
 Идентификация типа времени выполнения 327, 386
 Инициализация 57
 агрегатная 275
 списком 71
 цикла for 151
 Инкапсуляция 231, 283
- Инкремент 109
 Инструкция 106
 do...while 146
 for 148
 goto 143
 if 130
 вложенная 134
 switch 138
 while 145
 итеративная 146
 составная 107, 133
 Интегрированная среда разработки 34
 Интеллектуальный указатель 628
 Исключение 664
 Итератор 150, 431
 ввода 431
 вывода 431
 двунаправленный 432
 однонаправленный 431
 произвольного доступа 432
 Итерация 146
- К**
 Класс 230
 basic_string 453
 bitset 616
 deque 470
 exception 671
 forward_list 475
 fstream 652
 initialize_list 461
 istream 643
 list 475
 map 514
 multimap 514
 multiset 496
 mutex 682
 ostream 643
 priority_queue 608
 queue 605
 set 496
 stack 601
 string 439, 440
 stringstream 658
 thread 680
 tuple 418
 unique_ptr 637
 unordered_map 529
 unordered_multimap 530
 unordered_multiset 508
 unordered_set 508
 vector 458
 размер 468
 вставка 460, 461
 удаление 466
 доступ 464
 емкость 468
 инициализация списком 461
 очистка 472
 создание экземпляра 458
 vector<bool> 621
 wstring 440, 453
 абстрактный 328
 базовый 284
 друг 270
 некопируемый 258, 366
 несоздаваемый в стеке 262
 производный 284
 член 231
 шаблонный 409
 специализация 413
 Ключевое слово 80
 auto 72, 435, 450
 class 230
 const 75, 368
 constexpr 76, 246
 enum 78
 explicit 350
 final 311, 336
 friend 270
 inline 183, 184
 mutable 560
 operator 344
 override 335
 private 234, 303
 protected 288, 305
 public 234
 static 260
 struct 269
 template 406
 this 266
 throw 668
 typedef 73
 typename 406
 union 272
 virtual 318, 334
 Комментарий 46
 многострочный 52
 однострочный 52
 Компилятор 34
 Компиляция 34
 Композиция 308
 Компоновщик 34

Константа 55, 74
 литеральная 75, 376
 Конструктор 237
 копирующий 252
 перемещающий 258, 371
 порядок вызовов 300
 по умолчанию 240, 244
 преобразующий 264
 Контейнер 428
 deque 428
 forward_list 428
 list 428
 map 429
 multimap 429
 multiset 429
 set 429
 unordered_map 429
 unordered_multimap 429
 unordered_multiset 429
 unordered_set 429
 vector 428
 адаптер 428, 431
 адаптивный 599
 ассоциативный 429
 выбор 435
 последовательный 428
 Копирование
 глубокое 252, 255, 363, 631
 деструктивное 634
 поверхностное 250
 при записи 633
 Кортеж 418

Л

Лямбда-выражение 38, 186,
 553, 554
 mutable 560
 синтаксис 560
 список захвата 558

М

Макрос 396
 assert 403
 Макрофункция 400
 Массив 86
 динамический 87, 95
 индекс 89
 и указатель 209
 массивов 94
 многомерный 93
 символьный 97
 статический 86, 87
 Метод 231
 перекрытие 293

сокрытие в производном
 классе 298
 Многопоточность 679
 Множественное
 наследование 286, 309
 Множество 496
 битов 616
 Модификатор доступа 288, 305
 private 303
 protected 289, 305
 Мультимножество 496
 Мультиотображение 514
 Мьютекс 682

Н

Наследование 283
 виртуальное 332
 закрытое 303
 защищенное 305
 множественное 286, 309
 открытое 284

О

Область видимости 59
 Объединение 272
 Объектный файл 34
 Объект 231
 функциональный 370, 528, 537
 Объявление 48, 167
 using 45
 функции 48
 Оперативная память 56
 Оператор 107, 344
 % 108
 & 119, 193
 && 114
 () 369
 * 108, 197, 351
 + 108, 354
 ++ 109
 += 357
 - 108, 354
 -- 109
 -= 357
 . 232
 ! 114
 != 111, 359
 < 112
 << 43, 121, 642
 <= 112
 = 107
 == 111, 359

>= 112
 >> 50, 121, 642
 > 112, 232, 351
 ? 141
 | 119
 || 115
 / 108
 \ 238, 295, 376
 [] 366
 ^ 115, 119
 ~ 119
 break 138, 153
 const_cast 389
 continue 153
 delete 202
 dynamic_cast 385
 new 201, 215
 исключение 215
 reinterpret_cast 388
 return 168, 175
 sizeof 69, 124, 199, 267
 sizeof... 418
 static_cast 384
 throw 668
 бинарный 353
 больше или равно 112
 больше 112
 выбора поля 232
 выбора члена 351
 вывода в поток 43, 642
 вычитания
 с присваиванием 357
 вычитания 108
 декремента 109
 деления по модулю 108
 деления 108
 извлечения из потока 642
 индексации 366
 инкремента 109
 копирующего
 присваивания 363
 косвенного обращения 198
 меньше или равно 112
 меньше 112
 неперегружаемый 378
 неравенства 111, 359
 получения адреса 194
 постфиксный 109
 преобразования 348
 префиксный 109
 приведения 381
 приоритет 126, 702

присваивания 107
 перемещающий 371
 составной 122
 присваивания 256
 равенства 111, 359
 разрешения области
 видимости 238, 295
 разыменования 197, 351
 сдвига 121
 сложения с
 присваиванием 357
 сравнения 111
 суммирования 108
 тернарный условный 141
 точки 232
 указателя 232
 умножения 108
 унарный 345
 функции 369
 Определение 167
 функции 48
 Оптимизация 184
 Открытое наследование 284
 Отладка 34
 Отношение
 содержит 303
 является 284, 303
 Отображение 514
 Очередь 600
 с приоритетами 608
 Ошибка времени
 выполнения 39

П

Память
 динамическая 201, 204
 оперативная 56
 утечка 202, 212
 Переменная 55
 глобальная 61, 62
 инициализация 57
 локальная 60
 область видимости 59
 состояния 558
 Перемещающий
 конструктор 371
 Переполнение буфера 101
 Переполнение стека 173
 Перечисление 78
 Полиморфизм 283, 316
 подтипов 316
 реализация 324, 326

Пользовательский литерал 376
 Поток 43, 642
 cin 648
 cout 644
 выполнения 679
 синхронизация 681
 манипулятор 643
 файловый 652
 режим открытия 653
 Предикат 369, 538
 бинарный 487, 538, 563
 унарный 543, 557
 Препроцессор 42, 396
 директива 42, 396
 Приведение
 восходящее 384
 нисходящее 384
 Приоритет операторов 126
 Проблема ромба 334
 Производный класс 284
 Пространство имен 43, 44
 std 45
 Простые старые данные 69
 Процедурное
 программирование 229

Р

Размер 69, 468
 Рекурсия 173

С

Сборка мусора 212
 Связанный список 429, 476
 Семафор 682
 Символ завершающий
 нулевой 97
 Синглтон 259, 262
 Сложность 430
 Состояние гонки 682
 Специализация 288
 Список 429, 476
 двухсвязный 475
 захвата 558
 инициализации 244
 односвязный 475
 Срезка 309, 631
 Ссылка 218
 константная 221
 Стандартная библиотека
 шаблонов 421, 427, 439
 Стек 182, 600

Строка 440
 string 441
 wstring 453
 в стиле C 97
 конкатенация 445
 Строковый литерал 44, 74
 Структура 269
 Счетчик ссылок 633

Т

Таблица виртуальных
 функций 325
 Тернарный условный
 оператор 141

Тип

bool 64
 char 64
 double 69
 float 69
 int 66
 long 66
 long long 66
 short 66
 string 100
 unsigned int 66
 unsigned long 66
 unsigned long long 66
 unsigned short 66
 void 168
 безопасность 383
 преобразование 264
 приведение 381
 фундаментальный 63

У

Узел 476
 Указатель 192
 this 266
 арифметика 204
 висящий 214
 и массив 209
 интеллектуальный 351, 628
 со списком ссылок 634
 передача в функцию 208
 Унарный оператор 345
 Утечка памяти 202, 212

Ф

Файл
 выполнимый 33
 заголовочный 399
 объектный 34

Функтор 370, 528, 537

адаптивный 538

Функция 47, 165

main() 43

аргумент 43, 168

бинарная 538

виртуальная 318

таблица 325

возвращаемое значение 44

встраиваемая 183

вызов 167, 168

лямбда. См. Лямбда-

выражение

объявление 167

определение 167, 168

параметр 168

массив 178

входной 222

необязательный 172

передача по ссылке 180, 221

по умолчанию 171

перегрузка 177

передача указателя 208

прототип 167

реализация 168

рекурсивная 173

унарная 538

чисто виртуальная 328

шаблонная 407

Х

Хеш-таблица 528

Хеширование 507, 528

коллизия 529

Ц

Циклическая зависимость 634

Цикл 142

do...while 146

for 148

для диапазона 151

while 145

бесконечный 154

вложенный 157

Ш

Шаблон 405

вариативный 417

инстанцирование 410

параметр типа 406

по умолчанию 411

с переменным числом

параметров 417

специализация 413

Я

Язык программирования

С 32

С# 212

С++. См. С++

Java 212

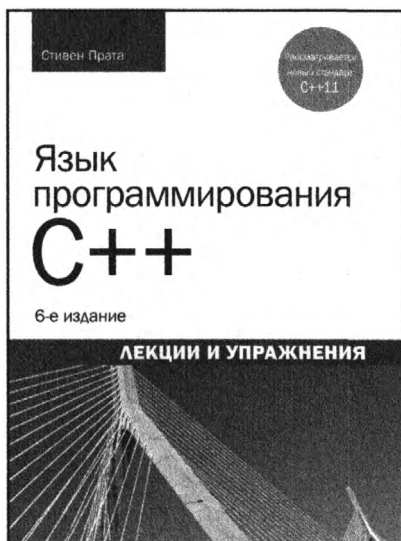
интерпретируемый 39

ЯЗЫК ПРОГРАММИРОВАНИЯ C++

ЛЕКЦИИ И УПРАЖНЕНИЯ

6-Е ИЗДАНИЕ

Стивен Прата



www.williamspublishing.com

Книга представляет собой тщательно проверенный, качественно составленный полноценный учебник по одной из ключевых тем для программистов и разработчиков. Эта классическая работа по вычислительной технике обучает принципам программирования, среди которых структурированный код и нисходящее проектирование, а также использованию классов, наследования, шаблонов, исключений, лямбда-выражений, интеллектуальных указателей и семантики переноса. Автор и преподаватель Стивен Прата создал поучительное, ясное и строгое введение в C++. Фундаментальные концепции программирования излагаются вместе с подробными сведениями о языке C++. Множество коротких практических примеров иллюстрируют одну или две концепции за раз, стимулируя читателей осваивать новые темы за счет непосредственной их проверки на практике.

ISBN 978-5-8459-2048-5

в продаже

Освой самостоятельно

C++

по одному часу в день

Перед вами новое 8-е издание ставшей уже популярной книги *Освой самостоятельно C++ за 21 день!*

Выделив всего один час на урок вы можете приобрести квалификацию, необходимую для начала программирования на языке C++. В книге представлен полный курс обучения программированию, который позволит быстро овладеть основами языка и перейти к более сложным понятиям и концепциям.

Эта книга, полностью переработанная с учетом стандарта C++14 и готовящегося стандарта C++17, представляет язык C++ с практической точки зрения — как средство создания быстрых, простых и эффективных приложений на C++.

Особенности книги

- Изучение фундаментальных принципов языка C++ и объектно-ориентированного программирования
- Овладение возможностями языка C++, помогающими писать компактный и эффективный код с помощью таких концепций, как лямбда-выражения, конструкторы перемещения и операторы присваивания
- Полезные советы и рекомендации, позволяющие избежать проблем
- Изучение стандартной библиотеки шаблонов, включая контейнеры и алгоритмы, используемые в большинстве реальных приложений C++
- Проверка знаний и опыта с использованием упражнений в конце каждого занятия

Сиддхартха РАО является вице-президентом по вопросам безопасности в SAP SE — ведущем мировом поставщике корпоративного программного обеспечения. По мере развития языка программирования C++ он постоянно убеждается в том, что с помощью C++ можно создавать более мощные приложения быстрее и проще, чем когда-либо прежде.

Категория: программирование

Содержание: C++11, C++14, C++17

Уровень: для начинающих и пользователей средней квалификации



www.williams



informit.com/s



кодом
ачать
распо-
изда-

ublishing.com/
45-6-5.html

УЧИТЕСЬ, КОГДА ВАМ УДОБНО, В СОБСТВЕННОМ ТЕМПЕ

- Опыт программирования необязателен
- Пишите быстрые и мощные программы на C++, компилируйте код и создавайте выполнимые файлы
- Изучите концепции объектно-ориентированного программирования, такие как инкапсуляция, абстракция, наследование и полиморфизм
- Используйте алгоритмы и контейнеры стандартной библиотеки шаблонов для написания многофункциональных надежных приложений на C++
- Изучите, как автоматический вывод типов помогает упрощать исходные тексты на языке C++
- Разрабатывайте сложные программные решения, используя лямбда-выражения, интеллектуальные указатели и конструкторы перемещения
- Овладейте средствами C++, используя опыт ведущих экспертов по программированию на языке C++
- Изучите возможности C++, позволяющие создавать компактные и высокопроизводительные приложения на C++
- Узнайте, что нового ожидается в стандарте C++17

ISBN 978-5-9909445-6-5



9 785990 944565