Сиддхартха Рао

СЕДЬМОЕ ИЗДАНИЕ

ОПИСАН C++11

Освой самостоятельно C++

за 21 день

SAMS

Sams Teach Yourself C++ in One Hour a Day

SEVENTH EDITION

Siddhartha Rao

SAMS

800 East 96th Street, Indianapolis, Indiana 46240

Освой самостоятельно C++ за 21 день

СЕДЬМОЕ ИЗДАНИЕ

Освой самостоятельно C++ за 21 день

СЕДЬМОЕ ИЗДАНИЕ

Сиддхартха Рао

Москва · Санкт-Петербург · Киев
2013

Оглавление

Содержание

Введение

2011 год был особенным для языка C++. Ратификация нового стандарта C++11 позволяет писать лучший код с использованием новых ключевых слов и конструкций, улучшающих эффективность программирования. Эта книга поможет изучить язык C++11 маленькими шагами. Она специально разделена на отдельные занятия, излагающие основные принципы этого языка объектно-ориентированного программирования с практической точки зрения. Вы сможете овладеть языком C++11, занимаясь всего по одному часу.

Приведенные здесь фрагменты кода были проверены с использованием последних версий доступных компиляторов, а именно компилятора Microsoft Visual C++ 2010 для C++ и компилятора C++ GNU версии 4.6, которые поддерживают новые средства языка C++11.

Для кого написана эта книга

Книга начинается с самых основ языка C++. Необходимо 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. К этой же книге имеет смысл обратиться, если вы уже знаете язык C++, но хотите изучить добавления, которые были внесены в язык C++11. Если вы профессиональный программист, то третья часть книги поможет узнать, как лучше на практике создавать приложения C++11.

Структура книги

В зависимости от текущего уровня квалификации вы можете начать изучение с любого раздела. Книга состоит из пяти частей.

- Часть I, "Основы", 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, "Фундаментальные принципы объектно-ориентированного программирования 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 новые концепции C++11, включая конструктор перемещения, а занятие 12, "Типы операторов и их перегрузка", — оператор присваивания при перемещении. Эти эффективные средства помогают сократить ненужные и нежелательные этапы копирования, увеличивая производительность приложения. Занятие 14, "Макросы и шаблоны", является краеугольным камнем для написания мощного обобщенного кода на C++.

- Часть III, "Знакомство со стандартной библиотекой шаблонов (STL)", поможет писать эффективный код C++, использующий класс STL `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, "Подробней о библиотеке 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 размера кода при использовании алгоритмов библиотеки STL.
- Часть V, "Передовые концепции языка 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.

Соглашения, принятые в книге

Здесь используются соглашения, общепринятые в компьютерной литературе.

- Новые термины в тексте выделяются *курсивом*. Чтобы обратить внимание читателя на отдельные участки текста, также применяется *курсив*.
- Текст программ, функций, переменных, URL веб-страниц и другой код представлен моноширинным шрифтом.
- Все, что придется вводить с клавиатуры, выделено **полужирным моноширинным** 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: `BINDSIZE=` (*максимальная ширина колонки*) * (*номер колонки*).
- Пункты меню и названия диалоговых окон представлены следующим образом: Menu Option (Пункт меню).
- В листингах каждая строка имеет номер. Это сделано исключительно для удобства описания. В реальном коде нумерация отсутствует. Текст некоторых строк кода в листингах иногда бывает слишком длинным и не помещается на странице книги. Таким образом, отсутствие номера в начале строки свидетельствует о том, что строка является продолжением предыдущей и в реальном коде их разрывать не следует.

Текст некоторых абзацев выделен специальным шрифтом. Это примечания, советы и предостережения, которые помогут обратить внимание на наиболее важные моменты в изложении материала и избежать ошибок в работе.

ПРИМЕЧАНИЕ Примечания содержат дополнительную информацию, связанную с излагаемым материалом.

C++11

Здесь приведены новые возможности языка C++11. Вполне вероятно, что для использования этих возможностей языка придется использовать более новые версии компиляторов.

ВНИМАНИЕ! Предостережения обращают ваше внимание на побочные эффекты или проблемы, которые могут возникнуть в определенных ситуациях.

СОВЕТ Здесь содержатся полезные советы по написанию программ на языке C++.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте эти рекомендации для поиска наболее эффективного решения поставленных задач	**Не пропускайте** важные замечания и предупреждения, показанные в этом столбце

Примеры кода

Примеры кода, приведенные в этой книге, доступны на веб-сайте издательства Sams.

От издательства

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 интересно услышать и любые другие замечания, которые вам хотелось бы высказать авторам.

Мы ждем ваших комментариев. Вы можете прислать письмо по электронной почте или просто посетить наш веб-сервер, оставив на нем свои замечания. 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 подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш e-mail. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию следующих книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, Москва, ул. Лесная, д. 43, стр 1
в Украине: 03150, Киев, а/я 152

ЧАСТЬ I
Основы

ЗАНЯТИЕ 1
Первые шаги

Добро пожаловать на страницы седьмого издания книги *Освой самостоятельно C++ за 21 день*! Сегодня начинается долгий путь, который позволит вам достичь уровня профессионального программирования на языке C++.

На сегодняшнем занятии.

- Почему язык C++ стал стандартом в области разработки программного продукта.
- Как набрать, откомпилировать и скомпоновать первую рабочую программу C++.
- Что нового в C++11.

Краткий экскурс в историю языка C++

Задача языка программирования в том, чтобы упростить использование вычислительных ресурсов. Язык C++ отнюдь не нов, но он еще весьма популярен и продолжает совершенствоваться. Последняя версия языка C++, принятая международным комитетом по стандартам ISO Standards Committee в 2011 называется C++11.

Связь с языком C

Первоначально разработанный Бьярне Страуструпом в 1979 году, язык C++ был задуман как преемник языка C. C — процедурный язык, где каждая функция предпринимает определенное действие. Язык C++, напротив, был задуман как объектно-ориентированный, но он реализует такие концепции, как наследование, абстракция, полиморфизм и инкапсуляция. Классы языка C++ используют свойства для содержания данных и методы для обработки этих данных. (Методы являются аналогом функций в языке C.) В результате программист больше думает о данных и о том, что с ними следует сделать. Компиляторы C++ традиционно поддержали также программы на языке C. Преимуществом этого была совместимость с устаревшим кодом, а недостатком — чрезвычайно высокая сложность компиляторов, вынужденных обеспечивать программистам эту совместимость, одновременно реализуя все новые средства, которых требует развитие языка.

Преимущества языка C++

C++ считается языком программирования промежуточного уровня, т.е. 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, сохраняя разработчику возможность обеспечить им максимальную производительность за счет подробного 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 определенные средства, такие как управление памятью за счет сбора "мусора", реализованное в компоненте исполняющей среды, которые нравятся некоторым программистам. Однако там, где нужен подробный контроль за производительностью создаваемого приложения, эти же программисты все еще выбрали бы язык C++. Многоуровневая архитектура, где веб-сервер создается на языке C++, а пользовательская часть приложения на HTML, Java или .NET, является в настоящее время вполне обыденной.

Развитие стандарта C++

Годы развития сделали язык C++ весьма популярным, хоть и существующем во многих разных формах из-за множества различий компиляторов, каждый из которых имеет собственные особенности. Эта популярность и различия в доступных версиях привели к большому количеству проблем совместимости и переносимости кода. Следовательно, появилась потребность стандартизировать все это.

В 1998 году первая стандартная версия языка C++ была ратифицирована международной организацией по стандартизации ISO Committee в виде стандарта ISO/IEC 14882:1998. Затем, в 2003 году, последовала версия ISO/IEC 14882:200). Текущая версия стандарта языка C++ была ратифицирована в августе 2011 года. Официально она называется C++11 (ISO/IEC 14882:2011) и содержит некоторые из самых честолюбивых и прогрессивных изменений, которые когда-либо видел стандарт.

ПРИМЕЧАНИЕ

Многие документы в Интернете все еще ссылаются на версию языка C++0x. Ожидалось, что новый стандарт будет ратифицирован в 2008 или 2009 году, а знак *x* использовался как обозначение года. И наконец, в августе 2011 года был предложен и принят новый стандарт и соответственно назван C++11.

Другими словами, C++11 является новым стандартом C++ 0x.

Кто использует программы, написанные на C++?

Независимо от того, кто вы или что вы делаете — закаленный программист или тот, кто использует компьютер для определенных целей, — вы постоянно используете приложения и библиотеки C++. Это операционные системы, драйверы устройств, офисные приложения, веб-серверы, приложения на базе облака (сетевой среды), поисковые механизмы и даже некоторые из более новых языков программирования, для создания которых, как правило, выбирают язык C++.

Создание приложения C++

Когда вы запускаете Блокнот (Notepad) или редактор `vi` на своем компьютере, вы фактически указываете процессору запустить исполняемый файл этой программы. *Исполняемый файл* (executable) — это готовый продукт, который может быть запущен и должен сделать то, чего намеревался достичь программист.

Этапы создания исполняемого файла

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

1. Написать (или запрограммировать) код C++, используя текстовый редактор.
2. Откомпилировать код, используя компилятор C++, который преобразовывает его в версию машинного языка и запишет в *объектный файл* (object file).
3. Скомпоновать результат работы компилятора, используя компоновщик, и получить исполняемый файл (`.exe` в Windows, например).

Обратите внимание, что микропроцессор не может использовать текстовые файлы, которые вы, по существу, и создаете, когда пишете программы. *Компиляция* (compilation) — этап, на котором код C++, содержащийся обычно в текстовых файлах с расширением `.cpp`, преобразуется в бинарный код, который может быть понят процессором. *Компилятор* (compiler) преобразует по одному файлу кода за раз, создавая объектный

файл с расширением `.o` или `.obj` и игнорируя зависимости, которые код в этом файле может иметь с кодом в другом файле. Распознавание этих зависимостей и объединение кода является задачей *компоновщика* (linker). Кроме объединения различных объектных файлов, он устанавливает зависимости и в случае успешной компоновки создает исполняемый файл, который можно выполнять и в конечном счете распространять.

Анализ и устранение ошибок

Большинство сложных приложений, особенно разработанных коллективом программистов, редко компилируются и начинают хорошо работать сразу. Большое или сложное приложение, написанное на любом языке (включая C++), зачастую требует множества запусков и перезапусков, чтобы проанализировать проблемы и обнаружить ошибки. Затем ошибки исправляются, программа перекомпилируется и процесс продолжается. Таким образом, кроме трех этапов (программирование, компиляция и компоновка), разработка зачастую подразумевает этап *отладки* (debugging), на котором программист анализирует аномалии и ошибки в приложении, используя инструментальные средства просмотра и отладчики, а также построчное выполнение.

Интегрированная среда 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

Ныне доступно множество бесплатных интегрированных сред разработки и компиляторов C++. Наиболее популярные — Microsoft Visual C++ 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++.

ВНИМАНИЕ!

Хотя на момент написания этой книги ни один из компиляторов не поддерживал стандарт C++11 полностью, большинство его основных средств вышеупомянутыми компиляторами уже поддерживалось.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте для создания исходного кода либо простой текстовый редактор, такой как Блокнот или gedit, либо интегрированную среду разработки	**Не используйте** сложные редакторы текста, поскольку они зачастую добавляют собственную разметку в текст кода
Сохраняйте свои файлы исходного кода в файлах с расширением `.cpp`	**Не используйте** расширение `.c`, поскольку большинство компиляторов рассматривают такие файлы как содержащие код языка C, а не C++

Создание первого приложения C++

Теперь, когда известны инструментальные средства и задействованные этапы, пришло время создать первое приложение C++, которое по традиции выводит на экран текст `Hello World!` ("Привет, мир!").

Если вы работаете под управлением операционной системы Windows и используете IDE Microsoft Visual C++ Express, то можете следовать этапам, описанным ниже.

1. Создайте новый проект, используя пункт меню File⇨New⇨Project (Файл⇨Новый⇨Проект).
2. Выберите тип Win32 Console Application (Консольное приложение Win32) и сбросьте флажок Use Precompiled Header (Использовать предварительно скомпилированный заголовок).
3. Назовите проект **`Hello`** и замените автоматически созданное содержимое в файле `Hello.cpp` фрагментом кода, представленным в листинге 1.1.

Если вы работаете на Linux, для создания файла `.cpp` с таким содержимым, как в листинге 1.1, используйте простой текстовый редактор (я на Ubuntu использовал gedit).

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 строку, а оператор `return 0` обеспечивает завершение работы приложения и возврат операционной системе кода 0.

ПРИМЕЧАНИЕ

Понять программу при чтении будет легче, если знаешь, как произносятся специальные символы и ключевые слова. Например, директиву `#include` можно прочитать как hash-include (хеш инклюд), или как sharp-include (шарп инклюд), или как pound-include (фунт инклюд), в зависимости от ваших привычек.

Аналогично оператор `std::cout` можно прочитать как standard-c-out (стандард-си-аут).

ВНИМАНИЕ!

Помните, дьявол – в деталях, значит, код необходимо вводить точно так же, как он представлен в листинге. Компиляторы известны своей придирчивостью. Если вы по ошибке поместите в конце оператора `:`, где ожидается `;`, то весь ад, безусловно, развернется!

Создание и запуск вашего первого приложения C++

Если вы используете IDE Microsoft Visual C++ Express, нажмите для запуска программы непосредственно в интегрированной среде разработки комбинацию клавиш <Ctrl+5>. В результате программа будет откомпилирована, скомпонована и запущена на выполнение. Эти же этапы можно также пройти индивидуально.

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

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

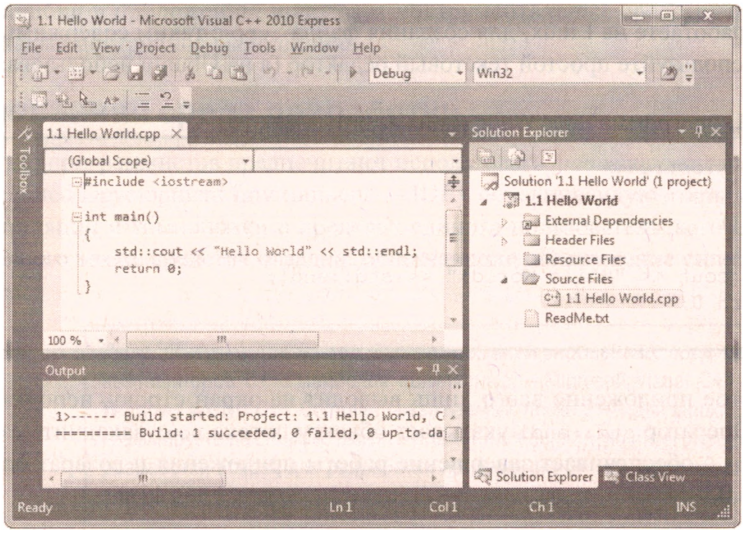


РИС. 1.1. Пример программы C++ "Hello World" в среде разработки Microsoft Visual C++ 2010 Express

При работе под Linux вызовите компилятор g++ и компоновщик, используя следующую командную строку:

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

Ввод этой строки приказывает компилятору g++ создавать исполняемый файл `hello` и откомпилировать файл исходного кода C++ `Hello.cpp`. Запуск файла `.\hello` на 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 соответствующего стандарту компилятора C++. Таким образом, если необходимо создать продукт, который способен выполняться как на операционной системе Windows, так и на 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, вероятней 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: оно указывает имя файла, в котором содержится ошибка, номер строки (в данном случае 6), где пропущена точка с запятой, и описание самой ошибки, предваряемое номером ошибки (в данном случае C2143). Хотя знак препинания был удален из пятой строки кода примера, в сообщении об ошибке упоминается следующая строка, поскольку для компилятора ошибка стала очевидной, только когда он проанализировал оператор `return` и понял, что предыдущий оператор должен был быть закончен перед переходом к оператору `return`. Можете попробовать добавить точку с запятой в начале шестой строки, и программа будет прекрасно откомпилирована!

ПРИМЕЧАНИЕ

Здесь конец строки не считается автоматически концом оператора, как в некоторых других языках, таких как VBScript.

В C++ оператор может распространяться на несколько строк кода.

Что нового в C++11

Если вы опытный программист C++, то, вероятно, уже обратили внимание на то, что в примере программы C++ из листинга 1.1 ни один бит не изменился. Хотя язык C++11 остается полностью совместимым с предыдущими версиями языка C++, на самом деле при создании языка было проделано очень много работы, чтобы упростить его использование.

Такое средство, как ключевое слово `auto`, позволяет определить переменную, тип которой компилятор выясняет автоматически, уплотняя многословные объявления (например,

итератора) и не нарушая безопасность типов. Лямбда-функции — это функции без имени. Они позволяют писать компактные объекты функций без длинных определений класса, значительно сокращая строки кода. Стандарт C++11 обещал программистам способность писать переносимые, многопоточные и все же соответствующие стандарту приложения C++. Эти приложения, при правильном построении, поддерживают парадигму параллельного исполнения 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++11, обсуждаемых в этой книге.

Резюме

На этом занятии вы узнали, как написать, откомпилировать, скомпоновать и выполнить вашу первую программу C++. Здесь приведен также краткий обзор развития языка C++ и продемонстрирована эффективность стандарта на примере того, как 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 улучшить свою программу соответственно. Не игнорируйте предупреждающие сообщения, если не уверены абсолютно, что они ошибочны.

- **Чем интерпретирующий язык отличается от компилирующего?**

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

- **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 исполняемый файл.

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 Г, "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?
4. Как стандарт C++11 улучшает поддержку многоядерных процессоров?

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

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

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

ЗАНЯТИЕ 2

Структура программы на C++

Программы C++ состоят из классов, 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 сегодняшнем занятии.

- Элементы программы C++.
- Взаимодействие элементов.
- 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 эта программа содержит некоторые из наиболее важных фундаментальных составляющих частей 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 программы C++

```
 1: // Директива препроцессора, подключающая заголовок 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:     // Возвращение 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 программы, которая начинается с `int main()`.

ПРИМЕЧАНИЕ

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

Более подробная информация о комментариях приведена в следующем разделе.

Директива препроцессора `#include`

Как и предлагает его название, *препроцессор* (preprocessor) — это инструмент, запускающийся перед фактическим началом компиляции. *Директивы препроцессора* (preprocessor directive) — это команды препроцессору, они всегда начинаются со знака фунта (#). В строке 2 листинга 2.1 директива `#include` *<имяфайла>* указывает препроцессору взять содержимое файла (в данном случае `iostream`) и включить его в строку, где расположена директива. `iostream` — это стандартный файл заголовка, 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 оператор `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 заголовки. Сложные приложения, как правило, имеют несколько исходных файлов, причем некоторым требуется включить другие. Так, если некий объект, объявленный в файле `FileA`, должен использоваться в файле `FileB`, то первый необходимо включить в последний. Обычно для этого в файл `FileA` помещают следующую директиву `#include`:

```
#include "...relative path to FileB\FileB"
```

В данном случае при включении самодельного заголовка мы используем кавычки, а не угловые скобки. Угловые скобки (<>), как правило, используются при включении стандартных заголовков.

Тело программы — функция `main()`

После директивы (директив) препроцессора следует тело программы, расположенное в функции `main()`. Исполнение программ C++ всегда начинается здесь. По стандартному соглашению, 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 также совместимо со стандартом и вполне приемлемо, поскольку функция `main()` возвращает тип `int`, а содержимое круглых скобок – это *аргументы* (argument), передаваемые программе. Эта программа, вероятно, позволяет пользователю запускать ее с аргументом командной строки, таким, например, как

```
program.exe /DoSomethingSpecific
```

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

Обсудим 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 оператором, фактически выводящем на экран строку `Hello World`. `cout` — это поток, определенный в стандартном пространстве имен (поэтому и `std::cout`), а то, что мы делаем, — так это помещаем текст строки `Hello World` в данный поток, используя оператор вывода в поток <<. Оператор `std::endl` используется для завершения строки, а ввод его в поток эквивалентен вставке знака абзаца. Обратите внимание: *оператор вывода в поток* (stream insertion operator) используется при необходимости вывода в поток каждого нового элемента.

Преимущество потоков C++ заключается в одинаковой семантике, используемой потоками разного типа. В результате различные операции, осуществляемые с тем же текстом, например вывод в файл, а не на консоль, выглядят одинаково. Таким образом, работа с потоками становится интуитивно понятной и, когда вы привыкаете к одному потоку

(такому как cout, выводящему текст на консоль), то без проблем можете работать с другими (таким как fstream, записывающим текстовые файлы на диск).

Более подробная информация о потоках приведена на занятии 27, "Применение потоков для ввода и вывода".

ПРИМЕЧАНИЕ

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

Возвращение значения

Функции в языке C++ должны возвратить значение, если противное не указано явным образом. main() — это функция, всегда и обязательно возвращающая целое число. Это значение возвращается операционной системе (OS) и, в зависимости от характера вашего приложения, может быть очень полезным, поскольку большинство операционных систем предусматривает возможность обратиться к возвращенному значению другим приложениям. 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 и Std::Cout вместо std::cout.

Концепция пространств имен

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

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

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 полезны пространства имен. Пространства имен — это имена, присвоенные частям кода, помогающие снизить вероятность конфликтов имен. При вызове оператора std::cout вы указываете компилятору использовать именно тот объект cout, который доступен в пространстве имен std.

ПРИМЕЧАНИЕ

Вы используете пространство имен `std` (произносится "standard" (стандарт)) для вызова функций, потоков и утилит, которые были утверждены комитетом по стандартам ISO Standards Committee, а следовательно, объявлены в его пределах.

Многие программисты находят утомительным регулярный ввод в коде спецификатора `std` при использовании оператора `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:     // Возвращение 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 явно в строке 11 при использовании операторов `td::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:     // Возвращение 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 операторами `using namespace std` и `using std::cout` в том, что первый позволяет использовать все элементы пространства имен `std`, без явного указания спецификатора пространства имен `std::`. Удобство последнего в том, что без необходимости устранять неоднозначность пространств имен явно можно использовать только операторы `std::cout` и `std::endl`.

Комментарии в коде C++

Строки 1, 4, 10 и 13 листинга 2.3 содержат текст на человеческом языке, но программа все равно компилируется. Они также не влияют на вывод программы. Такие строки называются *комментариями* (comment). Комментарии игнорируются компилятором и обычно используются программистами для объяснений 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

Могло бы 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.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Добавляйте комментарии, объясняющие работу сложных алгоритмов и частей вашей программы	**Не используйте** комментарии для повторения или объяснения очевидного
Оформляйте комментарии в стиле, принятом вашим коллективом программистов	**Не забывайте**, что добавление комментариев не сделает понятней запутанный код
	Не забывайте изменять комментарии при изменении кода

Функции в C++

Функции в языке C++ такие же, как и в языке C. *Функции* (function) — это элементы, позволяющие разделить содержимое вашего приложения на функциональные блоки, которые могут быть вызваны по вашему выбору. При вызове функция обычно возвращает значение вызывающей функции. Самая известная функция, конечно, — `main()`. Она

распознается компилятором как отправная точка приложения C++ и должна возвратить 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 оператор `std::cout` с различными параметрами.

ЛИСТИНГ 2.4. Объявление, определение и вызов функции, демонстрирующей некоторые возможности оператора `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 функции
16: int DemoConsoleOutput()
17: {
18:     cout << "This is a simple string literal" << endl;
19:     cout << "Writing number five: " << 5 << endl;
20:     cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
21:     cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
22:     cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
23:
24:     return 0;
25: }
```

Результат

```
This is a simple string literal
Writing number five: 5
Performing division 10 / 5 = 2
Pi when approximated is 22 / 7 = 3
Pi more accurately is 22 / 7 = 3.14286
```

Анализ

Интерес представляют строки 5, 10 и 15–25. 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, с учетом, что далее (в строках 15–25) следует *определение функции* (function definition), т.е. ее реализация.

Фактически эта функция отображает различные возможности оператора cout. Обратите внимание: она выводит не только текст, как в приложении Hello World в предыдущих примерах, но и результаты простых арифметических вычислений. Две строки, 21 и 22, отображают результат вычисления числа Пи (22 / 7), но последний точней просто потому, что при делении 22.0 на 7 вы указываете компилятору вычислить результат как вещественное число (тип float в терминах C++), а не как целое.

Обратите внимание, что функция предусмотрена как возвращающая целое число и возвращает она значение 0. Поскольку никаких решений эта функция не принимает, нет никакой необходимости возвращать какое либо другое значение. Точно так же функция main() возвращает значение 0. Поскольку функция main() делегирует все свои действия функции DemoConsoleOutput(), имело бы смысл использовать возвращаемое ею значение для возвращения 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 << "This is a simple string literal" << endl;
 8:     cout << "Writing number five: " << 5 << endl;
 9:     cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
10:     cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
11:     cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
12:
13:     return 0;
14: }
15:
16: int main()
17: {
18:     // Вызов функции с возвращением реультата при выходе
19:     return DemoConsoleOutput();
20: }
```

Анализ

Вывод этого приложения такой же, как у предыдущего. Небольшие изменения есть 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 одновременно объявление и определение функции. Функция main() также немного короче. В строке 19 осуществляется вызовов 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, вы можете 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 при помощи функций".

Простые операторы ввода `std::cin` и вывода `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 форме сложной графики, может быть напечатана принтером на бумаге или просто сохранена в файловой системе для последующего использования. В данном разделе рассматривается самая простая форма ввода и вывода в языке C++ — использование консоли для отображения и ввода информации.

Для записи простых текстовых данных на консоль используются оператор `std::cout` (произносится как standard see-out (стандарт си-аут)) и оператор `std::cin` (произносится как standard see-in (стандарт си-ин)) для чтения текста и чисел с консоли (как правило, с клавиатуры). Фактически при отображении слов `Hello World` на экране в листинге 2.1 вы уже встречались с оператором `cout`:

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

Здесь оператор `cout` сопровождается оператором вывода << (позволяющим вставить данные в поток вывода), который подлежит выводу строковым литералом `"Hello World"` и символом новой строки в форме оператора `std::endl` (произносится как standard end-line (стандарт енд-лайн)).

Применение оператора `cin` также очень просто, он сопровождается переменной, в которую следует поместить вводимые данные:

```
std::cin >> Переменная;
```

Таким образом, оператор `cin` сопровождается оператором извлечения значения >> (данные извлекаются из входного потока) и переменной, в которую следует поместить данные. Если вводимые данные, разделенные пробелом, следует сохранить в двух переменных, то можно использовать один оператор:

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

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

ЛИСТИНГ 2.6. Использование операторов `cin` и `cout` для отображения числовых и текстовых данных

```
 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 << "Enter an integer: ";
11:
12:     // Сохранить введенное пользователем целое число
13:     cin >> InputNumber;
14:
15:     // Аналогично с текстовыми данными
16:     cout << "Enter your name: ";
17:     string InputName;
18:     cin >> InputName;
19:
20:     cout << InputName << " entered " << InputNumber << endl;
21:
22:     return 0;
23: }
```

Результат

```
Enter an integer: 2011
Enter your name: Siddhartha
Siddhartha entered 2011
```

Анализ

В строке 8 переменная `InputNumber` объявляется как способная хранить данные типа `int`. В строке 10 пользователя просят ввести число, используя оператор `cout`, а введенное значение сохраняется в целочисленной переменной с использованием оператора `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 можно заметить в строках 17 и 18. Именно поэтому, чтобы использовать тип `string` далее в функции `main()`, в строке 2 была включена директива `#include <string>`. И наконец, в строке 20 оператор `cout` используется для отображения введенного имени и числа с промежуточным текстом, чтобы получить вывод `Siddhartha entered 2011`.

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

Резюме

Это занятие знакомит с основными частями простых программ C++. Здесь было 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, указанного в угловых скобках <> после нее, в текущую строку, как будто оно было введено в этом месте исходного кода.
- **В чем разница между комментариями `//` и `/*`?**
  Комментарий после двойной косой черты (`//`) оканчивается в конце строки. Комментарий после косой черты со звездочкой (`/*`) продолжается до тех пор, пока не встретится завершающий знак комментария (`*/`). Комментарии двойной косой черты называют также *однострочными комментариями*, а косой черты со звездочкой — *многострочными комментариями*. Помните, 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 позволить пользователю 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 указали бы путь к ним, используя аргументы командной строки, как при вызове в `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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "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:** Наберите и откомпилируйте эту программу. Почему она не компилируется? Как ее можно исправить?

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

2. Исправьте ошибки в программе упражнения 1 и, перекомпилировав, запустите ее снова.
3. Модифицируйте код листинга 2.4 так, чтобы продемонстрировать вычитание (используя оператор -) и умножение (используя оператор *).

ЗАНЯТИЕ 3
Использование переменных, объявление констант

Переменная (variable) — это средство, позволяющее программисту временно сохранить данные. *Константа* (constant) — это средство, позволяющее программисту определить элемент, которому не позволено изменяться.

На сегодняшнем занятии.

- Как в C++11 использовать ключевые слова `auto` и `constexpr`.
- Как объявить и определить переменные и константы.
- Как присваивать значения переменным и манипулировать ими.
- Как вывести значение переменной на экран.

Что такое переменная

Прежде чем перейти к рассмотрению потребности в использовании переменных в языке программирования, сделаем небольшое отступление и рассмотрим, как компьютер содержит 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 его загрузки, а также для 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 в памяти, скажем 578-й, процессору нужно при помощи специальной инструкции попросить выбрать оттуда значение или записать значение в нее.

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 множитель и множимое, один за другим, и каждое из этих значений необходимо хранить до момента умножения. В зависимости от того, что вы хотите делать с результатом умножения, их может понадобится хранить и для более позднего использования в программе. Если бы для хранения чисел вы должны были явно определять адреса областей памяти (такой как 578), это было бы медленно и подвержено ошибкам, поскольку вы должны были бы позаботиться о предотвращении перезаписи данных, уже существующих в этой области, и перезаписи ваших данных другими впоследствии.

При программировании на таких языках, как C++, для хранения значений определяют переменные. Определение переменной очень просто и осуществляется по такому шаблону:

```
тип_переменной имя_переменной;
```

или

```
тип_переменной имя_переменной = исходное_значение;
```

Атрибут типа переменной указывает компилятору характер данных, которые может хранить переменная, и компилятор резервирует для этого необходимое пространство. Выбранное программистом имя переменной является более осмысленной заменой для адреса области в памяти, где хранится значение переменной. Если исходное значение не применяется, вы не можете быть уверены в содержимом этой области памяти, что может быть плохо для программы. Поэтому, будучи необязательной, инициализация зачастую является хорошей практикой программирования. Листинг 3.1 демонстрирует объявление переменных, их инициализацию и использование в программе, которая умножает два числа, предоставленных 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 << "This program will help you multiply two numbers"
           << endl;
 7:
 8:     cout << "Enter the first number: ";
 9:     int FirstNumber = 0;
10:     cin >> FirstNumber;
11:
12:     cout << "Enter the second number: ";
13:     int SecondNumber = 0;
14:     cin >> SecondNumber;
15:
16:     // Умножение двух чисел, сохранение результата в переменной
17:     int MultiplicationResult = FirstNumber * SecondNumber;
18:
19:     // Отображение результата
20:     cout << FirstNumber << " x " << SecondNumber;
21:     cout << " = " << MultiplicationResult << endl;
22:
23:     return 0;
24: }
```

Результат

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

Анализ

Это приложение просит пользователя ввести два числа, результат умножения которых и отображается. Чтобы приложение могло использовать введенные пользователем числа, оно должно хранить их в памяти. Переменные `FirstNumber` и `SecondNumber`, объявленные в строках 9 и 13, решают задачу временного хранения введенных пользователем целочисленных значений. Операторы `std::cin` в строках 10 и 14 используются для получения введенных пользователем значений и сохранения их в двух целочисленных переменных. Оператор `cout` в строке 21 используется для отображения результата на консоли.

Давайте проанализируем объявление переменной подробнее:

```
 9:     int FirstNumber = 0;
```

Эта строка объявляет переменную типа `int`, который означает целое число, по имени `FirstNumber`. В качестве исходного переменной присваивается нулевое значение.

Таким образом, по сравнению с программированием на ассемблере, где необходимо явно просить процессор сохранить множитель в области памяти, скажем 578, язык C++

позволяет обратиться к области памяти для сохранения и получения данных, используя более дружественные концепции, такие как переменная по имени `FirstNumber`. Компилятор сам выполнит задачу по сопоставлению имени этой переменной с адресом области памяти и позаботится о соответствующих действиях за вас.

Таким образом, программист работает с понятными человеку именами, предоставляя компилятору право преобразовать переменную в адрес и создать инструкции для работы микропроцессора с оперативной памятью.

ВНИМАНИЕ!

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

Имена переменных могут состоять из букв и цифр, но не могут начинаться с цифр, а также содержать пробелы и арифметические операторы (такие как +, - и т.д.). Вы можете использовать в именах переменных символ подчеркивания.

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

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. По желанию 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 можно заметить, язык C++ позволяет объявить несколько переменных одного типа сразу и даже объявлять переменные в начале функции. Но все же объявление переменной перед ее первым применением зачастую оказывается лучше, поскольку это делает код более читабельным и объявление типа переменной оказывается ближе к месту ее первого применения.

ВНИМАНИЕ!

Данные, хранимые в переменных, находятся в оперативной памяти. Они теряются при отключении компьютера или завершении работы приложения, если программист не сохранит их специально на таком носителе данных, как жесткий диск.

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 есть четко определенная *область видимости* (scope), в пределах которой они допустимы и применимы. При использовании вне своих областей видимости имена переменных не будут распознаны компилятором, и ваша программа не будет

откомпилирована. Вне своей области видимости переменная — неопознанная сущность, о которой компилятор ничего не знает.

Чтобы лучше понять концепцию области видимости переменной, реорганизуем программу листинга 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 << "Enter the first number: ";
 7:     int FirstNumber = 0;
 8:     cin >> FirstNumber;
 9:
10:     cout << "Enter the second number: ";
11:     int SecondNumber = 0;
12:     cin >> SecondNumber;
13:
14:     // Умножение двух чисел, сохранение результата в переменной
15:     int MultiplicationResult = FirstNumber * SecondNumber;
16:
17:     // Отображение результата
18:     cout << FirstNumber << " x " << SecondNumber;
19:     cout << " = " << MultiplicationResult << endl;
20: }
21: int main ()
22: {
23:     cout << "This program will help you multiply two numbers"
          << 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

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

Анализ

Код листинга 3.2 делает то же самое, что и код листинга 3.1, создавая 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 снять комментарий со 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 и ограничиваются той функцией, в которой они объявлены, в данном случае функцией MultiplyNumbers(). *Локальная переменная* (local variable) применима в функции от места ее объявления до конца функции. Фигурная скобка (}), означающая конец функции, означает также конец области видимости объявленных в ней переменных. Когда функция заканчивается, все ее локальные переменные ликвидируются, а занимаемая ими память освобождается.

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

ВНИМАНИЕ!

Если в функции main() объявить другой набор переменных с теми же именами, то не надейтесь, что они будут содержать то значение, которое, возможно, было присвоено 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 эти переменные разграничиваются своими областями видимости.

Глобальные переменные

Если бы переменные, используемые в функции MultiplyNumbers() листинга 3.2, были объявлены не в ней, а вне ее, 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 глобальных целых числа
 5: int FirstNumber = 0;
 6: int SecondNumber = 0;
 7: int MultiplicationResult = 0;
 8:
 9: void MultiplyNumbers ()
10: {
11:     cout << "Enter the first number: ";
12:     cin >> FirstNumber;
13:
14:     cout << "Enter the second number: ";
15:     cin >> SecondNumber;
16:
17:     // Умножение двух чисел, сохранение результата в переменной
18:     MultiplicationResult = FirstNumber * SecondNumber;
```

```
19:
20:     // Отображение результата
21:     cout << "Displaying from MultiplyNumbers(): ";
22:     cout << FirstNumber << " x " << SecondNumber;
23:     cout << " = " << MultiplicationResult << endl;
24: }
25: int main ()
26: {
27:     cout << "This program will help you multiply two numbers"
          << endl;
28:
29:     // Вызов функции, выполняющей всю работу
30:     MultiplyNumbers();
31:
32:     cout << "Displaying from main(): ";
33:
34:     // Теперь эта строка компилируется и работает!
35:     cout << FirstNumber << " x " << SecondNumber;
36:     cout << " = " << MultiplicationResult << endl;
37:
38:     return 0;
39: }
```

Результат

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 19
Displaying from MultiplyNumbers(): 51 x 19 = 969
Displaying from main(): 51 x 19 = 969
```

Анализ

Листинг 3.3 демонстрирует результат умножения в двух функциях, причем переменные `FirstNumber`, `SecondNumber` и `MultiplicationResult` объявлены вне их. Эти переменные *глобальны* (global), поскольку они 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` применяется и в функции `MultiplyNumbers()`, и повторно в функции `main()`.

ВНИМАНИЕ!

Безосновательное использование глобальных переменных обычно считается плохой практикой программирования.

Значение глобальной переменной может быть присвоено в любой функции, и это значение может оказаться непредсказуемо, особенно если разные функциональные модули разрабатываются разными программистами группы.

Корректный способ получения результата умножения в функции `main()` листинга 3.3 подразумевает возвращение в нее результата вызова функции `MultiplyNumbers()`.

Популярные 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 фундаментальных типов переменных, поддерживаемых самим 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 и 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	Значения
`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
`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 из которых являются зарезервированными ключевыми словами C++. Этот тип особенно полезен при хранении параметров и флагов, которые могут быть установлены или сброшены, существовать или отсутствовать, могут быть доступными или недоступными.

Типичное объявление инициализированной логической переменной имеет следующий вид:

```
bool AlwaysOnTop = false;
```

Выражение, обрабатывающее логический тип, выглядит так:

```
bool DeleteFile = (UserSelection == "yes");
// истино, только если UserSelection содержит "yes",
// в противном случае ложно
```

Использование типа char для хранения символьных значений

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

```
char UserInput = 'Y'; // инициализированный символ 'Y'
```

Обратите внимание, что память состоит из битов и байтов. Биты могут содержать значения 0 или 1, а байты могут хранить числовые представления, используя 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, <DEL>) и специальных символов (таких, как возврат на один символ) было стандартизировано в стандартный американский код обмена информацией (American Standard Code for Information Interchange), называемый также ASCII.

Вы можете открыть таблицу в приложении Д, "Коды ASCII", и увидеть, что символ 'Y', присвоенный переменный UserInput, — это десятичное значение 89, согласно стандарту ASCII. Таким образом, компилятор просто сохраняет значение 89 в области памяти, зарезервированной для переменной UserInput.

Концепция знаковых и беззнаковых целых чисел

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

Если эти значения должны быть беззнаковыми, чтобы 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).

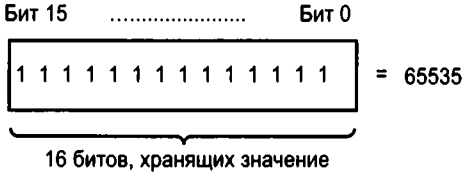


РИС. 3.1. Организация битов в 16-разрядном коротком беззнаковом целом числе

Но как же представить в этой области отрицательные числа? Один из способов — "пожертвовать" для знака одним из разрядов, который указывал бы, положительное или отрицательное значение содержится в других битах (рис. 3.2). Знаковый разряд должен быть *самым старшим битом* (Most-Significant-Bit — MSB), поскольку *самый младший бит* (Least-Significant-Bit — LSB) нужен для обозначения нечетных чисел. Когда бит MSB содержит информацию о знаке, предполагается, что значение 0 означает положительное число, а значение 1 — отрицательное. Другие байты содержат абсолютное значение.

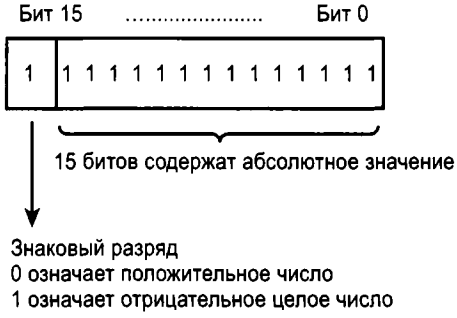


РИС. 3.2. Организация битов в 16-разрядном коротком знаковом целом числе

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 пространстве.

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

Эти типы отличаются по своим размерам, а следовательно, и по диапазону значений, которые они могут содержать. Тип `int`, возможно, самый популярный тип размером в 32 бита на большинстве компиляторов. Используйте подходящий тип в зависимости от максимального значения, которое определенная переменная предположительно будет содержать.

Объявление переменной знакового типа очень просто:

```
short int SmallNumber = -100;
int LargerNumber = -70000;
long PossiblyLargerThanInt = -70000;
// на некоторых платформах long совпадает с int
long long LargerThanInt = -70000000000;
```

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

В отличие от знаковых аналогов, типы беззнаковых целочисленных переменных не могут содержать информацию о знаке, а следовательно, могут содержать вдвое больше положительных значений.

Объявление переменной беззнакового типа тоже очень просто:

```
unsigned short int SmallNumber = 255;
unsigned int LargerNumber = 70000;
// на некоторых платформах long совпадает с int
unsigned long PossiblyLargerThanInt = 70000;
unsigned long long LargerThanInt = 70000000000;
```

ПРИМЕЧАНИЕ

Беззнаковый тип переменной используется тогда, когда ожидаются только позитивные значения. Так, если вы подсчитываете количество яблок, не используйте тип `int`, а используйте тип `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.

Типы с плавающей точкой `float` и `double`

Числа с плавающей запятой вы, возможно, изучали в школе как вещественные числа. Эти числа могут быть положительными и отрицательными. Они могут содержать десятичные значения. Так, если в переменной C++ необходимо сохранить значение числа Пи (22/7, или 3,14), вы использовали бы 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 = 3.14;
```

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

```
double MorePrecisePi = 22 / 7;
```

ПРИМЕЧАНИЕ

Упомянутые в таблице типы данных зачастую называют *простыми старыми данными* (Plain Old Data – POD). К этой категории относятся также объединения этих типов (структуры, перечисления, объединения и классы).

Определение размера переменной с использованием оператора `sizeof`

Размер (size) — это объем памяти, резервируемый компилятором при объявлении программистом переменной для содержания присваиваемых ей данных. Размер переменной зависит от ее типа, и в языке C++ есть очень удобный оператор `sizeof`, который сообщает размер в байтах переменной или типа.

Применение оператора `sizeof` очень просто. Чтобы определить размер целого числа, вызывайте оператор `sizeof` с параметром `int` (тип), как показано в листинге 3.4.

```
cout << "Size of an int: " << sizeof (int);
```

ЛИСТИНГ 3.4. Поиск размера стандартных типов переменных языка C++

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     using namespace std;
 6:     cout << "Computing the size of some C++ inbuilt variable types"
          << endl;
 7:
 8:     cout << "Size of bool: " << sizeof(bool) << endl;
 9:     cout << "Size of char: " << sizeof(char) << endl;
10:     cout << "Size of unsigned short int: " << sizeof(unsigned short)
          << endl;
11:     cout << "Size of short int: " << sizeof(short) << endl;
12:     cout << "Size of unsigned long int: " << sizeof(unsigned long)
          << endl;
13:     cout << "Size of long: " << sizeof(long) << endl;
14:     cout << "Size of int: " << sizeof(int) << endl;
15:     cout << "Size of unsigned long long: "
          << sizeof(unsigned long long)<< endl;
16:     cout << "Size of long long: " << sizeof(long long) << endl;
17:     cout << "Size of unsigned int: " << sizeof(unsigned int) << endl;
18:     cout << "Size of float: " << sizeof(float) << endl;
19:     cout << "Size of double: " << sizeof(double) << endl;
20:
21:     cout << "The output changes with compiler, hardware and OS"
          << endl;
22:
23:     return 0;
24: }
```

Результат

```
Computing the size of some C++ inbuilt variable types
Size of bool: 1
Size of char: 1
Size of unsigned short int: 2
Size of short int: 2
Size of unsigned long int: 4
Size of long: 4
Size of int: 4
Size of unsigned long long: 8
Size of long long: 8
Size of unsigned int: 4
Size of float: 4
Size of double: 8
```

Конкретные значения вывода зависят от компилятора, аппаратных средств и операционной системы.

Анализ

Вывод листинга 3.4 демонстрирует размеры различных типов в байтах и зависит от конкретной платформы: компилятора, операционной системы и аппаратных средств. Данный конкретный вывод — это результат выполнения программы в 32-битовом режиме (32-битовый компилятор) на 64-битовой операционной системе. Обратите внимание, что 64-битовый компилятор, вероятно, даст другие результаты, а 32-битовый компилятор автор выбрал потому, что должен был иметь возможность запускать приложение как на 32-битовых, так и на 64-битовых системах. Вывод оператора `sizeof` свидетельствует о том, что размер переменной знакового и беззнакового типа одинаков; единственным различием у этих двух типов MSB является наличие информации о знаке.

ПРИМЕЧАНИЕ

Все размеры в выводе приведены в байтах. Размер объекта – важный параметр при резервировании памяти для нее, особенно когда резервирование осуществляется динамически.

C++11

Возможности компилятора по выведению типов и ключевое слово `auto`

В некоторых случаях, когда тип переменной очевиден по присваиваемому при инициализации значению, именно он и применяется. Например, если переменная инициализируется значением `true`, типом переменной может быть только `bool`. В языке C++11 есть возможность определять тип неявно, с использованием вместо него ключевого слова `auto`:

```
auto Flag = true;
```

Здесь задача определения конкретного типа переменной `Flag` оставлена компилятору. Компилятор просто проверяет характер 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 `Flag` и внутренне рассматривает ее как имеющую тип `bool`, что демонстрирует листинг 3.5.

ЛИСТИНГ 3.5. Использование ключевого слова `auto` для выведения типов компилятором

```
#include <iostream>
using namespace std;

int main()
{
    auto Flag = true;
    auto Number = 2500000000000;

    cout << "Flag = " << Flag;
    cout << " , sizeof(Flag) = " << sizeof(Flag) << endl;
    cout << "Number = " << Number;
    cout << " , sizeof(Number) = " << sizeof(Number) << endl;

    return 0;
}
```

Результат

```
Flag = 1 , sizeof(Flag) = 1
Number = 2500000000000 , sizeof(Number) = 8
```

Анализ

Как можно заметить, вместо явного указания типа `bool` для переменной `Flag` и типа `long long` для переменной `Number` в строках 6 и 7, где они объявляются, было использовано ключевое слово `auto`. Это делегирует решение о типе переменных компилятору, который использует для этого инициализирующее значение. Чтобы проверить, создал ли компилятор фактически предполагаемые типы, используется оператор `sizeof`, позволяющий увидеть в выводе листинга 3.4, 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. Возьмем, например, случай, когда объявляется динамический массив целых чисел `MyNumbers` в форме `std::vector`:

```
std::vector<int> MyNumbers;
```

Вы обращаетесь к элементам массива (или перебираете их) и отображаете полученные значения, используя следующий код:

```
for ( vector<int>::const_iterator Iterator = MyNumbers.begin();
      Iterator < MyNumbers.end();
      ++Iterator )
    cout << *Iterator << " ";
```

Ни тип `std::vector`, ни оператор цикла `for` еще не были объяснены, поэтому не волнуйтесь, если код кажется непонятным. Он лишь перебирает все элементы вектора, начиная с начала до конца, и отображает их значения с помощью оператора `cout`. Посмотрите на сложность первой строки, где объявляется переменная `Iterator` и ей присваивается исходное значение, возвращенное методом `begin()`. Переменная `Iterator` имеет тип `vector<int>::const_iterator`, изучать и писать который программистам весьма сложно. Вместо того чтобы знать его наизусть, программист может положиться на тип возвращаемого значения метода `begin()` и упростить код цикла `for` до следующего:

```
for( auto Iterator = MyNumbers.begin();
     Iterator < MyNumbers.end();
     ++Iterator )
    cout << *Iterator << " ";
```

Обратите внимание, насколько короче стала первая строка. Компилятор проверяет инициализирующее значение переменной `Iterator`, которым является возвращаемое

значение метода `begin()`, и назначает его как тип переменной. Это упрощает программирование на языке C++, особенно когда вы используете шаблоны не очень часто.

Использование ключевого слова typedef для замены типа переменной

Язык C++ позволяет переименовывать типы переменных в нечто, что вы могли бы найти 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 PosNumber = 4532;
```

При компиляции первая строка указывает компилятору, что `STRICLY_POSITIVE_INTEGER` — это только название типа `unsigned int`. Впоследствии, когда компилятор встречает уже определенный тип `STRICLY_POSITIVE_INTEGER`, он заменяет его типом `unsigned int` и продолжает компиляцию.

ПРИМЕЧАНИЕ

Определение или подстановка типа особенно удобна при работе со сложными типами, у которых может быть громоздкий синтаксис. Например, при использовании шаблонов.

Что такое константа

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

```
Площадь = Pi * Радиус * Радиус;
Периметр = 2 * Pi * Радиус круга
```

В данных формулах `Pi` — это константа со значением 22/7[1]. Вы не хотите, чтобы значение `Pi` изменилось где-нибудь в вашей программе. Вы также не хотите случайно присвоить `Pi` неправильное значение, скажем, при небрежном копировании и вставке или при контекстном поиске и замене. Язык C++ позволяет определить `Pi` как константу, которая не может быть изменена после объявления. Другими словами, после того как значение константы определено, оно не может быть изменено. Попытки присвоения 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 переменным, константы также занимают пространство в памяти и имеют имя для идентификации адреса зарезервированной для нее области. Однако содержимое этой области не может быть перезаписано. В языке C++ возможны следующие константы.

---

[1] Здесь и далее используется приближенное значение числа Pi с точностью до двух знаков после запятой. Более точное значение этого числа равно 3,14159265. — *Примеч. ред.*

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

Литеральные константы

Вернемся к листингу 3.1 — простой программе, умножающей два числа. Целочисленная переменная по имени `FirstNumber` там объявлялась так:

```
9: int FirstNumber = 0;
```

Целочисленной переменной `FirstNumber` присваивается исходное нулевое значение. Здесь нуль — это часть кода, компилируемая в приложение, она является неизменной и называется *литеральной константой* (literal constant). Литеральные константы бывают множества типов: `bool`, `integer`, `string` и т.д. В самой первой программе C++ (см. листинг 1.1) текст `"Hello World"` отображался с использованием следующего кода:

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

где `"Hello World"` — это константа *строковый литерал* (string literal).

Объявление переменных как констант с использованием ключевого слова `const`

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

```
const имя_типа имя_константы;
```

Давайте рассмотрим простое приложение, которое отображает значение константы по имени `Pi` (листинг 3.6).

ЛИСТИНГ 3.6. Объявление константы по имени `Pi`

```
 1: #include <iostream>
 2:
 3: int main()
 4: {
 5:     using namespace std;
 6:
 7:     const double Pi = 22.0 / 7;
 8:     cout << "The value of constant Pi is: " << Pi << endl;
 9:
10:     // Снятие комментария со следующей строки приведет к ошибке
11:     // Pi = 345;
12:
13:     return 0;
14: }
```

Результат

```
The value of constant Pi is: 3.14286
```

Анализ

Обратите внимание на объявление константы `Pi` в строке 7. Ключевое слово `const` позволяет указать компилятору, что `Pi` — это константа типа `double`. Если снять комментарий со строки 11, где осуществляется 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 прекрасный способ гарантировать неизменность определенных данных.

ПРИМЕЧАНИЕ

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

Это особенно полезно там, где программистов несколько.

Константы полезны при объявлении массивов постоянной длины, которые неизменны во время компиляции. Листинг 4.2, приведенный в занятии 4, "Массивы и строки", содержит пример, демонстрирующий использование синтаксиса `const int` при определении длины массива.

C++11

Объявление констант с использованием ключевого слова `constexpr`

Концепция *константных выражений* (constant expression) всегда существовала в языке C++, еще до появления C++11, однако она не была формализована в ключевое слово. Обратите внимание на пример кода в листинге 3.5, в котором константное выражение `22.0 / 7` вполне поддерживается компиляторами и до 2011 года выпуска. Однако предыдущие компиляторы не учитывали определение функций, которые могли быть выполнены во время компиляции. В языке C++11 вы можете определить это так:

```
constexpr double GetPi() {return 22.0 / 7;}
```

Объявление функции `GetPi()`, используемой в комбинации с другой константой, такой, как здесь, делает допустимым следующий оператор:

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

Различие между константой и константным выражением на первый взгляд незначительно; однако с точки зрения компилятора и приложения — это новые возможности по оптимизации. Второй оператор (без `constexpr`) после компиляции прежним компилятором обрабатывался бы приложением во время выполнения, а компилятор, совместимый

со стандартом C++11, обработал бы выражение во время компиляции, и приложение выполнялось бы быстрее.

ПРИМЕЧАНИЕ

На момент написания этой книги ключевое слово `constexpr` не поддерживалось компилятором Microsoft Visual C++ Express Compiler. Оно поддерживалось только компилятором g++.

Перечисляемые константы

Иногда некая переменная может принимать значение 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. *Перечисляемые константы* (enumerated constant) — это именно то, что необходимо в данной ситуации, и характеризуются они ключевым словом `enum`.

Вот пример перечисляемой константы, которая определяет цвета радуги:

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

А вот другой пример — направление компаса:

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

Обратите внимание, что эти перечисляемые константы могут теперь использоваться как типы переменных, которые могут принимать значения, ограниченные объявленным ранее содержимым. Так, при определении переменной, которая содержит цвета радуги, вы объявили бы ее так:

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

В приведенной выше строке кода объявляется перечисляемая переменная `MyWorldsColor` типа `RainbowColors`. Эта перечисляемая переменная ограничена содержанием только одного из семи цветов радуги, и не допускает никаких других значений.

ПРИМЕЧАНИЕ

При объявлении перечисляемой константы компилятор преобразует перечисляемые значения, такие как `Violet` и другие, в целые числа. Каждое последующее значение перечисления оказывается на единицу больше предыдущего. Начальное значение вы можете задать сами, но если не сделаете этого, то компилятор начнет 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 3.7 демонстрирует использование перечисляемых констант для содержания четырех направлений при инициализации первого значения.

ЛИСТИНГ 3.7. Использование перечисляемых значений для указания направлений ветра

```
#include <iostream>
using namespace std;

enum CardinalDirections
{
    North = 25,
    South,
    East,
    West
};

int main()
{
    cout << "Displaying directions and their symbolic values"
         << endl;
    cout << "North: " << North << endl;
    cout << "South: " << South << endl;
    cout << "East: " << East << endl;
    cout << "West: " << West << endl;

    CardinalDirections WindDirection = South;
    cout << "Variable WindDirection = " << WindDirection << endl;

    return 0;
}
```

Результат

```
Displaying directions and their symbolic values
North: 25
South: 26
East: 27
West: 28
Variable WindDirection = 26
```

Анализ

Обратите внимание на то, как определены четыре перечисляемые константы направления, но исходное значение 25 было присвоено только первому, `North` (см. строку 6). Это автоматически гарантирует, что следующим константам будут соответствовать значения 26, 27 и 28, как представлено в выводе. В строке 20 создается переменная типа `CardinalDirections`, которой присваивается исходное значение `South`. При выводе на экран в строке 21 компилятор передает целочисленное значение, связанное со значением `South`, которым является 26.

СОВЕТ

Имеет смысл обратиться к листингам 6.4 и 6.5 занятия 6, "Ветвление процесса выполнения программ". Там перечисление используется для дней недели, а условное выражение позволяет указать выбранный пользователем день.

Определение констант с использованием директивы #define

Главное, не используйте это при написании новых программ. Единственная причина упоминания определения констант с использованием директивы `#define` в этой книге заключается в том, чтобы помочь вам понять некоторые устаревшие программы, в которых для определения числа Пи мог бы использоваться такой синтаксис:

```
#define Pi 3.14286
```

Это макрокоманда препроцессора, предписывающая компилятору заменять все упоминания `Pi` значением `3.14286`. Обратите внимание: это текстовая замена (читай: неинтеллектуальная), осуществляемая препроцессором. Компилятор не знает и не заботится о фактическом типе рассматриваемой константы.

ВНИМАНИЕ!

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

Именование переменных и констант

Существует множество разных способов и соглашений по именованию переменных. Некоторые программисты предпочитают приписывать к именам переменных несколько символов, означающих тип. Например:

```
bool bIsLampOn = false;
```

где `b` — префикс, который программист добавил для указания на то, что переменная имеет тип `bool`. Подобная форма записи называется Венгерской нотацией; первоначально она была разработана и применена корпорацией Microsoft. Однако язык C++ — строго типизированный, и компилятор сам знает тип переменной, причем не по префиксу в ее имени, а по типу в ее объявлении, которым является `bool`. Таким образом, программистам ныне настоятельно не рекомендуется следовать Венгерской нотации. Имя переменной должно быть понятным, даже если оно станет немного длинным. Учитывая,

что логическая переменная в данном примере использовалась при программировании электрооборудования автомобиля, немного лучшим вариантом будет следующий:

```
bool IsHeadLampOn = false;
```

Заметим, что и эти варианты лучше и понятней, чем нечто вроде следующего:

```
bool b = false;
```

Таких неинформативных имен переменных нужно избегать любой ценой.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Присваивайте переменным осмысленные имена, даже если они становятся длинными	**Не присваивайте** переменным имена, которые слишком коротки или содержат только один символ
Удостоверьтесь, что имя переменной объясняет ее цель	**Не присваивайте** переменным имена, которые используют экзотические акронимы, известные только вам
Встаньте на место того, кто еще не видел ваш код, и подумайте, имеют ли смысл применяемые в ней имена	**Не присваивайте** переменным имена, совпадающие с зарезервированными ключевыми словами языка C++, поскольку такой код не будет компилироваться
Используйте в своей группе соглашение об именовании и строго придерживайтесь его	

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 для 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	else	New	this
auto	enum	Operator	throw
bool	explicit	Private	true
break	export	Protected	try
case	extern	Public	typedef
catch	false	Register	typeid
char	float	reinterpret_cast	typename
class	for	Return	union
const	friend	Short	unsigned
const_cast	goto	Signed	using
continue	if	Sizeof	virtual

Окончание табл. 3.2

default	inline	Static	void
delete	int	static_cast	volatile
do	long	Struct	wchar_t
double	mutable	Switch	while
dynamic_cast	namespace	Template	

Кроме того, зарезервированы следующие слова:

and	bitor	not_eq	xor
and_eq	compl	Or	xor_eq
bitand	not	or_eq	

Резюме

На этом занятии речь шла об использовании памяти для временного хранения значений в переменных и константах. Вы узнали, что размер переменных определяет их тип и что оператор `sizeof` позволяет выяснить его. Вы узнали о существовании различных типов переменных, таких как `bool`, `int` и т.д., а также 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 ошибкой оболочки или переполнением регистров. Вы познакомились с новым ключевым словом C++11 `auto`, который позволяет компилятору самостоятельно вывести тип данных на основе инициализирующего значения переменной.

Кроме того, мы рассмотрели различные типы констант и применение самых важных из них (использующих ключевые слова `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 указать компилятору, что значение определенной переменной должно быть постоянным и не должно изменяться. Следовательно, компилятор гарантирует, что переменной, объявленной постоянной, никогда не будет присвоено другое значение, даже если другой программист, поправляя вашу работу, по неосторожности попытается перезаписывать значение. Так, если вы знаете, что значение переменной не должно изменяться, хорошая практика программирования подразумевает объявление ее как константы, что увеличивает качество приложения.

- **Зачем инициализировать значение переменной?**

  Если вы не инициализируете переменную, то не знаете, какое значение она содержит первоначально. Начальное значение — это только содержимое области памяти, зарезервированной для переменной. Инициализация переменной, такая как

  ```
  int MyFavoriteNumber = 0;,
  ```

записывает в область памяти, зарезервированной для переменной MyFavoriteNumber, исходное значение по вашему выбору, в данном случае 0. Нередки ситуации, когда осуществляется обработка условного выражения в зависимости от значения переменной (как правило, проверка на отличие от нуля). Без инициализации такая логика работает ненадежно, поскольку вновь зарезервированная область памяти содержит то, что в ней было раньше, т.е. случайное значение, которое зачастую отличается от нуля.

- **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, где вычислительные возможности и доступная память весьма ограничены. В данном случае программист может сэкономить 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 — даже отсутствовать.

- **Почему не следует широко использовать глобальные переменные? Разве не правда, что они пригодны для использования повсюду в приложении, и я могу 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 глобально. 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, на ваш код это тоже повлияет. Поэтому экономия нескольких секунд или минут не должна быть критерием, и вы не должны использовать глобальные переменные без разбора, чтобы гарантировать стабильность своего кода.

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

  Произойдет *переполнение* (wrapping). Декремент значения 0 беззнаковой целочисленной переменной на 1 превратит его в самое высокое значение, которое она способна содержать! Посмотрите табл. 3.1 и убедитесь, что переменная типа `unsigned short` способна содержать значения от 0 до 65535. Итак, объявим переменную типа `unsigned short`, осуществим декремент и увидим нечто неожиданное:

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

  Обратите внимание: это проблема не типа `unsigned short`, а вашего способа ее применения. Целочисленный тип без знака (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 резервированию 64 Кбайт! Хуже того, если бы переменная `MyShortInt` использовалась как индекс при доступе к областям памяти, ваше приложение, вероятней всего, обратилось бы к внешней области памяти и потерпело неудачу!

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 Г, "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 целым числом?
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 обоих их меньше, чем у длинного целого числа.
3. Напишите программу для вычисления площади и периметра круга, радиус которого вводится пользователем.
4. Что если бы в коде упражнения 3 площадь и периметр хранились в целочисленных переменных, а результат — в переменной любого другого типа?
5. **Отладка:** Что не так 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 стаю котов.

На сегодняшнем 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 полный набор, например массив солнечных панелей".

Ниже приведены характеристики массива.

- Массив — это коллекция элементов.
- Все содержащиеся в массиве элементы имеют одинаковый вид.
- Такая коллекция формирует полный набор.

В языке C++ массивы позволяют сохранить в памяти элементы данных одинакового типа в последовательном порядке.

Необходимость в массивах

Предположим, вы пишете программу, где пользователь может ввести пять целых чисел и отобразить их на экране. Один из способов подразумевал бы объявление в программе пяти отдельных целочисленных переменных и сохранение в них отображаемых значений. Объявления выглядели бы следующим образом:

```
int FirstNumber = 0;
int SecondNumber = 0;
int ThirdNumber = 0;
int FourthNumber = 0;
int FifthNumber = 0;
```

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. Это потребует огромной работы и терпения на ее выполнение. А что делать, если пользователь попросит обеспечить 500 000 целых чисел вместо 5?

Правильно было бы объявить массив из пяти целых чисел, каждое из которых инициализировалось бы нулем:

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

Таким образом, если бы вас попросили обеспечить 500 000 целых чисел, то ваш массив без проблем увеличился бы так:

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

Массив из пяти символов был бы определен следующим образом:

```
char MyCharacters [5];
```

Такие массивы называются *статическими* (static array), поскольку количество содержащихся в них элементов, а также размер его области в 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 также инициализировать одним значением:

```
int MyNumbers [5] = {100}; // инициализировать все элементы
                           // значением 100
```

Вы можете также инициализировать только часть элементов массива:

```
int MyNumbers [5] = {34, 56}; // инициализировать первые два элемента
```

Вы можете определить длину массива (т.е. 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 элементов в нескольких местах можно избежать необходимости выяснения его длины каждый раз — достаточно лишь исправить инициализирующее значение при объявлении const int.

ПРИМЕЧАНИЕ

При частичной инициализации массивов некоторые компиляторы инициализируют проигнорированные вами элементы исходным значением 0.

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

```
int MyNumbers [] = {2011, 2052, -525};
```

Приведенный выше код создает массив из трех целых чисел с исходными значениями 2011, 2052 и –525.

ПРИМЕЧАНИЕ

Массивы, которые мы объявляли до сих пор, называются *статическими массивами* (static array), поскольку их длина фиксируется программистом во время компиляции. Такой массив не может принять больше данных, чем определил программист. Он также не может задействовать меньше памяти, если используется только наполовину или вообще не используется.

Как данные хранятся в массиве

Рассмотрим книги, стоящие рядом на полке. Это пример одномерного массива, поскольку он располагается только в одной размерности, которой является количество книг в ней. Каждая книга — это элемент массива, а полка сродни области памяти, которая была зарезервирована для хранения этой коллекции книг (рис. 4.1).

РИС. 4.1. Книги на полке: одномерный массив

Это не ошибка, мы начинаем нумеровать книги с нуля. Поскольку, как вы увидите позже, индексы в языке C++ начинаются с 0, а не с 1. Подобно пяти книгам на полке, массив `MyNumbers`, содержащий пять целых чисел, выглядит очень похоже на рис. 4.2.

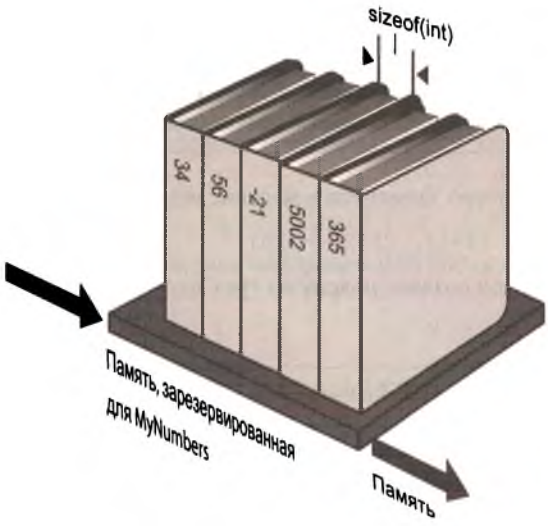


РИС. 4.2. Организация массива `MyNumbers` из пяти целых чисел

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, объявление констант". Объем памяти, зарезервированной компилятором для массива MyNumbers, составит sizeof(int) * 5. В общем виде объем памяти в байтах, резервируемой компилятором для массива, составляет:

```
Байты_массива = sizeof(тип_элемента) * количество_элементов
```

Доступ к данным, хранимым в массиве

Для обращения к элементам массива можно использовать отсчитываемый от нуля *индекс* (index). Индексы называются отсчитываемыми от нуля потому, что 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 — MyNumbers[4]. Другими словами, индекс последнего элемента в массиве всегда на единицу меньше его длины.

Когда запрашивается доступ к элементу по индексу N, компилятор использует адрес области памяти первого элемента (позиция нулевого индекса) как отправную точку, а затем пропускает N элементов, добавляя к нему смещение, вычисленное как N*sizeof(*тип_элемента*), чтобы получить адрес области, содержащей N+1-й элемент. Компилятор C++ не проверяет, находится ли индекс в пределах фактически определенных элементов массива. Вы можете попытаться выбрать элемент по индексу 1001 в массиве, содержащем только 10 элементов, поставив под угрозу безопасность и стабильность вашей программы. Ответственность за предотвращение обращения к элементам за пределами массива лежит исключительно на программисте.

ВНИМАНИЕ!

Результат доступа к массиву за его пределами непредсказуем. Как правило, это нарушает работу программы. Этого нужно избегать любой ценой.

Листинг 4.1 демонстрирует объявление массива целых чисел, инициализацию 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:
 8:     cout << "First element at index 0: " << MyNumbers [0] << endl;
 9:     cout << "Second element at index 1: " << MyNumbers [1] << endl;
10:     cout << "Third element at index 2: " << MyNumbers [2] << endl;
11:     cout << "Fourth element at index 3: " << MyNumbers [3] << endl;
12:     cout << "Fifth element at index 4: " << MyNumbers [4] << endl;
13:
14:     return 0;
15: }
```

Результат

```
First element at index 0: 34
Second element at index 1: 56
Third element at index 2: -21
Fourth element at index 3: 5002
Fifth element at index 4: 365
```

Анализ

В строке 6 объявляется массив из пяти целых чисел с исходными значениями, определенными для каждого из них. Последующие строки просто отображают целые числа, используя оператор `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, с листинга 4.1 строки кода нумеруются начиная с нуля, а не с единицы.

Изменение хранимых в массиве данных

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

Например, присвоение значения 2011 целочисленной переменной выглядит так:

```
int AnIntegerValue;
AnIntegerValue = 2011;
```

Присвоение значения 2011 четвертому элементу в рассматриваемом массиве выглядит так:

```
MyNumbers [3] = 2011; // Присвоение 2011 четвертому элементу
```

Листинг 4.2 демонстрирует использование констант в объявлении длины массива, а также присвоение значений отдельным элементам массива во время выполнения программы.

Листинг 4.2. Присвоение значений элементам массива

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     const int ARRAY_LENGTH = 5;
 6:
 7:     // Массив из 5 целых чисел, инициализированных нулями
 8:     int MyNumbers [ARRAY_LENGTH] = {0};
 9:
10:     cout << "Enter index of the element to be changed: ";
11:     int nElementIndex = 0;
12:     cin >> nElementIndex;
13:
14:     cout << "Enter new value: ";
15:     cin >> MyNumbers [nElementIndex];
16:
```

```
17:     cout << "First element at index 0: " << MyNumbers [0] << endl;
18:     cout << "Second element at index 1: " << MyNumbers [1] << endl;
19:     cout << "Third element at index 2: " << MyNumbers [2] << endl;
20:     cout << "Fourth element at index 3: " << MyNumbers [3] << endl;
21:     cout << "Fifth element at index 4: " << MyNumbers [4] << endl;
22:
23:     return 0;
24: }
```

Результат

```
Enter index of the element to be changed: 2
Enter new value: 2011
First element at index 0: 0
Second element at index 1: 0
Third element at index 2: 2011
Fourth element at index 3: 0
Fifth element at index 4: 0
```

Анализ

Синтаксис объявления массива в строке 8 использует константу `const integer ARRAY_LENGTH`, инициализированную предварительно значением пять. Поскольку это статический массив, его длина фиксируется во время компиляции. Компилятор заменяет константу `ARRAY_LENGTH` значением 5 и компилирует код, полагая, что `MyArray` — это целочисленный массив, содержащий пять элементов. В строках 10–12 пользователя спрашивают, какой элемент массива он хочет изменить, а введенный индекс сохраняется в целочисленной переменной `ElementIndex`. Это значение используется в строке 14 для изменения содержимого массива. Вывод демонстрирует, что изменился элемент по индексу 2, а фактически это был третий элемент массива, поскольку индексы отсчитываются от нуля. Вы должны привыкнуть к этому.

ПРИМЕЧАНИЕ

Многие новички в программировании на языке C++ присваивают значение пятому элементу массива из пяти целых чисел по индексу пять. Обратите внимание: это уже за пределами массива, и откомпилированный код попытается обратиться к шестому элементу массива, состоящему из пяти элементов.

Этот вид ошибки вызывается *ошибкой поста охранения* (fence-post error). Данное название происходит из того факта, что количество постов для построения охранения всегда на один больше, чем количество охраняемых участков.

ВНИМАНИЕ!

В листинге 4.2 отсутствует нечто фундаментальное: проверка введенного пользователем индекса на соответствие границам массива. Фактически предыдущая программа должна проверять, находится ли значение переменной `nElementIndex` в пределах от 0 до 4, и отбрасывать все остальные значения. Отсутствие этой проверки позволяет пользователю присвоить значение вне границ массива. Потенциально это может привести к нарушению работы приложения, а в самом плохом случае и операционной системы.

Более подробная информация о проверках приведена на занятии 6, "Ветвление процесса выполнения программ".

Использование циклов для доступа к элементам массива

При работе с элементами массива в последовательном порядке для обращения к ним (перебора) используются циклы. Чтобы быстро научиться эффективно работать с элементами массива, используя цикл `for`, обратитесь к листингу 6.10 занятия 6, "Ветвление процесса выполнения программ".

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Инициализируйте массивы всегда, иначе они будут содержать непредвиденные значения	**Никогда не обращайтесь** к элементу номер N, используя индекс N, в массиве из N элементов
Проверяйте всегда, используются ли ваши массивы в пределах их границ	**Не забывайте**, что к первому элементу в массиве обращаются по индексу 0

Многомерные массивы

Массивы, которые мы рассматривали до сих пор, напоминали книги на полке. Может быть больше книг на более длинной полке или меньше на более короткой. Таким образом, длина полки — единственная размерность, определяющая ее емкость, следовательно, она одномерна. Но что если теперь нужно использовать массив для моделирования солнечных панелей, как показано 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 по три столбца. С одной стороны вы можете рассматривать такое расположение 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 также создать в памяти многомерные массивы.

Объявление и инициализация многомерных массивов

Язык C++ позволяет объявлять многомерные массивы, указав количество элементов, которое необходимо зарезервировать по каждой размерности. Следовательно, 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 пределах от 0 до 5. Если бы пришлось инициализировать целочисленный массив в том же порядке, то это выглядело бы так:

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

Как видите, синтаксис инициализации подобен используемому при инициализации двух одномерных массивов. Обратите внимание: это не два массива, поскольку массив двумерный, это два его ряда. Если бы этот массив состоял из трех рядов и трех столбцов, его объявления и инициализация выглядели бы следующим образом:

```
int ThreeRowsThreeColumns [3][3] = {{-501, 206, 2011}, {989, 101, 206},
{303, 456, 596}};
```

ПРИМЕЧАНИЕ

Даже при том, что язык C++ позволяет использовать модель многомерных массивов, в памяти массив содержится как одномерный. Компилятор раскладывает многомерный массив в области памяти, которая расширяется только в одном направлении.

Если инициализировать тот же массив `SolarPanelIDs` следующим образом, то результат был бы тем же:

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

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

Доступ к элементам в многомерном массиве

Считайте многомерный массив массивом массивов. Поскольку рассмотренный ранее двумерный массив включал три ряда и три столбца, содержащих целые числа, вы можете представить 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, 2011}, {989, 101, 206},
{303, 456, 596}};
```

Он инициализирован способом, который 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][2].

Листинг 4.3 демонстрирует, как можно обращаться к целочисленным элементам в этом массиве.

ЛИСТИНГ 4.3. Доступ к элементам в многомерном массиве

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     int ThreeRowsThreeColumns [3][3] = \
 6:     {{-501, 206, 2011}, {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 2011
Row 1: 989 101 206
Row 2: 303 456 596
```

Анализ

Обратите внимание на способ обращения к элементам в построчном массиве, начинающемся с массива `Row 0` (первый ряд с индексом 0) и заканчивающимся массивом `Row 2` (третий ряд с индексом 2). Поскольку каждый из рядов — это массив, синтаксис для обращения к третьему элементу в первом ряду такой, как в строке 10.

ПРИМЕЧАНИЕ

Длина кода в листинге 4.3 существенно увеличивается при увеличении количества элементов в массиве или его размерностей. Фактически в профессиональной среде разработки такой код неприменим.

Более эффективный способ обращения к элементам многомерного массива приведен в листинге 6.14 занятия 6, "Ветвление процесса выполнения программ". 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, превышение которого маловероятно. 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 заполнение его значениями

```
 0: #include <iostream>
 1: #include <vector>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     vector<int> DynArrNums (3);
 8:
 9:     DynArrNums[0] = 365;
10:     DynArrNums[1] = -421;
11:     DynArrNums[2]= 789;
12:
13:     cout << "Number of integers in array: " << DynArrNums.size()
            << endl;
14:
15:     cout << "Enter another number for the array" << endl;
16:     int AnotherNum = 0;
17:     cin >> AnotherNum;
18:     DynArrNums.push_back(AnotherNum);
19:
20:     cout << "Number of integers in array: " << DynArrNums.size()
            << endl;
21:     cout << "Last element in array: ";
22:     cout << DynArrNums[DynArrNums.size() - 1] << endl;
23:
24:     return 0;
25: }
```

Результат

```
Number of integers in array: 3
Enter another number for the array
2011
Number of integers in array: 4
Last element in array: 2011
```

Анализ

Не волнуйтесь о синтаксисе векторов и шаблонов в листинге 4.4, — они пока еще не были объяснены. Попробуйте посмотреть 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 интересней всего, в строке 18 можете добавить его в вектор, используя метод `push_back()`. Вектор 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 статическому массиву 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) при помощи индекса, который для последнего элемента имеет значение `размер - 1`, а метод `size()` как раз и возвращает общее количество элементов вектора.

ПРИМЕЧАНИЕ

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

```
#include <vector>
```

Более подробная информация о векторах приведена на занятии 17, "Классы динамических массивов библиотеки STL".

Строки 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 такой 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 *знаком окончания строки* (string-terminating character), поскольку указывает компилятору, что строка закончилась. 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 демонстрирует листинг 4.5.

ПРИМЕЧАНИЕ

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

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

Чтобы увидеть этот и другие значения кодов ASCII, обратитесь к таблице в приложении Д, "Коды ASCII".

ЛИСТИНГ 4.5. Анализ завершающейся нулем строки в стиле C

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     char SayHello[] = {'H', 'e', 'l', 'l', 'o', ' ',
                           'W', 'o', 'r', 'l', 'd', '\0'};;
 6:     cout << SayHello << endl;
 7:     cout << "Size of array: " << sizeof(SayHello) << endl;
 8:
 9:     cout << "Replacing space with null" << endl;
10:     SayHello[5] = '\0';
11:     cout << SayHello << endl;
12:     cout << "Size of array: " << sizeof(SayHello) << endl;
13:
14:     return 0;
15: }
```

Результат

```
Hello World
Size of array: 12
Replacing space with null
Hello
Size of array: 12
```

Анализ

Код строки 10 заменяет пробел в строке `Hello World` нулевым символом. Теперь у массива есть два нулевых завершающих символа, но используется первый, что и создает эффект. Когда пробел заменяется нулевым символом, отображаемая строка усекается до части `Hello`. Метод `sizeof()` в строках 7 и 12 указывает, что размер массива не изменился, несмотря на изменение отображаемых данных.

ВНИМАНИЕ!

Если при объявлении и инициализации символьного массива в строке 5 листинга 4.5 вы забываете добавить символ `'\0'`, то после части `Hello World` вывод будет содержать случайный набор символов. Дело в том, что оператор `std::cout` не остановится на выводе массива, он будет продолжать вывод, пока не достигнет нулевого символа, даже если для этого придется перейти границы массива.

Эта ошибка может нарушить вашу программу, а в некоторых случаях поставить под угрозу стабильность системы.

Строки в стиле C чреваты опасностью. Листинг 4.6 демонстрирует риски, связанные с их применением.

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

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Enter a word NOT longer than 20 characters:" << endl;
 6:
 7:     char UserInput [21] = {'\0'};
 8:     cin >> UserInput;
 9:
10:     cout << "Length of your input was: " << strlen (UserInput)
          << endl;
11:
12:     return 0;
13: }
```

Результат

```
Enter a word NOT longer than 20 characters:
Don't UseThisProgram
Length of your input was: 19
```

Анализ

Опасность видна в выводе. Программа просит пользователя не вводить больше двадцати символов. Причина в том, что объявленный в строке 7 символьный буфер, предназначенный для хранения пользовательского ввода, имеет фиксированную статическую длину в 21 символ. Поскольку последний символ в строке должен быть нулевым `'\0'`, максимальная длина текста, хранимого буфером, ограничивается двадцатью символами. Обратите внимание на применение оператора `strlen` в строке 10 для вычисления длины строки. Он перебирает символьный буфер и подсчитывает количество символов, пока не достигает нулевого, который означает конец строки. Этот символ был вставлен в конец введенных пользователем данных оператором `cin`. Подобное поведение оператора `strlen` делает это опасным, поскольку он может легко пересечь границы символьного массива, если пользователь введет текст 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, "Ветвление процесса выполнения программ".

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

Стандартная строка C++ — самый эффективный способ работы и с текстовым вводом, и при манипуляции строками, например, при их конкатенации.

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 преодолеть границы символьного массива, если программист не гарантировал наличие завершающего нулевого символа.

Язык C++ предоставляет мощное и в то же время безопасное средство манипулирования строками — класс `std::string`, представленный в листинге 4.7. Класс `std::string` реализован не на статическом массиве типа `char` неизменного размера, как строки в стиле C, и допускает увеличение размера, когда в нем необходимо сохранить больше данных.

ЛИСТИНГ 4.7. Использование типа `std::string` для инициализации и хранения пользовательского ввода, а также копирование, конкатенация и определение длины строки

```
 0: #include <iostream>
 1: #include <string>
 2:
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     string Greetings ("Hello std::string!");
 8:     cout << Greetings << endl;
 9:
10:     cout << "Enter a line of text: " << endl;
11:     string FirstLine;
12:     getline(cin, FirstLine);
13:
14:     cout << "Enter another: " << endl;
15:     string SecLine;
16:     getline(cin, SecLine);
17:
18:     cout << "Result of concatenation: " << endl;
19:     string Concat = FirstLine + " " + SecLine;
20:     cout << Concat << endl;
21:
22:     cout << "Copy of concatenated string: " << endl;
23:     string Copy;
24:     Copy = Concat;
```

```
25:     cout << Copy << endl;
26:
27:     cout << "Length of concat string: " << Concat.length() << endl;
28:
29:     return 0;
30: }
```

Результат

```
Hello std::string!
Enter a line of text:
I love
Enter another:
C++ strings
Result of concatenation:
I love C++ strings
Copy of concatenated string:
I love C++ strings
Length of concat string: 18
```

Анализ

Старайтесь понять вывод и связать его с соответствующими элементами в коде. Не беспокойтесь пока о новых синтаксических средствах. Программа начинается с отображения инициализированной в строке 7 строки `Hello std::string!`. Затем, в строках 12 и 16, она просит пользователя ввести две строки текста, которые сохраняются в переменных `FirstLine` и `SecLine`. Фактически конкатенация очень проста и выглядит как арифметическая сумма в строке 19, где даже пробел был добавлен к первой строке. Действие копирования — это простое присвоение в строке 24. Определение длины строки осуществляется при вызове метода `length()` в строке 27.

ПРИМЕЧАНИЕ

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

```
#include <string>
```

Это можно заметить в строке 1 листинга 4.7.

Чтобы подробней изучить различные функции класса `std::string`, обратитесь к занятию 16, "Классы строк 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 основами 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 называется *переполнением буфера* (buffer overflow), и проверка ввода перед его использованием для индексации элементов позволяет гарантировать нахождение в пределах массива без их пересечения.

Динамические массивы позволяют программисту не волноваться 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, где конец строки отмечается нулевым завершающим символом `'\0'`. Кроме того, вы узнали, что язык C++ обеспечивает намного лучшую возможность — класс `std::string`, — предоставляющий удобные вспомогательные функции и позволяющий определить длину строк, объединять их и выполнять подобные действия.

Вопросы и ответы

- **Зачем заботиться об инициализации элементов статического массива?**

  Если не инициализировать массив, в отличие от переменной любого другого типа, 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?**

  Фактически 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, нуждающиеся в завершающем нулевом символе?**

  Только если 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 если 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 необходимо для нулевого символа в конце самой длинной строки. Если бы строка `Hello World` была наибольшей, которую предстоит содержать символьному массиву, то ее длина составила бы 11 + 1 символ, т.е. 12 символов.

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 Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Проверьте массив `MyNumbers` в листинге 4.1. Каковы индексы первых и последних его элементов?
2. Если необходимо позволить пользователю вводить строки, использовали бы вы строки в стиле C?
3. Сколько символов насчитывает компилятор, встретив `'\0'`?
4. Вы забываете завершить строку в стиле C нулевым символом. Что может случиться при ее использовании?
5. Посмотрите объявление вектора в листинге 4.4 и попытайтесь создать динамический массив, содержащий элементы типа `char`.

Упражнения

1. Объявите массив, представляющий клетки на шахматной доске; типом массива может быть перечисление, определяющее характер фигур на доске.
2. **Отладка:** Что не так с этим фрагментом кода?
```
int MyNumbers[5] = {0};
MyNumbers[5] = 450; // Присвоение значения 450 пятому элементу
```
3. **Отладка:** Что не так с этим фрагментом кода?
```
int MyNumbers[5];
cout << MyNumbers[3];
```

ЗАНЯТИЕ 5
Команды, выражения и операторы

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

На сегодняшнем занятии.

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

Выражения

Языки, разговорные или программирования, состоят из *выражений* (statement), которые следуют одно за другим. Давайте проанализируем первое выражение, которое вы изучили:

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

Выражение использует оператор `cout` для отображения текста на консоль (т.е. экран). Все выражения в языке C++ заканчиваются точкой с запятой (`;`), определяющей границу выражения. Это подобно точке (`.`), которую вы добавляете в конце предложения разговорного языка. Следующее выражение может начаться непосредственно после точки с запятой, но для удобства и удобочитаемости программисты записывают выражения с новой строки. Вот, например, два выражения в одной строке:

```
cout << "Hello World" << endl; cout << "Another hello" << endl;
   // Одна строка, два выражения
```

ПРИМЕЧАНИЕ

Отступы обычно не воспринимаются компилятором. К ним относятся пробелы, табуляция, символы новой строки и т.д. Тем не менее отступы в пределах строковых литералов отображаются в выводе.

Поэтому следующий код был бы 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 выражение точку с запятой (`;`). Если по каким-то причинам необходимо распространить оператор на несколько строк, достаточно добавить в конец символ обратной косой черты (\):

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

Еще один способ написать приведенный выше оператор в двух строках — это применить два строковых литерала вместо одного:

```
cout << "Hello "
    "World" << endl; // два строковых литерала тоже вполне допустимы
```

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

ПРИМЕЧАНИЕ

Разделение выражений на несколько строк может быть очень полезным, когда у вас есть длинные текстовые элементы или сложные выражения, состоящие из множества переменных, которые делают выражение намного длиннее, чем может вместить большинство экранов.

Составные выражения, или блоки

Сгруппировав операторы в фигурных скобках { . . . }, вы создаете *составной оператор* (compound statement), или *блок* (block).

```
{
    int Number = 365;
    cout << "This block contains an integer and a cout statement"
         << endl;
}
```

Как правило, блок объединяет множество операторов. Блоки особенно полезны при применении условного оператора `if` и циклов, которые рассматриваются на занятии 6, "Ветвление процесса выполнения программ".

Использование операторов

Операторы (operator) — это инструментальные средства, предоставляемые языком C++ для работы с данными, их преобразованием, обработкой и принятия решений на их основе.

Оператор присвоения (=)

Оператор присвоения (assignment operator) мы уже использовали в этой книге, он вполне интуитивно понятен:

```
int MyInteger = 101;
```

Приведенное выше выражение использует оператор присвоения для инициализации целочисленной переменной значением 101. Оператор присвоения заменяет значение, содержащееся операндом слева (называемого *l-значением* (l-value)), значением операнда справа (называемого *r-значением* (r-value)).

Понятие l- и r-значений

l-значения зачастую называют областями памяти. Такая переменная, как `MyInteger`, из приведенного выше примера фактически является дескриптором (описателем) области памяти и соответственно l-значением. r-значения, напротив, 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:

```
101 = MyInteger;
```

Операторы суммы (+), 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 addition = Num1 + Num2;         // 27
int subtraction = Num1 - Num2;      // 17
int multiplication = Num1 * Num2;   // 110
int division = Num1 / Num2;         // 4
int modulo = Num1 % Num2;           // 2
```

Оператор деления (/) возвращает результат деления двух операндов. Однако в случае целых чисел результат не содержит десятичной части, поскольку целые числа по определению не могут ее содержать. Оператор деления по модулю (%) возвращает только остаток деления и применим только к целочисленным значениям. Листинг 5.1 содержит простую программу, демонстрирующую 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 << "Enter two integers:" << endl;
 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

```
Enter two integers:
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. Интересней 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 необходим *инкремент* (increment), или приращение, значения на единицу. Это особенно важно для переменных, контролирующих циклы, где значение переменной должно увеличиваться или уменьшаться на единицу при каждом выполнении цикла.

Для решения этой задачи 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) операторами инкремента или декремента.

Что значит постфиксный и префиксный

Сначала следует понять различие между префиксом и постфиксом, а затем использовать то, что нужно в данном случае. Результат выполнения постфиксных операторов заключается в том, что сначала l-значение присваивается r-значению, 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 одинаковое значение. Листинг 5.2 демонстрирует результат выполнения префиксных и постфиксных операторов инкремента и декремента на примере 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 MyInt = 101;
 6:     cout << "Start value of integer being operated: " << MyInt
           << endl;
 7:
 8:     int PostFixInc = MyInt++;
 9:     cout << "Result of Postfix Increment = " << PostFixInc << endl;
10:     cout << "After Postfix Increment, MyInt = " << MyInt << endl;
11:
12:     MyInt = 101; // Переустановка
13:     int PreFixInc = ++MyInt;
14:     cout << "Result of Prefix Increment = " << PreFixInc << endl;
15:     cout << "After Prefix Increment, MyInt = " << MyInt << endl;
```

```
16:
17:     MyInt = 101;
18:     int PostFixDec = MyInt--;
19:     cout << "Result of Postfix Decrement = " << PostFixDec << endl;
20:     cout << "After Postfix Decrement, MyInt = " << MyInt << endl;
21:
22:     MyInt = 101;
23:     int PreFixDec = --MyInt;
24:     cout << "Result of Prefix Decrement = " << PreFixDec << endl;
25:     cout << "After Prefix Decrement, MyInt = " << MyInt << endl;
26:
27:     return 0;
28: }
```

Результат

```
Start value of integer being operated: 101
Result of Postfix Increment = 101
After Postfix Increment, MyInt = 102
Result of Prefix Increment = 102
After Prefix Increment, MyInt = 102
Result of Postfix Decrement = 101
After Postfix Decrement, MyInt = 100
Result of Prefix Decrement = 100
After Prefix Decrement, MyInt = 100
```

Анализ

Результаты показывают, чем постфиксные операторы отличались от префиксных, присваиваемые в строках 8 и 18. l-значения содержат исходные значения целого числа, то, каким оно было до операций инкремента или декремента. Префиксные операторы в строках 13 и 23, напротив, присвоили 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:

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

Дело в том, что здесь нет присвоения исходного значения и конечный результат в обоих случаях — увеличенное на единицу целое число.

ПРИМЕЧАНИЕ

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

Это правда, по крайней мере, теоретически, поскольку при постфиксных операторах компилятор должен временно хранить исходное значение на случай его присвоения. Влияние на производительность в случае целых чисел незначительно, но в случае некоторых классов этот аргументе мог бы иметь смысл.

Избегайте переполнения, обдуманно выбирая типы данных

Такие типов данных, как `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long` и т.д., имеют ограниченную емкость для содержания чисел. Когда в ходе арифметической операции вы превышаете определенный типом предел, происходит *переполнение* (overflow).

Возьмем, к примеру, тип `unsigned short`. Тип данных `short` использует 16 битов, а следовательно, может содержать значения от 0 до 65535. Когда вы добавляете 1 к значению 65535, хранящемуся в переменной типа `unsigned short`, происходит переполнение и значение превращается в 0. Это очень похоже на спидометр автомобиля: в нем тоже происходит механическое переполнение, когда он может отображать только пять цифр, а автомобиль прошел 99 999 километров.

В данном случае тип `unsigned short` оказался бы неподходящим типом для такого счетчика. Для содержания чисел больше 65 535 следует использовать тип `unsigned int`.

В случае типа `signed short` допустимы целые числа в диапазоне от –32768 до 32767. Добавление 1 к числу 32767 в переменной типа `signed integer` зачастую дает в результате самое большое отрицательное значение, но это уже зависит от компилятора.

Листинг 5.3 демонстрирует ошибки переполнения, которые можно по неосторожности получить при арифметических операциях.

ЛИСТИНГ 5.3. Ошибка переполнения у знаковых и беззнаковых целочисленных переменных

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

Результат

```
Incrementing unsigned short 65535 gives: 0
Incrementing signed short 32767 gives: -32768
```

Анализ

Как можно заметить, непреднамеренные ситуации переполнения приводят к непредсказуемому и не интуитивно понятному поведению приложения. Если по причинам экономии

памяти вы использовали тип unsigned short, то когда потребуется число 65 536, вы фактически получите число 0.

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 MyNum = 20;
bool CheckEquality = (MyNum == 20);          // true
bool CheckInequality = (MyNum != 100);       // true

bool CheckEqualityAgain = (MyNum == 200);    // false
bool CheckInequalityAgain = (MyNum != 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	Описание
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, тип bool. Следующий пример кода демонстрирует применение операторов сравнения, приведенных в табл. 5.1:

```
int MyNum = 20;            // пример целочисленного значения
bool CheckLessThan = (MyNum < 100);                  // true
bool CheckGreaterThan = (MyNum > 100);               // false
bool CheckLessThanEqualTo = (MyNum <= 20);           // true
bool CheckGreaterThanEqualTo = (MyNum >= 20);        // true
bool CheckGreaterThanEqualToAgain = (MyNum >= 100); // false
```

Код листинга 5.4 демонстрирует использование этих операторов при отображении результата на экране.

ЛИСТИНГ 5.4. Операторы равенства и сравнения

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Enter two integers:" << endl;
    int Num1 = 0, Num2 = 0;
    cin >> Num1;
    cin >> Num2;

    bool Equality = (Num1 == Num2);
    cout << "Result of equality test: " << Equality << endl;

    bool Inequality = (Num1 != Num2);
    cout << "Result of inequality test: " << Inequality << endl;

    bool GreaterThan = (Num1 > Num2);
    cout << "Result of " << Num1 << " > " << Num2;
    cout << " test: " << GreaterThan << endl;

    bool LessThan = (Num1 < Num2);
    cout << "Result of " << Num1 << " < " << Num2 << " test: "
         << LessThan << endl;

    bool GreaterThanEquals = (Num1 >= Num2);
    cout << "Result of " << Num1 << " >= " << Num2;
    cout << " test: " << GreaterThanEquals << endl;

    bool LessThanEquals = (Num1 <= Num2);
    cout << "Result of " << Num1 << " <= " << Num2;
    cout << " test: " << LessThanEquals << endl;

    return 0;
}
```

Результат

```
Enter two integers:
365
-24
Result of equality test: 0
Result of inequality test: 1
Result of 365 > -24 test: 1
Result of 365 < -24 test: 0
Result of 365 >= -24 test: 1
Result of 365 <= -24 test: 0
```

Следующий запуск:

```
Enter two integers:
101
101
Result of equality test: 1
Result of inequality test: 0
```

```
Result of 101 > 101 test: 0
Result of 101 < 101 test: 0
Result of 101 >= 101 test: 1
Result of 101 <= 101 test: 1
```

Анализ

Программа отображает результат различных операций в двоичном виде. Интересно отметить в выводе случай, где сравниваются два одинаковых целых числа. Операторы ==, >= и <= дают идентичный результат.

Тот факт, что вывод операторов равенства и сравнения является двоичным, 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, "Ветвление процесса выполнения программ".

Логические операции NOT, AND, OR и XOR

Логическая операция NOT обеспечивается оператором ! и выполняется над одним операндом. Таблица истинности для логической операции NOT, которая просто инвертирует значение логического флага, приведена в табл. 5.2.

ТАБЛИЦА 5.2. Таблица истинности логической операции NOT

Операнд	Результат операции NOT (Операнд)
`False`	`True`
`True`	`False`

Для других операций, таких как AND, OR или XOR, необходимы два операнда. Логическая операция AND возвращает значение `true` только тогда, когда каждый операнд содержит значение `true`. Действие логической операции AND приведено в табл. 5.3.

ТАБЛИЦА 5.3. Таблица истинности логической операции AND

Операнд 1	Операнд 2	Результат операции Операнд 1 AND Операнд 2
`False`	`False`	`False`
`True`	`False`	`False`
`False`	`True`	`False`
`True`	`True`	`True`

Логическая операция AND обеспечивается оператором `&&`.

Логическая операция OR возвращает значение `true` тогда, когда по крайней мере один из операндов содержит значение `true`. Действие логической операции OR приведено в табл. 5.4.

Логическая операция OR обеспечивается оператором `||`.

Логическая операция XOR (exclusive OR, или исключающего ИЛИ) немного отличается от логической операции OR и возвращает значение `true` тогда, когда любой из операндов содержит значение `true`, но не оба. Действие логической операции XOR приведено в табл. 5.5.

ТАБЛИЦА 5.4. Таблица истинности логической операции OR

Операнд 1	Операнд 2	Результат операции Операнд 1 OR Операнд 2
False	False	False
True	False	True
False	True	True
True	True	True

ТАБЛИЦА 5.5. Таблица истинности логической операции XOR

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 OR обеспечивается в языке C++ оператором ^. Результат получается при выполнении операции XOR над битами операндов.

Использование логических операторов C++ NOT (!), AND (&&) и OR (||)

Рассмотрим следующие выражения.

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

В программировании необходима некоторая логическая конструкция, где результат двух операций используется в логическом контексте для принятия решения о последующем направлении потока программы. Язык C++ предоставляет логические операторы AND и OR, которые можно использовать в условных операторах, а следовательно, условно изменять поток выполнения программы.

Листинг 5.5 демонстрирует работу логических операторов AND и OR.

ЛИСТИНГ 5.5. Анализ логических операторов C++ && и ||

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Enter true(1) or false(0) for two operands:" << endl;
 6:     bool Op1 = false, Op2 = false;
 7:     cin >> Op1;
 8:     cin >> Op2;
 9:
10:     cout << Op1 << " AND " << Op2 << " = " << (Op1 && Op2) << endl;
11:     cout << Op1 << " OR " << Op2 << " = " << (Op1 || Op2) << endl;
```

```
12:
13:     return 0;
14: }
```

Результат

```
Enter true(1) or false(0) for two operands:
1
0
1 AND 0 = 0
1 OR 0 = 1
```

Следующий запуск:

```
Enter true(1) or false(0) for two operands:
1
1
1 AND 1 = 1
1 OR 1 = 1
```

Анализ

Программа фактически демонстрирует, что позволяют логические операции AND и OR. Однако она не показывают, как их использовать для принятия решений.

В листинге 5.6 представлена программа, которая, используя условные и логические операторы, выполняет разные строки кода в зависимости от значений, содержавшихся в переменных.

ЛИСТИНГ 5.6. Использование логических операторов NOT (`!`) и AND (`&&`) в условных операторах для изменения потока выполнения

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Use boolean values(0 / 1) to answer the questions"
           << endl;
 6:     cout << "Is it raining? ";
 7:     bool Raining = false;
 8:     cin >> Raining;
 9:
10:     cout << "Do you have buses on the streets? ";
11:     bool Buses = false;
12:     cin >> Buses;
13:
14:     // Условный оператор использует логические операторы AND и NOT
15:     if (Raining && !Buses)
16:         cout << "You cannot go to work" << endl;
17:     else
18:         cout << "You can go to work" << endl;
19:
20:     if (Raining && Buses)
21:         cout << "Take an umbrella" << endl;
```

```
22:
23:     if ((!Raining) && Buses)
24:         cout << "Enjoy the sun and have a nice day" << endl;
25:
26:     return 0;
27: }
```

Результат

```
Use boolean values(0 / 1) to answer the questions
Is it raining? 1
Do you have buses on the streets? 1
You can go to work
Take an umbrella
```

Следующий запуск:

```
Use boolean values(0 / 1) to answer the questions
Is it raining? 1
Do you have buses on the streets? 0
You cannot go to work
```

Последний запуск:

```
Use boolean values(0 / 1) to answer the questions
Is it raining? 0
Do you have buses on the streets? 1
You can go to work
Enjoy the sun and have a nice day
```

Анализ

Код листинга 5.6 использует условные операторы в форме конструкции `if`, которая пока еще не рассматривалась. Но все же попробуйте понять поведение этой конструкции, сопоставив ее с выводом. Строка 15 содержит логическое выражение `(Raining && !Buses)`, которое можно прочитать как "идет дождь И НЕТ автобусов". Логический оператор AND здесь использован для объединения отсутствия автобусов (обозначенного логическим оператором NOT перед наличием автобусов) и присутствия дождя.

ПРИМЕЧАНИЕ

> Более подробная информация о конструкции `if` будет приведена на занятии 6, "Ветвление процесса выполнения программ".

Код листинга 5.7 использует логические операторы NOT (`!`) и OR (`||`) для демонстрации условной обработки.

ЛИСТИНГ 5.7. Использование логических операторов NOT и OR для принятия решения о том, можете ли вы купить автомобиль своей мечты

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
```

```
 5:     cout << "Answer questions with 0 or 1" << endl;
 6:     cout << "Is there a deep discount on your favorite car? ";
 7:     bool Discount = false;
 8:     cin >> Discount;
 9:
10:     cout << "Did you get a fantastic bonus? ";
11:     bool FantasticBonus = false;
12:     cin >> FantasticBonus;
13:
14:     if (Discount || FantasticBonus)
15:         cout << "Congratulations, you can buy that car!" << endl;
16:     else
17:         cout << "Sorry, waiting a while is a good idea" << endl;
18:
19:     return 0;
20: }
```

Результат

```
Answer questions with 0 or 1
Is there a deep discount on your favorite car? 0
Did you get a fantastic bonus? 1
Congratulations, you can buy that car!
```

Следующий запуск:

```
Answer questions with 0 or 1
Is there a deep discount on your favorite car? 0
Did you get a fantastic bonus? 0
Sorry, waiting a while is a good idea
```

Последний запуск:

```
Answer questions with 0 or 1
Is there a deep discount on your favorite car? 1
Congratulations, you can buy that car!
```

Анализ

В строке 14 конструкция `if` используется вместе с конструкцией `else` в строке 16. Конструкция `if` выполняет следующий оператор в строке 15, когда условие `(Discount || FantasticBonus)` возвращает значение `true`. Это выражение содержит логический оператор OR и возвращает значение `true`, даже если нет никакой скидки на ваш любимый автомобиль. Когда выражение возвращает значение `false`, выполняется оператор в строке 17 после конструкции `else`.

Побитовые операторы NOT (~), AND (&), OR (|) и XOR (^)

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 битами операндов. Язык C++ позволяет выполнять такие операции, как NOT, OR, AND и исключающее OR (XOR), в побитовом режиме, позволяя манипулировать отдельными битами, инвертируя их при помощи оператора ~, применяя

операцию OR при помощи оператора |, применяя операцию AND при помощи оператора & и операцию XOR при помощи оператора ^. Последние три выполняются с числами (обычно битовой маской).

Некоторые битовые операции полезны в тех случаях, когда каждый из битов, содержащихся в целом числе, например, определяет состояние некоего флага. Так, целое число размером в 32 бита можно использовать для хранения 32-х логических флагов. Использование побитовых операторов показано в листинге 5.8.

ЛИСТИНГ 5.8. Использование побитовых операторов для выполнения операций NOT, AND, OR и XOR с отдельными битами целого числа

```
 1: #include <iostream>
 2: #include <bitset>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     cout << "Enter a number (0 - 255): ";
 8:     unsigned short InputNum = 0;
 9:     cin >> InputNum;
10:
11:     bitset<8> InputBits (InputNum);
12:     cout << InputNum << " in binary is " << InputBits << endl;
13:
14:     bitset<8> BitwiseNOT = (~InputNum);
15:     cout << "Logical NOT |" << endl;
16:     cout << "~" << InputBits << " = " << BitwiseNOT << endl;
17:
18:     cout << "Logical AND, & with 00001111" << endl;
19:     bitset<8> BitwiseAND = (0x0F & InputNum);
                          // 0x0F шестнадцатеричная форма числа 0001111
20:     cout << "0001111 & " << InputBits << " = " << BitwiseAND << endl;
21:
22:     cout << "Logical OR, | with 00001111" << endl;
23:     bitset<8> BitwiseOR = (0x0F | InputNum);
24:     cout << "00001111 | " << InputBits << " = " << BitwiseOR << endl;
25:
26:     cout << "Logical XOR, ^ with 00001111" << endl;
27:     bitset<8> BitwiseXOR = (0x0F ^ InputNum);
28:     cout << "00001111 ^ " << InputBits << " = " << BitwiseXOR
29:          << endl;
30:
31:     return 0;
32: }
```

Результат

```
Enter a number (0 - 255): 181
181 in binary is 10110101
Logical NOT |
~10110101 = 01001010
Logical AND, & with 00001111
```

```
0001111 & 10110101 = 00000101
Logical OR, | with 00001111
00001111 | 10110101 = 10111111
Logical XOR, ^ with 00001111
00001111 ^ 10110101 = 10111010
```

Анализ

Эта программа использует *набор битов* (`bitset`) — тип, который еще не рассматривался, — для облегчения отображения двоичных данных. Роль класса `std::bitset` здесь исключительно вспомогательная — он помогает с отображением, и не более того. В строках 10, 13, 17 и 22 вы фактически присваиваете целое число объекту набора битов, используемому для отображения того же целочисленного значения в двоичном виде. Операции выполняются с целыми числами. Сначала сосредоточьтесь на выводе, отображающем введенное пользователем исходное число 181 в двоичном виде, а затем перейдите к результату выполнения различных побитовых операторов `~`, `&`, `|` и `^` с этим целым числом. Как можно заметить, побитовое NOT, используемое в строке 13, просто инвертирует отдельные биты. Программа демонстрирует также работу операторов `&`, `|` и `^`, использующих каждый бит двух операндов для создания результата. Сопоставьте полученные результаты с приведенными ранее таблицами истинности, и они станут понятнее.

ПРИМЕЧАНИЕ

> Если хотите узнать больше о манипулировании битовыми флагами в языке C++, обратитесь к занятию 25, "Работа с битовыми флагами при использовании библиотеки STL", там класс `std::bitset` обсуждается подробней.

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

Операторы сдвига перемещают всю последовательность битов вправо или влево, позволяя осуществить умножение и деление на два, а также многие другие действия.

А это 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 приведено в листинге 5.9.

ЛИСТИНГ 5.9. Использование побитового оператора сдвига вправо (>>) для получения четверти и половины значения, а также оператора сдвига влево (<<) для удвоения значения и умножения его на четыре

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
```

```
5:     cout << "Enter a number: ";
6:     int Input = 0;
7:     cin >> Input;
8:
9:     int Half = Input >> 1;
10:    int Quarter = Input >> 2;
11:    int Double = Input << 1;
12:    int Quadruple = Input << 2;
13:
14:    cout << "Quarter: " << Quarter << endl;
15:    cout << "Half: " << Half << endl;
16:    cout << "Double: " << Double << endl;
17:    cout << "Quadruple: " << Quadruple << endl;
18:
19:    return 0;
20: }
```

Результат

```
Enter a number: 16
Quarter: 4
Half: 8
Double: 32
Quadruple: 64
```

Анализ

Пользователь вводит число 16, которое в двоичном представлении выглядит как 1000. В строке 9 осуществляется его смещение вправо на один бит, и получается 0100, что в десятичном виде составляет 8 — фактически половина исходного значения. В строке 10 осуществляется смещение вправо на два бита, 1000 превращается в 00100, что составляет 4. Результат операторов сдвига влево в строках 11 и 12 прямо противоположен. Смещение 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 переносят значения. Кроме того, результат сдвига знаковых чисел зависит от конкретного компилятора. На некоторых из них самый старший разряд при сдвиге влево не присваивается самому младшему биту; последний получает значение нуль.

Составные операторы присвоения

Составные операторы присвоения (compound assignment operator) — это операторы присвоения, где результат операции присваивается операнду слева.

Рассмотрим следующий код:

```
int Num1 = 22;
int Num2 = 5;
Num1 += Num2; // после операции Num1 содержит значение 27
```

Это эквивалентно следующей строке кода:

```
Num1 = Num1 + Num2;
```

Таким образом, результат оператора += — это сумма этих двух операндов, присвоенная затем операнду слева (Num1). Краткий перечень составных операторов присвоения с объяснением их работы приведен в табл. 5.6.

ТАБЛИЦА 5.6. Составные операторы присвоения

Оператор	Применение	Эквивалент
Присвоение с добавлением	`Num1 += Num2;`	`Num1 = Num1 + Num2;`
Присвоение с вычитанием	`Num1 -= Num2;`	`Num1 = Num1 - Num2;`
Присвоение с умножением	`Num1 *= Num2;`	`Num1 = Num1 * Num2;`
Присвоение с делением	`Num1 /= Num2;`	`Num1 = Num1 / Num2;`
Присвоение с делением по модулю	`Num1 %= Num2;`	`Num1 = Num1 % Num2;`
Присвоение с побитовым сдвигом влево	`Num1 <<= Num2;`	`Num1 = Num << Num2;`
Присвоение с побитовым сдвигом вправо	`Num1 >>= Num2;`	`Num1 = Num >> Num2;`
Присвоение с побитовым AND	`Num1 &= Num2;`	`Num1 = Num1 & Num2;`
Присвоение с побитовым OR	`Num1 |= Num2;`	`Num1 = Num1 | Num2;`
Присвоение с побитовым XOR	`Num1 ^= Num2;`	`Num1 = Num1 ^ Num2;`

Применение этих операторов демонстрирует листинг 5.10.

ЛИСТИНГ 5.10. Использование составных операторов присвоения для добавления, вычитания, деления и деления по модулю, а также побитовых операций сдвига, OR, AND и XOR

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Enter a number: ";
 6:     int Value = 0;
 7:     cin >> Value;
 8:
 9:     Value += 8;
10:     cout << "After += 8, Value = " << Value << endl;
11:     Value -= 2;
12:     cout << "After -= 2, Value = " << Value << endl;
13:     Value /= 4;
14:     cout << "After /= 4, Value = " << Value << endl;
15:     Value *= 4;
16:     cout << "After *= 4, Value = " << Value << endl;
17:     Value %= 1000;
18:     cout << "After %= 1000, Value = " << Value << endl;
19:
20:     // Примечание: далее присвоение происходит в пределах cout
21:     cout << "After <<= 1, value = " << (Value <<= 1) << endl;
22:     cout << "After >>= 2, value = " << (Value >>= 2) << endl;
23:
24:     cout << "After |= 0x55, value = " << (Value |= 0x55) << endl;
```

```
15:     cout << "After ^= 0x55, value = " << (Value ^= 0x55) << endl;
16:     cout << "After &= 0x0F, value = " << (Value &= 0x0F) << endl;
17:
18:     return 0;
19:}
```

Результат

```
Enter a number: 440
After += 8, Value = 448
After -= 2, Value = 446
After /= 4, Value = 111
After *= 4, Value = 444
After %= 1000, Value = 444
After <<= 1, value = 888
After >>= 2, value = 222
After |= 0x55, value = 223
After ^= 0x55, value = 138
After &= 0x0F, value = 10
```

Анализ

Обратите внимание, как последовательно изменяется значение переменной `Value` по мере применения в программе различных операторов присвоения. Каждая операция осуществляется с использованием переменной `Value`, а ее результат снова присваивается переменной `Value`. Следовательно, в строке 9 введенное пользователем значение 440 добавляется к 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, занятого переменной

Этот оператор возвращает объем памяти в байтах, использованной 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. Этот оператор не может быть определен программистом, а следовательно, не может быть перегружен.

Более подробная информация о собственных операторах приведена на занятии 12, "Типы операторов и их перегрузка".

Листинг 5.11 демонстрирует применение оператора `sizeof` для определения объема памяти, занятого массивом. Кроме того, можно вернуться к листингу 3.4 и проана-

лизировать применение оператора sizeof для определения объема памяти, занятого переменными наиболее распространенных типов.

ЛИСТИНГ 5.11. 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 << "Use sizeof of determine memory occupied by arrays"
           << endl;
 6:     int MyNumbers [100] = {0};
 7:
 8:     cout << "Bytes occupied by an int: " << sizeof(int) << endl;
 9:     cout << "Bytes occupied by array MyNumbers: "
           << sizeof(MyNumbers) << endl;
10:     cout << "Bytes occupied by each element: "
           << sizeof(MyNumbers[0]) << endl;
11:
12:     return 0;
13: }
```

Результат

```
Use sizeof of determine memory occupied by arrays
Bytes occupied by an int: 4
Bytes occupied by array MyNumbers: 400
Bytes occupied by each element: 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 когда их тип создан вами самостоятельно. Вы использовали бы результат выполнения оператора sizeof для определения объема памяти, занятого каждым объектом, а затем динамически зарезервировали бы для него память, используя оператор new.

Более подробная информация о динамическом распределении памяти приведена на занятии 8, "Указатели и ссылки".

Приоритет операторов

Возможно, вы помните из школьного курса, что арифметические операции имеют порядок, в котором они должны выполняться в сложном арифметическом выражении.

В языке C++ используются такие операторы и выражения:

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

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

ТАБЛИЦА 5.7. Приоритет операторов

№	Название	Оператор
1	Область видимости	`::`
2	Прямое и косвенное обращение к члену класса, вызов функции, постфиксный инкремент и декремент	`. -> () ++ --`
3	Префиксный инкремент и декремент, инверсия и не унарные "минус" и "плюс", получение адреса и ссылки, а также операторы `new`, `new[]`, `delete`, `delete[]`, `casting`, `sizeof()`	`++ -- ^ ! - + & * ()`
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 AND	`&`
11	Побитовое исключающее OR	`^`
12	Побитовое OR	`|`
13	Логическое AND	`&&`
14	Логическое OR	`||`
15	Троичный условный оператор	`?:`
16	Операторы присвоения	`= *= /= %= += -= <<= >>=`
17	Запятая	`,`

Давайте еще раз рассмотрим сложное выражение, приведенное для примера ранее:

```
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 все сводится к следующему:

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

Поскольку сложение и вычитание имеют приоритет над сдвигом, дальнейшее сводится к следующему:

```
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 сказать, что выражение сводится к `295 * 4`, а результат составляет 1180.

ВНИМАНИЕ!

Чтобы сделать код понятней, используйте круглые скобки.

Приведенное выше выражение просто плохо написано. Это компилятору все просто понять, но написанный код должен быть понятен и людям.

То же выражение будет намного понятней, если записать его так:

```
int MyNumber = ((10 * 30) - (5 * 5) + 20) << 2; // не оставляйте никаких
                                                //поводов для сомнения
```

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте круглые скобки, чтобы сделать ваш код понятнее	**Не создавайте** сложные выражения, полагающиеся на таблицу приоритета операторов; ваш код должен быть понятен и людям
Используйте правильные типы переменных во избежание ситуаций переполнения	**Не заблуждайтесь**, что выражения *++Переменная* и *Переменная++* равнозначны. Они различаются при использовании в присвоении
Помните, что все l-значения (например, переменные) могут быть r-значениями, но не все r-значения (например, `"Hello World"`) могут быть l-значениями	

Резюме

На этом занятии вы узнали, что такое команды, операторы и выражения языка C++. Вы научились выполнять простые арифметические операции, такие как сложение, вычитание, умножение и деление. Был также приведен краткий обзор таких логических операторов, как NOT, AND, OR и XOR. Мы рассмотрели логические операторы `!`, `&&` и `||`, используемые в условных выражениях, и такие побитовые операторы, как `~`, `&`, `|` и `^`, которые позволяют манипулировать данными по одному биту за раз.

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

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, следует выбрать более емкий тип, например `unsigned int`, если вы не уверены абсолютно, что значение останется значительно ниже его предела.

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

  Да! Поскольку вы не использовали круглые скобки, чтобы сделать эту строку проще для понимания поддерживающим программистам. Добавление комментария тоже не повредило бы.

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

```
int Num1 = 5, Num2 = 2;
int result = Num1 / Num2;
```

При запуске результат содержит значение 2. Разве так и должно быть?

Вовсе нет. Целые числа не предназначены для содержания десятичных данных. Поэтому результат этой операции 2, а не 2.5. Если вы ожидаете результат 2.5, 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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "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. `sizeof(...)` — это функция?
5. Я должен вычислить удвоенное число, добавить к нему 5, а затем удвоить его снова. Все ли я сделал правильно?

```
int Result = number << 1 + 5 << 1;
```

6. Каков результат операции XOR, когда 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

Ветвление процесса выполнения программ

Большинство приложений должно действовать по-разному в зависимости от ситуации или введенных пользователем данных. Чтобы позволить приложению реагировать по-другому, необходимы условные операторы, выполняющие разные сегменты кода в разных ситуациях.

На сегодняшнем занятии.

- Как заставить программу вести себя по-разному в определенных условиях.
- Как неоднократно выполнить раздел кода в цикле.
- Как лучше контролировать поток выполнения в цикле.

ое выполнение с использованием укции `if...else`

ы, которые вы видели и создавали до сих пор, имели последовательный по-нения — сверху вниз. Все строки выполнялись, и ни одна не игнорирова- последовательное выполнение всех строк кода редко используется в при-

жим, программа должна умножать два числа, если пользователь нажимает , или суммировать их, если он нажимает другую клавишу.
о заметить на рис. 6.1, не все участки кода выполняются при каждом запу-ьзователь нажимает клавишу <м>, выполняется код, умножающий два числа. ит нечто другое, выполняется код суммирования. Ни при какой ситуации обе ьности кода не выполняются.

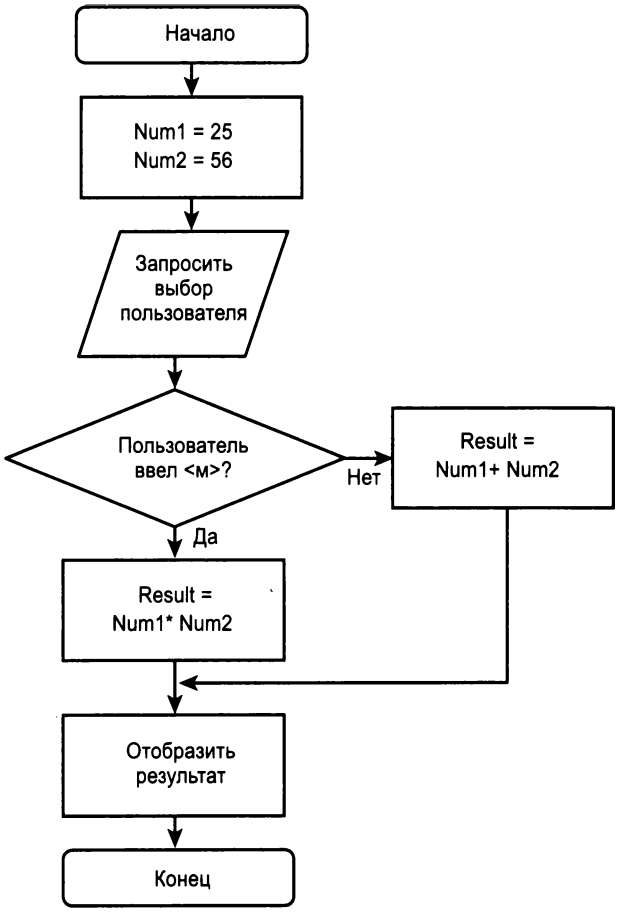


РИС. 6.1. Пример условного выполнения кода на основе пользовательского ввода

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

Условное выполнение кода реализовано в языке C++ на базе конструкции `if ... else`, синтаксис который выглядит следующим образом:

```
if (условное выражение)
    Сделать нечто, когда условное выражение возвращает true;
else // Необязательно
    Сделать нечто другое, когда условное выражение возвращает false;
```

Таким образом, конструкция `if ... else`, позволяющая программе умножать, если пользователь нажимает клавишу <м>, и добавлять в противном случае, выглядит следующим образом:

```
if (UserSelection == 'm')
    Result = Num1 * Num2; // умножение
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, любое выражение, возвращающее любое ненулевое число, положительное или отрицательное, по существу рассматривается как возвращающее значение `true`, когда оно используется в условном выражении.

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 << "Enter two integers: " << endl;
 6:     int Num1 = 0, Num2 = 0;
 7:     cin >> Num1;
 8:     cin >> Num2;
 9:
10:     cout << "Enter \'m\' to multiply, anything else to add: ";
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 << "Result is: " << Result << endl;
21:
22:     return 0;
23: }
```

Результат

```
Enter two integers:
25
56
Enter 'm' to multiply, anything else to add: m
Result is: 1400
```

Следующий запуск:

```
Enter two integers:
25
56
Enter 'm' to multiply, anything else to add: a
Result is: 81
```

Анализ

Обратите внимание на использование оператора `if` в строке 15 и оператора `else` в строке 17. В строке 15 мы инструктируем компилятор выполнить умножение, если следующее за оператором `if` выражение `(UserSelection == 'm')` истинно (возвращает значение `true`), или выполнять 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-другому при иной ситуации.

ПРИМЕЧАНИЕ

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

ВНИМАНИЕ!

Если бы строка 15 в листинге 6.1 выглядела так:

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

то конструкция `if` была бы бессмысленна, поскольку она завершилась бы в той же строке пустым оператором (точка с запятой). Будьте внимательны и избегайте такой ситуации, поскольку вы не получите от компилятора сообщения об ошибке, если за оператором `if` не последует часть `else`.

Некоторые хорошие компиляторы в такой ситуации предупреждают об этом сообщением "`empty control statement`" (пустой управляющий оператор).

Условное выполнение нескольких операторов

Если вы хотите выполнить несколько операторов в зависимости от условий, необходимо заключить их в блок операторов. По существу, это фигурные скобки ({ . . . }), включающие несколько операторов, которые будут выполняться как блок. Рассмотрим пример:

```
if (условие)
{
    // блок при истинном условии
    Оператор 1;
    Оператор 2;
}
else
{
    // блок при ложном условии
    Оператор 3;
    Оператор 4;
}
```

Такие блоки называются также *составными операторам* (compound statement).

На занятии 4, "Массивы и строки", объяснялась опасность использования статических массивов и пересечения их границ. Чаще всего эта проблема проявляется в символьных массивах. При записи строки в символьный массив или при его копировании важно проверять, является ли массив достаточно большим, чтобы содержать все символы. Листинг 6.2 демонстрирует выполнение такой проверки, позволяющей избежать переполнения буфера.

ЛИСТИНГ 6.2. Проверка емкости перед копированием строки в символьный массив

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:     char Buffer[20] = {'\0'};
 7:
 8:     cout << "Enter a line of text: " << endl;
 9:     string LineEntered;
10:     getline (cin, LineEntered);
11:
12:     if (LineEntered.length() < 20)
13:     {
14:         strcpy(Buffer, LineEntered.c_str());
15:         cout << "Buffer contains: " << Buffer << endl;
16:     }
17:
18:     return 0;
19: }
```

Результат

```
Enter a line of text:
This fits buffer!
Buffer contains: This fits buffer!
```

Анализ

Обратите внимание, как в строке 12 проверяются длины строки и буфера перед копированием. Интересным в этой проверке является также присутствие блока операторов в строках 13–16 (называемого также ставным оператором).

ВНИМАНИЕ!

В конце строки `if (условие)` НЕТ точки с запятой. Это сделано намеренно и гарантирует, что оператор после оператора `if` выполнится в случае истинности условия.

Следующий фрагмент кода вполне корректен:

```
if(условие);
    оператор;
```

Но вы не получите ожидаемого результата, поскольку конструкция `if` завершается последующей точкой с запятой и вне зависимости от условия не делает ничего, зато следующий оператор выполняется всегда.

Вложенные операторы `if`

Нередки ситуации, когда необходимо проверить несколько разных условий, некоторые из которых зависят от результата оценки предыдущего условия. Язык C++ позволяет вкладывать операторы `if` для выполнения таких требований.

Вложенные операторы `if` имеют следующий синтаксис:

```
if (условие1)
{
    СделатьНечто1;
    if(условие2)
        СделатьНечто2;
    else
        СделатьНечтоДругое2;
}
else
    СделатьНечтоДругое1;
```

Считайте приложение, подобное представленному в листинге 6.1, где пользователь, нажимая клавишу <d> или <m>, может указать приложению, следует ли поделить или умножить значения. Далее, деление должно быть разрешено, только если делитель отличается от нуля. Поэтому в дополнение к проверке вводимой пользователем команды следует проверить, не является ли делитель нулем, когда пользователь требует деления. Для этого в листинге 6.3 используется вложенная конструкция `if`.

Листинг 6.3. Использование вложенных операторов `if` в приложении умножения или деления чисел

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

```
 6:     float Num1 = 0, Num2 = 0;
 7:     cin >> Num1;
 8:     cin >> Num2;
 9:
10:     cout << "Enter 'd' to divide, anything else to multiply: ";
11:     char UserSelection = '\0';
12:     cin >> UserSelection;
13:
14:     if (UserSelection == 'd')
15:     {
16:         cout << "You want division!" << endl;
17:         if (Num2 != 0)
18:         {
19:             cout << "No div-by-zero, proceeding to calculate"
                    << endl;
20:             cout << Num1 << " / " << Num2 << " = " << Num1 / Num2
                    << endl;
21:         }
22:         else
23:             cout << "Division by zero is not allowed" << endl;
24:     }
25:     else
26:     {
27:         cout << "You want multiplication!" << endl;
28:         cout << Num1 << " x " << Num2 << " = " << Num1 * Num2
                 << endl;
29:     }
30:
31:     return 0;
32: }
```

Результат

```
Enter two numbers:
45
9
Enter 'd' to divide, anything else to multiply: m
You want multiplication!
45 x 9 = 405
```

Следующий запуск:

```
Enter two numbers:
22
7
Enter 'd' to divide, anything else to multiply: d
You want division!
No div-by-zero, proceeding to calculate
22 / 7 = 3.14286
```

Последний запуск:

```
Enter two numbers:
365
0
```

```
Enter 'd' to divide, anything else to multiply: d
You want division!
Division by zero is not allowed
```

Анализ

Результаты содержат вывод трех запусков программы с тремя разными наборами ввода. Как можно заметить, программа использовала различные пути выполнении кода для каждого из трех запусков. По сравнению с листингом 6.1 эта программа имеет довольно много изменений.

- Числа хранятся в переменных типа `float`, способных хранить десятичные числа, которые понадобятся при делении.
- Условие оператора `if` отличается от использованного в листинге 6.1. Вы больше не проверяете, нажал ли пользователь клавишу <м>; выражение `(UserSelection == 'd')` в строке 14 возвращает значение `true`, если пользователь ввел значение `d`. Если это так, то затребовано деление.
- С учетом того, что эта программа делит два числа и делитель вводится пользователем, важно удостовериться, что делитель не является ненулевым. Это осуществляется в строке 17 вложенным оператором `if`.

Таким образом, эта программа демонстрирует, как вложенные конструкции `if` могут оказаться очень полезными при решении различных задач, связанных с оценкой нескольких параметров.

СОВЕТ

> Отступы со смещением, сделанные в коде, необязательны, но они существенно содействуют удобочитаемости вложенных конструкций `if`.

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 << "Find what days of the week are named after!" << endl;
17:     cout << "Enter a number for a day (Sunday = 0): ";
18:
19:     int Day = Sunday; // Инициализация днем Sunday
20:     cin >> Day;
21:
22:     if (Day == Sunday)
23:         cout << "Sunday was named after the Sun" << endl;
24:     else if (Day == Monday)
25:         cout << "Monday was named after the Moon" << endl;
26:     else if (Day == Tuesday)
27:         cout << "Tuesday was named after Mars" << endl;
28:     else if (Day == Wednesday)
29:         cout << "Wednesday was named after Mercury" << endl;
30:     else if (Day == Thursday)
31:         cout << "Thursday was named after Jupiter" << endl;
32:     else if (Day == Friday)
33:         cout << "Friday was named after Venus" << endl;
34:     else if (Day == Saturday)
35:         cout << "Saturday was named after Saturn" << endl;
36:     else
37:         cout << "Wrong input, execute again" << endl;
38:
39:     return 0;
40: }
```

Результат

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 5
Friday was named after Venus
```

Следующий запуск:

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 9
Wrong input, execute again
```

Анализ

Конструкция `if...else...if`, используемая в строках 22–37, проверяет ввод пользователя и создает соответствующий вывод. Вывод при втором запуске свидетельствует, что программа в состоянии указать пользователю, что он ввел номер вне диапазона 0–6, а следовательно, не соответствующий 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 перечисляемой константы `DaysOfWeek`, объявленной в строке 5 и используемой повсюду в операторе `if`. Можно было бы просто сравнивать пользовательский ввод с целочисленными значениями; так, например, 0 соответствовал бы воскресенью и т.д. Однако использование перечисляемой константы `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 возможных констант 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 МеткаA:
    СделатьНечто;
    break;
case МеткаB:
    СделатьНечтоДругое;
    break;
// И так далее...
default:
    СделатьНечтоЕслиВыражениеНеСоответствуетНичемуВыше;
    break;
}
```

Код вычисляет результат выражения и сравнивает его на равенство с каждой из меток частей case ниже, каждая из которых должна быть константой, а затем выполняет код после этой метки. Когда результат выражения не соответствует метке *МеткаA*, он сравнивается с меткой *МеткаB*. Если значения совпадают, выполняется часть *СделатьНечтоДругое*. Выполнение продолжается до тех пор, пока не встретится оператор break. Вы впервые встречаете оператор break. Он означает выход из блока кода. Операторы break не обязательны; однако без них выполнение продолжится в коде следующей метки и так далее, до конца блока. Такого явления, как правило, желательно избегать. Часть default также является необязательной, она выполняется тогда, когда результат выражения не соответствует ни одной из меток в конструкции switch-case.

СОВЕТ

> Конструкции switch-case хорошо подходят для использования с перечисляемыми константами. Ключевое слово enum было введено на занятии 3, "Использование переменных, объявление констант".

Код листинга 6.5 является эквивалентом программы дней недели из листинга 6.4, но с использованием конструкции switch-case и перечисляемых констант.

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 << "Find what days of the week are named after!" << endl;
17:     cout << "Enter a number for a day (Sunday = 0): ";
18:
19:     int Day = Sunday; // Инициализация днем Sunday
20:     cin >> Day;
21:
22:     switch(Day)
23:     {
24:     case Sunday:
25:         cout << "Sunday was named after the Sun" << endl;
26:         break;
27:
28:     case Monday:
29:         cout << "Monday was named after the Moon" << endl;
30:         break;
31:
32:     case Tuesday:
33:         cout << "Tuesday was named after Mars" << endl;
34:         break;
35:
36:     case Wednesday:
37:         cout << "Wednesday was named after Mercury" << endl;
38:         break;
39:
40:     case Thursday:
41:         cout << "Thursday was named after Jupiter" << endl;
42:         break;
43:
44:     case Friday:
45:         cout << "Friday was named after Venus" << endl;
46:         break;
47:
48:     case Saturday:
49:         cout << "Saturday was named after Saturn" << endl;
50:         break;
51:
52:     default:
53:         cout << "Wrong input, execute again" << endl;
54:         break;
55:     }
56:
57:     return 0;
58: }
```

Результат

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 5
Friday was named after Venus
```

Следующий запуск:

```
Find what days of the week are named after!
Enter a number for a day (Sunday = 0): 9
Wrong input, execute again
```

Анализ

Строки 22–55 содержат конструкцию `switch-case`, осуществляющую различный вывод в зависимости от того, введено ли пользователем целое число, содержавшееся в переменной `Day`. Когда пользователь вводит число 5, приложение проверяет выражение `Day` оператора `switch`, которое составляет 5, и сравнивает с четырьмя метками, являющимися перечисляемыми константами от `Sunday` (значение 0) до `Thursday` (значение 4), пропуская код ниже каждого из них, поскольку ни один из них не равен 5. По достижении метки `Friday`, значение 5 которой равно выражению `Day` оператора `switch`, выполняется содержащийся ниже ее код, пока не встретится оператор `break`, предписывающий выйти из конструкции `switch`. При следующем запуске, когда вводится недопустимое значение, достигается часть `default` и выполнение кода ниже его отображает сообщение с просьбой повторить ввод.

В этой программе конструкция `switch-case` используется для получения такого же вывода, что и созданного конструкцией `if...else...if` в листинге 6.4. Все же версия с конструкцией `switch-case` выглядит немного более структурированной и, возможно, лучше подходящей к ситуациям, когда необходимо сделать больше, чем просто вывести строку на экран (в этом случае вы заключали бы код ниже меток в фигурные скобки, создавая блоки).

Троичный условный оператор (?:)

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

Троичный условный оператор, называемый также *условным оператором* и *троичным оператором*, использует три операнда:

```
(логическое условное выражение) ? выражение1 при true : выражение2 при false;
```

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

```
int Max = (Num1 > Num2)? Num1 : Num2;
// Max содержит большее число из Num1 и Num2
```

В листинге 6.6 показана условная обработка с использованием оператора `:?`.

ЛИСТИНГ 6.6. Использование условного оператора (?:) для поиска большего из двух чисел

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

```
 2:
 3: int main()
 4: {
 5:     cout << "Enter two numbers" << endl;
 6:     int Num1 = 0, Num2 = 0;
 7:     cin >> Num1;
 8:     cin >> Num2;
 9:
10:     int Max = (Num1 > Num2)? Num1 : Num2;
11:     cout << "The greater of " << Num1 << " and " \
12:          << Num2 << " is: " << Max << endl;
13:
14:     return 0;
15: }
```

Результат

```
Enter two numbers
365
-1
The greater of 365 and -1 is: 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, троичный условный оператор сэкономил несколько строк кода! Однако экономия строк кода не должна быть приоритетом. Одни программисты предпочитают троичные условные операторы, другие нет. Главное, чтобы пути выполнения кода были легко понятны.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте константы и перечисления для выражений оператора `switch`, чтобы сделать код читабельным	**Не добавляйте** два оператора `case` с одинаковой меткой — это не имеет смысла и не будет компилироваться
Используйте раздел `default` оператора `switch`, только если не уверены в его совершенной ненужности	**Не усложняйте** свои операторы `case`, отказавшись от оператора `break` и разрешив последовательное выполнение, поскольку последующее перемещение оператора `case` может нарушить порядок выполнения кода
Проверяйте, не забыли ли вы включить оператор `break` в конец каждого оператора `case`	**Не используйте** в операторах `:?` сложные условия или выражения

Выполнение кода в циклах

Недавно вы узнали, как заставить программу вести себя по-разному, когда переменные содержат разные значения. Например, код листинга 6.2 осуществлял умножение, когда пользователь нажал клавишу <m>, а в противном случае — суммирование. Но что если пользователь не хочет, чтобы программа закончила выполнение? Что если он хочет выполнить еще одну операцию суммирования или умножения, или возможно еще пять? Вот когда необходимо повторное выполнение уже существующего кода.

Вот когда программе необходим цикл.

Рудиментарный цикл с использованием оператора goto

Как и подразумевает название оператора `goto`, он приказывает указателю команд продолжать исполнение с определенной точки в коде. Вы можете использовать его для перехода назад и повторного выполнения определенных операторов. Синтаксис оператора `goto` таков:

```
SomeFunction()
{
    JumpToPoint: // Называется меткой
    CodeThatRepeats;
    goto JumpToPoint;
}
```

Вы объявляете метку `JumpToPoint` и используете оператор `goto` для повторного исполнения кода с этого момента, как показано в листинге 6.7. Если вы не вызываете оператор `goto` с учетом условия, способного вернуть значение `false` при определенных обстоятельствах, или если повторяемый код содержит оператор `return`, выполняемый при определенных условиях, то часть кода между командой `goto` и меткой будет повторяться бесконечно и не позволит программе завершиться.

ЛИСТИНГ 6.7. Запрос пользователю, не хочет ли он повторить вычисления, используя оператор `goto`

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     JumpToPoint:
 6:     int Num1 = 0, Num2 = 0;
 7:
 8:     cout << "Enter two integers: " << 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 << "Do you wish to perform another operation (y/n)?"
            << endl;
16:     char Repeat = 'y';
```

```
17:     cin >> Repeat;
18:
19:     if (Repeat == 'y')
20:         goto JumpToPoint;
21:
22:     cout << "Goodbye!" << endl;
23:
24:     return 0;
25: }
```

Результат

```
Enter two integers:
56
25
56 x 25 = 1400
56 + 25 = 81
Do you wish to perform another operation (y/n)?
y
Enter two integers:
95
-47
95 x -47 = -4465
95 + -47 = 48
Do you wish to perform another operation (y/n)?
n
Goodbye!
```

Анализ

Обратите внимание на основное различие между кодом листинга 6.7 и листинга 6.1: 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, желает ли он выполнить другую операцию. Код, фактически обеспечивающий это повторение, находится в строке 20, где вызывается оператор `goto`, если пользователь вводит символ `'y'`. В результате использования оператора `goto` в строке 20 программа переходит к метке `JumpToPoint`, объявленной в строке 5, что фактически перезапускает программу.

ВНИМАНИЕ!

Использование оператора `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 также переменные в непредсказуемых состояниях.

Тяжелый случай программы, использующей операторы `goto`, называется *запутанная программа* (spaghetti code). Объясняемые на следующих страницах циклы `while`, `do...while` и `for` вполне позволяют избежать использования оператора `goto`.

Единственная причина упоминания здесь оператора `goto` в том, чтобы объяснить код, который его использует.

Цикл while

Ключевое слово while языка C++ позволяет получить то, что оператор goto делал в листинге 6.7, но правильным способом. Синтаксис таков:

```
while(выражение)
{
    // Результат выражения = true
    БлокОператоров;
}
```

Выполнение блока операторов повторяется до тех пор, пока *выражение* возвращает значение true. Очень важно, чтобы в коде была ситуация, где *выражение* возвращает 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 оператора goto.

ЛИСТИНГ 6.8. Использование цикла while позволяет пользователю повторно выполнять вычисления

```
 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 << "Enter the two integers: " << 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 << "Press x to exit(x) or any other key to recalculate"
                 << endl;
18:         cin >> UserSelection;
19:     }
20:
21:     cout << "Goodbye!" << endl;
22:
23:     return 0;
24: }
```

Результат

```
Enter the two integers:
56
```

```
25
56 x 25 = 1400
56 + 25 = 81
Press x to exit(x) or any other key to recalculate
r
Enter the two integers:
365
-5
365 x -5 = -1825
365 + -5 = 360
Press x to exit(x) or any other key to recalculate
x
Goodbye!
```

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 возвращает значение `true`. Для обеспечения первого запуска символьная переменная `UserSelection` инициализируется в строке 5 значением `'i'`. Это должно было быть любым значением, кроме `'x'` (иначе условие не будет выполняться с самого первого цикла и приложение закончит выполнение, не позволив пользователю ничего сделать). Первый запуск очень прост — у пользователя в строке 17 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 оператора `while` в строке 7 и цикл повторяется, если пользователь ввел значение, отличное от `'x'`. Если пользователь нажал клавишу <x>, то выражение в строке 7 вернет значение `false` и цикл `while` окончится. Приложение после этого также закончит работу, вежливо попрощавшись.

ПРИМЕЧАНИЕ

Цикл также называется *итерацией* (iteration). Выражения, задействовавшие циклы `while`, `do...while` и `f_or`, также называют итерационными выражениями.

Цикл do...while

Бывают ситуации (как в листинге 6.8), когда определенному сегменту кода, повторяемому в цикле, необходимо гарантировать выполнение по крайней мере однажды. Для этого используется цикл `do...while`.

Его синтаксис таков:

```
do
{
    БлокОператоров; // выполняется как минимум раз
} while(условие);   // закончить цикл, если условие ложно
```

Обратите внимание, что строка, содержащая часть `while(`*условие*`)`, заканчивается точкой с запятой. Это является отличием от цикла `while`, в котором точка с запятой фактически завершила бы цикл в первой строке, приведя к пустому оператору.

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 << "Enter the two integers: " << 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 << "Press x to exit(x) or any other key to recalculate"
                 << endl;
17:         cin >> UserSelection;
18:     } while (UserSelection != 'x');
19:
20:     cout << "Goodbye!" << endl;
21:
22:     return 0;
23: }
```

Результат

```
Enter the two integers:
654
-25
654 x -25 = -16350
654 + -25 = 629
Press x to exit(x) or any other key to recalculate
m
Enter the two integers:
909
101
909 x 101 = 91809
909 + 101 = 1010
Press x to exit(x) or any other key to recalculate
x
Goodbye!
```

Анализ

Эта программа очень похожа по действию и выводу на предыдущую. Действительно, единственное различие — в ключевом слове do в строке 6 и использовании цикла while в строке 18. Выполнение кода происходит последовательно, одна строка за другой, пока в строке 18 не встретится оператор while, проверяющий выражение (UserSelection = 'x'). Когда оно возвращает значение true (т.е. пользователь не нажал клавишу <x> для выхода), цикл повторяется. Когда выражение возвращает значение false (т.е. пользователь нажал клавишу <x>), выполнение покидает цикл и продолжается до окончания приложения, включая вывод прощания.

Цикл for

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

Синтаксис цикла for таков:

```
for (выражение инициализации, выполняемое только раз;
     условие выхода, проверяемое в начале каждого цикла;
     выражение цикла, выполняемое в конце каждого цикла)
{
    БлокОператоров;
}
```

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

В листинге 6.10 показан эффективный способ доступа к элементам массива при помощи цикла for.

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

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     const int ARRAY_LENGTH = 5;
7:     int MyInts[ARRAY_LENGTH] = {0};
8:
9:     cout << "Populate array of " << ARRAY_LENGTH << " integers"
10:         << endl;
11:
12:    for (int ArrayIndex = 0; ArrayIndex < ARRAY_LENGTH; ++ArrayIndex)
13:    {
14:       cout << "Enter an integer for element " << ArrayIndex << ": ";
15:       cin >> MyInts[ArrayIndex];
16:    }
17:
```

```
16:     cout << "Displaying contents of the array: " << endl;
17:
18:     for (int ArrayIndex = 0; ArrayIndex < ARRAY_LENGTH; ++ArrayIndex)
19:         cout << "Element " << ArrayIndex << " = "
                 << MyInts[ArrayIndex] << endl;
20:
21:     return 0;
22: }
```

Результат

```
Populate array of 5 integers
Enter an integer for element 0: 365
Enter an integer for element 1: 31
Enter an integer for element 2: 24
Enter an integer for element 3: -59
Enter an integer for element 4: 65536
Displaying contents of the array:
Element 0 = 365
Element 1 = 31
Element 2 = 24
Element 3 = -59
Element 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 `ArrayIndex` для доступа к элементам массива. Значение этой переменной увеличивается в конце каждого цикла; поэтому на следующей итерации цикла она позволяет обратиться к следующему элементу. Среднее выражение в цикле `for` — это условие выхода. Оно проверяет, находится ли значение переменной `ArrayIndex`, увеличенное в конце каждого цикла, все еще в пределах границ массива (сравнивая его со значением `ARRAY_LENGTH`). Этим гарантируется, что цикл `for` никогда не превысит длину массива.

ПРИМЕЧАНИЕ

Такая переменная, как `ArrayIndex`, 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, которая была объявлена повторно, фактически является новой переменной.

Однако применение инициализации, условия выхода и выражения, выполняемого в конце каждого цикла, является необязательным. Вполне возможно получить цикл `for` без некоторых или любого из них, как показано в листинге 6.11.

Листинг 6.11. Использование цикла `for` без выражения цикла для повторения вычислений до просьбы пользователя

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:     // без выражения цикла (третье выражение пропущено)
 7:     for(char UserSelection = 'm'; (UserSelection != 'x'); )
 8:     {
 9:         cout << "Enter the two integers: " << 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 << "Press x to exit or any other key to recalculate"
                 << endl;
18:         cin >> UserSelection;
19:     }
20:
21:     cout << "Goodbye!" << endl;
22:
23:     return 0;
24: }
```

Результат

```
Enter the two integers:
56
25
56 x 25 = 1400
56 + 25 = 81
Press x to exit or any other key to recalculate
m
Enter the two integers:
789
-36
789 x -36 = -28404
789 + -36 = 753
Press x to exit or any other key to recalculate
x
Goodbye!
```

Анализ

Это идентично коду листинга 6.8, который использовал цикл `while`; единственное отличие в использовании цикла `for` в строке 8. Самое интересное в этом цикле `for` то, что он содержит только выражение инициализации и условие выхода, без возможности изменять значения переменной в конце каждого цикла.

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 Index = 0, AnotherInt = 5; Index < ARRAY_LENGTH;
++Index, --AnotherInt)
```

Обратите внимание на новую переменную `AnotherInt`, которая инициализируется значением 5.

В выражении цикла, выполняемом на каждой итерации, вполне можно осуществлять и декремент.

Изменение поведения цикла с использованием операторов `continue` и `break`

В некоторых случаях (особенно в сложных циклах с большим количеством параметров в условии) вы можете не суметь грамотно сформулировать условие выхода из цикла, тогда вам придется изменять поведение программы уже в пределах цикла. В этом могут помочь операторы `continue` и `break`.

Оператор `continue` позволяет возобновить выполнение с вершины цикла. Он просто пропускает код, расположенный в блоке 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 повторной проверкой условия выхода выполняется выражение цикла (третье выражение в операторе `for`, которое обычно увеличивает значение счетчика).

Оператор `break` осуществляет выход из блока цикла, фактически завершая цикл, в котором он был вызван.

ВНИМАНИЕ!

Обычно программисты ожидают, что пока условия цикла выполняются, выполняется и весь код в цикле. Операторы `continue` и `break` изменяют это поведение и могут привести к интуитивно непонятному коду.

Поэтому операторы `continue` и `break` следует использовать только тогда, когда вы на самом деле не можете сообразить, как правильно и эффективно организовать цикл, не используя их.

Следующий далее листинг 6.12 демонстрирует использование оператора `continue` для запроса у пользователя повторного ввода обрабатываемых чисел и оператора `break` для выхода из цикла.

Циклы, которые не заканчиваются никогда, т.е. бесконечные циклы

Помните, что у циклов `while`, `do...while` и `for` есть условия, результат `false` вычисления которых 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 while зафиксировано в true
{
    СделатьНечтоНеоднократно;
}
```

Бесконечный цикл `do...while` выглядит так:

```
do
{
    СделатьНечтоНеоднократно;
} while(true); // выражение do...while никогда не вернет false
```

Бесконечный цикл `for` можно создать следующим образом:

```
for (;;) // нет условия выхода, значит, цикл for бесконечный
{
    СделатьНечтоНеоднократно;
}
```

Как ни странно, но у таких циклов действительно есть цель. Представьте операционную систему, которая должна непрерывно проверять, подключено ли устройство USB к порту USB. Это действие должно выполняться регулярно, пока запущена операционная система. Такие случаи гарантируют популярность бесконечных циклов.

Контроль бесконечных циклов

Для выхода из бесконечного цикла (скажем, перед завершением работы операционной системы в предыдущем примере) вы можете вставить оператор `break` (как правило, в пределах блока `if (`*условие*`)`).

Вот пример использования оператора `break` для контроля бесконечного цикла `while`:

```
while(true) // выражение while зафиксировано в true
{
    СделатьНечтоНеоднократно;
    if(выражение)
        break; // выход из цикла, когда выражение возвращает true
}
```

Использование оператора `break` в бесконечном цикле `do..while`:

```
do
{
    СделатьНечтоНеоднократно;
    if(выражение)
        break; // выход из цикла, когда выражение возвращает true
} while(true); // выражение do...while никогда не вернет false
```

Использование оператора `break` в бесконечном цикле `for`:

```
for (;;) // нет условия выхода, значит, цикл for бесконечный
{
    СделатьНечтоНеоднократно;
    if(выражение)
        break; // выход из цикла, когда выражение возвращает true
}
```

Листинг 6.12 демонстрирует использование операторов `continue` и `break` для контроля критерия выхода из бесконечного цикла.

ЛИСТИНГ 6.12. Использование оператора `continue` для перезапуска и оператора `break` для выхода из бесконечного цикла `for`

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     for(;;) // бесконечный цикл
 6:     {
 7:         cout << "Enter two integers: " << endl;
 8:         int Num1 = 0, Num2 = 0;
 9:         cin >> Num1;
10:         cin >> Num2;
11:
12:         cout << "Do you wish to correct the numbers? (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 << "Press x to exit or any other key to recalculate"
                 << endl;
23:         char UserSelection = '\0';
24:         cin >> UserSelection;
25:
26:         if (UserSelection == 'x')
27:             break; // выход из бесконечного цикла
28:     }
29:
30:     cout << "Goodbye!" << endl;
31:
32:     return 0;
33: }
```

Результат

```
Enter two integers:
560
25
Do you wish to correct the numbers? (y/n): y
Enter two integers:
56
25
Do you wish to correct the numbers? (y/n): n
56 x 25 = 1400
56 + 25 = 81
Press x to exit or any other key to recalculate
r
Enter two integers:
95
-1
Do you wish to correct the numbers? (y/n): n
95 x -1 = -95
95 + -1 = 94
Press x to exit or any other key to recalculate
x
Goodbye!
```

Анализ

Цикл `for` в строке 5 отличается от такового в листинге 6.11 тем, что он бесконечный, в цикле отсутствует условие выхода, проверяемое на каждой итерации. Другими словами, без исполнения оператора `break` этот цикл (а следовательно, это приложение) никогда не завершится. Обратите внимание на вывод, который отличается от представленного до сих пор, — он позволяет пользователю поправить 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 нажимает клавишу <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 спрашивают, не желает ли он ввести два целых числа. Аналогично в конце цикла, когда в ответ на предложение выйти из программы пользователь вводит символ `'x'`, условие в строке 26 выполняется и выполняется следующий далее оператор `b_reak`, заканчивающий цикл.

ПРИМЕЧАНИЕ

Для создания бесконечного цикла в листинге 6.12 использован пустой оператор `for(;;)`. Вы можете заменить его оператором `while(true)` или `do... while(true);` и получить тот же результат, хотя и будет использован цикл другого типа.

РЕКОМЕНДУЕТСЯ	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, содержащегося в цикле, чтобы улучшить его удобочитаемость	**Не используйте** оператор `goto` **Не используйте** операторы `continue` и `break` без крайней необходимости **Не используйте** бесконечные циклы с оператором `break`, если этого можно избежать

Программирование вложенных циклов

Как вы уже видели в начале этого занятия, вложенные операторы `if` позволяют вложить один цикл в другой. Предположим, есть два массива целых чисел. Программа поиска произведения каждого элемента массива `Array1` и каждого элемента массива `Array2` будет проще, если использовать вложенный цикл. Первый цикл перебирает элементы массива `Array1`, а второй цикл, внутри первого, перебирает элементы массива `Array2`.

Листинг 6.13 демонстрирует, как можно вложить один цикл в другой.

ЛИСТИНГ 6.13. Использование вложенных циклов для умножения каждого элемента одного массива на каждый элемент другого

```
 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 MyInts1[ARRAY1_LEN] = {35, -3, 0};
 9:     int MyInts2[ARRAY2_LEN] = {20, -1};
10:
11:     cout << "Multiplying each int in MyInts1 by each in MyInts2:"
          << endl;
12:
13:     for(int Array1Index = 0; Array1Index < ARRAY1_LEN; ++Array1Index)
14:         for(int Array2Index = 0;
                Array2Index < ARRAY2_LEN; ++Array2Index)
15:             cout << MyInts1[Array1Index] << " x "
                  << MyInts2[Array2Index] \
16:                 << " = " << MyInts1[Array1Index] * MyInts2[Array2Index]
                    << endl;
17:
18:     return 0;
19: }
```

Результат

```
Multiplying each int in MyInts1 by each in MyInts2:
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 массив `MyInts1`, а второй — массив `MyInts2`. Первый цикл `for` запускает второй в пределах каждой итерации. Второй цикл `for` перебирает все элементы массива `MyInts2`, причем при каждой итерации он умножает этот элемент на элемент, проиндексированный переменной `Array1Index` из первого, внешнего, цикла. Так, для каждого элемента массива `MyInts1` второй цикл переберет все элементы массива `MyInts2`, в результате первый элемент массива `MyInts1` (со смещением 0) перемножается со всеми элементами массива `MyInts2`. Затем второй элемент массива `MyInts1` перемножается со всеми элементами массива `MyInts2`. И наконец, третий элемент массива `MyInts1` перемножается со всеми элементами массива `MyInts2`.

ПРИМЕЧАНИЕ Для удобства (и чтобы не отвлекаться от циклов) содержимое массивов в листинге 6.13 инициализируется. В предыдущих примерах, например в листинге 6.10, показано, как позволить пользователю ввести числа в целочисленный массив.

Использование вложенных циклов для перебора многомерного массива

На занятии 4 вы узнали о многомерных массивах. В листинге 4.3 происходит обращение к элементам двумерного массива из трех рядов и трех столбцов. Там обращение к каждому элементу в каждом ряду осуществлялось индивидуально, и не было никакой автоматизации. Если бы массив стал большим или его размерностей стало больше, то для доступа к его элементам понадобилось бы много больше кода. Однако использование циклов может все это изменить, как показано в листинге 6.14.

ЛИСТИНГ 6.14. Использование вложенных циклов для перебора элементов двумерного массива

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     const int MAX_ROWS = 3;
6:     const int MAX_COLS = 4;
7:
```

```
 8:     // Двумерный массив целых чисел
 9:     int MyInts[MAX_ROWS][MAX_COLS] = { {34, -1, 879, 22},
10:                                        {24, 365, -101, -1},
11:                                        {-20, 40, 90, 97} };
12:
13:     // перебор всех рядов массива
14:     for (int Row = 0; Row < MAX_ROWS; ++Row)
15:     {
16:         // перебор всех чисел в каждом ряду (столбцов)
17:         for (int Column = 0; Column < MAX_COLS; ++Column)
18:         {
19:             cout << "Integer[" << Row << "][" << Column \
20:                  << "] = " << MyInts[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 перебора двумерного массива целых чисел и получения доступа к его элементам. В действительности двумерный массив — это массив массивов целых чисел. Обратите внимание, что первый цикл `for` обращается к рядам (каждый из которых является массивом целых чисел), а второй — к его столбцам, т.е. осуществляет доступ к каждому элементу в этом массиве.

ПРИМЕЧАНИЕ

Скобки в листинге 6.14 вокруг вложенного цикла `for` использованы только для удобочитаемости. Эти вложенные циклы прекрасно сработают и без фигурных скобок, поскольку оператор цикла – это только один оператор, а не составной оператор, который требует использования фигурных скобок.

Использование вложенных циклов для вычисления чисел Фибоначчи

Знаменитая прогрессия Фибоначчи — это ряд чисел, начинающихся с 0 и 1, где каждое последующее число — сумма предыдущих двух. Таким образом, прогрессия Фибоначчи начинается с такой последовательности:

0, 1, 1, 2, 3, 5, 8, ... и так далее

В листинге 6.15 показано, как получить прогрессию Фибоначчи из любого желаемого количества чисел, ограниченного только размером целочисленной переменной, хранящей последнее число.

ЛИСТИНГ 6.15. Использование вложенных циклов для вычисления чисел прогрессии Фибоначчи

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     const int NumsToCal = 5;
 6:     cout << "This program will calculate " << NumsToCal \
 7:          << " Fibonacci Numbers at a time" << endl;
 8:
 9:     int Num1 = 0, Num2 = 1;
10:     char WantMore = '\0';
11:     cout << Num1 << " " << Num2 << " ";
12:
13:     do
14:     {
15:         for (int Index = 0; Index < NumsToCal; ++Index)
16:         {
17:             cout << Num1 + Num2 << " ";
18:
19:             int Num2Temp = Num2;
20:             Num2 = Num1 + Num2;
21:             Num1 = Num2Temp;
22:         }
23:
24:         cout << endl << "Do you want more numbers (y/n)? ";
25:         cin >> WantMore;
26:     }while (WantMore == 'y');
27:
28:     cout << "Goodbye!" << endl;
29:
30:     return 0;
31: }
```

Результат

```
This program will calculate 5 Fibonacci Numbers at a time
0 1 1 2 3 5 8
Do you want more numbers (y/n)? y
```

```
13 21 34 55 89
Do you want more numbers (y/n)? y
144 233 377 610 987
Do you want more numbers (y/n)? y
1597 2584 4181 6765 10946
Do you want more numbers (y/n)? n
Goodbye!
```

Анализ

Внешний цикл `do...while` в строке 13 является основным, он запрашивает у пользователя, хочет ли он получить следующие числа. Внутренний цикл `for` в строке 15 решает задачу вычисления и отображения за один раз пяти следующих 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 нажмет клавишу <y>.

Резюме

На этом занятии вы узнали, что можно писать код, выполняющийся не только сверху вниз; операторы условного выполнения кода позволяют создавать альтернативные пути выполнения и повторять блоки кода в цикле. Теперь вы знаете, как использовать конструкцию `if...else` и оператор `switch-case`, чтобы справиться с различными ситуациями, когда переменные содержат различные значения.

Для объяснения концепции циклов был представлен оператор `goto`, однако сразу было сделано предупреждение не использовать его 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 следующих операторов в 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)`. Цикл будет продолжаться, пока значением переменной `Integer` не станет -1, не так ли?**
  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 `true`. Поскольку -1 — это не нуль, условие выхода из цикла `while` возвращает значение `true`, и цикл продолжает выполняться. Если вы хотите, чтобы цикл был выполнен только для положительных чисел, перепишите

выражение как `while(Integer>0)`. Это правило истинно для всех условных операторов и циклов.

- **Эквивалентны ли пустой цикл `while` и оператор `for(;;)`?**

  Нет, оператор `while` всегда нуждается в последующем условии выхода.

- **Я изменил код `do...while(exp);` на `while(exp);` копированием и вставкой. Не возникнет ли каких-нибудь проблем?**

  Да, и большие! Код `while(exp);` — вполне допустимый, хоть и пустой цикл `while`, поскольку перед точкой с запятой нет никаких операторов, даже если за ним следует блок операторов. Блок кода перед первым в вопросе циклом выполняется как минимум однажды. 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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "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 операторов, вложенных циклов операторов `if`, если код вполне нормально компилируется и без них?
2. Вы можете быстро реализовать переход, используя оператор `goto`. Почему следует избегать его применения?
3. Возможно ли написать цикл `for`, где значение счетчика уменьшается? Как бы он выглядел?
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 6.13, но добавляющий элементы в два массива в обратном порядке.
3. Напишите программу, которая, подобно листингу 6.15, отображает числа Фибоначчи, но спрашивает пользователя, сколько чисел он хочет вычислить.
4. Напишите конструкцию `switch-case`, которая сообщает, есть ли в радуге такой цвет или нет. Используйте перечисляемую константу.
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 << "Enter a number between 0 and 4" << endl;
int Input = 0;
cin >> Input;
switch (Input)
{
case 0:
case 1:
case 2:
case 3:
case 4:
    cout << "Valid input" << endl;
default:
    cout << "Invalid input" << endl;
}
```

ЗАНЯТИЕ 7
Организация кода при помощи функций

До сих пор в этой книге вы видели простые программы, все действия которых содержались в функции `main()`. В небольших программах и приложениях это работает 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 разграничивают содержимое приложения на логические блоки, которые вызываются при необходимости.

Функция (function) — это подпрограмма, которая может получать параметры и возвращает значение. Чтобы выполнить свою задачу, функция должна быть вызвана.

На сегодняшнем 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. Можно, конечно, поместить весь код в функцию `main()`, но можно разделить это приложение на логические блоки, а именно: вычисляющий площадь по данному радиусу и, соответственно, периметр (листинг 7.1).

ЛИСТИНГ 7.1. Две функции, вычисляющие площадь и периметр круга, заданного радиусом

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: const double Pi = 3.14159;
 4:
 5: // Объявления функций (прототипы)
 6: double Area(double InputRadius);
 7: double Circumference(double InputRadius);
 8:
 9: int main()
10: {
11:     cout << "Enter radius: ";
12:     double Radius = 0;
13:     cin >> Radius;
14:
15:     // Вызов функции "Area"
16:     cout << "Area is: " << Area(Radius) << endl;
17:
18:     // Вызов функции "Circumference"
19:     cout << "Circumference is: " << Circumference(Radius) << endl;
20:
21:     return 0;
22: }
23:
24: // Определения функций (реализации)
25: double Area(double InputRadius)
26: {
27:     return Pi * InputRadius * InputRadius;
28: }
29:
30: double Circumference(double InputRadius)
31: {
32:     return 2 * Pi * InputRadius;
33: }
```

Результат

```
Enter radius: 6.5
Area is: 132.732
Circumference is: 40.8407
```

Анализ

На первый взгляд это выглядит как то же содержимое, но в разных упаковках. Впоследствии вы оцените, что разделение вычисления площади и периметра на отдельные функции позволяет многократно использовать этот код, поскольку функции можно вызывать неоднократно, как и когда понадобится. Функция `main()` весьма компактна, она делегирует действия 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 компилятор узнает 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 25–38, а функция `Circumference()` — в строках 30–33.

Что такое прототип функции

Рассмотрим строки 6 и 7 в листинге 7.1:

```
double Area(double InputRadius);
double Circumference(double InputRadius);
```

Состав прототипа функции представлен на рис. 7.1.

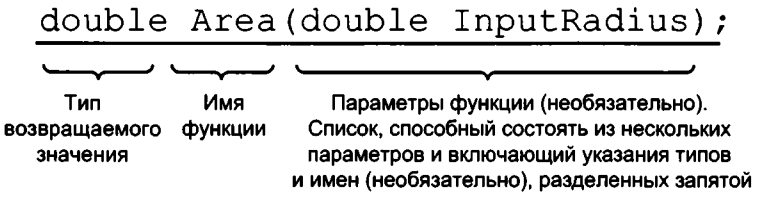


РИС. 7.1. Части прототипа функции

Прототип функции в основном указывает имя функции (в данном случае `Area`), список получаемых ею параметров (в данном случае один параметр типа `double` по имени `InputRadius`) 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 это за термины такие — `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 операторы и считает их допустимыми, а свою задачу понимает как необходимость связать функции с их реализациями и гарантировать, чтобы при выполнении программа фактически вызывала именно их.

ПРИМЕЧАНИЕ

Функции могут иметь список из нескольких параметров, разделенных запятыми, но только один тип возвращаемого значения.

При создании функции, которая не должна возвращать никаких значений, укажите ее тип возвращаемого значения как `void` (пусто).

Что такое определение функции

Фактическая суть функции в ее *реализации* (implementation), называемой также *определением* (definition). Проанализируем определение функции `Area`:

```
25: double Area(double InputRadius)
26: {
27:     return Pi * InputRadius * InputRadius;
28: }
```

Определение функции всегда состоит из блока операторов. Оператор `return` необходим, если функция не объявлена с типом возвращаемого значения `void`. В данном случае функция `Area` нуждается в операторе `return` для возвращения значения, поскольку типом возвращаемого значения функции был объявлен не тип `void`. *Блок операторов* (statement block) содержит операторы в фигурных скобках (`{...}`), которые выполняются при вызове функции. Функция `Area()` использует входной параметр `InputRadius`, содержащий радиус как аргумент, передаваемый вызывающей стороной для вычисления площади круга.

Что такое вызов функции и аргументы

Применение функции называется *вызовом функции* (function call). Когда функция объявлена так же, как здесь, с *параметрами* (parameter), при вызове ей нужно передать *аргументы* (argument), являющиеся значениями, затребованными функцией в ее списке параметров. Проанализируем вызов функции `Area()` в листинге 7.1:

```
16:     cout << "Area is: " << Area(Radius) << endl;
```

где `Area(Radius)` — это вызов функции; `Radius` — аргумент, переданный в функцию `Area()`. При вызове выполнение переходит к функции `Area()`, которая использует переданный радиус для вычисления площади круга. Когда функция завершает работу, она возвращает значение типа `double`, которое затем отображается на экране оператором `cout`.

Создание функций с несколькими параметрами

Предположим, вы пишете программу, которая вычисляет площадь цилиндра, как показано на рис. 7.2.

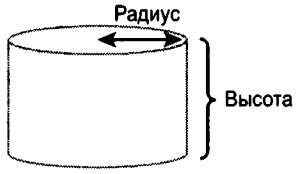


РИС. 7.2. Цилиндр

Вы использовали бы для этого следующую формулу:

```
Площадь цилиндра = Площадь верхнего круга + Площадь нижнего круга +
                   + Площадь боковой поверхности
                 = Pi * Radius^2 + Pi * Radius ^2 + 2 * Pi * Radius * Height
                 = 2 * Pi * Radius^2 + 2 * Pi * Radius * Height
```

Таким образом, при вычислении площади цилиндра необходимо работать с двумя переменными — радиусом и высотой. Поэтому при объявлении такой функции в списке ее параметров определяется по крайней мере два параметра. В списке параметров их отделяют запятой, как показано в листинге 7.2.

Листинг 7.2. Функция, получающая два параметра и вычисляющая площадь поверхности цилиндра

```
1: #include <iostream>
2: using namespace std;
3:
4: const double Pi = 3.14159;
5:
6: // Объявление содержит два параметра
7: double SurfaceArea(double Radius, double Height);
8:
9: int main()
10: {
11:     cout << "Enter the radius of the cylinder: ";
12:     double InRadius = 0;
13:     cin >> InRadius;
14:     cout << "Enter the height of the cylinder: ";
15:     double InHeight = 0;
16:     cin >> InHeight;
17:
18:     cout << "Surface Area: " << SurfaceArea(InRadius, InHeight)
          << endl;
19:
20:     return 0;
21: }
22:
23: double SurfaceArea(double Radius, double Height)
24: {
25:     double Area = 2 * Pi * Radius * Radius + 2 * Pi * Radius \
                    * Height;
26:     return Area;
27: }
```

Результат

```
Enter the radius of the cylinder: 3
Enter the height of the cylinder: 6.5
Surface Area: 179.071
```

Анализ

Строка 6 содержит объявление функции `SurfaceArea()` с двумя параметрами: `Radius` и `Height`, оба имеют тип `double` и отделены 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 стороне.

ПРИМЕЧАНИЕ

Параметры функций похожи на локальные переменные. Они допустимы только в пределах функции. Так, параметры Radius и Height функции SurfaceArea() в листинге 7.2 допустимы и пригодны для использования только в пределах самой функции SurfaceArea(), но не вне ее.

Создание функций без параметров и возвращаемых значений

Если вы делегируете задачу вывода фразы "Hello World" функции, которая делает только это и ничего больше, то сделать это можно без всяких параметров (поскольку она не должна делать ничего, кроме вывода слов "Hello World"), возвращать значения ей тоже не нужно (так как вы ничего не ожидаете от возвращения такой функции). 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. она не возвращает никакого значения. Следовательно, в определении функции (строки 11–14) нет никакого оператора выхода. Вызов этой функции в строке 7 функции main() не присваивает ни какой переменной возвращаемое значение и не используется ни в каком выражении, поскольку данная функция ничего не возвращает.

Параметры функций со значениями по умолчанию

До сих пор мы использовали в примерах фиксированное значение числа Пи как константу и не предоставляли пользователю возможности изменить его. Однако пользователя функции может интересовать более или менее точное чтение. Когда вы создаете функцию, использующую число Пи, как позволить ее пользователю применить собственное значение, а при его отсутствии задействовать стандартное?

Один из способов решения этой проблемы подразумевает создание в функции `Area()` дополнительного параметра для числа Пи и присвоение ему *значения по умолчанию* (default value). Такая адаптация функции `Area()` из листинга 7.1 выглядела бы так:

```
double Area(double InputRadius, double Pi = 3.14);
```

Обратите внимание на второй параметр `Pi` и присвоенное ему по умолчанию значение 3,14. Этот второй параметр является теперь *необязательным параметром* (optional parameter) для вызывающей стороны. Так, вызывающая функция все еще может вызвать функцию `Area()`, используя такой синтаксис:

```
Area(Radius);
```

В данном случае второй параметр был проигнорирован, поэтому используется значение по умолчанию 3,14. Но если пользователь захочет задействовать другое значение числа Пи, то сделать это можно, вызвав функцию `Area()` так:

```
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 Пи 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 InputRadius, double Pi = 3.14); // Pi со значением
 5:                                                    // по умолчанию
 6: int main()
 7: {
 8:     cout << "Enter radius: ";
 9:     double Radius = 0;
10:     cin >> Radius;
11:
12:     cout << "Pi is 3.14, do you wish to change this (y / n)? ";
13:     char ChangePi = 'n';
14:     cin >> ChangePi;
15:
16:     double CircleArea = 0;
17:     if (ChangePi == 'y')
18:     {
19:         cout << "Enter new Pi: ";
20:         double NewPi = 3.14;
```

```
21:         cin >> NewPi;
22:         CircleArea = Area (Radius, NewPi);
23:     }
24:     else
25:         CircleArea = Area(Radius); // 2-й параметр игнорируется,
26:                         // значит, использовать значение по умолчанию
27:     // Вызов функции "Area"
28:     cout << "Area is: " << CircleArea << endl;
29:
30:     return 0;
31: }
32:
33: // В определении функции значение по умолчанию не задается снова
34: double Area(double InputRadius, double Pi)
35: {
36:     return Pi * InputRadius * InputRadius;
37: }
```

Результат

```
Enter radius: 1
Pi is 3.14, do you wish to change this (y / n)? n
Area is: 3.14
```

Следующий запуск:

```
Enter radius: 1
Pi is 3.14, do you wish to change this (y / n)? y
Enter new Pi: 3.1416
Area is: 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 Пи, а следовательно, вычисленная площадь стала немного иной. Обратите внимание, что в обоих случаях в строке 22 и 25 происходит вызов той же функции. В строке 25 при вызове нет второго параметра, `Pi`, поэтому для него 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 погрязнет 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 прогрессии Фибоначчи, как представлено в листинге 7.5. Эта прогрессия 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, энное значение последовательности (для n>1) определяется следующей (рекурсивной) формулой:

```
Fibonacci(n) = Fibonacci(n - 1) + Fibonacci(n - 2)
```

В результате прогрессия Фибоначчи расширяется до

```
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8 и так далее.
```

ЛИСТИНГ 7.5. Использование рекурсивной функции для вычисления чисел прогрессии 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 << "Enter 0-based index of desired Fibonacci Number: ";
14:     int Index = 0;
15:     cin >> Index;
16:
17:     cout << "Fibonacci number is: " << GetFibNumber(Index) << endl;
18:     return 0;
19: }
```

Результат

```
Enter 0-based index of desired Fibonacci Number: 6
Fibonacci number is: 8
```

Анализ

Функция `GetFibNumber()`, определенная в строках 3–9, рекурсивна, поскольку она вызывает сама себя в строке 8. Обратите внимание на условие выхода, расположенное 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 определенной 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 и 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 `rturn` в одной функции

```
 0: #include <iostream>
 1: using namespace std;
 2: const double Pi = 3.14159;
 3:
 4: void QueryAndCalculate()
 5: {
 6:     cout << "Enter radius: ";
 7:     double Radius = 0;
 8:     cin >> Radius;
 9:
10:     cout << "Area: " << Pi * Radius * Radius << endl;
11:
12:     cout << "Do you wish to calculate circumference (y/n)? ";
13:     char CalcCircum = 'n';
14:     cin >> CalcCircum;
15:
16:     if (CalcCircum == 'n')
17:         return;
18:
19:     cout << "Circumference: " << 2 * Pi * Radius << endl;
20:     return;
21: }
22:
23: int main()
24: {
25:     QueryAndCalculate ();
26:
27:     return 0;
28: }
```

Результат

```
Enter radius: 1
Area: 3.14159
Do you wish to calculate circumference (y/n)? y
Circumference: 6.28319
```

Следующий запуск:

```
Enter radius: 1
Area: 3.14159
Do you wish to calculate circumference (y/n)? n
```

Анализ

Функция `QueryAndCalculate()` содержит несколько операторов `return`: один в строке 17, а второй в строке 20. Эта функция спрашивает у пользователя, не хочет ли он также вычислить периметр. Если пользователь вводит символ `'n'`, отказываясь от вычисления, происходит выход из программы с использованием 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 наверху и заканчивается внизу, чем функцию, которая имеет несколько выходов в разных местах.

Чтобы избежать использования нескольких выходов в листинге 7.6, достаточно изменить условие оператора `if` на проверку значения `'y'`:

```
if (CalcCircum == 'y')
    cout << "Circumference: " << 2 * Pi * Radius << endl;
```

Использование функций для работы с данными различных форм

Функции не ограничивают вас передачей значений по одному; вы можете передать функции в массив значений. Вы можете создать две функции с одинаковым именем и возвращаемым значением, но с различными параметрами. Можете создать функцию, параметры которой не создаются и не удаляются при вызове функции; вместо них используются ссылки, которые остаются допустимыми даже после завершения работы функции, что позволяет манипулировать при вызове функции большими объемами данных или параметров. В этом разделе вы узнаете о передаче функциям массивов, о перегрузке функций и передаче функции аргументов по ссылке.

Перегрузка функции

Функции с одинаковым именем и типом возвращаемого значения, но с разными параметрами или набором параметров называют *перегруженными функциями* (overloaded function). Перегруженные функции могут быть весьма полезными в приложениях, где функция с определенным именем осуществляет определенный тип вывода, но может быть вызвана с различными наборами параметров. Предположим, необходимо написать приложение, которое вычисляет площадь круга и площадь цилиндра. Функция, которая вычисляет площадь круга, нуждается в одном параметре — радиусе. Другая функция, которая вычисляет площадь цилиндра, нуждается кроме радиуса во втором параметре — высоте цилиндра. Обе функции должны возвратить данные одинакового типа — площадь. Язык

C++ позволяет определить две перегруженные функции, обе по имени Area, и обе, возвращающие значение типа double, однако одна из них получает только радиус, а другая радиус и высоту, как представлено в листинге 7.7.

ЛИСТИНГ 7.7. Использование перегруженной функции для вычисления площади круга или цилиндра

```
#include <iostream>
using namespace std;

const double Pi = 3.14159;

double Area(double Radius); // для круга
double Area(double Radius, double Height); // перегружено для
                                           // цилиндра
int main()
{
    cout << "Enter z for Cylinder, c for Circle: ";
    char Choice = 'z';
    cin >> Choice;

    cout << "Enter radius: ";
    double Radius = 0;
    cin >> Radius;

    if (Choice == 'z')
    {
        cout << "Enter height: ";
        double Height = 0;
        cin >> Height;

        // Вызов перегруженной версии Area для цилиндра
        cout << "Area of cylinder is: " << Area (Radius, Height)
             << endl;
    }
    else
        cout << "Area of cylinder is: " << Area (Radius) << endl;

    return 0;
}

// для круга
double Area(double Radius)
{
    return Pi * Radius * Radius;
}

// перегружено для цилиндра
double Area(double Radius, double Height)
{
    // повторное использование версии для площади круга
    return 2 * Area(Radius) + 2 * Pi * Radius * Height;
}
```

Результат

```
Enter z for Cylinder, c for Circle: z
Enter radius: 2
Enter height: 5
Area of cylinder is: 87.9646
```

Следующий запуск:

```
Enter z for Cylinder, c for Circle: c
Enter radius: 1
Area of cylinder is: 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, функция перегружена. Определения перегруженных функций находятся в строках 34–44, где реализованы две функции вычисления площади: круга по радиусу и цилиндра по радиусу и высоте соответственно. Интересно то, что, поскольку площадь цилиндра состоит из площади двух кругов (один сверху, второй снизу) и площади боковой стороны, перегруженная версия для цилиндра способна повторно использовать функцию `Area()` для круга, как показано в строке 43.

Передача функции массива значений

Функция, которая отображает целое число, может быть представлена так:

```
void DisplayInteger(int Number);
```

У функции, способной отобразить массив целых чисел, прототип немного другой:

```
void DisplayIntegers(int[] Numbers, int Length);
```

Первый параметр указывает, что передаваемые функции данные являются массивом, а второй параметр предоставляет его длину[1], чтобы, используя массив, вы не пересекли 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] << " ";
```

---

[1] Не поступайте так никогда. Неправильно указав длину массива или изменив впоследствии его размер и забыв исправить абсолютно все вызовы функции, вы получите в лучшем случае ошибку выхода за границы массива, а в худшем — очень трудно обнаруживаемую ошибку перебора массива не до конца. Для определения размера массива лучше использовать оператор *sizeof()* в функции. — *Примеч. ред.*

```
 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 MyNumbers[4] = {24, 58, -1, 245};
22:     DisplayArray(MyNumbers, 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

Здесь есть две перегруженные версии функции `DisplayArray()`: одна отображает содержимое элементов целочисленного массива, а вторая символьного. В строках 22 и 25 вызываются две функции, используя как параметр массива целых чисел и 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 сейчас массив не используется как строка в операторе `cout` или подобном (`cout << MyStatement;`), но впоследствии это может произойти.

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 InputRadius)
26: {
27:     return Pi * InputRadius * InputRadius;
28: }
```

Здесь параметр `InputRadius` содержит значение, которое копируется в него при вызове функции `main()`:

```
15:     // Вызов функции "Area"
16:     cout << "Area is: " << Area(Radius) << endl;
```

Это означает, что вызов функции `Area()` никак не воздействует на переменную `Radius` в функции `main()`, поскольку работает она с копией значения переменной `Radius`, содержащейся в переменной `InputRadius`. Иногда нужны функции, способные воздействовать на исходную переменную или изменять значение, которое доступно вне функции, скажем, в вызывающей функции. Вот когда вы объявляете параметр как получающий аргумент *по ссылке* (by reference). Версия функции `Area()`, вычисляющей и возвращающей площадь как передаваемый по ссылке параметр, выглядит следующим образом:

```
// выходной параметр Result по ссылке
void Area(double Radius, double& Result)
{
    Result = Pi * Radius * Radius;
}
```

Обратите внимание: функция `Area()` в этой версии получает два параметра. Не пропустите амперсанд (`&`) рядом со вторым параметром `Result`. Этот знак указывает компилятору, что второй аргумент не должен быть скопирован в функцию, что это ссылка на передаваемую переменную. Тип возвращаемого значения был изменен на `void`, поскольку функция теперь возвращает вычисленную площадь не как возвращаемое значение, а как выходной параметр по ссылке. Возвращение значений по ссылке демонстрирует код листинга 7.9, вычисляющий площадь круга.

ЛИСТИНГ 7.9. Возврат площади круга по ссылке, а не в качестве возвращаемого значения

```
#include <iostream>
using namespace std;

const double Pi = 3.1416;

// выходной параметр Result по ссылке
void Area(double Radius, double& Result)
{
    Result = Pi * Radius * Radius;
}

int main()
{
    cout << "Enter radius: ";
    double Radius = 0;
    cin >> Radius;

    double AreaFetched = 0;
    Area(Radius, AreaFetched);

    cout << "The area is: " << AreaFetched << endl;
    return 0;
}
```

Результат

```
Enter radius: 2
The area is: 12.5664
```

Анализ

Обратите внимание на строки 17 и 18, где функция `Area()` вызывается с двумя параметрами; второй параметр тот, который должен содержать результат. Поскольку функция `Area()` получает второй параметр по ссылке, переменная `Result`, используемая в строке 8 в пределах функции `Area()`, указывает на ту же область памяти, что и переменная `double AreaFetched`, объявленная в строке 17 вызывающей стороны (функции `main()`). Таким образом, вычисленный в функции `Area()` результат (строка 8) доступен в функции `main()` и отображается на экране в строке 20.

ПРИМЕЧАНИЕ

Используя оператор `return`, функция может возвратить только одно значение. Но если функция должна выполнять операции, затрагивающие много значений, которые необходимо вернуть вызывающей стороне, то передача аргументов по ссылке является единственным способом, позволяющим функции вернуть это множество модификаций назад вызывающей стороне.

Как процессор обрабатывает вызовы функций

Хотя и не чрезвычайно важно знать во всех подробностях, как вызов функции реализуется на уровне процессора, понятие об этом все же стоит иметь. Это поможет понять, почему язык C++ позволяет создавать встраиваемые функции, которые рассматриваются в этом разделе позже.

Вызов функции, по существу, означает, что микропроцессор переходит к выполнению следующей инструкции, принадлежащей вызываемой функции, расположенной в области памяти непоследовательно. После выполнения инструкций в функции выполнение возвращается туда, откуда был совершен переход. Для реализации этой логики компилятор преобразует вызов функции в инструкцию процессора `CALL`, определяющую адрес следующей инструкции для выполнения, и этот адрес принадлежит вашей функции. При компиляции самой функции компилятор преобразует оператор `return` в инструкцию процессора `RET`.

Когда процессор встречает инструкцию `CALL`, он запоминает в стеке позицию инструкции, которая будет выполнена после вызова функции, и переходит к области памяти, содержавшейся в инструкции `CALL`.

Эта область памяти содержит инструкции, принадлежащие функции. Процессор выполняет их до тех пор, пока не встретит инструкцию `RET` (код процессора для оператора `return` в программе). Инструкция `RET` требует от процессора извлечь из стека адрес, сохраненный во время выполнения инструкции `CALL`. Это адрес области в вызывающей функции, откуда выполнение должно продолжиться. Таким образом, процессор возвращает выполнение вызывающей стороне, и оно продолжается оттуда, где было остановлено.

Понятие стека

Стек (stack) — это структура в памяти, действующая по принципу *последним вошел, первым вышел* (Last-In-First-Out — LIFO), весьма похожая на стопку тарелок, из которой вы берете сверху ту тарелку, 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, и он всегда указывает на вершину стека (рис. 7.3).

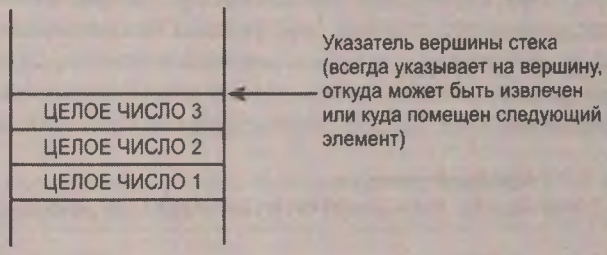


РИС. 7.3. Визуальное представление стека, содержащего три целых числа

Характер стека делает его оптимальным для обработки вызовов функций. При вызове функции экземпляры всех ее локальных переменных создаются в стеке, т.е. они заталкиваются в стек. Когда функция заканчивает работу, они просто вытягиваются из него и указатель вершины стека возвращается туда, где он находился первоначально.

Встраиваемые функции

Вызов обычной функции преобразуется в инструкцию CALL, которая приводит к операциям со стеком и переходу процессора к области хранения кода функции и т.д. Это дополнительная работа, выполняемая внутренне, однако весьма быстро в большинстве случаев. Но что если функция очень проста, как эта?

```
double GetPi()
{
    return 3.14159;
}
```

Дополнительные затраты времени на выполнение фактического вызова функции в этом случае весьма высоки по сравнению с периодом времени, затраченным на выполнение содержимого функции `GetPi()`. Вот почему компиляторы C++ позволяют программисту объявлять такие функции как *встраиваемые* (inline). Ключевое слово `inline` — это просьба встроить реализацию функции в место ее вызова.

```
inline double GetPi()
{
    return 3.14159;
}
```

Аналогично функция, которая только удваивает число или выполняет похожие простые операции, является хорошим кандидатом на встраивание. 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 << "Enter an integer: ";
12:     int InputNum = 0;
13:     cin >> InputNum;
14:
15:     // Вызов встраиваемой функции
16:     cout << "Double is: " << DoubleNum(InputNum) << endl;
17:
18:     return 0;
19: }
```

Результат

```
Enter an integer: 35
Double is: 70
```

Анализ

Рассматриваемое ключевое слово, `inline`, используется в строке 4. Компиляторы, как правило, рассматривают это ключевое слово как просьбу поместить содержимое функции `DoubleNum()` непосредственно по месту 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

Самые современные компиляторы C++ обладают различными возможностями по оптимизации производительности. Некоторые, такие как компилятор Microsoft C++ Compiler, позволяют оптимизировать по размеру или скорости приложения. Первое весьма важно при разработке программного обеспечения для различных устройств, как правило, мобильных, где память может быть очень ценна. При оптимизации по размеру 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) и делает это иногда даже в тех случаях, когда никто его об этом не просит.

C++11

Лямбда-функции

Этот раздел — только введение в сложную для новичков концепцию. Попробуйте изучить ее, но не расстраивайтесь, если это не получится. Более подробная информация о лямбда-функциях приведена на занятии 22, "Лямбда-выражения языка C++11".

Лямбда-функции очень полезны, если вы часто используете алгоритмы STL для сортировки и обработки данных, содержащихся, например, в таких контейнерах STL, как `std::vector` (динамический массив). Как правило, сортировка требует предоставить двоичный предикат, реализованный как оператор в классе, что требует весьма кропотливого программирования. Компиляторы, совместимые со стандартом C++11, позволяют создавать лямбда-функции и не тратить много усилий, как показано в листинге 7.11.

ЛИСТИНГ 7.11. Использование лямбда-функции для сортировки и отображения элементов массива

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void DisplayNums(vector<int>& DynArray)
{
    for_each (DynArray.begin(), DynArray.end(), \
            [](int Element) {cout << Element << " ";} );// лямбда

    cout << endl;
}

int main()
{
    vector<int> MyNumbers;
    MyNumbers.push_back(501);
    MyNumbers.push_back(-1);
    MyNumbers.push_back(25);
    MyNumbers.push_back(-35);

    DisplayNums(MyNumbers);

    cout << "Sorting them in descending order" << endl;

    sort (MyNumbers.begin(), MyNumbers.end(), \
         [](int Num1, int Num2) {return (Num2 < Num1); } );

    DisplayNums(MyNumbers);

    return 0;
}
```

Результат

```
501 -1 25 -35
Sorting them in descending order
501 25 -1 -35
```

Анализ

Программа помещает целые числа в динамический массив, предоставленный стандартной библиотекой C++ в виде вектора `std::vector` (строки 5–19). Функция `DisplayNums()` использует алгоритм STL для перебора и отображения значений всех элементов массива. При этом в строке 8 вместо громоздкого предиката используется лямбда-функция. При сортировке в строке 25 также используется двоичный предикат (строка 26) в форме лямбда-функции, возвращающей значение `true`, если второе число меньше первого, фактически сортируя коллекцию в порядке возрастания.

Синтаксис лямбда-функций следующий:

```
[необязательные параметры](список параметров){ операторы; }
```

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 сторона может переопределить, а также иметь параметры, аргументы которых передаются по ссылке. Вы также узнали, как передать функции массив, как создавать перегруженные функции с одинаковым именем и типом возвращаемого значения, но разными списками параметров.

И наконец, вы получили некоторое представление о лямбда-функциях. У лямбда-функций C++11 есть совершенно новый потенциал, позволяющий изменить способ создания приложений C++, особенно при 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 объема стека, что в конечном счете приведет 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. Однако результатом встраивания каждой функции 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 параметров 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 сработает?**
  Для перегрузки обе функции нуждаются в одинаковом имени и одинаковом типе возвращаемого значения. В данном случае компилятор сообщит об ошибке, поскольку две разные функции не могут использовать одинаковое имя.

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 Г, "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 конец стека?

Упражнения

1. Напишите перегруженные функции, которые вычисляют объем сферы и цилиндра. Формулы таковы:
   ```
   Объем сферы = (4 * Pi * Радиус * Радиус * Радиус) / 3
   Объем цилиндра = Pi * Радиус * Радиус * 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 << "Enter radius: ";
    double Radius = 0;
    cin >> Radius;

    double AreaFetched = 0;
    Area(Radius, AreaFetched);

    cout << "The area is: " << AreaFetched << endl;
    return 0;
}
```

4. **Отладка:** Что не так со следующим объявлением функции?

```
double Area(double Pi = 3.14, double Radius);
```

5. Напишите функцию с типом возвращаемого значения `void`, которая все же вполне способна вернуть вызывающей стороне вычисленную площадь и периметр круга по радиусу.

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 работать, по мере необходимости, близко к аппаратным средствам. Действительно, язык C++ позволяет контролировать производительность приложения на уровне байтов и битов. Понимание работы указателей и ссылок — один из этапов на пути к способности писать программы, эффективно использующие системные ресурсы.

На сегодняшнем занятии.

- Что такое указатель.
- Что такое динамическая память.
- Как использовать операторы `new` и `delete` для резервирования 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).

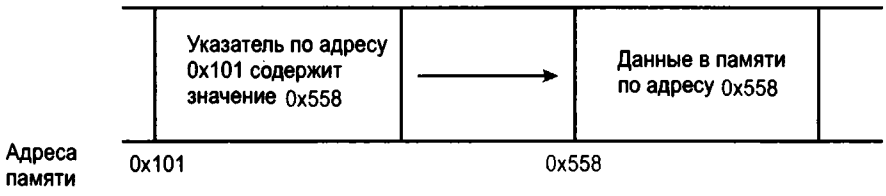


РИС. 8.1. Визуализация указателя

Таким образом, указатель — это переменная, и, как все переменные, он занимает пространство в памяти (в случае рис. 8.1 — по адресу 0x101). Но особенным в указателях является то, что содержащееся в нем значение (в данном случае 0x558) интерпретируется как адрес области памяти. Следовательно, указатель — это специальная переменная, которая *указывает на* область в памяти.

Объявление указателя

Поскольку указатель является переменной, его следует объявить. Обычно вы объявляете, что указатель указывает на значение определенного типа (например, типа `int`). Это значит, что содержавшийся в указателе адрес указывает на область в памяти, содержащую целое число. Можно также определить указатель на блок памяти (называемый также *пустым указателем* (void pointer)).

Указатель должен быть объявлен, как и все остальные переменные:

```
ТипУказателя * ИмяПеременнойУказателя;
```

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 `NULL`. Наличие значения `NULL` можно проверить, и оно не может быть адресом области памяти:

```
ТипУказателя * ИмяПеременнойУказателя = NULL; // инициализирующее значение
```

Таким образом, объявление указателя на целое число было бы таким:

```
int *pInteger = NULL; //
```

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 оно означает адрес области в памяти. Неинициализированные указатели способны заставить вашу программу обратиться к недопустимой области памяти, приведя к аварийному отказу.

Определение адреса переменной с использованием оператора ссылки (&)

Переменные — это средство, предоставляемое языком для работы с данными в памяти. Эта концепция была подробно рассмотрена на занятии 3, "Использование переменных, объявление констант". Указатели — это тоже переменные, но специального типа, использующиеся исключительно для содержания адресов памяти.

Если VarName — переменная, то оператор &VarName возвращает адрес в памяти, где хранится ее значение.

Так, если вы объявили целочисленную переменную, используя хорошо знакомый вам синтаксис:

```
int Age = 30;
```

то оператор &Age вернет адрес области в памяти, куда помещается значение (30). Листинг 8.1 демонстрирует концепцию адреса целочисленной переменной в области памяти, используемого для доступа к содержащемуся в ней значению.

ЛИСТИНГ 8.1. Определение адресов переменных типа int и double

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     int Age = 30;
7:     const double Pi = 3.1416;
8:
9:     // Использование & для поиска адреса в памяти
10:    cout << "Integer Age is at: 0x" << hex << &Age << endl;
11:    cout << "Double Pi is located at: 0x" << hex << &Pi << endl;
12:
13:    return 0;
14: }
```

Результат

```
Integer Age is at: 0x0045FE00
Double Pi is located at: 0x0045FDF8
```

Анализ

Обратите внимание, как оператор ссылки (&) используется в строках 9 и 10 для получения адресов переменной Age и константы Pi. Часть 0x была добавлена согласно соглашению, используемому при отображении шестнадцатеричных чисел.

ПРИМЕЧАНИЕ

Вы уже знаете, что объем памяти, используемый переменной, зависит от ее типа. Применение оператора sizeof() в коде листинга 3.4 показало, что размер целочисленной переменной составляет 4 байта (на системе автора при использовании его компилятора). Таким образом, приведенный выше вывод свидетельствует о том, что значение целочисленной переменной Age находится по адресу 0x0045FE08, а зная, что sizeof(int) составляет 4 бита, можно сделать вывод, что четыре байта, расположенные в диапазоне 0x0045FE00–0x0045FE04, принадлежат целочисленной переменной Age.

ПРИМЕЧАНИЕ

Оператор ссылки (referencing operator) (&) называется также *оператором обращения к адресу* (address-of 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 обращения к адресу (&).

С синтаксисом объявления переменной определенного типа вы уже знакомы:

```
// Объявление переменной
Тип ИмяПеременной = ИсходноеЗначение;
```

Чтобы сохранить адрес этой переменной в указателе, следует объявить указатель на тот же *Тип* и инициализировать его, используя оператор обращения к адресу (&):

```
// Объявление указателя на тот же тип и его инициализация
Тип* Указатель = &ИмяПеременной;
```

Предположим, вы объявили переменную Age типа int так:

```
int Age = 30;
```

Указатель на тип int, хранящий для последующего использования адрес значения переменной Age, объявили бы так:

```
int* pInteger = &Age; // Указатель на целочисленную переменную Age
```

Листинг 8.2 демонстрирует применение указателя для хранения адреса, полученного с использованием оператора обращения к адресу (&).

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* pInteger = &Age; // указатель на тип int, инициализированный
 7:                           // результатом &Age
 8:     // Отображение значения указателя
 9:     cout << "Integer Age is at: 0x" << hex << pInteger << endl;
10:
11:     return 0;
12: }
```

Результат

```
Integer Age is at: 0x0045FE00
```

Анализ

По существу, вывод этого фрагмента кода тот же, что и у предыдущего, поскольку оба примера отображают 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) его значение (адрес) отображается оператором cout.

ПРИМЕЧАНИЕ У вас адрес в выводе примеров, вероятно, будет иным. Фактически адрес переменной может изменяться при каждом запуске приложения на том же компьютере.

Теперь, когда вы знаете, как сохранить адрес в переменной указателя, вполне логично предположить, что тому же указателю может быть присвоен другой адрес области памяти и он будет указывать на другое значение, как представлено в листинге 8.3.

ЛИСТИНГ 8.3. Переназначение указателя другой переменной

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:
7:     int* pInteger = &Age;
8:     cout << "pInteger points to Age now" << endl;
9:
10:    // Отображение значения указателя
11:    cout << "pInteger = 0x" << hex << pInteger << endl;
12:
13:    int DogsAge = 9;
14:    pInteger = &DogsAge;
15:    cout << "pInteger points to DogsAge now" << endl;
16:
17:    cout << "pInteger = 0x" << hex << pInteger << endl;
18:
19:    return 0;
20: }
```

Результат

```
pInteger points to Age now
pInteger = 0x002EFB34
pInteger points to DogsAge now
pInteger = 0x002EFB1C
```

Анализ

Эта программа свидетельствует, что один указатель pInteger на целочисленную переменную способен указать на любую целочисленную переменную. В строке 7 этот указатель был инициализирован как &Age. Следовательно, он содержал адрес переменной Age. В строке 14 тому же указателю присваивается результат оператора &DogsAge, и он

уже указывает на другую область в памяти, содержащую значение переменной DogsAge. Таким образом, вывод демонстрирует, что значение указателя, первоначально бывшее адресом переменной Age, сменилось адресом переменной DogsAge. Значения этих переменных, естественно, хранятся в разных областях памяти, 0x002EFB34 и 0x002EFB1C соответственно.

Доступ к данным с использованием оператора обращения к значению (*)

Предположим, у вас есть указатель, содержащий вполне допустимый адрес. Как теперь обратиться к этой области, чтобы записать или прочитать содержащиеся в ней данные? Для этого используется *оператор обращения к значению* (de-referencing operator) (*). По существу, если есть допустимый указатель pData, использование оператора *pData позволяет получить доступ к значению, хранящемуся по адресу, содержащемуся в указателе. Использование оператора (*) показано в листинге 8.4.

ЛИСТИНГ 8.4. Использование оператора обращения к значению (*) 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 << "Integer Age = " << Age << endl;
 9:     cout << "Integer DogsAge = " << DogsAge << endl;
10:
11:     int* pInteger = &Age;
12:     cout << "pInteger points to Age" << endl;
13:
14:     // Отображение значения указателя
15:     cout << "pInteger = 0x" << hex << pInteger << endl;
16:
17:     // Отображение значения в указанной области
18:     cout << "*pInteger = " << dec << *pInteger << endl;
19:
20:     pInteger = &DogsAge;
21:     cout << "pInteger points to DogsAge now" << endl;
22:
23:     cout << "pInteger = 0x" << hex << pInteger << endl;
24:     cout << "*pInteger = " << dec << *pInteger << endl;
25:
26:     return 0;
27: }
```

Результат

```
Integer Age = 30
Integer DogsAge = 9
pInteger points to Age
```

```
pInteger = 0x0025F788
*pInteger = 30
pInteger points to DogsAge now
pInteger = 0x0025F77C
*pInteger = 9
```

Анализ

Кроме изменения адреса, хранимого в указателе, как в предыдущем примере (см. листинг 8.3), здесь используется также оператор обращения к значению (*) с тем же указателем pInteger для отображения значения двух адресов. Обратите внимание на строки 18 и 24.

В обеих этих строках осуществляется доступ к целочисленному значению, на которое указывает указатель pInteger, с использованием оператора обращения к значению (*). Поскольку адрес, содержавшийся в указателе pInteger, изменяется в строке 20, тот же указатель после этого позволяет обратиться к переменной DogsAge и отобразить значение 9.

Когда выполняется оператор обращения к значению (*), приложение использует хранящийся в указателе адрес как отправную точку для выборки из памяти 4 байтов, принадлежащих целому числу (поскольку это указатель на тип int и оператор sizeof(int) дает 4). Таким образом, допустимость адреса, содержавшегося в указателе, является абсолютно необходимой. При инициализации указателя результатом оператора &Age в строке 11 мы гарантировали, что указатель содержит допустимый адрес. Если не инициализировать указатель, он будет содержать случайное значение, которое существовало в области памяти при создании переменной указателя. Обращение к значению такого указателя обычно приводит к ошибке с сообщением Access Violation (Нарушение прав доступа), свидетельствующем о попытке обращения к области памяти, доступ к которой вашему приложению не разрешен.

ПРИМЕЧАНИЕ

Оператор обращения к значению (*) называется также *оператором косвенного доступа* (indirection operator).

Указатель в приведенном выше примере использовался для чтения (получения) значения из области памяти, на которую указывает указатель. В листинге 8.5 показано, что происходит, когда оператор *pInteger используется как l-значение, т.е. для присвоения значения, а не для его чтения.

ЛИСТИНГ 8.5. Манипулирование данными при помощи указателя и оператора обращения к значению (*)

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     int DogsAge = 30;
 6:     cout << "Initialized DogsAge = " << DogsAge << endl;
 7:
 8:     int* pAge = &DogsAge;
 9:     cout << "pAge points to DogsAge" << endl;
10:
11:     cout << "Enter an age for your dog: ";
```

```
12:
13:     // сохранить ввод в области памяти, на которую указывает pAge
14:     cin >> *pAge;
15:
16:     // Отобразить адрес, по которому хранится возраст
17:     cout << "Input stored using pAge at 0x" << hex << pAge << endl;
18:
19:     cout << "Integer DogsAge = " << dec << DogsAge << endl;
20:
21:     return 0;
22: }
```

Результат

```
Initialized DogsAge = 30
pAge points to DogsAge
Enter an age for your dog: 10
Input stored using pAge at 0x0025FA18
Integer DogsAge = 10
```

Анализ

Ключевой этап здесь в строке 14, где введенное пользователем целое число сохраняется в области, на которую указывает указатель `pAge`. Обратите внимание, несмотря на то, что введенное число было сохранено при помощи указателя `pAge`, строка 19 отображает это же значение при помощи переменной `DogsAge`. Это связано с тем, что указатель `pAge` указывает на переменную `DogsAge`, как было инициализировано в строке 8. Любое изменение в области памяти, где хранится значение переменной `DogsAge`, и на которую указывает указатель `pAge`, отражается на обоих.

Каков результат выполнения оператора `sizeof()` для указателя?

Вы уже знаете, что указатель — это только переменная, содержащая адрес области памяти. Следовательно, независимо от типа, на который он указывает, содержимое указателя — числовой адрес. 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, как демонстрирует листинг 8.6.

ЛИСТИНГ 8.6. Указатели на различные типы имеют одинаковый размер

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     int Age = 30;
6:     double Pi = 3.1416;
7:     char SayYes = 'y';
8:
```

```
    // инициализация указателей адресами переменных
    int* pInt = &Age;
    double* pDouble = &Pi;
    char* pChar = &SayYes;

    cout << "sizeof fundamental types -" << endl;
    cout << "sizeof(int) = " << sizeof(int) << endl;
    cout << "sizeof(double) = " << sizeof(double) << endl;
    cout << "sizeof(char) = " << sizeof(char) << endl;

    cout << "sizeof pointers to fundamental types -" << endl;
    cout << "sizeof(pInt) = " << sizeof(pInt) << endl;
    cout << "sizeof(pDouble) = " << sizeof(pDouble) << endl;
    cout << "sizeof(pChar) = " << sizeof(pChar) << endl;

    return 0;
}
```

Результат

```
sizeof fundamental types -
sizeof(int) = 4
sizeof(double) = 8
sizeof(char) = 1
sizeof pointers to fundamental types -
sizeof(pInt) = 4
sizeof(pDouble) = 4
sizeof(pChar) = 4
```

Анализ

Вывод однозначно демонстрирует, что даже при том, что `sizeof(char)` составляет 1 байт, а `sizeof(double)` — 8 байтов, размер их указателей всегда остается постоянным — 4 байта. Это связано с тем, что объем памяти, необходимый для хранения адреса, является тем же, независимо от того, указывает ли адрес на 1 байт или на 8 байтов.

ПРИМЕЧАНИЕ

Вывод листинга 8.6 свидетельствует, что размер указателей составляет 4 байта, но на вашей системе результат может быть иным. Вывод был получен при компиляции кода на 32-битовом компиляторе. Если вы используете 64-битовый компилятор и запустите программу на 64-битовой системе, то размер вашего указателя может составить 64 бита, т.е. 8 байтов.

Динамическое распределение памяти

Когда вы пишете программу, содержащее такое объявление массива, как

```
int Numbers[100]; // статический массив для 100 целых чисел
```

у вашей программы есть две проблемы.

1. Так вы фактически ограничиваете емкость своей программы, поскольку она не сможет хранить больше 100 чисел.

2. Вы ухудшаете производительность системы в случае, когда храниться должно только 1 число, а пространство все равно резервируется для 100 чисел.

Причиной этих проблем является статическое, фиксированное резервирование памяти для массива компилятором, как уже обсуждалось ранее.

Чтобы программа могла оптимально использовать ресурсы памяти, в зависимости от потребностей пользователя, необходимо использовать динамическое распределение памяти. Это позволит при необходимости резервировать больше памяти и освобождать ее, когда необходимости в ней больше нет. Язык C++ предоставляет два оператора, `new` и `delete`, позволяющие подробно контролировать использование памяти в вашем приложении. Указатели, являющиеся переменными, хранящими адреса памяти, играют критически важную роль в эффективном динамическом распределении памяти.

Использование операторов `new` и `delete` для динамического резервирования и освобождения памяти

Оператор `new` используется для резервирования (распределения) новых блоков памяти. Чаще всего используется версия оператора `new`, возвращающая указатель на затребованную область память в случае успеха, и передающая исключение в противном случае. При использовании оператора `new` необходимо указать тип данных, для которого резервируется память:

```
Тип* Указатель = new Тип; // запрос памяти для одного элемента
```

Вы можете также определить количество элементов, для которых хотите зарезервировать память (если нужно резервировать память для нескольких элементов):

```
Тип* Указатель = new Тип[КолЭлементов]; // запрос памяти для КолЭлементов
```

Таким образом, если необходимо разместить в памяти целые числа, используйте следующий код:

```
int* pNumber = new int;          // получить указатель на целое число
int* pNumbers = new int[10];     // получить указатель на блок из 10
                                 // целых чисел
```

ПРИМЕЧАНИЕ

Обратите внимание на то, что оператор `new` только запрашивает область памяти. Нет никакой гарантии, что запрос всегда будет удовлетворен успешно, поскольку это зависит от состояния системы и доступности ресурсов памяти.

Каждая область памяти, зарезервированная оператором `new`, должна быть в конечном счете освобождена (очищена) соответствующим оператором `delete`:

```
Тип* Указатель = new Тип;
delete Указатель; // освобождение памяти, зарезервированной
                  // ранее для одного экземпляра Типа
```

Это правило применимо также при запросе памяти для нескольких элементов:

```
Тип* Указатель = new Тип[КолЭлементов];
delete[] Указатель; // освободить зарезервированный ранее блок
```

ПРИМЕЧАНИЕ

> Обратите внимание на применение оператора `delete[]` при резервировании блока с использованием оператора `new[...]` и оператора `delete` при резервировании только одного элемента с использованием оператора `new`.

Если не освободить зарезервированную память после прекращения ее использования, то она так и останется зарезервированной для вашего приложения даже после его завершения. Это, в свою очередь, сократит объем системной памяти, доступной для использования другими приложениями, а возможно, даже замедлит выполнение вашего приложения. Это называется утечкой памяти, и ее следует избегать любой ценой.

В листинге 8.7 показано динамическое распределение и освобождение памяти.

ЛИСТИНГ 8.7. Использование оператора (*) для доступа к области памяти, зарезервированной оператором `new`, и ее освобождение оператором `delete`

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Запрос области памяти для int
6:     int* pAge = new int;
7:
8:     // Использование распределенной память для хранения числа
9:     cout << "Enter your dog's age: ";
10:    cin >> *pAge;
11:
12:    // использование оператора косвенного доступа * для
       // обращения к значению
13:    cout << "Age " << *pAge << " is stored at 0x" << hex << pAge
           << endl;
14:
15:    delete pAge; // освобождение памяти
16:
17:    return 0;
18: }
```

Результат

```
Enter your dog's age: 9
Age 9 is stored at 0x00338120
```

Анализ

Строка 6 демонстрирует использование оператора `new` для запроса области под целое число, в которой планируется хранить введенный пользователем возраст собаки. Обратите внимание, что оператор `new` возвращает указатель, вот почему осуществляется присвоение. Введенный пользователем возраст сохраняется в этой только что зарезервированной области памяти, а оператор `cin` обращается к ней в строке 10, используя оператор *. Строка 13 отображает значение возраста, снова используя оператор обращения к значению (*), а также отображает адрес области памяти, где оно хранится. Содержавшийся в указателе `pAge` адрес в строке 13 остается все тем же, который был возвращен оператором `new` в строке 6, — он с тех пор не изменился.

ВНИМАНИЕ!

Оператор `delete` не может быть вызван ни для какого адреса, содержащегося в указателе, кроме тех, и только тех, которые были возвращены оператором `new` и еще не были освобождены оператором `delete`.

Таким образом, несмотря на то что указатели в листинге 8.6 содержат допустимые адреса, все же их нельзя освобождать при помощи оператора `delete`, поскольку они не были возвращены вызовом оператора `new`.

Обратите внимание, что при резервировании памяти для диапазона элементов с использованием оператора `new...[...]` вы освобождаете ее, используя оператор `delete[]`, как показано в листинге 8.8.

ЛИСТИНГ 8.8. Резервирование с использованием оператора `new[...]` и освобождение с использованием оператора `delete[]`

```
0: #include <iostream>
1: #include <string>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Enter your name: ";
7:     string Name;
8:     cin >> Name;
9:
10:    // Добавить 1 к резервируемому объему памяти для завершающего
       // нулевого символа
11:    int CharsToAllocate = Name.length() + 1;
12:
13:    // запрос памяти для содержания копии ввода
14:    char* CopyOfName = new char [CharsToAllocate];
15:
16:    // strcpy копирует из строки с завершающим нулевым символом
17:    strcpy(CopyOfName, Name.c_str());
18:
19:    // Отобразить скопированную строку
20:    cout << "Dynamically allocated buffer contains: " << CopyOfName
           << endl;
21:
22:    // Буфер больше не используется? Удалить его
23:    delete[] CopyOfName;
24:
25:    return 0;
26: }
```

Результат

```
Enter your name: Siddhartha
Dynamically allocated buffer contains: Siddhartha
```

Анализ

Самыми важными являются строки 11 и 23, где используются операторы new и delete[] соответственно. По сравнению с листингом 8.7, где резервировалось место только для одного элемента, здесь резервируется блок памяти для нескольких элементов. Такому резервированию массива элементов должно соответствовать освобождение с использованием оператора delete[], чтобы освободить память по завершении ее использования. Количество резервируемых символов вычисляется в строке 11 как на единицу большее, чем количество символов, введенных пользователем, чтобы разместить завершающий символ NULL, который важен в строках стиля C. Потребность в завершающем символе NULL объяснялась на занятии 4, " Массивы и строки". Фактическое копирование осуществляется в строке 17 оператором strcpy, который использует метод c_str() строки Name, предоставляемый классом std::string, чтобы скопировать ее в символьный буфер CopyOfName.

ПРИМЕЧАНИЕ

Операторы new и delete резервируют область в динамической памяти. *Динамическая память* (free store) – это абстракция памяти в форме пула памяти, который ваше приложение может *распределять* (allocate) (т.е. *резервировать* (reserve) области) и *возвращать* (de-allocate) (т.е. *освобождать* (release)).

Воздействие операторов инкремента и декремента (++ и -) на указатели

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 места от 0x002EFB34 до 0x002EFB37. Приращение значения этого указателя с использованием оператора (++) *не даст* значения 0x002EFB35, указывающего на середину целого числа, что 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 он имеет тот же тип, а *не* на следующий байт (если тип значения не имеет длину в 1 байт, как, например тип char).

Так, результатом инкремента такого указателя, как pInteger 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 перевести его на следующее расположенное далее целое число. Следовательно, после приращения указатель будет указывать на адрес 0x002EFB38. Точно так же добавление 2 к этому указателю привело бы к его переводу на 2 целых числа далее, что составит 8 байтов вперед. Впоследствии вы увидите корреляцию между этим поведением, демонстрируемым указателями, и индексами, используемыми в массивах.

Декремент указателя с использованием оператора (-) демонстрирует тот же эффект — значение, содержавшееся в указателе адреса, 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 (и не обязательно 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 так:

```
Тип* pТип = Адрес;
```

то оператор `++pТип` означал бы, что указатель `pТип` содержит адрес (указывает на) `Адрес + sizeof(Тип)`.

В листинге 8.9 показан результат инкремента указателей или добавления смещений к ним.

ЛИСТИНГ 8.9. Динамическое резервирование на основании потребности, исследование приращения указателей при помощи значений смещения и оператора ++

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "How many integers you wish to enter? ";
 6:     int InputNums = 0;
 7:     cin >> InputNums;
 8:
 9:     int* pNumbers = new int [InputNums]; // резервирование требуемого
                                             // количества целых чисел
10:     int* pCopy = pNumbers;
11:
12:     cout<<"Successfully allocated memory for "<<InputNums
            << " integers" << endl;
13:     for(int Index = 0; Index < InputNums; ++Index)
14:     {
15:         cout << "Enter number "<< Index << ": ";
16:         cin >> *(pNumbers + Index);
17:     }
18:
19:     cout << "Displaying all numbers input: " << endl;
20:     for(int Index = 0, int* pCopy = pNumbers;
                Index < InputNums; ++Index)
21:         cout << *(pCopy++) << " ";
22:
23:     cout << endl;
24:
25:     // указатель больше не используется? Освободить память
26:     delete[] pNumbers;
27:
28:     return 0;
29: }
```

Результат

```
How many integers you wish to enter? 2
Successfully allocated memory for 2 integers
Enter number 0: 789
Enter number 1: 575
Displaying all numbers input:
789 575
```

Другой запуск:

```
How many integers you wish to enter? 5
Successfully allocated memory for 5 integers
Enter number 0: 789
Enter number 1: 12
Enter number 2: -65
Enter number 3: 285
Enter number 4: -101
Displaying all numbers input:
789 12 -65 285 -101
```

Анализ

Программа запрашивает у пользователя количество целых чисел, которые он хочет ввести в систему, прежде чем резервировать память для них в строке 9. Обратите внимание, как мы сохраняем резервную копию этого адреса в строке 10, которая используется впоследствии при освобождении этого блока памяти оператором `delete` в строке 26. Эта программа демонстрирует преимущество использования указателей и динамического распределения памяти перед статическим массивом. Когда пользователь желает хранить меньше чисел, данное приложение использует меньше памяти; когда чисел больше, он резервирует больше памяти, но никогда не растрачивает ее впустую. Благодаря динамическому распределению нет никакого верхнего предела для количества хранимых чисел, если только они полностью не исчерпают системные ресурсы. Строки 13–17 содержат цикл `for`, где пользователя просят ввести числа, которые затем, в строке 16, сохраняют их последовательно в памяти, используя выражение. Именно здесь отсчитываемое от нуля значение смещения (`Index`) добавляется к указателю, заставляя вставлять введенное пользователем значение в соответствующую область памяти, не перезаписывая предыдущее значение. Другими словами, выражение `(pNumber + Index)` возвращает указатель на целое число в отсчитываемой от нуля индексной области в памяти (т.е. индекс 1 принадлежит второму числу), а следовательно, оператор обращения к значению `* (pNumber + Index)` и является тем выражением, которое оператор `cin` использует для доступа к значению по отсчитываемому от нуля индексу. Цикл `for` в строках 20 и 21 подобным образом отображает эти значения, сохраненные предыдущим циклом. Создавая копию в указателе `pCopy` и увеличивая ее содержимое в строке 21, чтобы отобразить значение, цикл `for` использует несколько выражений инициализации.

Причина создания копии в строке 10 в том, что цикл изменяет указатель, используемый в операторе инкремента (++). Исходный указатель, возвращенный оператором `new`, должен храниться неповрежденным для использования в операторе `delete[]` (строка 26), где должен быть использован адрес, возвращенный оператором `new`, а не любое произвольное значение.

Использование ключевого слова const с указателями

На занятии 3, "Использование переменных, объявление констант", вы узнали, что объявление переменной как const фактически гарантирует, что значение переменной фиксируется после ее инициализации. Значение такой переменной не может быть изменено, и она не может использоваться как l-значение.

Указатели — это тоже переменные, а следовательно, ключевое слово const также вполне уместно для них. Однако указатели — это особый вид переменной, которая содержит адрес области памяти и позволяет модифицировать совокупность данных в памяти. Таким образом, когда дело доходит до указателей и констант, возможны следующие комбинации.

- Данные, на которые указывает указатель, являются постоянными 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* pInteger = &HoursInDay; // нельзя использовать pInteger
                                   // для изменения HoursInDay
int MonthsInYear = 12;
pInteger = &MonthsInYear;              // OK!
*pInteger = 13;                        // Ошибка компиляции: нельзя
                                       // изменять данные
int* pAnotherPointerToInt = pInteger;  // Ошибка компиляции: нельзя
                                  // присвоить константу не константе
```

- Содержащийся в указателе адрес является постоянным и не может быть изменен, однако данные, на которые он указывает, вполне могут быть изменены:

```
int DaysInMonth = 30;
// pInteger не может указать ни на что иное
int* const pDaysInMonth = &DaysInMonth;
*pDaysInMonth = 31;                // OK! Значение может быть изменено
int DaysInLunarMonth = 28;
pDaysInMonth = &DaysInLunarMonth; // Ошибка компиляции: нельзя
                                  // изменить адрес!
```

- И содержащийся в указателе адрес, и значение, на которое он указывает, являются константами и не могут быть изменены (самый ограничивающий вариант):

```
int HoursInDay = 24;
// указатель может указать только на HoursInDay
const int* const pHoursInDay = &HoursInDay;
*pHoursInDay .= 25;       // Ошибка компиляции: нельзя изменить
                          // значение, на которое указывает указатель
int DaysInMonth = 30;
pHoursInDay = &DaysInMonth; // Ошибка компиляции: нельзя изменить
                            // значение указателя
```

Эти разнообразные формы константности особенно полезны при передаче указателей функциям. Параметры функций следует объявлять, обеспечивая максимально возможный (ограничивающий) уровень константности, чтобы гарантировать невозможность функции изменить значение, на которое указывает указатель, если это не предполагалось. Это облегчает сопровождение функций, особенно со временем и с учетом возможных изменений, внесенных в программу.

Передача указателей в функции

Указатели — это эффективный способ передачи функции областей памяти, содержащих значения и способных содержать результат. При использовании указателей с функциями важно гарантировать, что вызываемой функции позволено изменять только те параметры, которые вы хотите изменить, но не другие. Например, функции, вычисляющей площадь круга по радиусу, переданному как указатель, нельзя позволить изменять радиус. Вот где пригодятся константные указатели, позволяющие эффективно контролировать то, что функции позволено изменять, а что — нет (листинг 8.10).

Листинг 8.10. Использование ключевого слова `const` при вычислении площади круга, когда радиус и число Пи передаются как указатели

```
1: #include <iostream>
2: using namespace std;
3:
4: void CalcArea(const double* const pPi,   // Константный указатель на
5:                const double* const pRadius, // константные данные,
6:                                   // т.е. ничто не может быть изменено
7:                double* const pArea) // изменяемо значение, но не адрес
8: {
9:     // проверить указатели перед использованием!
10:    if (pPi && pRadius && pArea)
11:        *pArea = (*pPi) * (*pRadius) * (*pRadius);
12: }
13:
14: int main()
15: {
16:    const double Pi = 3.1416;
17:
18:    cout << "Enter radius of circle: ";
19:    double Radius = 0;
20:    cin >> Radius;
21:
22:    double Area = 0;
23:    CalcArea (&Pi, &Radius, &Area);
24:
25:    cout << "Area is = " << Area << endl;
26:
27:    return 0;
28: }
```

Результат

```
Enter radius of circle: 10.5
Area is = 346.361
```

Анализ

Строки 3–5 демонстрируют две формы константности, где оба указателя, `pRadius` и `pPi`, передаются как "константные указатели на константные данные", так что ни адрес в указателе, ни данные, на которые он указывает, не могут быть изменены. Указатель `pArea`,

вполне очевидно, — это параметр, предназначенный для хранения вывода. Значение указателя (адрес) не может быть изменено, но данные, на которые он указывает, могут. Строка 7 демонстрирует проверку на допустимость переданных функции указателей перед их использованием. Вы бы, наверное, не хотели, чтобы функция вычисляла площадь, если вызывающая сторона по неосторожности передаст пустой указатель как любой из этих трех параметров, поскольку это привело бы к аварийному отказу приложения.

Сходство между массивами и указателями

Разве не кажется, что у примера в листинге 8.9, где осуществлялся инкремент указателя с использованием отсчитываемого от нуля индекса для доступа к следующему числу в памяти, слишком много сходств со способом индексирования массива? Когда вы объявляете массив целых чисел так:

```
int MyNumbers[5];
```

компилятор сам резервирует фиксированный объем памяти для содержания пяти целых чисел и предоставляет вам указатель на первый элемент в этом массиве, идентифицируемый именем, назначенным переменной типа массива. Другими словами, `MyNumbers` — это указатель на первый элемент `MyNumber[0]`. Листинг 8.11 подчеркивает эту корреляцию.

ЛИСТИНГ 8.11. Демонстрация того, что переменная типа 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* pNumbers = MyNumbers;
10:
11:     // Отображение адреса, содержащегося в указателе
12:     cout << "pNumbers = 0x" << hex << pNumbers << endl;
13:
14:     // Адрес первого элемента массива
15:     cout << "&MyNumbers[0] = 0x" << hex << &MyNumbers[0] << endl;
16:
17:     return 0;
18: }
```

Результат

```
pNumbers = 0x004BFE8C
&MyNumbers[0] = 0x004BFE8C
```

Анализ

Эта простая программа демонстрирует, что переменная типа 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). Эта программа подтверждает, что массив — это указатель на первый его элемент.

Если необходимо получить доступ ко второму элементу, `MyNumbers[1]`, то можно воспользоваться указателем `pNumbers` в операторе `*(pNumbers + 1)`. К третьему элементу статического массива обращаются, используя синтаксис `MyNumbers[2]`, тогда как к третьему элементу динамического массива можно обратиться с помощью синтаксиса `*(pNumbers + 2)`.

Поскольку переменные типа массива — это, по существу, указатели, при работе с массивами вполне возможно использовать оператор обращения к значению (*), как при работе с обычными указателями. Точно так же вполне возможно использовать оператор массива ([]) при работе с указателями, как показано в листинге 8.12.

ЛИСТИНГ 8.12. Доступ к элементам массива с использованием оператора обращения к значению (*) и использование оператора массива ([]) при работе с указателем

```
1: #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* pNumbers = MyNumbers;
12:
13:    cout << "Displaying array using pointer syntax, operator*"
           << endl;
14:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
15:        cout << "Element " << Index << " = " << *(MyNumbers + Index)
               << endl;
16:
17:    cout <<
          "Displaying array using pointer with array syntax, operator[]"
          << endl;
18:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
19:        cout << "Element " << Index << " = " << pNumbers[Index]
               << endl;
20:
21:    return 0;
22: }
```

Результат

```
Displaying array using pointer syntax, operator*
Element 0 = 24
Element 1 = -1
Element 2 = 365
Element 3 = -999
Element 4 = 2011
Displaying array using pointer with array syntax, operator[]
Element 0 = 24
Element 1 = -1
Element 2 = 365
Element 3 = -999
Element 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 типа массива и оператора косвенного доступа (*) в строке 15, а также с использованием переменной указателя и оператора массива ([]) в строке 19.

Таким образом, эта программа свидетельствует, что и массив `MyNumbers`, и указатель `pNumbers` 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 в строке 11, но нельзя присвоить указатель массиву, поскольку массив имеет статической характер, а следовательно, не может быть l-значением.

ВНИМАНИЕ!

> Не забывайте, что указатели, созданные динамически при помощи оператора `new`, следует освободить при помощи оператора `delete`, даже если вы использовали такой синтаксис, как у статического массива.
>
> Если вы забудете, то произойдет утечка памяти, а это плохо.

Наиболее распространенные ошибки при использовании указателей

Язык C++ позволяет резервировать память динамически, чтобы использование памяти вашим приложением было оптимальным. В отличие от более новых языков, таких как C# и Java, работающих на базе среды времени выполнения, язык C++ не использует автоматический сборщик мусора, который очищает зарезервированную вашей программой память, когда она уже не используется. Поскольку указатели способны на разные трюки, у программиста есть масса возможностей сделать ошибки.

Утечки памяти

Вероятно, это одна из самых распространенных проблем приложений C++: чем дольше они выполняются, тем больший объем памяти используют и замедляют систему. Это, как правило, случается, когда программист не гарантировал в приложении освобождение памяти, зарезервированной динамически оператором new, при помощи вызова оператора delete по завершении ее использования.

Это задача программиста, т.е. ваша, обеспечить освобождение всей зарезервированной вашим приложением памяти. Кое-чему никогда нельзя позволять случаться:

```
int* pNumbers = new int [5]; // начальное резервирование
// использование указателя pNumbers
...
// забыто освобождение при помощи оператора delete[] pNumbers;
...
// следующее резервирование и перезапись указателя
pNumbers = new int[10]; // утечка предварительно зарезервированной памяти
```

Когда указатели указывают на недопустимые области памяти

Когда вы используете оператор (*) для доступа к значению, на которое указывает указатель, необходимо быть на 100% уверенным в том, что указатель содержит допустимый адрес области памяти, иначе поведение вашей программы будет непредсказуемым. Как ни странно это звучит, но недопустимые указатели являются наиболее частой причиной аварийных отказов приложения. Указатели могут оказаться недопустимыми по ряду причин, которые связаны с плохим управлением памятью. Типичный случай, где указатель мог бы оказаться недопустимым, приведен в листинге 8.13.

ЛИСТИНГ 8.13. Пример плохого программирования с использованием недопустимых указателей

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // неинициализированный указатель (плохо)
6:     int* pTemperature;
7:
8:     cout << "Is it sunny (y/n)?" << endl;
9:     char UserInput = 'y';
10:    cin >> UserInput;
11:
12:    if (UserInput == 'y')
13:    {
14:        pTemperature = new int;
15:        *pTemperature = 30;
16:    }
17:
18:    // pTemperature содержит недопустимое значение,
       // если пользователь ввел 'n'
```

```
19:     cout << "Temperature is: " << *pTemperature;
20:
21:     // оператор delete также может быть вызван для указателя
        // без применения оператора new
22:     delete pTemperature;
23:
24:     return 0;
25: }
```

Результат

```
Is it sunny (y/n)? y
Temperature is: 30
```

Второй запуск:

```
Is it sunny (y/n)? n
<АВАРИЯ!>
```

Анализ

В программе много проблем, некоторые из которых уже прокомментированы в коде. Обратите внимание, что память в строке 14 резервируется и ее адрес присваивается указателю в блоке `if` только тогда, когда его условие выполняется, т.е. когда пользователь, соглашаясь, нажимает клавишу <y>. При любом другом нажатии пользователем этот блок `if` не выполняется, и указатель `pTemperature` остается недопустимым. Таким образом, когда во второй раз пользователь нажимает клавишу <n>, приложение завершается отказом, поскольку указатель `pTemperature` содержит недопустимый адрес области памяти и обращение к значению по недопустимому указателю в строке 19 создает проблему.

Аналогично вызов оператора `delete` в строке 22 для этого указателя, который не был инициализирован оператором `new`, также является неправильным. Обратите внимание на то, что, если у вас есть копия указателя, вызов оператора `delete` необходим только для одного из них (наличие неконтролируемых копий указателей также следует избегать).

Лучшая (более безопасная и стабильная) версия программы из листинга 8.13 подразумевала бы инициализацию указателей, проверку допустимости их значений и освобождение только однажды и только когда допустимо.

Потерянные указатели (они же беспризорные или дикие)

Любой допустимый указатель становится недопустимым после того, как он был освобожден оператором `delete`. Так, если бы указатель `pTemperature` был использован после строки 22, даже в том случае, если пользователь действительно нажал клавишу <y> и указатель был допустимым до этого момента, после вызова оператора `delete` он становится неприменимым и не должен использоваться.

Чтобы избежать этой проблемы, большинство программистов присваивают указателю при инициализации и после его освобождения значение NULL, а затем, используя его, проверяют указатель на допустимость перед обращением к значению с использованием оператора (*).

Полезные советы по применению указателей

Когда дело доходит до использования указателей в приложении, придерживайтесь простых правил, которые упростят вам жизнь.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
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 `new` или другой допустимой переменной, инициализируйте его значением NULL	**Не используйте** указатель для обращения к блоку памяти после того, как он был освобожден оператором `delete`
Проверяйте указатели на значение NULL перед их использованием. Это гарантирует, что указатели, не получившие допустимый адрес в выражениях после их объявления (где они были инициализированы значением NULL), не будут использованы (как это произошло в листинге 8.13)	**Не применяйте** оператор `delete` для того же адреса повторно
Используйте указатели правильно, гарантируя их допустимость, иначе ваша программа столкнется с отказом	**Не допускайте** утечки памяти, забыв вызвать оператор `delete` для зарезервированного ранее блока памяти
Помните о необходимости освобождать память, зарезервированную оператором `new`, при помощи оператора `delete`, иначе ваше приложение создаст утечку памяти и уменьшит производительность системы	

Ознакомившись с приведенными выше полезными советами по использованию указателей, мы должны исправить чрезвычайно дефектный код листинга 8.13 и использовать код, представленный в листинге 8.14.

ЛИСТИНГ 8.14. Более безопасная программа, исправленный листинг 8.13

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Is it sunny (y/n)? ";
7:     char UserInput = 'y';
8:     cin >> UserInput;
9:
10:    if (UserInput == 'y')
11:    {
```

```
11:         // инициализированный указатель (хорошо)
12:         int* pTemperature = new int;
13:         *pTemperature = 30;
14:
15:         cout << "Temperature is: " << *pTemperature << endl;
16:
17:         // указатель больше не используется? удалить
18:         delete pTemperature;
19:     }
20:
21:     return 0;
22: }
```

Результат

```
Is it sunny (y/n)? y
Temperature is: 30
```

Следующий запуск:

```
Is it sunny (y/n)? n
(Удачное завершение)
```

Анализ

Основное различие здесь в том, что указатель создается только по мере необходимости (т.е. когда пользователь нажимает клавишу <y>) и при создании инициализируется (см. строку 12). Освобождение памяти происходит в том же блоке, поэтому нет никаких случаев использования указателя (при обращении к значению или в вызове оператора `delete`), когда ему не был присвоен адрес допустимой области памяти.

Проверка успешности запроса с использованием оператора new

Оператор `new` выполняется успешно, только если не был запрошен необычно большим объемом памяти или если система не находится в критическом состоянии, когда у нее есть повод экономить память. 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 допустимости указателя. Основной способ подразумевает использование исключения. При неудаче резервирования памяти передается исключение класса `std::bad_alloc`. В результате выполнение приложения будет прервано с сообщением об ошибке наподобие `unhandled exception` (необработанное исключение), если только вы не создали *обработчик исключения* (exception handler).

Подробно решение этой проблемы обсуждается на занятии 28, "Обработка исключений". Листинг 8.15 дает общее представление о применении обработки исключений для проверки успеха запроса на резервирование.

ЛИСТИНГ 8.15. Обработка исключения как изящный выход из ситуации при неудаче оператора `new`

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     try
 6:     {
 7:         // Запрос большой области памяти
 8:         int* pAge = new int [536870911];
 9:
10:         // Использование предоставленной памяти
11:
12:         delete[] pAge;
13:     }
14:     catch (bad_alloc)
15:     {
16:         cout << "Memory allocation failed. Ending program" << endl;
17:     }
18:     return 0;
19: }
```

Результат

```
Memory allocation failed. Ending program
```

Анализ

На вашем компьютере эта программа могла бы выполниться по-другому. Система автора не смогла успешно зарезервировать пространство для 536870911 целых чисел. Без обработчика исключения (блока `catch`, расположенного в строках 14–17) программа закончилась бы отказом. В режиме отладки среды разработки Microsoft Visual Studio выполнение программы привело бы к результату, показанному на рис. 8.2.

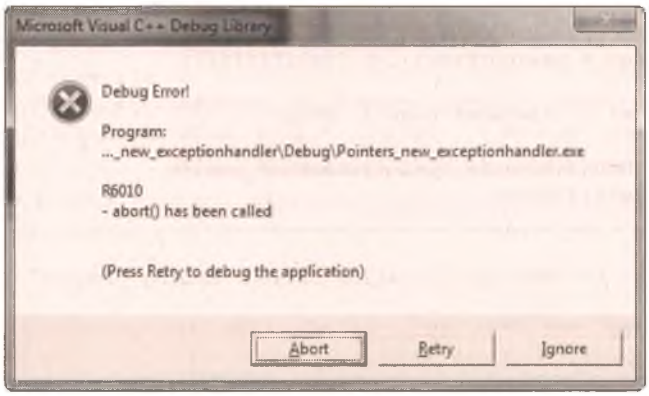


РИС. 8.2. Аварийный отказ программы при отсутствии обработки исключений в листинге 8.15 (отладка с использованием компилятора MSVC)

Отладчик в среде разработки имеет обработчик исключений, и он вмешался в процесс выполнения, отобразив приведенное выше сообщение. Выполнение откомпилированной версии операционной системой (в данном случае Windows) заканчивается весьма просто, как показано на рис. 8.3.

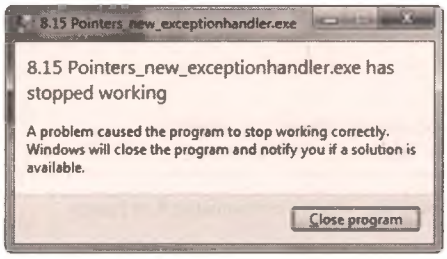


РИС. 8.3. Аварийный отказ программы при отсутствии обработки исключений в листинге 8.15 (откомпилированная версия)

Когда ваше приложение откажет, как показано выше, операционная система закроет его, и в отсутствии обработчика исключения у вас не будет шанса даже попрощаться.

Наличие обработчика исключения предоставило бы приложению способ корректного выхода после извещения пользователя о возникшей проблеме, а не позволяло бы операционной системе самой отображать сообщение об аварийном отказе.

Есть также вариант оператора new, называемый new(nothrow), который не передает исключение, а возвращает указателю значение NULL, которое можно заметить при проверке на допустимость, прежде чем использовать указатель (листинг 8.16).

ЛИСТИНГ 8.16. Использование оператора new(nothrow), возвращающего при неудаче значение NULL

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     // Запрос большой области памяти с использованием версии nothrow
 6:     int* pAge = new(nothrow) int [0x1fffffff];
 7:
 8:     if (pAge) // проверка pAge != NULL
 9:     {
10:         // Использование предоставленной памяти
11:         delete[] pAge;
12:     }
13:     else
14:         cout << "Memory allocation failed. Ending program" << endl;
15:
16:     return 0;
17: }
```

Результат

```
Memory allocation failed. Ending program
```

Анализ

Та же программа, использующая оператор new(nothrow), возвращающий при неудаче распределения памяти значение NULL, а не передающий исключение класса std::bad_alloc, как в листинге 8.15. Обе формы хороши, и выбор остается за вами.

Что такое ссылка

Ссылка (reference) — это псевдоним переменной. При объявлении ссылки ее необходимо инициализировать переменной. Таким образом, ссылочная переменная — это только другой способ доступа к данным, хранимым в переменной.

Для объявления ссылки используется оператор ссылки (&), 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. Демонстрация того, что ссылки — это псевдонимы для значений примененных

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     int Original = 30;
7:     cout << "Original = " << Original << endl;
8:     cout << "Original is at address: " << hex << &Original << endl;
9:
10:    int& Ref = Original;
11:    cout << "Ref is at address: " << hex << &Ref << endl;
12:
13:    int& Ref2 = Ref;
14:    cout << "Ref2 is at address: " << hex << &Ref2 << endl;
15:    cout << "Ref2 gets value, Ref2 = " << dec << Ref2 << endl;
16:
17:    return 0;
18: }
```

Результат

```
Original = 30
Original is at address: 0044FB5C
Ref is at address: 0044FB5C
Ref2 is at address: 0044FB5C
Ref2 gets value, Ref2 = 30
```

Анализ

Вывод показывает, что ссылки, независимо от того, инициализируются ли они исходной переменной, как можно заметить в строке 9, или присвоением ссылки, как в строке 12, ссылаются на ту же область в памяти, где содержится исходное значение. Таким образом, ссылки — это настоящие псевдонимы, т.е. просто другое имя 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 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 уже было сказано на занятии 7, "Организация кода при помощи функций", типичная функция объявляется так:

```
ТипВозвращаемогоЗначения СделатьНечто(Тип Параметр);
```

Функция *СделатьНечто*() вызывается так:

```
ТипВозвращаемогоЗначения Результат = СделатьНечто(аргумент); // вызов функции
```

Приведенный выше код привел бы к копированию аргумента в параметр, который затем использовался бы функцией *СделатьНечто*(). Этап копирования может быть весьма продолжительным, если рассматриваемый *аргумент* занимает много памяти. Аналогично, когда функция *СделатьНечто*() возвращает значение, оно копируется в *Результат*. Было бы хорошо, если бы удалось избежать или устранить этапы копирования, позволив функции воздействовать непосредственно на данные в стеке вызывающей стороны. Ссылки позволяют сделать именно это.

Версия функции без этапа копирования выглядит следующим образом:

```
ТипВозвращаемогоЗначения СделатьНечто(Тип& Параметр); // обратите внимание на ссылку&
```

Эта функция была бы вызвана так:

```
ТипВозвращаемогоЗначения Результат = СделатьНечто(аргумент);
```

Поскольку аргумент передается по ссылке, параметр не будет копией аргумента, а скорее псевдонимом последнего, как `Ref` в листинге 8.17. Кроме того, функция, которая получает параметр как ссылку, может возвращать значение, используя ссылочные параметры. Рассмотрим листинг 8.18, чтобы понять, как функции могут использовать ссылки вместо возвращаемых значений.

ЛИСТИНГ 8.18. Функция вычисляет квадрат числа и возвращает его в параметре по ссылке

```
0: #include <iostream>
1: using namespace std;
2:
3: void ReturnSquare(int& Number)
4: {
5:     Number *= Number;
6: }
7:
8:
```

```
int main()
{
    cout << "Enter a number you wish to square: ";
    int Number = 0;
    cin >> Number;

    ReturnSquare(Number);
    cout << "Square is: " << Number << endl;

    return 0;
}
```

Результат

```
Enter a number you wish to square: 5
Square is: 25
```

Анализ

Функция, вычисляющая квадрат числа, находится на 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 отметить параметр `Number` как ссылку (`&`), то результат не достигнет вызывающей функции `main()`, поскольку функция `ReturnSquare()` выполнит свои действия с локальной копией переменной `Number`, которая будет удалена при выходе из функции. Используя ссылки, вы позволяете функции `ReturnSquare()` работать в том же пространстве адресов, где определена переменная `Number` в функции `main()`. Таким образом, результат доступен в функции `main()` даже после того, как функция `ReturnSquare()` закончила работу.

В этом примере был изменен входной параметр, содержащий число, переданное пользователем. При необходимости оба значения, исходное и квадрат, могут быть двумя параметрами функции: один получает ввод, а другой предоставляет квадрат.

Использование ключевого слова `const` со ссылками

Возможны ситуации, когда ссылке не позволено 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 воздействовать на параметры, которые не копируются 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, позволяют сделать именно это, как показано в листинге 8.19. Константная ссылка не может использоваться как l-значение, поэтому любая попытка его присвоения вызовет ошибку компиляции.

ЛИСТИНГ 8.19. Использование константной ссылки для гарантии невозможности вызываемой функции изменить значение, переданное по ссылке

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: void CalculateSquare(const int& Number, int& Result)
                                                    // заметьте, "const"
 4: {
 5:     Result = Number*Number;
 6: }
 7:
 8: int main()
 9: {
10:     cout << "Enter a number you wish to square: ";
11:     int Number = 0;
12:     cin >> Number;
13:
14:     int Square = 0;
15:     CalculateSquare(Number, Square);
16:     cout << Number << "^2 = " << Square << endl;
17:
18:     return 0;
19: }
```

Результат

```
Enter a number you wish to square: 27
27^2 = 729
```

Анализ

В отличие от предыдущей программы, где передававшая число переменная содержала также и результат, здесь используются две переменные: одна для передачи числа, которое будет возведено в квадрат, а вторая для содержания результата. Для гарантии неизменности передаваемого числа оно было помечено как константная ссылка ключевым словом const (см. строку 3). Это автоматически делает параметр Number *входным параметром* (input parameter), значение которого не может быть изменено.

В качестве эксперимента можете изменить строку 5 так, чтобы возвратить 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 обозначения 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 пишет первую версию, второй — вторую, третий исправляет или совершенствует ее, 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 манипулирования ими, а также узнали, что указатели — это инструмент динамического распределения памяти. Вы изучили операторы `new` и `delete`, применяемые при резервировании памяти для элемента, а также их варианты `new...[]` и `delete[]`, позволяющие резервировать 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. Ссылки — это псевдонимы, являющиеся мощной альтернативой использованию указателей при передаче аргументов функциям, поскольку они гарантируют их допустимость. Вы узнали о "правильности констант" при использовании указателей и ссылок, и надеюсь, впредь будете объявлять функции с самым ограничивающим уровнем константности параметром из всех возможных.

Вопросы и ответы

- **Зачем нужно динамическое резервирование, когда все необходимое можно сделать со статическими массивами и не нужно заботиться об их освобождении?**

  Статические массивы имеют фиксированный размер; они не будут увеличиваться, если приложению понадобится больше памяти, и при этом окажутся слишком расточительными, если приложение нуждается в меньшем количестве элементов. Вот где динамическое распределение памяти имеет значение.

- **У меня есть два указателя:**

```
int* pNumber = new int;
int* pCopy = pNumber;
```

  Полагаю, после использования будет лучше освободить их оба, чтобы гарантированно избежать утечки памяти?

  Это было бы неправильным. Разрешено применять оператор `delete` только один раз для каждого адреса, возвращенного оператором `new`. Кроме того, желательно избегать наличия двух указателей на тот же адрес, поскольку выполнение оператора `delete` для любого из них сделает недействительным другой. Не нужно допускать в программах никаких неопределенностей в допустимости 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 NULL при неудаче резервирования памяти.

- **Я могу вызвать функцию для вычисления площади, используя следующие два способа:**

```
void CalculateArea (const double* const pRadius, double* const pArea);
void CalculateArea (const double& radius, double& area);
```

Какой вариант следует предпочесть?

Последний, с использованием ссылок, так как он вполне допустим при отсутствии указателей. Кроме того, он проще.

- **У меня есть два указателя:**

```
int Number = 30;
const int* pNumber = &Number;
```

Я понимаю, что не могу изменить значение `Number`, используя указатель `pNumber`, поскольку он объявлен константным. Могу ли я присвоить указатель `pNumber` непостоянному указателю, а затем использовать его для манипулирования значением, содержавшимся в целочисленной переменной `Number`?

Нет, вы не можете изменить константность указателя:

```
int* pAnother = pNumber; // нельзя присвоить указатель на
                         // константу неконстантному указателю
```

- **Зачем мне передавать значения в функцию по ссылке?**

Это и не нужно, пока не настанет необходимость. Но если параметрами функции являются очень большие объекты (в байтах), то передача по значению потребовала бы весьма дорогого копирования. Вызов функции стал бы намного эффективней при использовании ссылок. Не забудьте использовать ключевое слово `const`, если только функция не должна сохранить результат в переменной.

- **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 (по индексу 0) из 100 целых чисел. Это статическая альтернатива для следующего:

```
int* MyNumbers = new int [100]; // динамически распределенный массив
// использование MyNumbers
delete MyNumbers;
```

`MyArrays`, напротив, 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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "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. `new` и `delete` — это функции?

3. Каков характер значения, содержащегося в переменной указателя?
4. Какой оператор используется для доступа к данным, на которые указывает указатель?

Упражнения

1. Что будет отображено при выполнении этих операторов:

```
0: int Number = 3;
1: int* pNum1 = &Number;
2: _*pNum1 = 20;
3: int* pNum2 = pNum1;
4: Number *= 2;
5: cout << *pNum2;
```

2. В чем подобие и различие между этими 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 присвоение в строке 3 недопустимым? (*Подсказка:* это имеет некоторое отношение к обеспечению невозможности изменения данных, на которые указывает указатель pNum1.)
4. **Отладка:** Что не так с этим кодом?

```
#include <iostream>
using namespace std;
int main()
{
    int *pNumber = new int;
    pNumber = 9;
    cout << "The value at pNumber: " << *pNumber;
    delete pNumber;
    return 0;
}
```

5. **Отладка:** Что не так с этим кодом?

```
#include <iostream>
using namespace std;
int main()
{
    int pNumber = new int;
    int* pNumberCopy = pNumber;
    *pNumberCopy = 30;
    cout << *pNumber;
    delete pNumberCopy;
    delete pNumber;
    return 0;
}
```

6. Каков вывод приведенной выше программы после исправления?

ЧАСТЬ II

Фундаментальные принципы объектно-ориентированного программирования на C++

ЗАНЯТИЕ 9
Классы и объекты

До сих пор мы исследовали структуру простой программы, которая выполняет операторы в функции `main()`, позволяющие объявлять ло-альные и глобальные переменные, константы, структурировать логику выполнения в функциях, которые могут получать параметры и возвра-ать значения. Все это очень похоже на такой процедурный язык, как C. у которого нет ориентации на объекты. Другими словами, пришло время узнать об управлении данными и методах, связанных с этим.

На сегодняшнем занятии.

- Что такое классы.
- Как классы позволяют объединить данные с методами (аналогом функций) для работы с ними.
- Конструкторы, конструкторы копий и деструкторы.
- Как язык C++11 позволяет улучшить производительность при помощи конструктора перемещения.
- Объектно-ориентированные концепции инкапсуляции и абстракции.
- Что такое указатель `this`.
- Что такое структура и чем она отличается от класса.

Концепция классов и объектов

Предположим, вы пишете программу, моделирующую такого человека, как вы сами. У человека должна быть индивидуальность: имя, дата рождения, место рождения и пол. Человек может выполнять определенные действия, например говорить и представляться другим людям. Таким образом, первая часть информации — это данные о человеке, а вторая — это его функции (рис. 9.1).

РИС. 9.1. Общее представление человека

Чтобы смоделировать человека, нужна конструкция, позволяющая сгруппировать атрибуты, определяющие человека (данные), *и* действия, которые он может выполнять (методы, подобные функциям), используя доступные атрибуты. Эта конструкция называется *класс* (class).

Объявление класса

При объявлении класса используется ключевое слово `class`, сопровождаемое именем класса и блоком операторов `{...}`, который заключает в фигурные скобки набор атрибутов и методов, а завершается точкой с запятой.

Объявление класса сродни объявлению функции. Оно оповещает компилятор о классе и его свойствах. Только объявление класса никак не влияет на выполнение программы, поскольку класс следует использовать, как и функцию следует вызвать.

Моделирующий человека класс выглядит следующим образом (игнорируйте пока недостатки в синтаксисе):

```
class Human
{
    // Атрибуты данных:
    string Name;
    string DateOfBirth;
    string PlaceOfBirth;
    string Gender;
```

```
    // Методы:
    void Talk(string TextToTalk);
    void IntroduceSelf();
    ...
};
```

Само собой разумеется, метод `IntroduceSelf()` использует метод `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) — это способ логической группировки данных и методов, которые используют их; она является важнейшим качеством объектно-ориентированного программирования.

ПРИМЕЧАНИЕ *Методы* (method) – это, по существу, функции, являющиеся членами класса.

Создание экземпляра объекта класса

Класс похож на чертеж, и только его объявление не окажет никакого влияния на выполнение программы. Реальный предмет класса, который можно использовать в исполняющей среде, — это *объект* (object). Для использования средств класса создается экземпляр объекта этого класса, позволяющий получить доступ к его методам и атрибутам.

Экземпляр объекта типа `Human` можно создать в стеке, подобно экземпляру любого другого типа, скажем, `double`:

```
double Pi = 3.1415; // Объявление локальной переменной типа double
Human Tom;      // Объект класса Human объявлен как локальная переменная
```

В качестве альтернативы можно зарезервировать место для экземпляра класса `Human` в динамической памяти, используя оператор `new`, как и для типа `int`:

```
int* pNumber = new int;              // распределение динамической памяти
                                     // для целого числа
delete pNumber;                      // освобождение памяти
Human* pAnotherHuman = new Human(); // распределение динамической памяти
                                     // для объекта класса Human
delete pAnotherHuman;                // освобождение памяти, занятой
                                     // объектом класса Human
```

Доступ к членам класса с использованием точечного оператора (.)

Рассмотрим человека по имени Том, мужчину, родившегося в 1970 году в Алабаме. Экземпляр `Tom` — это объект класса `Human`, т.е. экземпляр класса, который существует в действительности и может использоваться в исполняющей среде:

```
Human Tom; // экземпляр класса Human
```

Как свидетельствует объявление класса, у объекта Tom есть атрибуты, такие как DateOfBirth, к которым можно обратиться, используя *точечный оператор* (dot operator) (.):

```
Tom.DateOfBirth = "1970";
```

Дело в том, что атрибут DateOfBirth принадлежит классу Human и является его элементом, как можно заметить в объявлении класса. Этот атрибут существует в действительности, т.е. в исполняющей среде, только когда создан объект. Точечный оператор (.) позволяет обратиться к атрибутам объекта, а также к таким методам, как IntroduceSelf():

```
Tom.IntroduceSelf();
```

Если у вас есть указатель pTom на экземпляр класса Human, то для доступа к его членам можно использовать оператор указателя (->), как описано в следующем разделе, или использовать оператор косвенного доступа (*) для ссылки на объект с последующим точечным оператором.

```
Human* pTom = new Human();
(*pTom).IntroduceSelf();
```

Доступ 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* pTom = new Human();
pTom->DateOfBirth = "1970";
pTom->IntroduceSelf();
delete pTom;

// Альтернативно при наличии указателя:
Human Tom;
Human* pTom = &Tom;  // Присвоить адрес, используя оператор ссылки &
pTom->DateOfBirth = "1970"; // эквивалентно Tom.DateOfBirth = "1970";
pTom->IntroduceSelf();      // эквивалентно Tom.IntroduceSelf();
```

Готовая для компиляции форма класса Human с ключевыми словами private и 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: private:
 7:     string Name;
 8:     int Age;
 9:
10: public:
```

```
11:     void SetName (string HumansName)
12:     {
13:         Name = HumansName;
14:     }
15:
16:     void SetAge(int HumansAge)
17:     {
18:         Age = HumansAge;
19:     }
20:
21:     void IntroduceSelf()
22:     {
23:         cout << "I am " + Name << " and am ";
24:         cout << Age << " years old" << endl;
25:     }
26: };
27:
28: int main()
29: {
30:     // Создание объекта класса Human со значением "Adam"
        // атрибута Name
31:     Human FirstMan;
32:     FirstMan.SetName("Adam");
33:     FirstMan.SetAge(30);
34:
35:     // Создание объекта класса Human со значением "Eve"
        // атрибута Name
36:     Human FirstWoman;
37:     FirstWoman.SetName("Eve");
38:     FirstWoman.SetAge (28);
39:
40:     FirstMan.IntroduceSelf();
41:     FirstWoman.IntroduceSelf();
42: }
```

Результат

```
I am Adam and am 30 years old
I am Eve and am 28 years old
```

Анализ

Строки 4–26 демонстрируют создание очень простого класса C++. Рассмотрим его с практической точки зрения, игнорируя термины и концепции, которые вы не понимаете с первого взгляда, поскольку подробно они обсуждаются далее на этом занятии. Сосредоточьтесь на структуре класса `Human` и его использовании в функции `main()`.

Этот класс содержит две закрытые (`private`) переменные: одна типа `string` по имени `Name` в строке 7, другая типа `int` по имени `Age` в строке 8, а также несколько открытых (`public`) функций (называемых *методами* (method)): `SetName()`, `SetAge()` и `IntroduceSelf()` в строках 11, 16 и 21, которые используют закрытые переменные. Строки 31 и 36 в функции `main()` создают два объекта класса `Human` соответственно. Следующие строки устанавливают значения переменных-членов объектов `FirstMan` и `FirstWoman`,

используя методы `SetName()` и `SetAge()`, которые называют *методами доступа* (accessor method). Обратите внимание, как в строках 40 и 41 вызов метода `IntroduceSelf()` этих двух объектов позволил создать две разные строки в выводе, использующих *переменные-члены* (member variable), значения которых были установлены в предыдущих строках.

Вы обратили внимание на ключевые слова `private` и `public` в листинге 9.1? Пришло время изучить эти средства, предоставляемые языком C++, для защиты атрибутов, которые ваш класс должен оставить скрытыми от тех, кто их использует.

Ключевые слова `public` и `private`

У каждого из нас есть много информации, часть которой доступна для окружающих людей, например наши имена. Эта информация может быть названа открытой. Но есть информация, которую вы не захотите предоставлять всему миру или даже соседу, например ваш доход. Эта информация является закрытой и часто хранится в тайне.

Язык C++ позволяет объявлять атрибуты и методы класса открытыми (допускающими доступ к ним извне объекта класса) или закрытыми (допускающими доступ только членам этого класса (или его "друзьям")). Ключевые слова C++ `public` (открытый) и `private` (закрытый) позволяют разработчику класса решать, что в классе может быть доступно извне его, например, из функции `main()`, а что 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 InputAge)
    {
        Age = InputAge;
    }
    // ...Другие члены и объявления
};
```

Предположим, экземпляр класса `Human` называется `Eve`:

```
Human Eve;
```

Когда пользователь этого экземпляра попытается получить доступ к возрасту Евы следующим образом:

```
cout << Eve.Age; // ошибка компиляции
```

он получит при компиляции ошибку с 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(); // OK
```

Если разработчик класса Human пожелает, он может реализовать метод GetAge() так, чтобы Ева представлялась моложе, чем есть! 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); // OK
```

У этого подхода много преимуществ. Текущая реализация метода SetAge() не делает ничего, кроме собственно присвоения 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 InputAge)
    {
        if (InputAge > 0)
            Age = InputAge;
    }
};
```

Таким образом, язык C++ позволяет разработчику класса контролировать возможность обращения к атрибутам данных класса и манипулировать ими.

Абстракция данных при помощи ключевого слова private

Позволяя разработать класс как контейнер, инкапсулирующий данные, и работающие с ними методы, язык C++ разрешает вам при помощи ключевого слова private решить, какая информация недоступна внешнему миру (т.е. недоступна вне класса). В то же время при помощи методов, объявленных как public, у вас есть возможность предоставить контролируемый доступ даже к информации, объявленной закрытой. Таким образом, реализация вашего класса позволяет абстрагировать то, что, по вашему мнению, внешний мир (другие классы и функции, такие как main()) не должен знать.

Возвращаясь к примеру с переменной-членом Human::Age, являющейся закрытой, вы знаете, что в действительности многим людям не нравится разглашать свой истинный возраст. Если бы классу Human понадобилось сообщить возраст на два года моложе, чем на самом деле, то это легко позволила бы сделать открытая функция GetAge(), использующая параметр Human::Age, но уменьшающая его на два и возвращающая результат, как представлено в листинге 9.2.

ЛИСТИНГ 9.2. Модель класса Human, где истинный возраст абстрагируется от пользователя и сообщается более молодой 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 << "Age of FirstMan " << FirstMan.GetAge() << endl;
34:     cout << "Age of FirstWoman " << FirstWoman.GetAge() << endl;
35:
36:     return 0;
37: }
```

Результат

```
Age of FirstMan 33
Age of FirstWoman 22
```

Анализ

Обратите внимание на открытый метод `Human::GetAge()` в строке 16. Поскольку фактический возраст, содержавшийся в закрытой целочисленной переменной-члене `Human::Age`, недоступен непосредственно, единственным способом для внешнего пользователя объекта класса `Human` обратиться к его атрибуту `Age` является метод `GetAge()`. Таким образом, фактический возраст, содержащийся в переменной-члене `Human::Age`, абстрагируется от внешнего мира.

Абстракция (abstraction) — это очень важная концепция 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 при создании 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 так:

```
class Human
{
public:
    Human(); // объявление конструктора
};
```

Конструктор может быть реализован в классе или вне объявления класса. Реализация (определение) в классе выглядит следующим образом:

```
class Human
{
public:
    Human()
    {
        // код конструктора здесь
    }
};
```

Вариант определения конструктора вне объявления класса выглядит следующим образом:

```
class Human
{
public:
    Human(); // объявление конструктора
};
// определение конструктора (реализация)
Human::Human()
{
    // код конструктора здесь
}
```

ПРИМЕЧАНИЕ

Оператор :: называется *оператором области видимости* (scope resolution operator). Например, синтаксис `Human::DateOfBirth` относится к переменной `DateOfBirth`, объявленной в пределах класса `Human`. Синтаксис `::DateOfBirth`, напротив, относится к другой переменной `DateOfBirth`, объявленной в глобальной области видимости.

Когда и как использовать конструкторы

Конструктор всегда вызывается при создании объекта. 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 (попробуйте, что получится, закомментировав строки 28 и 31). Для реализации улучшенной версии класса `Human`, где переменная `Age` инициализируется, в листинге 9.3 используется конструктор.

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:     // Закрытые данные-члены:
 8:     string Name;
 9:     int Age;
10:
11: public:
12:     // конструктор
13:     Human()
14:     {
15:         Age = 0; // инициализация гарантирует отсутствие
                     // случайного значения
16:         cout << "Constructed an instance of class Human" << endl;
17:     }
18:
19:     void SetName (string HumansName)
20:     {
21:         Name = HumansName;
22:     }
23:
24:     void SetAge(int HumansAge)
25:     {
26:         Age = HumansAge;
27:     }
28:
29:     void IntroduceSelf()
```

```
10:     {
11:         cout << "I am " + Name << " and am ";
12:         cout << Age << " years old" << endl;
13:     }
14: };
15:
16: int main()
17: {
18:     Human FirstMan;
19:     FirstMan.SetName("Adam");
20:     FirstMan.SetAge(30);
21:
22:     Human FirstWoman;
23:     FirstWoman.SetName("Eve");
24:     FirstWoman.SetAge (28);
25:
26:     FirstMan.IntroduceSelf();
27:     FirstWoman.IntroduceSelf();
28: }
```

Результат

```
Constructed an instance of class Human
Constructed an instance of class Human
I am Adam and am 30 years old
I am Eve and am 28 years old
```

Анализ

В выводе вы видите добавление двух весьма уместных первых строк по сравнению с листингом 9.1. Теперь обратите внимание на функцию `main()`, определенную в строках 36–48. Вы видите, что эти две строки были созданы косвенно при создании двух объектов `FirstMan` и `FirstWoman` в строках 38 и 42. Поскольку эти два объекта имеют тип класса `Human`, их создание автоматически привело к выполнению конструктора класса `Human`, определенного в строках 13–17. У этого конструктора есть оператор `cout`, создавший этот вывод. Кроме того, конструктор также 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, означающее неудачу установки атрибута класса `Human::Age`.

ПРИМЕЧАНИЕ

Конструктор, который может быть вызван без аргумента, называется *стандартным конструктором* (default constructor). Собственноручно создавать стандартный конструктор необязательно.

Если вы не создали конструктор сами, как в листинге 9.1, то компилятор предоставит его автоматически (он создает переменные-члены класса, но не инициализирует простые старые типы данных, такие как `int`).

Перегрузка конструкторов

Поскольку конструкторы могут быть перегружены как функции, вполне можно получить конструктор, позволяющий создать объект класса Human, с именем в качестве параметра, например:

```
class Human
{
public:
    Human()
    {
        // здесь код стандартного конструктора
    }

    Human(string HumansName)
    {
        // здесь 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:     // Закрытые данные-члены:
 8:     string Name;
 9:     int Age;
10:
11: public:
12:     // конструктор
13:     Human()
14:     {
15:         Age = 0; // инициализация гарантирует отсутствие
                     // случайного значения
16:         cout << "Default constructor creates an instance of Human"
                 << endl;
17:     }
18:
19:     // перегруженный конструктор, получающий Name
20:     Human(string HumansName)
21:     {
22:         Name = HumansName;
23:         Age = 0; // инициализация гарантирует отсутствие
                     // случайного значения
24:         cout << "Overloaded constructor creates " << Name << endl;
25:     }
```

```
26:
27:     // перегруженный конструктор, получающий Name и Age
28:     Human(string HumansName, int HumansAge)
29:     {
30:         Name = HumansName;
31:         Age = HumansAge;
32:         cout << "Overloaded constructor creates ";
33:         cout << Name << " of " << Age << " years" << endl;
34:     }
35:
36:     void SetName (string HumansName)
37:     {
38:         Name = HumansName;
39:     }
40:
41:     void SetAge(int HumansAge)
42:     {
43:         Age = HumansAge;
44:     }
45:
46:     void IntroduceSelf()
47:     {
48:         cout << "I am " + Name << " and am ";
49:         cout << Age << " years old" << endl;
50:     }
51: };
52:
53: int main()
54: {
55:     Human FirstMan; // использование стандартного конструктора
56:     FirstMan.SetName("Adam");
57:     FirstMan.SetAge(30);
58:
59:     Human FirstWoman ("Eve"); // использование перегруженного
                                  // конструктора
60:     FirstWoman.SetAge (28);
61:
62:     Human FirstChild ("Rose", 1);
63:
64:     FirstMan.IntroduceSelf();
65:     FirstWoman.IntroduceSelf();
66:     FirstChild.IntroduceSelf();
67: }
```

Результат

```
Default constructor creates an instance of Human
Overloaded constructor creates Eve
Overloaded constructor creates Rose of 1 years
I am Adam and am 30 years old
I am Eve and am 28 years old
I am Rose and am 1 years old
```

Анализ

Адам создается с использованием стандартного конструктора; Ева — с использованием перегруженного конструктора, который в виде параметра принимает строку, присваиваемую переменной-члену `Human::Name`, тогда как Роза создается с использованием третьего перегруженного конструктора, получающего в качестве параметров строку и число, присваиваемое переменной-члену `Human::Age`. Эта программа демонстрирует, что перегрузка конструкторов может оказаться весьма полезной, помогая инициализировать переменные.

СОВЕТ

> Вы можете решить не реализовать стандартный конструктор, чтобы заставить создавать экземпляры объектов с определенным минимальным набором параметров.

Класс без стандартного конструктора

В листинге 9.5 представлен класс `Human` без стандартного конструктора, что заставляет создавать его объекты, предоставляя имя и возраст.

ЛИСТИНГ 9.5. Класс с перегруженным конструктором, но без стандартного конструктора

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:     // Закрытые данные-члены:
 8:     string Name;
 9:     int Age;
10:
11: public:
12:
13:     // перегруженный конструктор (без стандартного конструктора)
14:     Human(string HumansName, int HumansAge)
15:     {
16:         Name = HumansName;
17:         Age = HumansAge;
18:         cout << "Overloaded constructor creates " << Name;
19:         cout << " of age " << Age << endl;
20:     }
21:
22:     void IntroduceSelf()
23:     {
24:         cout << "I am " + Name << " and am ";
25:         cout << Age << " years old" << endl;
26:     }
27: };
28:
29: int main()
```

```
11: {
12:     // Закомментирована следующая строка, пытающаяся создать объект
        // с использованием стандартного конструктора
13:     // Human FirstMan;
14:
15:     Human FirstMan("Adam", 30);
16:     Human FirstWoman("Eve", 28);
17:
18:     FirstMan.IntroduceSelf();
19:     FirstWoman.IntroduceSelf();
20: }
```

Результат

```
Overloaded constructor creates Adam of age 30
Overloaded constructor creates Eve of age 28
I am Adam and am 30 years old
I am Eve and am 28 years old
```

Анализ

Строка 32 в функции `main()` заслуживает внимания. Она очень похожа на код, создающий объект `FirstMan`, в листинге 9.3, но если снять с нее комментарий, компиляция потерпит неудачу с сообщением `error: 'Human' : no appropriate default constructor available` (ошибка: `'Human'` : нет подходящего стандартного конструктора). Дело в том, что у этой версии класса `Human` есть только один конструктор, получающий входные параметры типа `string` и `int`, как можно заметить в строке 14. Никакого стандартного конструктора нет, а при наличии перегруженного конструктора компилятор C++ не предоставляет стандартный конструктор автоматически.

Таким образом, этот пример демонстрирует возможность разработки классов, объекты которых можно создать, только предоставив определенные параметры, в данном случае `Name` и `Age`. Пример в листинге 9.5 демонстрирует также возможность присвоения значения переменной члену `Name` при создании объекта класса `Human` и невозможность изменить его впоследствии. Дело в том, что класс `Human` хранит атрибут имени в закрытой переменной типа `string` по имени `Name`, к которой нельзя обратиться или модифицировать из функции `main()` или любого другого места, кроме как из класса `Human`. Другими словами, перегруженный конструктор вынуждает пользователя класса `Human` определить имя (и возраст) для каждого создаваемого объекта и не позволяет изменять это имя, — весьма неплохая модель реального положения вещей, не так ли? Имя вам родители дали при рождении; людям разрешено знать его, но ни у кого (кроме вас) нет права изменить его.

Параметры конструктора со значениями по умолчанию

Как и функции, конструкторы способны иметь параметры со значениями, определенными по умолчанию. В следующем коде приведена немного модифицированная версия конструктора из строки 14 листинга 9.5, но где у параметра `Age` есть значение по умолчанию 25:

```
class Human
{
private:
    // Закрытые данные-члены:
    string Name;
    int Age;

public:
    // перегруженный конструктор (без стандартного конструктора)
    Human(string HumansName, int HumansAge = 25)
    {
        Name = HumansName;
        Age = HumansAge;
        cout << "Overloaded constructor creates " << Name;
        cout << " of age " << Age << endl;
    }
    // ... другие члены
};
```

Объект такого класса может быть создан следующим образом:

```
Human Adam("Adam"); // Adam.Age присваивается значение по умолчанию 25
Human Eve("Eve, 18); // Eve.Age присваивается указанное значение 18
```

ПРИМЕЧАНИЕ

Стандартный конструктор — это тот, который позволяет создавать экземпляры без аргументов, причем не обязательно тот, который не получает параметров. Ниже приведен конструктор с двумя параметрами, но оба они со значениями по умолчанию, поэтому он является стандартным конструктором.

```
class Human
{
private:
    // Закрытые данные-члены:
    string Name;
    int Age;

public:
    // Обратите внимание на значения по умолчанию для двух
входных параметров
    Human(string HumansName = "Adam", int HumansAge = 25)
    {
        Name = HumansName;
        Age = HumansAge;
        cout << "Overloaded constructor creates " << Name;
        cout << " of age " << 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. Другой способ инициализации членов — использование *списков инициализации* (initialization list). Ниже показан вариант конструктора из листинга 9.5, получающего два параметра, но выглядящего со списком инициализации следующим образом:

```
class Human
{
private:
    string Name;
    int Age;

public:
    // конструктор получает два параметра для инициализации
    // членов Age и Name
    Human(string InputName, int InputAge)
        :Name(InputName), Age(InputAge)
    {
        cout << "Constructed a Human called " << Name;
        cout << ", " << Age << " years old" << endl;
    }
    // ... другие члены класса
};
```

Таким образом, список инициализации характеризуется двоеточием (:) с последующим объявлением параметров, содержащихся в круглых скобках (...), индивидуальными переменными-членами и значениями для инициализации. Это инициализирующее значение может быть параметром, таким как InputName, или даже фиксированным значением. Списки инициализации могут также пригодиться при вызове конструкторов базового класса с определенными аргументами, как обсуждается на занятии 10, "Реализация наследования".

В листинге 9.6 представлен класс Human, оснащенный стандартным конструктором с параметрами, заданными по умолчанию, и списком инициализации.

ЛИСТИНГ 9.6. Стандартный конструктор, способный получать параметры, но со значениями по умолчанию и списком инициализации для установки значений членов

```
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: class Human
6: {
7: private:
8:     int Age;
9:     string Name;
10:
11: public:
12:     Human(string InputName = "Adam", int InputAge = 25)
13:         :Name(InputName), Age(InputAge)
14:     {
15:         cout << "Constructed a Human called " << Name;
16:         cout << ", " << Age << " years old" << endl;
```

```
16:     }
17: };
18:
19: int main()
20: {
21:     Human FirstMan;
22:     Human FirstWoman("Eve", 18);
23:
24:     return 0;
25: }
```

Результат

```
Constructed a Human called Adam, 25 years old
Constructed a Human called Eve, 18 years old
```

Анализ

Конструктор со списком инициализации расположен в строках 11–16, где можно также заметить, что параметрам были присвоены значения по умолчанию "Adam" для переменной-члена Name и 25 — для переменной-члена Age. Следовательно, когда в строке 21 создается экземпляр класса Human по имени FirstMan, его членам автоматически присваиваются значения по умолчанию. Экземпляр FirstWoman, напротив, был явно снабжен значениями для переменных Name и Age, как показано в строке 22.

Деструктор

Деструкторы (destructor), как и конструкторы, 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()
    {
        // здесь код деструктора
    }
};
```

Определение деструктора вне объявления класса выглядит следующим образом:

```
class Human
{
public:
    ~Human();// объявление деструктора
};

// определение деструктора (реализация)
Human::~Human()
{
    // здесь код деструктора
}
```

Как можно заметить, 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, а также освобождения зарезервированной динамической памяти и других ресурсов.

В этой книге регулярно рекомендуется применять строки класса `std::string`, а не символьные буфера 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 утилит (в дополнение к операторам, которые вы изучаете на занятии 12, "Типы операторов и их перегрузка"). Проанализируем пример класса `MyString`, представленный в листинге 9.7, который резервирует память для строки в конструкторе и освобождает ее в деструкторе.

ЛИСТИНГ 9.7. Пример класса, инкапсулирующего буфер в стиле C для гарантии его освобождения при помощи деструктора

```
1: #include <iostream>
1: using namespace std;
2:
3: class MyString
4: {
5: private:
6:     char* Buffer;
7:
8: public:
9:     // Конструктор
10:    MyString(const char* InitialInput)
11:    {
12:        if(InitialInput != NULL)
13:        {
14:            Buffer = new char [strlen(InitialInput) + 1];
15:            strcpy(Buffer, InitialInput);
16:        }
17:        else
18:            Buffer = NULL;
```

```
19:     }
20:     // Деструктор: освобождает буфер, зарезервированный в
        // конструкторе
21:     ~MyString()
22:     {
23:         cout << "Invoking destructor, clearing up" << endl;
24:         if (Buffer != NULL)
25:             delete [] Buffer;
26:     }
27:
28:     int GetLength()
29:     {
30:         return strlen(Buffer);
31:     }
32:
33:     const char* GetString()
34:     {
35:         return Buffer;
36:     }
37: }; // конец класса MyString
38:
39: int main()
40: {
41:     MyString SayHello("Hello from String Class");
42:     cout << "String buffer in MyString is " << SayHello.GetLength();
43:     cout << " characters long" << endl;
44:
45:     cout << "Buffer contains: ";
46:     cout << "Buffer contains: " << SayHello.GetString() << endl;
47: }
```

Результат

```
String buffer in MyString is 23 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
```

Анализ

Этот класс в основном инкапсулирует строку в стиле C в переменной `MyString::Buffer` и освобождает вас от задач резервирования и освобождения памяти, осуществляемых одинаково при каждой необходимости использовать строку. Особенно интересен код конструктора `MyString()` в строках 10–19 и деструктора `~MyString()` в строках 21–26. Конструктор требует передачи исходной строки, поскольку у него есть обязательный входной параметр, а затем копирует ее в строку в стиле C `Buffer` после резервирования памяти для нее в строке 14, где используется оператор `strlen` для определения длины исходной строки. Для копирования в эту недавно зарезервированную область памяти в строке 15 используется оператор `strcpy`. На случай, если пользователь класса в качестве аргумента для параметра `InitialInput` предоставит значение NULL, переменная `MyString::Buffer` также инициализируется со значением NULL (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, зарезервированной в конструкторе, и возвращение ее операционной

системе. Он проверяет, не содержит ли переменная `MyString::Buffer` значение NULL, и если это не так, то выполняет для нее оператор `delete[]`, дополняя оператор `new` в конструкторе. Обратите внимание, что нигде в функции `main()` не применяются операторы `new` или `delete`. Этот класс не только абстрагировал реализацию, но и гарантировал при этом техническую правильность освобождения зарезервированной памяти. Деструктор `~MyString()` вызывается автоматически при возвращении к функции `main()`, и это демонстрирует вывод, поскольку в деструкторе выполняется оператор `cout`.

Классы, облегчающие работу со строками, являются одним из многих примеров использования деструктора. На занятии 26, "Понятие интеллектуальных указателей", вы узнаете, что деструкторы играют критически важную роль в работе с указателями более сложным способом.

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 зарезервированной динамической памяти.

Конструктор копий

На занятии 7, "Организация кода при помощи функций", вы узнали, что переданные функции `Area()` аргументы копируются (см. листинг 7.1):

```
double Area(double InputRadius);
```

Так, аргумент, переданный в виде параметра `InputRadius`, копируется, когда вызывается функция `Area()`. Это правило относится также к объектам (экземплярам классов).

Поверхностное копирование и связанные с ним проблемы

Такие классы, как `MyString`, например, представленный в листинге 9.7, содержат в качестве члена указатель, который указывает на область в динамически распределяемой памяти, зарезервированную в конструкторе при помощи оператора `new` и освобождаемую в деструкторе с использованием оператора `delete[]`. Когда объект этого класса копируется, копируется и указатель, являющийся его членом, но буфер, на который он указывает, не копируется. В результате два объекта указывают на тот же буфер, находящийся в динамически распределяемой памяти. Это называется *поверхностным копированием* (shallow copy) и представляет угрозу стабильности программы, как показано в листинге 9.8.

ЛИСТИНГ 9.8. Проблема передачи объекта класса, такого, как `MyString`, по значению

```
1: #include <iostream>
2: using namespace std;
3:
4: class MyString
5: {
6: private:
7:     char* Buffer;
8:
9: public:
```

```
 9:     // Конструктор
10:     MyString(const char* InitialInput)
11:     {
12:         if(InitialInput != NULL)
13:         {
14:             Buffer = new char [strlen(InitialInput) + 1];
15:             strcpy(Buffer, InitialInput);
16:         }
17:         else
18:             Buffer = NULL;
19:     }
20:
21:     // Деструктор
22:     ~MyString()
23:     {
24:         cout << "Invoking destructor, clearing up" << endl;
25:         if (Buffer != NULL)
26:             delete [] Buffer;
27:     }
28:
29:     int GetLength()
30:     {
31:         return strlen(Buffer);
32:     }
33:
34:     const char* GetString()
35:     {
36:         return Buffer;
37:     }
38: };
39:
40: void UseMyString(MyString Input)
41: {
42:     cout << "String buffer in MyString is " << Input.GetLength();
43:     cout << " characters long" << endl;
44:
45:     cout << "Buffer contains: " << Input.GetString() << endl;
46:     return;
47: }
48:
49: int main()
50: {
51:     MyString SayHello("Hello from String Class");
52:
53:     // Передать SayHello функции как параметр
54:     UseMyString(SayHello);
55:
56:     return 0;
57: }
```

Результат

```
String buffer in MyString is 23 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
Invoking destructor, clearing up
<отказ, как можно заметить на рис. 9.2>
```

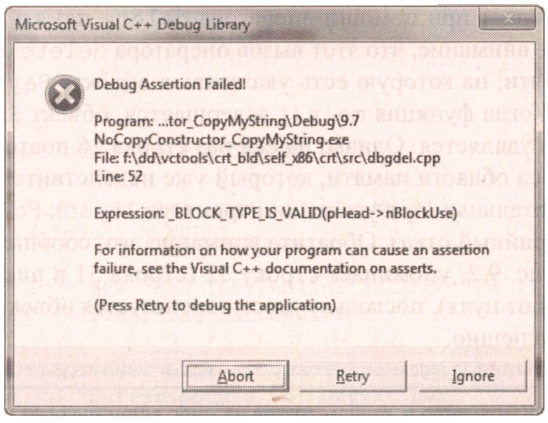


РИС. 9.2. Снимок экрана аварийного отказа, произошедшего при выполнении кода листинга 9.8 (режим отладки среды разработки Visual Studio)

Анализ

Почему класс, 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 задача использования объекта `SayHello` класса `MyString`, созданного в функции `main()`, была делегирована функции `UseMyString()`, вызываемой в строке 54. Делегирование работы этой функции привело к тому, что объект `SayHello` в функции `main()` копируется в аргумент параметра `Input`, используемого в функции `UseMyString()`. Эта копия создается компилятором, поскольку функция была объявлена как получающая параметр `Input` по значению, а не по ссылке. Компилятор создает двоичную копию простых старых данных, таких, как целые числа, символы и указатели. Таким образом, значение, содержащееся в указателе `SayHello.Buffer`, было просто скопировано в параметр `Input`, т.е. он теперь указывает на ту же область памяти, что и `Input.Buffer` (рис. 9.3).

РИС. 9.3. Поверхностное копирование объекта `SayHello` в параметр `Input` при вызове функции `UseMyString()`

Двоичная копия не обеспечивает *глубокого копирования* (deep copy) и не распространяется на указываемую область памяти, поэтому теперь есть два объекта класса `MyString`, указывающих на ту же область в памяти. Таким образом, по завершении работы функции `UseMyString()` переменная `Input` выходит из области видимости и удаляется. При этом вызывается деструктор класса `MyString`, и его код в строке 26

листинга 9.8 освобождает при помощи оператора delete память, зарезервированную для буфера. Обратите внимание, что этот вызов оператора delete объявляет 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 сей раз строка 26 повторно вызывает оператор delete для адреса области памяти, который уже недействителен (уже освобожден и объявлен недействительным при удалении параметра Input). Результатом повторного удаления и будет аварийный отказ. Обратите внимание, что сообщение режима отладки. представленное на рис. 9.2, упоминает строку 52 (строка 51 в листинге, ведь строки в книге отсчитываются от нуля), поскольку здесь используется объект SayHello, который не был освобожден успешно.

ПРИМЕЧАНИЕ

Компилятор в данном случае не смог автоматически обеспечить глубокое копирование, поскольку на момент компиляции ему неизвестно ни количество байтов, на которые указывает указатель-член MyString::Buffer, ни характер резервирования.

Обеспечение глубокого копирования с использованием конструктора копий

Конструктор копий (copy constructor) — это специальный перегруженный конструктор, который должен предоставить разработчик класса. Компилятор использует конструктор копий каждый раз, когда объект класса копируется, включая передачу объекта в функцию по значению.

Конструктор копий для класса MyString можно объявить так:

```
class MyString
{
    MyString(const MyString& CopySource); // конструктор копий
};

MyString::MyString(const MyString& CopySource)
{
    // Код реализации конструктора копий
}
```

Таким образом, конструктор копий получает как параметр по ссылке объект того же класса. Этот параметр — псевдоним исходного объекта, используемый при написании собственного специального кода копирования (где вы гарантировали бы глубокое копирование всех буферов оригинала), как показано в листинге 9.9.

ЛИСТИНГ 9.9. Определение конструктора копий, гарантирующего глубокое копирование буферов в динамически распределяемой памяти

```
0: #include <iostream>
1: using namespace std;
2:
3: class MyString
4: {
```

```
private:
    char* Buffer;

public:
    // конструктор
    MyString(const char* InitialInput)
    {
        cout << "Constructor: creating new MyString" << endl;
        if(InitialInput != NULL)
        {
            Buffer = new char [strlen(InitialInput) + 1];
            strcpy(Buffer, InitialInput);

            // Отображение адреса области памяти локального буфера
            cout << "Buffer points to: 0x" << hex;
            cout << (unsigned int*)Buffer << endl;
        }
        else
            Buffer = NULL;
    }

    // Конструктор копий
    MyString(const MyString& CopySource)
    {
        cout << "Copy constructor: copying from MyString" << endl;

        if(CopySource.Buffer != NULL)
        {
            // гарантировать глубокое копирование, создав сначала
            // собственный буфер
            Buffer = new char [strlen(CopySource.Buffer) + 1];

            // копирование из оригинала в локальный буфер
            strcpy(Buffer, CopySource.Buffer);

            // Отображение адреса области памяти локального буфера
            cout << "Buffer points to: 0x" << hex;
            cout << (unsigned int*)Buffer << endl;
        }
        else
            Buffer = NULL;
    }

    // Деструктор
    ~MyString()
    {
        cout << "Invoking destructor, clearing up" << endl;
        if (Buffer != NULL)
            delete [] Buffer;
    }

    int GetLength()
    {
        return strlen(Buffer);
```

```
58:     }
59:
60:     const char* GetString()
61:     {
62:         return Buffer;
63:     }
64: };
65:
66: void UseMyString(MyString Input)
67: {
68:     cout << "String buffer in MyString is " << Input.GetLength();
69:     cout << " characters long" << endl;
70:
71:     cout << "Buffer contains: " << Input.GetString() << endl;
72:     return;
73: }
74:
75: int main()
76: {
77:     MyString SayHello("Hello from String Class");
78:
79:     // Передача SayHello по значению (с копированием)
80:     UseMyString(SayHello);
81:
82:     return 0;
83: }
```

Результат

```
Constructor: creating new MyString
Buffer points to: 0x0040DA68
Copy constructor: copying from MyString
Buffer points to: 0x0040DAF8
String buffer in MyString is 17 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
Invoking destructor, clearing up
```

Анализ

Большая часть кода подобна коду листинга 9.8, кроме нескольких строк `cout`, добавленных в конструктор, и нового конструктора копий, расположенного в строках 27–45. Для начала сосредоточимся на функции `main()`, которая (как и прежде) создает объект `SayHello` в строке 77. Создание объекта `SayHello` приводит к отображению первой строки вывода, оператор `cout` которой расположен в строке 12 конструктора `MyString`. Для удобства конструктор отображает также адрес области памяти, на которую указывает `Buffer`. Затем, в строке 80, функция `main()` передает объект `SayHello` по значению функции `UseMyString()`, что автоматически приводит к вызову конструктора копий, как свидетельствует вывод. Код в конструкторе копий очень похож на таковой в конструкторе. Основная идея та же — выяснить длину строки в стиле C, которая содержится в буфере оригинала (строка 34), зарезервировать достаточно памяти в собственном экземпляре

ВНИМАНИЕ!

Использование ключевого слова `const` в объявлении конструктора копий гарантирует, что он не изменит то, на что указывает исходный объект.

Кроме того, параметр должен передаваться в конструктор копий по ссылке. Если бы он не передавался по ссылке, то конструктор сам вызвал бы копирование значения, приведя таким образом к поверхностному копированию данных оригинала, а именно этого мы и намеревались избежать.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Всегда создавайте конструктор копий и оператор присвоения копии, когда ваш класс содержит *простой указатель* (raw pointer) (например, `char*` и т.п.)	**Не используйте** простой указатель как член класса, если в этом нет абсолютно неизбежной необходимости
Всегда создавайте конструктор копий с константным параметром ссылки на оригинал	
Используйте как члены такие классы строк, как `std::string`, и классы интеллектуальных указателей вместо простых указателей, поскольку они реализуют конструкторы копий и экономят ваше время	

ПРИМЕЧАНИЕ

Класс `MyString` с простым указателем `char* Buffer` в качестве члена используется как пример для объяснения необходимости конструктора копий.

Если вам нужно создать класс, который должен содержать строковые данные для хранения имен, например, то используйте класс `std::string`, а не `char*`, ведь при отсутствии простых указателей даже конструктор копий не нужен. Дело в том, что стандартный конструктор копий, вставленный компилятором, гарантирует вызов всех доступных конструкторов копий членов класса объектов, таких как `std::string`.

C++11

Конструктор перемещения улучшает производительность

Бывают случаи, когда ваши объекты подвергаются копированию автоматически, в связи с характером языка и его потребностями. Рассмотрим следующий код:

```
class MyString
{
    // реализация в листинге 9.9
};

MyString Copy(MyString& Source)
{
    MyString CopyForReturn(Source.GetString());    // создание копии
    return CopyForReturn; // возвращение по значению вызывает конструктор
```

```
                                    // копий
}

int main()
{
    MyString sayHello ("Hello World of C++");
    MyString sayHelloAgain(Copy(sayHello)); // вызов конструктора копий
                                            // дважды

    return 0;
}
```

Как свидетельствует комментарий, при создании экземпляра `sayHelloAgain` конструктор копий был вызван дважды, следовательно, и глубокое копирование было выполнено дважды из-за вызова функции `Copy(sayHello)`, которая возвращает объект класса `MyString` по значению. Однако возвращение этого значения требует времени и не является доступным вне этого выражения. Таким образом, вызов конструктора копий, добросовестно выполненный компилятором C++, фактически приводит к сокращению производительности, которое может стать существенным, если массив объектов в динамической памяти имеет большой размер.

Чтобы избежать снижения производительности, создавайте в дополнение к конструктору копий *конструктор перемещения* (move constructor). Вот синтаксис конструктора перемещения:

```
// конструктор перемещения
MyString(MyString&& MoveSource)
{
    if(MoveSource.Buffer != NULL)
    {
        Buffer = MoveSource.Buffer; // взять в собственность,
                                    // т.е. 'переместить'
        MoveSource.Buffer = NULL; // установить источник перемещения
                                  // в NULL, т.е. освободить его
    }
}
```

Когда он доступен, компилятор C++11 автоматически выбирает конструктор перемещения для временного "перемещения" ресурса, а следовательно, избегает этапа глубокого копирования. При реализованном конструкторе перемещения комментарий должен быть соответственно заменен следующим:

```
MyString sayHelloAgain(Copy(sayHello)); // 1 вызов конструктора копий,
                                        // 1 конструктора перемещения
```

Конструктор перемещения обычно реализуют с *оператором присваивания при перемещении* (move assignment operator), который обсуждается подробно на занятии 12, "Типы операторов и их перегрузка". Улучшенная версия класса `MyString`, реализующая конструктор перемещения и оператор присваивания при перемещении, приведена в листинге 12.11.

Различные способы использования конструкторов и деструкторов

На этом занятии вы изучили ряд очень важных, фундаментальных концепций, таких как конструктор, деструктор, а также абстракция данных и методов при помощи ключевых слов public и private. Эти концепции позволяют создать классы, а также контролировать создание, копирование, удаление и предоставление данных.

Рассмотрим несколько интересных схем, которые помогут вам решить некоторые проблемы в 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); // дубликат, созданный при передаче
                           // по значению
President clone;
clone = OurPresident;      // дублирование при присвоении
```

Понятно, что таких ситуаций следует избегать. Кроме некой конституции, вы могли бы моделировать операционную систему, и необходимо было бы обеспечить одну локальную сеть, один процессор и т.д. Необходимо избежать ситуаций, где ресурсы могут быть скопированы. Если вы не объявите конструктор копий, компилятор C++ сам вставит стандартный открытый конструктор копий. Это нарушит ваш проект и создаст угрозу его реализации. Но все же язык предоставляет решение этой проблемы.

Объявление закрытого конструктора копий позволяет гарантировать, что объект вашего класса не может быть скопирован. Так, вызов функции DoSomething(OurPresident) приведет к неудаче при компиляции. Чтобы избежать присвоения, следует объявить закрытым оператор присвоения.

Таким образом, решение следующее:

```
class President
{
private:
    President(const President&);            // закрытый конструктор копий
    President& operator= (const President&); // закрытый оператор
                                             // присвоения копии

    // ... другие атрибуты
};
```

Нет никакой необходимости в реализации закрытого конструктора копий и оператора присвоения. Для предотвращения копирования объектов класса President вполне достаточно лишь объявления их как закрытых.

Синглетонный класс, разрешающий создание только одного экземпляра

Обсуждавшийся ранее класс President хорош, но у него есть недостаток: ему не избежать создания нескольких президентов при создании нескольких экземпляров объектов:

```
President One, Two, Three;
```

Благодаря закрытым конструкторам копий индивидуальные объекты не копируемы, однако на самом деле класс President нуждается в одном и только одном объекте, а создание дополнительных запрещается. Такова концепция *синглетона* (singleton), которая подразумевает использование закрытого конструктора, закрытого оператора присвоения и статических членов экземпляра класса для реализации этого мощнейшего (хоть это и спорно) 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 совместно используется всеми экземплярами класса.

Ключевое слово static — основной компонент в создании *синглетонного класса* (singleton class), как показано в листинге 9.10.

ЛИСТИНГ 9.10. Синглетонный класс President, запрещающий копирование, присвоение и создание нескольких экземпляров

```
#include <iostream>
#include <string>
using namespace std;

class President
{
private:
    // закрытый стандартный конструктор (запрет создания извне)
    President() {};

    // закрытый конструктор копий (запрет создания копии)
    President(const President&);

    // закрытый оператор присвоения (запрет присвоения)
    const President& operator=(const President&);

    // данные-члены: имя президента
    string Name;

public:
    // контролируемое создание экземпляра
    static President& GetInstance()
```

```
22:     {
23:         // статические объекты создаются только однажды
24:         static President OnlyInstance;
25:
26:         return OnlyInstance;
27:     }
28:
29:     // открытые методы
30:     string GetName()
31:     {
32:         return Name;
33:     }
34:
35:     void SetName(string InputName)
36:     {
37:         Name = InputName;
38:     }
39: };
40:
41: int main()
42: {
43:     President& OnlyPresident = President::GetInstance();
44:     OnlyPresident.SetName("Abraham Lincoln");
45:
46:     // чтобы увидеть, как отказ при компиляции запрещает
       // дублирование, снимите комментарии со следующих строк
47:     // President Second;                  // конструктор недоступен
48:     // President* Third= new President(); // конструктор недоступен
49:     // President Fourth = OnlyPresident;  // конструктор копий
                                              // недоступен
50:     // OnlyPresident = President::GetInstance(); // оператор =
                                                     // недоступен
51:
52:     cout << "The name of the President is: ";
53:     cout << President::GetInstance().GetName() << endl;
54:
55:     return 0;
56: }
```

Результат

```
The name of the President is: Abraham Lincoln
```

Анализ

Обратите внимание на функцию `main()`: в ней всего несколько строк кода и ряд закомментированных строк, которые демонстрируют все комбинации создания новых экземпляров или копий класса `President`, которые не сработают. Давайте проанализируем их одна за другой:

```
47:     // President Second;                  // конструктор недоступен
48:     // President* Third= new President(); // конструктор недоступен
```

Строки 47 и 48 — это попытки создания объекта в стеке и распределяемой памяти соответственно, с использованием стандартного конструктора, который недоступен, поскольку в строке 8 он объявлен закрытым.

```
49:     // President Fourth = OnlyPresident;  // конструктор копий
                                              // недоступен
```

В строке 49 предпринимается попытка создания копии существующего объекта при помощи конструктора копий (присвоение во время создания вызывает конструктор копий), который недоступен в функции `main()`, поскольку в строке 11 она объявлена закрытой.

```
50:     // OnlyPresident = President::GetInstance(); // оператор =
                                                     // недоступен
```

Строка 50 является попыткой создания копии через присвоение, которое не сработает, так как оператор присвоения объявлен закрытым в строке 14. Таким образом, функция `main()` никак не может создать экземпляр класса `President`, а единственная оставшаяся возможность получить экземпляр класса `President` — это использовать статическую функцию `GetInstance()`, как в строке 43. 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 21–27, использует статическую переменную `OnlyInstance` для гарантии наличия одного и только одного экземпляра класса `President`. Дело в том, что код строки 24 выполняется только однажды (статическая инициализация), а следовательно, функция `GetInstance()` возвращает единственный доступный экземпляр класса `President`, независимо от того, как он используется (строки 43 и 53) в функции `main()`.

ВНИМАНИЕ!

Используйте *шаблон синглетон* (singleton pattern) только там, где это абсолютно необходимо, учитывайте будущий рост приложения и его возможностей. Обратите внимание на то, что ограничение создания нескольких экземпляров может стать узким местом архитектуры, когда впоследствии понадобится несколько экземпляров класса.

Например, если наш проект перерастет от моделирования одной нации к Организации Объединенных Наций, которая в настоящее время насчитывает 192 нации с 192 президентами, проблема архитектуры становится очевидной.

Класс, запрещающий создание экземпляра в стеке

Пространство в стеке зачастую ограничено. Если вы пишете базу данных, способную содержать гигабайт 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 распределяемой памяти. Для этого следует объявить закрытым деструктор:

```
class MonsterDB
{
private:
    // закрытый деструктор
    ~MonsterDB();
    // ... коллекция, резервирующая огромный объем памяти
};
```

Таким образом, при попытке использовать класс MonsterDB не получится создать его экземпляр:

```
int main()
{
    MonsterDB myDatabase; // ошибка компиляции
    // ... остальной код
    return 0;
}
```

Этот экземпляр был бы успешно создан в стеке. Но поскольку компилятор знает, что экземпляр класса myDatabase должен быть удален, когда он выйдет из области видимости, он автоматически пробует вызывать его деструктор в конце функции main(), который оказывается недоступен, так как объявлен закрытым, что приводит к отказу при компиляции.

Однако закрытый деструктор не помешает 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 его там. Такой класс, как MonsterDB, нуждается в открытой статической функции-члене, которая удаляет экземпляр (как член класса она имеет доступ к 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
 7:
 8: public:
 9:     static void DestroyInstance(MonsterDB* pInstance)
10:     {
11:         // статический член класса может обратиться к закрытому
            // деструктору
12:         delete pInstance;
13:     }
14:
15:     // ... несколько других методов
16: };
17:
18: int main()
19: {
20:     MonsterDB* pMyDatabase = new MonsterDB();
```

```
    // pMyDatabase -> member methods (....);

    // снимите комментирий со следующих строк, чтобы получить
    // ошибку при компиляции
    // delete pMyDatabase; // закрытый деструктор не может быть
                           // вызван

    // для освобождения используйте статический метод
    MonsterDB::DestroyInstance(pMyDatabase);

    return 0;
}
```

Этот фрагмент кода не имеет вывода.

Анализ

Цель кода — продемонстрировать, что класс, запрещающий создание своих экземпляров в стеке, нуждается в закрытом деструкторе, как показано в строке 6, и статической функции `DestroyInstance()`, как показано в строках 9–13, используемой в функции `main()` (строка 28).

Указатель `this`

Указатель `this` — это важнейшая концепция языка C++; зарезервированное ключевое слово `this` применимо в рамках класса, который содержит адрес объекта. Другими словами, значение указателя `this` — это `&object`. В пределах метода класса, когда вы вызываете другой метод, компилятор неявно передает ему в вызове указатель `this` как невидимый параметр:

```
class Human
{
private:
    // ... объявления закрытых членов
    void Talk (string Statement)
    {
        cout << Statement;
    }
public:
    void IntroduceSelf()
    {
        Talk("Bla bla");
    }
};
```

Здесь представлен метод `IntroduceSelf();` использующий закрытый член `Talk()` для вывода на экран выражения. В действительности компилятор внедряет указатель `this` в вызов метода `Talk()`, который выглядит как `Talk(this, "Bla bla")`.

С точки зрения программирования у указателя `this` не слишком много областей применения, но иногда он оказывается удобным. Например, у кода доступа к переменной `Age` в пределах функции `SetAge()`, представленной в листинге 9.1, может быть такой вариант:

```
void SetAge(int HumansAge)
{
    this->Age = HumansAge; // то же, что и Age = HumansAge
}
```

ПРИМЕЧАНИЕ

Указатель `this` не передается в статические методы класса. Как и статические функции, они не связаны с экземпляром класса. Статические методы совместно используются всеми экземплярами.

Если хотите использовать переменные экземпляра в статической функции, явно объявите параметр, используемый вызывающей стороной, для передачи указателя `this` как аргумента.

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, объявление констант", используется для определения объема памяти в байтах, занимаемого переменной определенного типа. Этот оператор применим и для классов, он в основном сообщает сумму байтов, занимаемых каждым атрибутом данных, содержавшимся в пределах объявления 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 не участвуют в определении размера класса. Рассмотрим листинг 9.12.

ЛИСТИНГ 9.12. Результат применения оператора `sizeof()` к классам и их экземплярам

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:     char* Buffer;
 7:
 8: public:
 9:     // Конструктор
10:     MyString(const char* InitialInput)
11:     {
12:         if(InitialInput != NULL)
13:         {
14:             Buffer = new char [strlen(InitialInput) + 1];
15:             strcpy(Buffer, InitialInput);
16:         }
17:         else
18:             Buffer = NULL;
19:     }
20:
21:     // Конструктор копий
```

```
12:     MyString(const MyString& CopySource)
13:     {
14:         if(CopySource.Buffer != NULL)
15:         {
16:             Buffer = new char [strlen(CopySource.Buffer) + 1];
17:             strcpy(Buffer, CopySource.Buffer);
18:         }
19:         else
20:             Buffer = NULL;
21:     }
22:
23:     ~MyString()
24:     {
25:         if (Buffer != NULL)
26:             delete [] Buffer;
27:     }
28:
29:     int GetLength()
30:     {
31:         return strlen(Buffer);
32:     }
33:
34:     const char* GetString()
35:     {
36:         return Buffer;
37:     }
38: };
39:
40: class Human
41: {
42: private:
43:     int Age;
44:     bool Gender;
45:     MyString Name;
46:
47: public:
48:     Human(const MyString& InputName, int InputAge, bool InputGender)
49:         : Name(InputName), Age (InputAge), Gender(InputGender) {}
50:
51:     int GetAge ()
52:     {
53:         return Age;
54:     }
55: };
56:
57: int main()
58: {
59:     MyString FirstMan("Adam");
60:     MyString FirstWoman("Eve");
61:
62:     cout << "sizeof(MyString) = " << sizeof(MyString) << endl;
63:     cout << "sizeof(FirstMan) = " << sizeof(FirstMan) << endl;
64:     cout << "sizeof(FirstWoman) = " << sizeof(FirstWoman) << endl;
65:
```

```
76:     Human FirstMaleHuman(FirstMan, 25, true);
77:     Human FirstFemaleHuman(FirstWoman, 18, false);
78:
79:     cout << "sizeof(Human) = " << sizeof(Human) << endl;
80:     cout << "sizeof(FirstMaleHuman) = " << sizeof(FirstMaleHuman)
            << endl;
81:     cout << "sizeof(FirstFemaleHuman) = " << sizeof(FirstFemaleHuman)
            << endl;
82:
83:     return 0;
84: }
```

Результат

```
sizeof(MyString) = 4
sizeof(FirstMan) = 4
sizeof(FirstWoman) = 4
sizeof(Human) = 12
sizeof(FirstMaleHuman) = 12
sizeof(FirstFemaleHuman) = 12
```

Анализ

Пример, конечно, длинноват, поскольку он содержит класс `MyString`, представленный ранее в листинге 9.9 (правда, без большинства операторов вывода текста), 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(FirstMan)`, поскольку количество байтов, использованных классом, по существу фиксируется во время компиляции и известно уже при разработке класса. 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 он указывает.

Попробуйте вычислить размер типа `Human` вручную, который составляет 12. Строки 53–55 свидетельствуют, что класс `Human` содержит атрибуты типа `int`, `bool` и `MyString`. Чтобы освежить в памяти размер в байтах используемых встроенных типов, обратитесь к листингу 3.4. Таким образом, тип `int` использует 4 байта, тип `bool` — 1 байт, тип `MyString` — 4 байта на системе автора, что никак не дает в итоге 12, который отображен в выводе. Дело в том, что на результат оператора `sizeof()` влияют 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 оно обрабатывается компилятором C++ практически так же, как ключевое слово `class`. Различия кроются в заданном по умолчанию модификаторе доступа (`public` или `private`),

когда разработчик ничего не указывает. 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 у класса — закрытыми. Наследование рассматривается на занятии 10, "Реализация наследования".

Вариант структуры класса Human из листинга 9.12 был бы следующим:

```
struct Human
{
    // конструктор, открытый по умолчанию (поскольку никакой
    // спецификатор 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 у 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 при помощи ключевого слова friend объявлены *дружественными* (friend), как можно заметить в листинге 9.13.

ЛИСТИНГ 9.13. Использование ключевого слова friend, позволяющее внешней функции DisplayAge() обращаться к закрытым переменным-членам

```
 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:     friend void DisplayAge(const Human& Person);
11:
12: public:
13:     Human(string InputName, int InputAge)
```

```
14:     {
15:         Name = InputName;
16:         Age = InputAge;
17:     }
18: };
19:
20: void DisplayAge(const Human& Person)
21: {
22:     cout << Person.Age << endl;
23: }
24:
25: int main()
26: {
27:     Human FirstMan("Adam", 25);
28:     cout << "Accessing private member Age via friend: ";
29:     DisplayAge(FirstMan);
30:
31:     return 0;
32: }
```

Результат

```
Accessing private member Age via friend: 25
```

Анализ

Строка 10 содержит объявление, указывающее компилятору, что функции `DisplayAge()` в глобальной области видимости разрешен специальный доступ к закрытым членам класса `Human`. Закомментировав строку 10, вы сразу получите ошибку компиляции в строке 22.

Как и функции, внешние классы также могут быть объявлены дружественными, как показано в листинге 9.14.

ЛИСТИНГ 9.14. Использование ключевого слова `friend`, позволяющее внешнему вспомогательному классу обращаться к закрытым переменным-членам

```
 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:     friend class Utility;
11:
12: public:
13:     Human(string InputName, int InputAge)
14:     {
15:         Name = InputName;
```

```
16:         Age = InputAge;
17:     }
18: };
19:
20: class Utility
21: {
22: public:
23:     static void DisplayAge(const Human& Person)
24:     {
25:         cout << Person.Age << endl;
26:     }
27: };
28:
29: int main()
30: {
31:     Human FirstMan("Adam", 25);
32:     cout << "Accessing private member Age via friend class: ";
33:     Utility::DisplayAge(FirstMan);
34:
35:     return 0;
36: }
```

Результат

```
Accessing private member Age via friend class: 25
```

Анализ

Строка 10 объявляет класс `Utility` дружественным классу `Human`. Это позволяет всем методам класса `Utility` обращаться даже к закрытым переменным-членам и методам класса `Human`.

Резюме

Это занятие познакомило вас с одной из самых фундаментальных концепций языка C++ — классом и ключевым словом `class`. Вы узнали, что класс инкапсулирует данные-члены и функции-члены для их использования. Было показано, что такие модификаторы доступа, как `public` и `private`, позволяют абстрагировать данные и функции, которые сущностям вне класса не нужно видеть.

Вы изучили концепцию конструкторов копий и то, как с помощью конструкторов перемещения компилятор C++11 позволяет оптимизировать случаи нежелательных этапов копирования. Мы также рассмотрели некоторые частные случаи, где все эти элементы объединяются, позволяя реализовать такие шаблоны проектирования, как "синглетон".

Вопросы и ответы

- **В чем разница между экземпляром класса и объектом того же класса?**
  По существу, никакой. Когда вы создаете экземпляр класса, вы получаете объект.

- **Как лучше получить доступ к члену: используя точечный оператор (.) 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 точечный оператор.

- **Должен ли я всегда создавать конструктор копий?**

  Если среди переменных-членов вашего класса есть интеллектуальные указатели, строковые классы или контейнеры STL, такие как `std::vector`, то стандартный конструктор копий, предоставляемый компилятором, гарантирует вызов их конструкторов копий. Однако, если среди членов вашего класса есть простой указатель (такой как `int*` для динамического массива вместо `std::vector<int>`), необходимо предоставить конструктор копий, 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 стандартный конструктор?**

  Да. Если экземпляр класса может быть создан без аргументов, то считается, что у класса есть стандартный конструктор. У класса может быть только один стандартный конструктор.

- **Почему некоторые примеры данного занятия используют такие функции, как `SetAge()` для установки значения таких переменных, как `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 проекта данные-члены имеет смысл оставить закрытыми. Функции доступа, такие как `GetAge()` или `SetAge()`, являются корректным и рекомендуемым способом доступа к этим закрытым данным, позволяя выполнять проверки на ошибки прежде, чем, например, значение переменной `Human::Age` будет установлено или удалено.

- **Почему конструктору копий в качестве параметра по ссылке передается оригинал?**

  Прежде всего, такого конструктора копий ожидает компилятор. Причина в том, что конструктор копий вызвал бы сам себя, если бы получил оригинал по значению, что привело бы к бесконечному циклу копирования.

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 Г, "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 при помощи оператора `new`, где он создается, в стеке или в динамической памяти?

2. У моего класса есть простой указатель `int*` на динамический массив целых чисел. Разные ли размеры сообщит оператор `sizeof` в зависимости от количества целых чисел в динамическом массиве?
3. Все члены моего класса являются закрытыми, и для него не заявлен 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, где все параметры инициализируются с использованием списка инициализации в конструкторе.
4. Напишите класс `Circle`, который вычисляет площадь и периметр по радиусу, который передается классу как параметр во время создания экземпляра. Число Пи должно содержаться в константном закрытом члене, к которому нельзя обратиться извне класса.

ЗАНЯТИЕ 10
Реализация наследования

Объектно-ориентированное программирование основано на четырех важных аспектах: *инкапсуляция* (encapsulation), *абстракция* (abstraction), *наследование* (inheritance) и *полиморфизм* (polymorphism). *Наследование* — это мощнейший способ многократного использования атрибутов и краеугольный камень полиморфизма.

На сегодняшнем занятии.

- Наследование в контексте программирования.
- Синтаксис наследования C++.
- Открытое, закрытое и защищенное наследование.
- Множественное наследование.
- Проблемы, вызванные сокрытием методов базового класса и отсечением.

Основы наследования

То, что Том Смит унаследовал от своих предков, — это, прежде всего, фамилию, что и делает его Смитом. Кроме того, он унаследовал некие знания, которые его же 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, где у используемых компонентов есть подобные атрибуты с незначительными отличиями в деталях или поведении. Один из способов решения этой проблемы — сделать все компоненты классами, каждый из которых реализует все атрибуты, реализуя повторно даже общие. Другое решение — использовать наследование, чтобы позволить подобным классам получить общие функциональные возможности из базового класса, который реализует их, и переопределить их, чтобы реализовать то поведение, которое делает каждый класс индивидуальным. Последнее решение зачастую предпочтительней. Добро пожаловать в мир наследования объектно-ориентированного программирования (рис. 10.1).

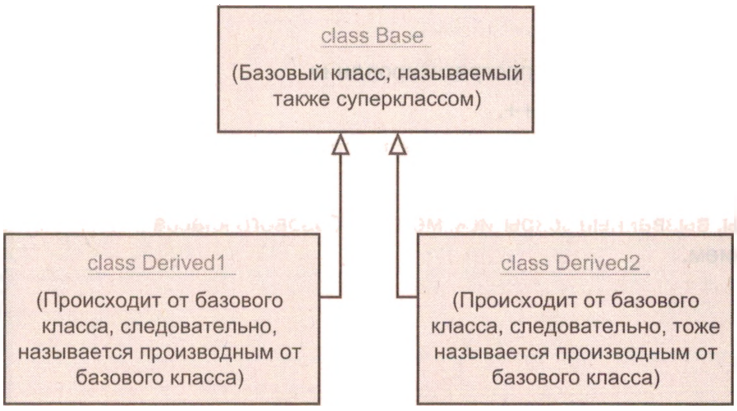


РИС. 10.1. Наследование классов

Наследование и происхождение

На рис. 10.1 приведена схема отношений между *базовым классом* (base class) и происходящими от него *производными классами* (derived class). Трудно представить прямо сейчас, чем могли бы быть базовый и производный классы; постарайтесь понять, что производный класс наследует от базового класса, и в том смысле базовый класс — то же самое, что и Смит для Тома.

ПРИМЕЧАНИЕ

Отношения между производным и базовым классами применимы только к *открытому наследованию* (public inheritance). Это занятие начинается с рассмотрения открытого наследования, чтобы объяснить саму концепцию наследования на примере его наиболее распространенной формы, прежде чем переходить к закрытому и защищенному наследованию.

Чтобы проще объяснить эту концепцию, рассмотрим базовый класс `Bird` (Птица). От класса `Bird` происходят классы `Crow` (Ворона), `Parrot` (Попугай) и `Kiwi` (Киви). Класс `Bird` определяет большинство основных атрибутов птицы, таких как наличие крыльев, откладывание яиц, способность лететь (у большинства). Производные классы, такие как `Crow`, `Parrot` и `Kiwi`, унаследовали бы эти атрибуты и скорректировали бы их (например, класс `Kiwi` не имел бы 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 *есть* рыба)
`Mammal` (Млекопитающее)	`Human` (Человек), `Elephant` (Слон), `Lion` (Лев), `Platypus` (Утконос) (Утконос *есть* млекопитающее)
`Bird` (Птица)	`Crow` (Ворона), `Parrot` (Попугай), `Ostrich` (Страус), `Kiwi` (Киви), `Platypus` (Утконос) (Утконос *есть* также и птица!)
`Shape` (Форма)	`Circle` (Круг), `Polygon` (Многоугольник) (Круг *есть* форма)
`Polygon` (Многоугольник)	`Triangle` (Треугольник), `Octagon` (Восьмиугольник) (Восьмиугольник *есть* многоугольник, который *есть* форма)

Эта таблица демонстрирует то, 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 хладнокровие. Однако Тунец отличается от Карпа внешним видом, скоростью плавания и тем фактом, что это морская рыба. Таким образом, классы `Tuna` и `Carp` наследуют общие характеристики от общего базового класса `Fish`, но все же специализируют атрибуты своего базового класса, чтобы отличатся друг от друга (рис. 10.2).

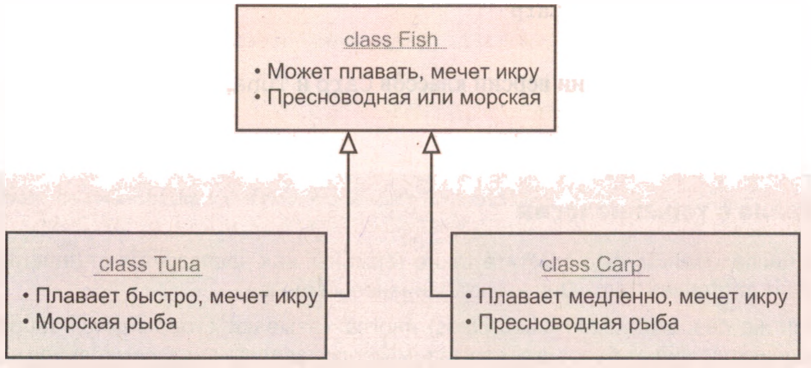


РИС. 10.2. Иерархические отношения между классами `Tuna`, `Carp` и `Fish`

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 наследником двух базовых

классов, класса Mammal и класса Bird, чтобы наследовать возможности млекопитающих и птиц. Это называется *множественным наследованием* (multiple inheritance) и обсуждается далее на этом занятии.

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: МодификаторДоступа Базовый
{
    // ... члены производного класса
};
```

МодификаторДоступа может быть любой, чаще всего используется модификатор public, для отношений "производный класс *есть* базовый класс" (is-a), или модификаторы private и protected для отношений "производный класс *содержит* базовый класс" (has-a).

Иерархическое представление наследования классом Carp класса Fish было бы таким:

```
class Fish
{
    // ... члены класса Fish
};

class Carp:public Fish
{
    // ... члены класса Carp
};
```

Пригодные для компиляции версии классов Carp и Tuna, производных от класса Fish, представлены в листинге 10.1.

Замечание о терминологии

Читая о наследовании, вы встретите такие термины, как *наследуется от* (inherits from) и *происходит от* (derives from). Они имеют одинаковый смысл.

Точно так же *базовый класс* (base class) иногда называют суперклассом (super class). Класс, происходящий от базового, называемый *производным классом* (derived class), может упоминаться как *подкласс* (subclass).

ЛИСТИНГ 10.1. Пример иерархии наследования

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: class Fish
 5: {
 6: public:
 7:     bool FreshWaterFish;
 8:
 9:     void Swim()
10:     {
11:         if (FreshWaterFish)
12:             cout << "Swims in lake" << endl;
13:         else
14:             cout << "Swims in sea" << endl;
15:     }
16: };
17:
18: class Tuna: public Fish
19: {
20: public:
21:     Tuna()
22:     {
23:         FreshWaterFish = false;
24:     }
25: };
26:
27: class Carp: public Fish
28: {
29: public:
30:     Carp()
31:     {
32:         FreshWaterFish = true;
33:     }
34: };
35:
36: int main()
37: {
38:     Carp myLunch;
39:     Tuna myDinner;
40:
41:     cout << "Getting my food to swim" << endl;
42:
43:     cout << "Lunch: ";
44:     myLunch.Swim();
45:
46:     cout << "Dinner: ";
47:     myDinner.Swim();
48:
49:     return 0;
50: }
```

Результат

```
Getting my food to swim
Lunch: Swims in lake
Dinner: Swims in sea
```

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 прошу свой завтрак и обед поплавать, вызвав их 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 заметить, эти классы весьма компактны, и ни один из них, кажется, не определяет метод, `Swim()`, который мы сумели успешно вызывать в функции `main()`. Очевидно, метод `Swim()` исходит от класса `Fish`, определенного в строках 3–15, и унаследованный ими. Поскольку класс `Fish` объявляет метод `Swim()` открытым, происходящие от него классы `Tuna` и `Carp` наследуют его (в ходе открытого наследования, осуществляемого в строках 17 и 26) и автоматически предоставляют. Обратите внимание, как конструкторы классов `Carp` и `Tuna` инициализирует флаг базового класса `FreshWaterFish`, который играет роль при решении, что отображает метод `Fish::Swim()`.

Модификатор доступа `protected`

В листинге 10.1 у класса `Fish` есть открытый атрибут `FreshWaterFish`, 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 обнаружился серьезный недостаток: если вы захотите, то даже в функции `main()` сможете вмешаться в значение этого флага, который был отмечен как `public`, а следовательно, открыт для манипулирования извне класса `Fish` при помощи, например, следующего кода:

```
myDinner.FreshWaterFish = true; // сделать тунца пресноводной рыбой!
```

Такого, очевидно, следует избегать. Необходимо средство, позволяющее определенным атрибутам в базовом классе быть доступными только для производного класса, но не для внешнего мира. Это означает, что логический флаг `FreshWaterFish` в классе `Fish` должен быть доступен для классов `Tuna` и `Carp`, которые происходят от него, но не для функции `main()`, где создаются экземпляры класса `Tuna` или `Carp`. Вот где пригодится ключевое слово `protected`.

ПРИМЕЧАНИЕ

> Ключевые слова `protected` (защищенный), а также `public` (открытый) и `private` (закрытый) являются модификаторами доступа. Когда вы объявляете 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 его 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

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: class Fish
 5: {
 6: protected:
 7:     bool FreshWaterFish; // доступно только производным классам
 8:
 9: public:
10:     void Swim()
11:     {
12:         if (FreshWaterFish)
13:             cout << "Swims in lake" << endl;
14:         else
15:             cout << "Swims in sea" << endl;
16:     }
17: };
18:
19: class Tuna: public Fish
20: {
21: public:
22:     Tuna()
23:     {
24:         FreshWaterFish = false; // установка значения защищенного
25:                                 // члена базового класса
26:     }
27: };
28:
29: class Carp: public Fish
30: {
31: public:
32:     Carp()
33:     {
34:         FreshWaterFish = false;
35:     }
36: };
37:
38: int main()
39: {
40:     Carp myLunch;
41:     Tuna myDinner;
42:
43:     cout << "Getting my food to swim" << endl;
44:
45:     cout << "Lunch: ";
46:     myLunch.Swim();
47:
48:     cout << "Dinner: ";
49:     myDinner.Swim();
50:
51:     // Снимите комментарий со строки ниже, чтобы убедиться в
52:     // недоступности защищенных членов извне иерархии класса
53:     // myLunch.FreshWaterFish = false;
54:
55:     return 0;
56: }
```

Результат

```
Getting my food to swim
Lunch: Swims in lake
Dinner: Swims in sea
```

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::FreshWaterFish` 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`, как показано в строках 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 для базового класса

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 класса 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::FreshWaterFish`. Так, базовый класс `Fish` может гарантировать, что каждый

производный класс вынужден будет указать, является ли рыба пресноводной или морской, как представлено в листинге 10.3.

ЛИСТИНГ 10.3. Конструктор производного класса со списками инициализации

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: class Fish
 5: {
 6: protected:
 7:     bool FreshWaterFish; // доступно только производным классам
 8:
 9: public:
10:     // конструктор класса Fish
11:     Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}
12:
13:     void Swim()
14:     {
15:         if (FreshWaterFish)
16:             cout << "Swims in lake" << endl;
17:         else
18:             cout << "Swims in sea" << endl;
19:     }
20: };
21:
22: class Tuna: public Fish
23: {
24: public:
25:     Tuna(): Fish(false) {}
26: };
27:
28: class Carp: public Fish
29: {
30: public:
31:     Carp(): Fish(true) {}
32: };
33:
34: int main()
35: {
36:     Carp myLunch;
37:     Tuna myDinner;
38:
39:     cout << "Getting my food to swim" << endl;
40:
41:     cout << "Lunch: ";
42:     myLunch.Swim();
43:
44:     cout << "Dinner: ";
45:     myDinner.Swim();
46:
47:     // Снимите комментарий со строки 48, чтобы убедиться в
48:     // недоступности защищенных членов извне иерархии класса
49:     // myLunch.FreshWaterFish = false;
50:
51:     return 0;
52: }
```

Результат

```
Getting my food to swim
Lunch: Swims in lake
Dinner: Swims in sea
```

Анализ

Теперь у класса `Fish` есть конструктор, который получает заданный по умолчанию параметр, инициализирующий переменную `Fish::FreshWaterFish`. Таким образом, единственная возможность создать объект класса `Fish` — это предоставить параметр, который инициализирует 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::FreshWaterFish`, несмотря на то, что она является защищенной, поскольку ее значение было установлено конструктором класса `Fish`.

Чтобы гарантировать максимальную защиту, если производные классы не нуждаются в доступе к атрибуту базового класса, отметьте его как `private`.

Производный класс, переопределяющий методы базового класса

Если производный класс реализует те же функции с теми же возвращаемыми значениями и сигнатурами, что и базовый класс, от которого он происходит, то он фактически переопределяет 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 другое
    }
};
```

Таким образом, если бы метод DoSomething() должен быть вызван с использованием экземпляра класса Derived, то это не задействовало бы функциональные возможности в классе Base.

Если классы Tuna и Carp должны реализовать собственный метод Swim(), который существует также и в базовом классе как Fish::Swim(), то его вызов в методе 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, переопределяет метод Fish::Swim() базового класса. Это демонстрирует листинг 10.4.

ЛИСТИНГ 10.4. Производные классы Tuna и Carp, переопределяющие метод Swim() базового класса Fish

```
1: #include <iostream>
2: using namespace std;
3:
4: class Fish
5: {
6: private:
7:     bool FreshWaterFish;
8:
9: public:
10:    // конструктор класса Fish
11:    Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}
12:
13:    void Swim()
14:    {
15:        if (FreshWaterFish)
16:            cout << "Swims in lake" << endl;
17:        else
18:            cout << "Swims in sea" << endl;
19:    }
20: };
21:
22: class Tuna: public Fish
23: {
24: public:
25:    Tuna(): Fish(false) {}
26:
27:    void Swim()
28:    {
29:        cout << "Tuna swims real fast" << endl;
30:    }
31: };
32:
33: class Carp: public Fish
34: {
35: public:
36:    Carp(): Fish(true) {}
```

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()` содержал бы обобщенную реализацию плавания, применимого ко всем рыбам, включая тунцов и карпов. Если специализированные реализации методов `Tuna::Swim()` и `Carp::Swim()` должны использовать обобщенную реализацию метода базового класса `Fish::Swim()`, используйте оператор области видимости (`::`), как показано в следующем коде:

```
class Carp: public Fish
{
public:
    Carp(): Fish(true) {}

    void Swim()
    {
        cout << "Carp swims real slow" << endl;
        Fish::Swim(); // использование оператора области видимости ::
    }
};
```

Этот подход используется в листинге 10.5.

ЛИСТИНГ 10.5. Использование оператора области видимости (`::`) для вызова методов базового класса из методов производных классов и функции `main()`

```
: #include <iostream>
: using namespace std;
:
: class Fish
: {
: private:
:     bool FreshWaterFish;
:
: public:
:     // конструктор класса Fish
:     Fish(bool IsFreshWater) : FreshWaterFish(IsFreshWater){}
:
:     void Swim()
:     {
:         if (FreshWaterFish)
:             cout << "Swims in lake" << endl;
:         else
:             cout << "Swims in sea" << endl;
:     }
: };
:
: class Tuna: public Fish
: {
: public:
:     Tuna(): Fish(false) {}
:
```

```
26:     void Swim()
27:     {
28:         cout << "Tuna swims real fast" << endl;
29:     }
30: };
31:
32: class Carp: public Fish
33: {
34: public:
35:     Carp(): Fish(true) {}
36:
37:     void Swim()
38:     {
39:         cout << "Carp swims real slow" << endl;
40:         Fish::Swim();
41:     }
42: };
43:
44: int main()
45: {
46:     Carp myLunch;
47:     Tuna myDinner;
48:
49:     cout << "Getting my food to swim" << endl;
50:
51:     cout << "Lunch: ";
52:     myLunch.Swim();
53:
54:     cout << "Dinner: ";
55:     myDinner.Fish::Swim();
56:
57:     return 0;
58: }
```

Результат

```
Getting my food to swim
Lunch: Carp swims real slow
Swims in lake
Dinner: Swims in sea
```

Анализ

Метод `Carp::Swim()` в строках 37–41 демонстрирует вызов функции `Fish::Swim()` базового класса с использованием оператора области видимости (`::`). Строка 55, с другой стороны, демонстрирует возможность использования оператора 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

Переопределение может принять критическую форму, и тогда метод `Tuna::Swim()` потенциально способен скрыть все доступные перегруженные версии функции `Fish::Swim()`, даже приведя к неудаче компиляции, когда перегружаются используемые версии (поэтому они и называется *скрытыми* (hidden)), как показано в листинге 10.6.

Листинг 10.6. Сокрытие методом `Tuna::Swim()` перегруженного метода `Fish::Swim(bool)`

```
#include <iostream>
using namespace std;

class Fish
{
public:
    void Swim()
    {
        cout << "Fish swims... !" << endl;
    }

    void Swim(bool FreshWaterFish)
    {
        if (FreshWaterFish)
            cout << "Swims in lake" << endl;
        else
            cout << "Swims in sea" << endl;
    }
};

class Tuna: public Fish
{
public:
    void Swim()
    {
        cout << "Tuna swims real fast" << endl;
    }
};

int main()
{
    Tuna myDinner;

    cout << "Getting my food to swim" << endl;

    // myDinner.Swim(false); // отказ компиляции: Fish::Swim(bool)
                             // скрыт методом Tuna::Swim()
    myDinner.Swim();

    return 0;
}
```

Результат

```
Getting my food to swim
Tuna swims real fast
```

Анализ

Эта версия класса Fish немного отличается от тех, которые вы видели до сих пор. Кроме минимизации версией, 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), не будет ошибкой ожидать, что обе версии метода Fish::Swim() будут доступны через экземпляр класса Tuna. Однако в результате того факта, что класс Tuna реализует собственную версию метода Tuna::Swim() (строки 23–26), функция Fish::Swim(bool) скрывается от компилятора. Если снять комментарий со строки 35, произойдет отказ компиляции.

Так, если необходимо вызвать функцию Fish::Swim(bool) через экземпляр класса Tuna, возможны следующие решения.

- Решение 1. Используйте оператор области видимости в функции main():

```
myDinner.Fish::Swim();
```

- Решение 2. Используйте в классе Tuna ключевое слово using, чтобы показать скрытые методы Swim() в классе Fish:

```
class Tuna: public Fish
{
public:
    using Fish::Swim; // показать скрытые методы Swim()
                      // в базовом классе Fish

    void Swim()
    {
        cout << «Tuna swims real fast» << endl;
    }
};
```

- Решение 3. Переопределите все перегруженные варианты метода Swim() в классе Tuna (если хотите, вызовите метод Fish::Swim(...) через Tuna::Fish(...)):

```
class Tuna: public Fish
{
public:
    void Swim(bool FreshWaterFish)
    {
        Fish::Swim(FreshWaterFish);
    }

    void Swim()
    {
        cout << «Tuna swims real fast» << endl;
    }
};
```

Порядок создания

При создании объекта класса Tuna, производного от класса Fish, конструктор класса Tuna будет вызван до или после конструктора класса Fish? Кроме того, каков порядок создания таких атрибутов класса, как Fish::FreshWaterFish, при создании экземпляра объектов в иерархии класса? Дело в том, что объекты базового класса создаются перед объектами производного. Таким образом, часть Fish объекта класса Tuna создается сначала, чтобы его члены, в частности открытые и защищенные, были готовы для использования, когда будет создаваться часть Tuna. В ходе создания экземпляра класса Fish и Tuna такие атрибуты, как Fish::FreshWaterFish, создаются до вызова конструктора Fish::Fish(), гарантируя существование атрибутов на момент работы конструктора с ними. То же самое относится к конструктору Tuna::Tuna().

Порядок удаления

Когда экземпляр класса Tuna выходит из области видимости, последовательность удаления противоположна последовательности создания. В листинге 10.7 приведен простой пример, демонстрирующий последовательность создания и удаления.

Листинг 10.7. Порядок создания и удаления базового класса, производного класса и его членов

```
#include <iostream>
using namespace std;

class FishDummyMember
{
public:
    FishDummyMember()
    {
        cout << "FishDummyMember constructor" << endl;
    }

    ~FishDummyMember()
    {
        cout << "FishDummyMember destructor" << endl;
    }
};

class Fish
{
protected:
    FishDummyMember dummy;

public:
    // Конструктор класса Fish
    Fish()
    {
        cout << "Fish constructor" << endl;
    }

    ~Fish()
```

```
30:     {
31:         cout << "Fish destructor" << endl;
32:     }
33: };
34:
35: class TunaDummyMember
36: {
37: public:
38:     TunaDummyMember()
39:     {
40:         cout << "TunaDummyMember constructor" << endl;
41:     }
42:
43:     ~TunaDummyMember()
44:     {
45:         cout << "TunaDummyMember destructor" << endl;
46:     }
47: };
48:
49:
50: class Tuna: public Fish
51: {
52: private:
53:     TunaDummyMember dummy;
54:
55: public:
56:     Tuna()
57:     {
58:         cout << "Tuna constructor" << endl;
59:     }
60:     ~Tuna()
61:     {
62:         cout << "Tuna destructor" << endl;
63:     }
64:
65: };
66:
67: int main()
68: {
69:     Tuna myDinner;
70: }
```

Результат

```
FishDummyMember constructor
Fish constructor
TunaDummyMember constructor
Tuna constructor
Tuna destructor
TunaDummyMember destructor
Fish destructor
FishDummyMember destructor
```

Анализ

Функция `main()`, представленная в строках 67–70, поразительно мала для объема создаваемого ею вывода. Создания экземпляра класса `Tuna` достаточно для этих строк вывода, поскольку операторы `cout` вставлены в конструкторы и деструкторы всех задействованных объектов. Для демонстрации создания и удаления переменных определены два вымышленных класса, `FishDummyMember` и `TunaDummyMember`, с операторами `cout` в конструкторах и деструкторах. Классы `Fish` и `Tuna` содержат члены для каждого из этих вымышленных классов (строки 20 и 53). Вывод указывает, что создание объекта класса `Tuna` фактически начинается сверху иерархии. Так, часть базового класса `Fish` в составе класса `Tuna` создается первой, при этом такие его члены, как `Fish::dummy`, создаются сначала. Далее следует конструктор класса `Fish`, который естественно выполняется после создания таких атрибутов, как `dummy`. После создания экземпляра базового класса создание экземпляра `Tuna` продолжается созданием экземпляра `Tuna::dummy` и завершается выполнением кода конструктора `Tuna::Tuna()`. Вывод демонстрирует, что последовательность удаления прямо противоположна.

Закрытое наследование

Закрытое наследование (private inheritance) отличается от открытого (которое рассматривалось до сих пор) тем, что в строке объявления производного класса как происходящего от базового 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-либо, еще 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 и потому, что класс `Tuna` происходит от класса `Fish` с использованием открытого наследования. Попробуйте переименовать ключевое слово `public` на `private` в строке 17, и вы получите сбой компиляции.

Таким образом, для мира вне иерархии наследования закрытое наследование по существу не означает отношение *есть* (is-a) (вообразите тунца, который не может плавать!). Поскольку закрытое наследование позволяет использовать атрибуты и методы базового класса только производным классам, которые происходят от него, создаются отношения,

называемые *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 *содержит* мотор)
`Heart` (Сердце)	`Mammal` (Млекопитающее *содержит* сердце)
`Refill` (Стержень)	`Pen` (Ручка *содержит* стержень)
`Moon` (Луна)	`Sky` (Небо *содержит* луну)

Давайте рассмотрим закрытое наследование на примере отношений автомобиля с его мотором (листинг 10.8).

ЛИСТИНГ 10.8. Класс `Car`, связанный с классом `Motor` через закрытое наследование

```
#include <iostream>
using namespace std;

class Motor
{
public:
    void SwitchIgnition()
    {
        cout << "Ignition ON" << endl;
    }
    void PumpFuel()
    {
        cout << "Fuel in cylinders" << endl;
    }
    void FireCylinders()
    {
        cout << "Vroooom" << endl;
    }
};

class Car:private Motor
{
public:
    void Move()
    {
        SwitchIgnition();
        PumpFuel();
        FireCylinders();
    }
};

int main()
{
    Car myDreamCar;
    myDreamCar.Move();

    return 0;
}
```

Результат

```
Ignition ON
Fuel in cylinders
Vroooom
```

Анализ

Класс Motor, определенный в строках 3–18, очень прост, он содержит три защищенные 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 произойдет другой класс, например SuperCar, то, независимо от характера наследования, у класса SuperCar 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, наиболее ограничивающий модификатор доступа доминирует при принятии компилятором решения о том, должен ли у некого класса быть доступ к открытым или защищенным членам базового класса.

Защищенное наследование

Защищенное наследование отличается от открытого наличием ключевого слова protected в строке объявления производного класс как происходящего от базового 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:

- Реализует отношения *содержит* (has-a).

- Позволяет производному классу обращаться ко всем открытым и защищенным членам базового.
- Вне иерархии наследования нельзя при помощи экземпляра производного класса обратиться к открытым членам базового класса.

Но защищенное наследование все же отличается от закрытого, когда дело доходит до следующего производного класса, унаследованного от него:

```
class Derived2: protected Derived
{
    // имеет доступ к членам Base
};
```

Иерархия защищенного наследования позволяет производному классу производного класса (т.е. классу `Derived2`) обращаться к открытым членам базового класса (листинг 10.9). Это не было бы возможно, если бы при наследовании классом `Derived` класса `Base` использовалось ключевое слово `private`.

ЛИСТИНГ 10.9. Класс `SuperCar`, производный от класса `Car`, происходящего от класса `Motor`, при защищенном наследовании

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Motor
 4: {
 5: public:
 6:     void SwitchIgnition()
 7:     {
 8:         cout << "Ignition ON" << endl;
 9:     }
10:     void PumpFuel()
11:     {
12:         cout << "Fuel in cylinders" << endl;
13:     }
14:     void FireCylinders()
15:     {
16:         cout << "Vroooom" << 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 SuperCar:protected Car
32: {
33: public:
34:     void Move()
```

```
35:     {
36:         SwitchIgnition(); // имеет доступ к членам базового благодаря
37:         PumpFuel();       // защищенному наследованию между Car и Motor
38:         FireCylinders();
39:         FireCylinders();
40:         FireCylinders();
41:     }
42: };
43:
44: int main()
45: {
46:     SuperCar myDreamCar;
47:     myDreamCar.Move();
48:
49:     return 0;
50: }
```

Результат

```
Ignition ON
Fuel in cylinders
Vroooom
Vroooom
Vroooom
```

Анализ

Класс `Car` защищенно наследует класс `Motor` (строка 20). Класс `SuperCar` защищенно наследует класс `Car` (строка 31). Как можно заметить, реализация метода `SuperCar::Move()` использует открытые методы, определенные в базовом классе `Motor`. Этот доступ к самому 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 выберет самый ограничивающий из использованных модификаторов доступа. Обратите внимание, что характер отношений между классами `Car` и `SuperCar` не имеет значения при доступе к базовому классу. Так, даже если в строке 31 заменить ключевое слово `protected` на `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 также может 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, попытайся мы последовать за такими разработками.

ПРИМЕЧАНИЕ

Наличие экземпляра класса `Motor` как закрытого члена, вместо наследования от него, называется *композиция* (composition) или *объединение* (aggregation). Такой класс `Car` выглядел бы следующим образом:

```
class Car
{
private:
    Motor heartOfCar;

public:
    void Move()
    {
        heartOfCar.SwitchIgnition();
        heartOfCar.PumpFuel();
        heartOfCar.FireCylinders();
    }
};
```

Это может быть хорошим ходом, поскольку позволяет легко добавлять к существующему классу `Car` больше моторов как атрибутов, не изменяя его иерархию наследования или предоставляемые клиентам возможности.

Проблема отсечения

Что будет, если программист сделает следующее?

```
Derived objectDerived;
Base objectBase = objectDerived;
```

Или следующее?

```
void FuncUseBase(Base input);
...
Derived objectDerived;
FuncUseBase(objectDerived); // objectDerived будет отсечен при
                            // копировании во время вызова функции
```

В обоих случаях объект производного класса копируется в другой объект, но уже базового класса явно, при присвоении, или косвенно, при передаче в качестве аргумента. В этих случаях компилятор скопирует в объект `objectDerived` только часть, соответствующую классу `Base`, а не весь объект. Как правило, это вовсе не то, чего ожидает программист, и это нежелательное сокращение части данных, специализирующей производный класс относительно базового, называется *отсечением* (slicing).

ВНИМАНИЕ!

Чтобы избежать проблемы отсечения, не передавайте параметры по значению. Передавайте их как указатели на базовый класс или как ссылку (можно `const`) на него же.

Множественное наследование

Ранее на этом занятии упоминалось о том, что иногда могло бы 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: модификаторДоступа Базовый1, модификаторДоступа Базовый2
{
    // 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).

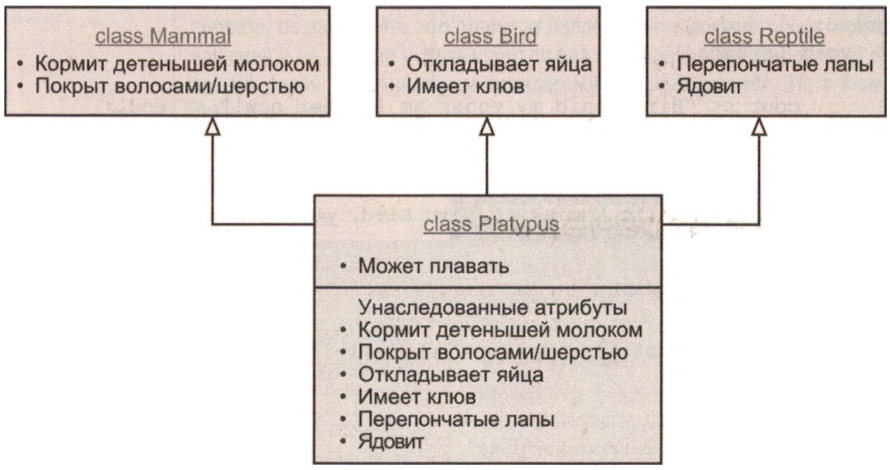


РИС. 10.3. Отношения между классом `Platypus` и классами `Mammal`, `Reptile` и `Bird`

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 << "Mammal: Baby says glug!" << endl;
 9:     }
10: };
11:
12: class Reptile
13: {
14: public:
15:     void SpitVenom()
16:     {
17:         cout << "Reptile: Shoo enemy! Spits venom!" << endl;
18:     }
19: };
20:
21: class Bird
22: {
23: public:
24:     void LayEggs()
25:     {
26:         cout << "Bird: Laid my eggs, am lighter now!" << endl;
27:     }
28: };
29:
30: class Platypus: public Mammal, public Bird, public Reptile
31: {
32: public:
33:     void Swim()
34:     {
35:         cout << "Platypus: Voila, I can swim!" << 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

```
Bird: Laid my eggs, am lighter now!
Mammal: Baby says glug!
Reptile: Shoo enemy! Spits venom!
Platypus: Voila, I can swim!
```

Анализ

Определение средств класса `Platypus` действительно компактно (строки 30–37). По существу, он просто наследует их от трех классов: `Mammal`, `Reptile` и `Bird`. Функция `main()` в строках 41–44 способна обратиться к трем индивидуальным характеристикам 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, как производный класс предоставляет все открытые члены (в данном случае методы) многих своих базовых классов.

ПРИМЕЧАНИЕ

Утконос может плавать, но это не рыба. Следовательно, в листинге 10.10 мы не наследовали класс `Platypus` также от класса `Fish` только для удобства использования существующего метода `Fish::Swim()`. Принимая решения в проекте, не забывайте, что открытое наследование должно также отражать отношения и не должно использоваться без разбора, лишь для решения определенных задач, связанных с многократным использованием. Этого можно достичь и по-другому.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Создавайте открытую иерархию наследования для установки отношений *есть*	**Не создавайте** иерархию наследования только для многократного использования обычных функций
Создавайте закрытую или защищенную иерархию наследования для установки отношений *содержит*	**Не используйте** закрытое и открытое наследование без разбора, поскольку впоследствии это может стать узким местом архитектуры вашего приложения
Помните, открытое наследование означает, что у классов, производных от производного класса, есть доступ к открытым и защищенным членам базового класса	**Не создавайте** функции производного класса (с тем же именем, но другим набором входных параметров), которые непреднамеренно скрывают таковые в базовом классе
Помните, закрытое наследование означает, что даже классы, производные от производного класса, не имеют доступа к базовому классу	
Помните, защищенное наследование означает, что у классов, производных от производного класса, есть доступ к защищенным и открытым методам базового класса	
Помните, что, независимо от характера наследственных отношений, закрытые члены в базовом классе недоступны никаким производным классам	

Резюме

На сегодняшнем занятии рассматривались основы наследования в языке C++. Вы узнали, что открытое наследование — это отношения *есть* между производным и базовым

классами, а закрытое и защищенное наследование создает отношения *имеет*. Вы видели, что применение модификатора доступа `protected` предоставляет члены базового класса только к производному, 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 переопределения методов и их сокрытия, а также узнали, как избежать нежелательного сокрытия метода с использованием 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 да, то какую?**

  Человек, лев и кит — все млекопитающие и по существу имеют отношения *есть*. Используйте открытое наследование, где класс `Mammal` будет базовым, а другие классы, `Human`, `Lion` и `Whale`, производными от него.

- **В чем разница между терминами *производный класс* и *подкласс*?**

  По сути, никакой. Оба подразумевают класс, который происходит (т.е. специализирует) от базового класса.

- **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 Г, "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 передаю объект производного класса по значению функции, ожидающей в виде 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 друг с другом.
3. Класс `D2` происходит от класса `D1`, который происходит от класса `Base`. Какой модификатор доступа 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, создания иерархии наследования и понимая, что открытое наследование по существу моделирует отношения *есть*, пришло время переходить к употреблению этих знаний в изучении святой чаши Грааля объектно-ориентированного программирования — полиморфизма.

На сегодняшнем 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 средство объектно-ориентированных языков, позволяющее обрабатывать подобным образом объекты разных типов. Это занятие посвящено полиморфному поведению, которое может быть реализовано на языке C++ через иерархию наследования, известную также как *полиморфизм подтипа* (subtype polymorphism).

Потребность в полиморфном поведении

На занятии 10, "Реализация наследования", вы видели, как классы `Tuna` и `Carp` наследовали открытый метод `Swim()` от класса `Fish` (см. листинг 10.1). Классы `Tuna` и `Carp` способны 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 было удалено все, что не является абсолютно необходимым для объяснения рассматриваемой темы, а количество строк кода сведено к минимуму, чтобы улучшить удобочитаемость.

При реальном программировании классы необходимо создавать правильно, а также разрабатывать осмысленные иерархии наследования, учитывающие цели проекта и приложения в перспективе.

ЛИСТИНГ 11.1. Вызов методов при помощи экземпляра базового класса `Fish`, который принадлежит классу `Tuna`

```
#include <iostream>
using namespace std;

class Fish
{
public:
    void Swim()
    {
        cout << "Fish swims!" << endl;
    }
};

class Tuna:public Fish
{
public:
    // переопределение Fish::Swim
    void Swim()
    {
        cout << "Tuna swims!" << endl;
    }
};
```

```
11:
12: void MakeFishSwim(Fish& InputFish)
13: {
14:     // вызов Fish::Swim
15:     InputFish.Swim();
16: }
17:
18: int main()
19: {
20:     Tuna myDinner;
21:
22:     // вызов Tuna::Swim
23:     myDinner.Swim();
24:
25:     // передача Tuna как Fish
26:     MakeFishSwim(myDinner);
27:
28:     return 0;
29: }
```

Результат

```
Tuna swims!
Fish swims!
```

Анализ

Класс Tuna специализирует класс Fish через открытое наследование, как показано в строке 12. Он также переопределяет метод Fish::Swim(). Функция main() напрямую вызывает метод Tuna::Swim() в строке 33 и передает объект myDinner (класса Tuna) как параметр для функции MakeFishSwim(), которая интерпретирует это как ссылку Fish&, как видно в ее объявлении (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(). Так, вторая строка вывода означает, что тот же объект класса Tuna создал вывод, как у класса Fish, без всякой специализации (с таким же успехом это мог быть класс Carp).

Однако пользователь, в идеале, ожидал бы, что объект класса Tuna поведет себя как тунец, даже если вызван метод Fish::Swim(). Другими словами, когда метод InputFish.Swim() вызывается в строке 25, он ожидает, что будет выполнен метод Tuna::Swim(). Такое полиморфное поведение, когда объект известного класса типа Fish может вести себя как объект фактического типа, а именно производный класс Tuna, может быть реализован, если сделать функцию Fish::Swim() виртуальной.

Полиморфное поведение, реализованное при помощи виртуальных функций

Доступ к объекту класса Fish возможен через указатель Fish* или по ссылке Fish&. Объект класса Fish может быть создан индивидуально или как часть объекта класса Tuna или Carp, производного от класса Fish. Неважно, как именно, но 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 такого специализированного класса, как `Tuna` или `Carp`. Вы можете гарантировать это, объявив функцию `Swim()` в базовом классе `Fish` как *виртуальную функцию* (virtual function):

```
class Base
{
    virtual ReturnType FunctionName (Parameter List);
};
class Derived
{
    ReturnType FunctionName (Parameter List);
};
```

Использование ключевого слова `virtual` означает, что компилятор гарантирует вызов любого переопределенного варианта затребованного 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 << "Fish swims!" << endl;
 9:     }
10: };
11:
12: class Tuna:public Fish
13: {
14: public:
15:     // переопределение Fish::Swim
16:     void Swim()
17:     {
18:         cout << "Tuna swims!" << endl;
19:     }
20: };
21:
22: class Carp:public Fish
23: {
24: public:
25:     // переопределение Fish::Swim
26:     void Swim()
27:     {
28:         cout << "Carp swims!" << 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 Tuna как Fish
44:     MakeFishSwim(myDinner);
45:
46:     // передача Carp как Fish
47:     MakeFishSwim(myLunch);
48:
49:     return 0;
50: }
```

Результат

```
Tuna swims!
Carp swims!
```

Анализ

Реализация функции `MakeFishSwim(Fish&)` никак не изменилась с листинга 11.1, а вывод получился совсем иной. Метод `Fish::Swim()` не был вызван вообще из-за присутствия переопределенных версий `Tuna::Swim()` и `Carp::Swim()`, которые получили преимущество над методом `Fish::Swim()`, поскольку последний был объявлен как виртуальная функция. Это очень важный момент. Он свидетельствует, 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 полиморфизм: обработка различных рыб как общего типа `Fish`, при гарантии выполнения правильной реализации метода `Swim()`, предоставляемого производными типами.

Потребность в виртуальных деструкторах

У средств, представленных в листинге 11.1, есть и оборотная сторона: неумышленный 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 << "Constructed Fish" << endl;
 9:     }
10:     ~Fish()
11:     {
12:         cout << "Destroyed Fish" << endl;
13:     }
14: };
15:
16: class Tuna:public Fish
17: {
18: public:
19:     Tuna()
20:     {
21:         cout << "Constructed Tuna" << endl;
22:     }
23:     ~Tuna()
24:     {
25:         cout << "Destroyed Tuna" << endl;
26:     }
27: };
28:
29: void DeleteFishMemory(Fish* pFish)
30: {
31:     delete pFish;
32: }
33:
34: int main()
35: {
36:     cout << "Allocating a Tuna on the free store:" << endl;
37:     Tuna* pTuna = new Tuna;
38:     cout << "Deleting the Tuna: " << endl;
39:     DeleteFishMemory(pTuna);
40:
41:     cout << "Instantiating a Tuna on the stack:" << endl;
42:     Tuna myDinner;
43:     cout << "Automatic destruction as it goes out of scope: "
           << endl;
44:
45:     return 0;
46: }
```

Результат

```
Allocating a Tuna on the free store:
Constructed Fish
Constructed Tuna
Deleting the Tuna:
Destroyed Fish
Instantiating a Tuna on the stack:
```

```
Constructed Fish
Constructed Tuna
Automatic destruction as it goes out of scope:
Destroyed Tuna
Destroyed Fish
```

Анализ

Функция `main()` создает экземпляр класса `Tuna` в динамической памяти, используя оператор `new` в строке 37, а затем сразу освобождает зарезервированную 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 создают операторы `cout` в конструкторах и деструкторах классов `Fish` и `Tuna`. Обратите внимание: несмотря на то, что обе части, `Tuna` и `Fish`, объекта были созданы в динамической памяти, поскольку использовался оператор `new`, при удалении был вызван только деструктор части `Fish`, а не класса `Tuna`. Это находится в абсолютном контрасте с созданием и удалением локального объекта `myDinner`, где 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.

Проблема в том, что код в деструкторе производного класса, объект которого был создан в динамической памяти при помощи оператора `new`, не будет вызван, если будет применен оператор `delete` для указателя типа `Base*`. В результате ресурсы не будут освобождены, произойдет утечка памяти и другие ненужные неприятности.

Чтобы избежать этой проблемы, используйте виртуальные деструкторы, как показано в листинге 11.4.

ЛИСТИНГ 11.4. Использование виртуальных деструкторов для гарантии вызова деструкторов производных классов при удалении указателя типа `Base*`

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6:     Fish()
7:     {
8:         cout << "Constructed Fish" << endl;
9:     }
10:    virtual ~Fish() // виртуальный деструктор!
11:    {
12:        cout << "Destroyed Fish" << endl;
13:    }
14: };
15:
16: class Tuna:public Fish
17: {
18: public:
19:     Tuna()
20:     {
```

```
21:         cout << "Constructed Tuna" << endl;
22:     }
23:     ~Tuna()
24:     {
25:         cout << "Destroyed Tuna" << endl;
26:     }
27: };
28:
29: void DeleteFishMemory(Fish* pFish)
30: {
31:     delete pFish;
32: }
33:
34: int main()
35: {
36:     cout << "Allocating a Tuna on the free store:" << endl;
37:     Tuna* pTuna = new Tuna;
38:     cout << "Deleting the Tuna: " << endl;
39:     DeleteFishMemory(pTuna);
40:
41:     cout << "Instantiating a Tuna on the stack:" << endl;
42:     Tuna myDinner;
43:     cout << "Automatic destruction as it goes out of scope: "
            << endl;
44:
45:     return 0;
46: }
```

Результат

```
Allocating a Tuna on the free store:
Constructed Fish
Constructed Tuna
Deleting the Tuna:
Destroyed Tuna
Destroyed Fish
Instantiating a Tuna on the stack:
Constructed Fish
Constructed Tuna
Automatic destruction as it goes out of scope:
Destroyed Tuna
Destroyed 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 изменение, по существу, заставило компилятор выполнить деструктор `Tuna::~Tuna()` в дополнение к деструктору `Fish::~Fish()`, когда был вызван оператор `delete` для указателя `Fish*`, который фактически указывает на объект класса `Tuna`, как показано в строке 31. Теперь вывод 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 никто с указателем Base* не сможет вызвать оператор delete способом, который не подразумевает вызова деструкторов производных классов.

Как работают виртуальные функции. Понятие таблицы виртуальной функции

ПРИМЕЧАНИЕ

Необязательно изучать этот раздел, чтобы использовать полиморфизм. Если вам не любопытно, можете не читать его.

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, он не в состоянии гарантировать выполнение различных методов Swim() в различные моменты времени. Очевидно, решение о том, какой метод Swim() должен быть вызван, принимается во время выполнения с использованием скрытой логики, реализующей полиморфизм, 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, переопределяет метод Base::Func2(). предоставляя другие виртуальные функции непосредственно из класса Base:

```
class Derived: public Base
{
public:
    virtual void Func1()
    {
        // Func2 переопределяет Base::Func2
    }
    // нет реализации для Func2
    virtual void FuncN()
    {
        // реализация FuncN
    }
};
```

Компилятор видит иерархию наследования и понимает, что класс Base определяет некоторые виртуальные функции, которые были переопределены в классе Derived. Теперь компилятор должен составить таблицу, называемую *таблицей виртуальной функции* (Virtual Function Table — VFT), для каждого класса, который реализует виртуальную функцию, и производного класса, который переопределяет ее. Другими словами, классы Base и Derived получают собственной экземпляр своей таблицы виртуальной функции. Когда создается объект этих классов, инициализируется скрытый указатель (назовем его VFT*) на соответствующую таблицу VFT. Таблицу виртуальной функции можно представить как статический массив, содержащий указатели на функцию, каждый из которых указывает на виртуальную функцию (или ее переопределенную версию), представляющую интерес (рис. 11.1).

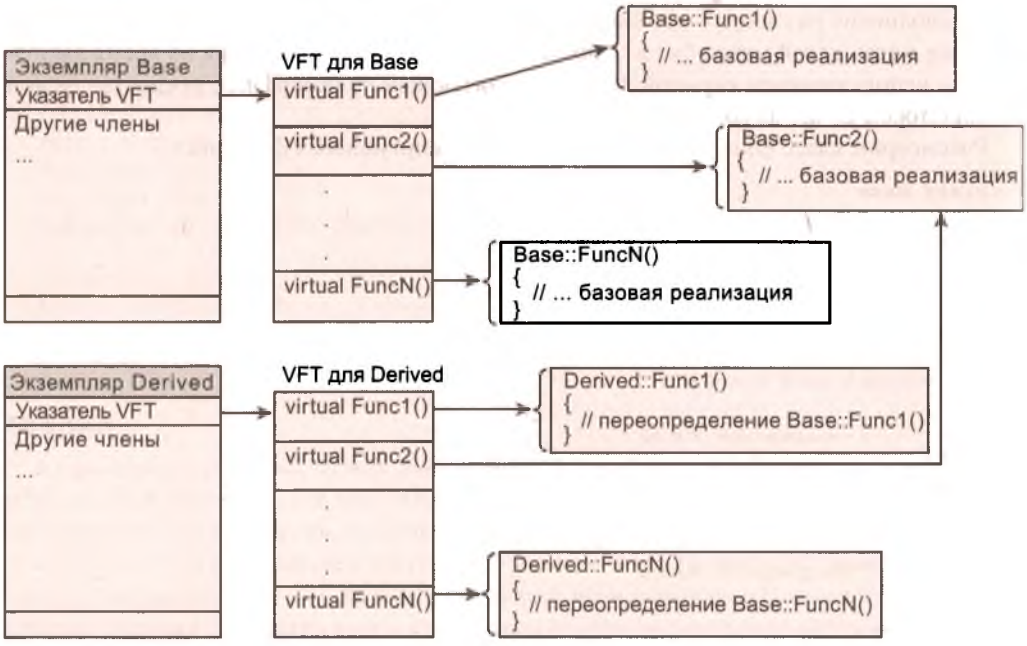


РИС. 11.1. Представление таблицы виртуальной функции для классов Derived и Base

Таким образом, каждая таблица состоит 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 следовательно, указатель на функцию указывает на реализацию в классе Base.

Это означает, что при следующем вызове пользователя класса Derived

```
CDerived objDerived;
objDerived.Func2();
```

компилятор осуществляет поиск в таблице VFT класса Derived и обеспечивает вызов реализации Base::Func2(). Это относится также к вызовам методов, которые были виртуально переопределены:

```
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 параметром objBase как экземпляр класса Base, указатель VFT в этом экземпляре все еще указывает на ту же таблицу, составленную для класса Derived. Таким образом, функцией Func1(), выполняемой через этот указатель VFT, является, конечно, Derived::Func1().

Вот как таблицы виртуальной функции помогают в реализации полиморфизма в C++.

Листинг 11.5 доказывает существование скрытого указателя VFT на примере сравнения размера двух идентичных классов, но у одного из них есть виртуальная функция, а у другого нет.

ЛИСТИНГ 11.5. Демонстрация наличия скрытого указателя VFT при сравнении двух одинаковых классов, функция одного из которых объявлена виртуальный

```
 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

```
sizeof(SimpleClass) = 8
sizeof(Base) = 12
```

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 виртуальной функции для класса `Base` и резервирует место для указателя на нее в том же классе `Base`, как его скрытый член. Этот указатель использует 4 дополнительных байта на 32-разрядной системе автора, что и является доказательством его существования.

ПРИМЕЧАНИЕ

Язык C++ позволяет запросить указатель `Base*`, если он имеет тип `Derived*`, при помощи оператора приведения типов `dynamic_cast` и последующего условного выполнения на основе результата запроса.

Это называется *идентификацией типа времени выполнения* (Run Time Type Identification – RTTI), и в идеале этого следует избегать, несмотря на поддержку большинством компиляторов C++. Дело в том, что необходимость узнать тип объекта производного класса по указателю базового класса обычно считается плохой практикой программирования.

Более подробно RTTI и оператор `dynamic_cast` обсуждаются на занятии 13, "Операторы приведения".

Абстрактные классы и чистые виртуальные функции

Базовый класс, экземпляр которого не может быть создан, называется *абстрактным классом* (abstract base class). Цель у такого базового класса только одна — от него получают производные классы. Язык C++ позволяет создать абстрактный класс, используя чистые виртуальные функции.

Метод называют *чистым виртуальным* (pure virtual), когда его объявление выглядит так:

```
class АбстрактныйБазовый
{
```

```
    public:
    virtual void СделатьНечто() = 0; // чистый виртуальный метод
};
```

Это объявление, по существу, говорит компилятору о том, что метод *СделатьНечто*() должен быть реализован классом, который наследует класс *АбстрактныйБазовый*.

```
class Производный: public АбстрактныйБазовый
{
public:
    void СделатьНечто() // чистый виртуальный метод
    {
        cout << "Implemented virtual function" << endl;
    }
}
```

Таким образом, класс *АбстрактныйБазовый* выполнил свою задачу — заставил класс *Производный* предоставить реализацию для виртуального метода *СделатьНечто*(). Такая возможность базового класса потребовать поддержки методов с определенным именем и сигнатурой в производных классах обеспечивает *интерфейс* (interface). 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 чистой 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 чистой виртуальной функции Swim
 7:     virtual void Swim() = 0;
 8: };
 9:
10: class Tuna:public Fish
11: {
12: public:
13:     void Swim()
14:     {
15:         cout << "Tuna swims fast in the sea!" << endl;
16:     }
17: };
18:
19: class Carp:public Fish
20: {
21:     void Swim()
22:     {
23:         cout << "Carp swims slow in the lake!" << endl;
24:     }
```

```
25: };
26:
27: void MakeFishSwim(Fish& inputFish)
28: {
29:     inputFish.Swim();
30: }
31:
32: int main()
33: {
34:     // Fish myFish; // Ошибка, нельзя создать экземпляр
                        // абстрактного класса
35:     Carp myLunch;
36:     Tuna myDinner;
37:
38:     MakeFishSwim(myLunch);
39:     MakeFishSwim(myDinner);
40:
41:     return 0;
42: }
```

Результат

```
Carp swims slow in the lake!
Tuna swims fast in the sea!
```

Анализ

Существенна первая (закомментированная) строка функции `main()` (строка 34). Это 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 чистой 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, где 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

Абстрактные базовые классы (Abstract Base Class) зачастую называют просто ABC.

Классы ABC накладывают на ваш 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.

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.

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
```

```
25: };
26:
27: void MakeFishSwim(Fish& inputFish)
28: {
29:     inputFish.Swim();
30: }
31:
32: int main()
33: {
34:     // Fish myFish; // Ошибка, нельзя создать экземпляр
                        // абстрактного класса
35:     Carp myLunch;
36:     Tuna myDinner;
37:
38:     MakeFishSwim(myLunch);
39:     MakeFishSwim(myDinner);
40:
41:     return 0;
42: }
```

Результат

```
Carp swims slow in the lake!
Tuna swims fast in the sea!
```

Анализ

Существенна первая (закомментированная) строка функции `main()` (строка 34). Это 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 чистой 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, где 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

Абстрактные базовые классы (Abstract Base Class) зачастую называют просто ABC.

Классы ABC накладывают на ваш 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.

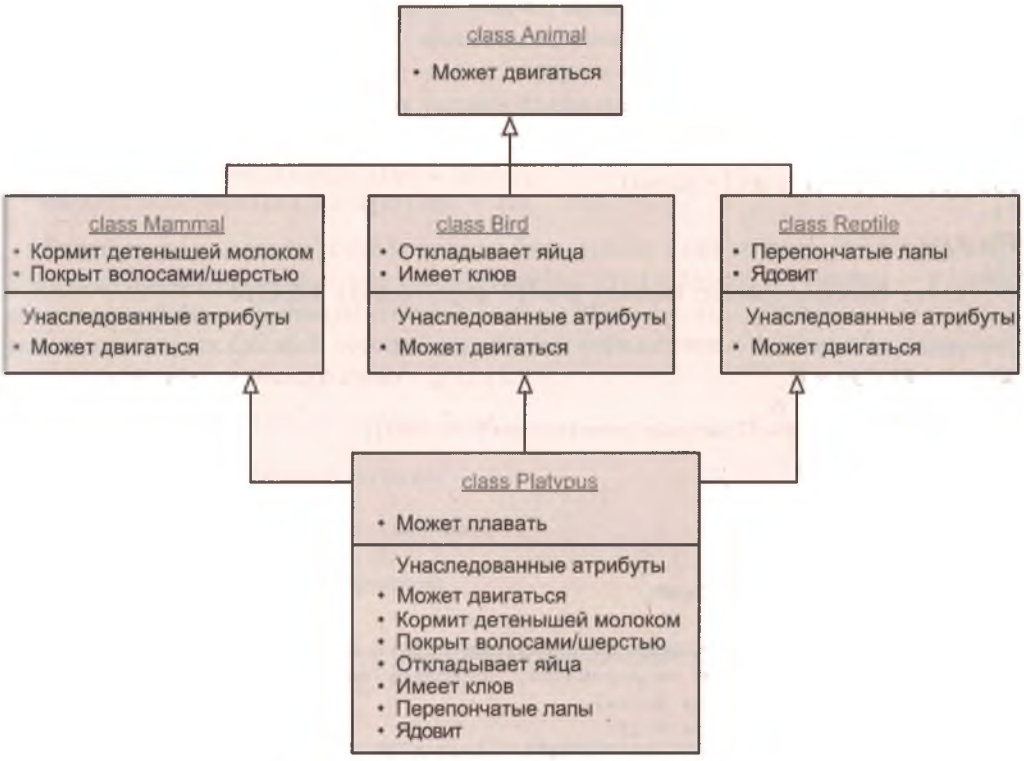


РИС. 11.2. Схема класса утконоса, демонстрирующего множественное наследование

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.

ЛИСТИНГ 11.7. Проверка количества экземпляров базового класса `Animal` в одном экземпляре класса `Platypus`

```
1: #include <iostream>
2: using namespace std;
3:
4: class Animal
```

```
 4: {
 5: public:
 6:     Animal()
 7:     {
 8:         cout << "Animal constructor" << 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 << "Platypus constructor" << endl;
33:     }
34: };
35:
36: int main()
37: {
38:     Platypus duckBilledP;
39:
40:     // Снимите комментирий со следующей строки и получите отказ
41:     // компияции. Age неоднозначен, поскольку есть три экземпляра
       // базового класса Animal
42:     // duckBilledP.Age = 25;
43:
44:     return 0;
45: }
```

Результат

```
Animal constructor
Animal constructor
Animal constructor
Platypus constructor
```

Анализ

Как демонстрирует вывод, благодаря множественному наследованию у всех трех базовых классов класса Platypus (происходящих, 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. 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, или Bird::Animal::Age, или Reptile::Animal::Age. Как ни смешно, но при желании вы можете установить значения для всех трех:

```
duckBilledP.Mammal::Animal::Age = 25;
duckBilledP.Bird::Animal::Age = 25;
duckBilledP.Reptile::Animal::Age = 25;
```

Безусловно, у одного утконоса должен быть только один возраст. Но все же класс Platypus должен происходить от классов Mammal, Bird и Reptile. Решение — в *виртуальном наследовании* (virtual inheritance). Если вы ожидаете, что производный класс будет использоваться как базовый, 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 (фактически улучшенные классы Mammal, Bird и Reptile) приведен в листинге 11.8.

ЛИСТИНГ 11.8. Как ключевое слово virtual в иерархии наследования позволяет ограничить количество экземпляров базового класса Animal до одного

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Animal
 4: {
 5: public:
 6:     Animal()
 7:     {
 8:         cout << "Animal constructor" << 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:public Mammal, public Bird, public Reptile
28: {
29: public:
30:     Platypus()
31:     {
32:         cout << "Platypus constructor" << endl;
33:     }
34: };
35:
36: int main()
37: {
38:     Platypus duckBilledP;
39:
40:     // нет ошибки компиляции, поскольку есть только один Animal::Age
41:     duckBilledP.Age = 25;
42:
43:     return 0;
44: }
```

Результат

```
Animal constructor
Platypus constructor
```

Анализ

Сравнив вывод с выводом листинга 11.7, можно сразу заметить, что количество экземпляров класса `Animal` уменьшилось до одного, что, наконец, отражает тот факт, что создан был только один утконос. Все дело в ключевом слове `virtual`, использованном в отношениях между классами `Mammal`, `Bird` и `Reptile`, 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.

ПРИМЕЧАНИЕ

Проблема иерархии наследования, содержащей два или больше базовых класса, которые происходят от одного общего базового класса, приводит к необходимости разрешения неоднозначности при отсутствии виртуального наследования, называется *проблемой ромба* (diamond problem).

Название "ромб", вероятно, возникло благодаря форме схемы классов (см. рис. 11.2), где 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 хотел сэкономить время на создании ключевого слова `apt`.) Вот краткое резюме.

Объявление функции *виртуальной* означает, что будет вызвана ее переопределенная версия, существующая в производном классе.

Отношения наследования, объявленные с использованием ключевого слова `virtual`, между классами `Derived1` и `Derived2`, происходящими от класса `Base`, означают, что экземпляр следующего класса, `Derived3`, происходящего от классов `Derived1` и `Derived2`, будет содержать только один экземпляр класса `Base`.

Таким образом, то же ключевое слово `virtual` используется для реализации двух разных концепций.

Виртуальные конструкторы копий?

Обратите внимание на вопросительный знак в конце заголовка данного раздела. Технически в языке C++ невозможно получить виртуальные конструкторы копий. Но все же можно создать коллекцию (например, статический массив) типа `Base*`, каждый элемент которого является специализацией этого типа:

```
// Классы Tuna, Carp и Trout открыто происходят от базового класса Fish
Fish* pFishes[3];
Fishes[0] = new Tuna();
Fishes[1] = new Carp();
Fishes[2] = new Trout();
```

Теперь присвоим его другому массиву того же типа, где виртуальный конструктор копий обеспечит глубокое копирование объектов производного класса, а также то, что объекты классов `Tuna`, `Carp` и `Trout` будут скопированы как объекты классов `Tuna`, `Carp` и `Trout`, несмотря на то, что используется конструктор копий для типа `Fish*`.

Но это все мечты.

Виртуальные конструкторы копий не возможны, поскольку ключевое слово `virtual` в контексте методов базового класса, переопределяемых реализациями, доступными в производном классе, свидетельствует о полиморфном поведении во время выполнения. Конструкторы, напротив, не полиморфны по своей природе, так как способны создавать экземпляр только фиксированного типа, а следовательно, язык C++ не позволяет использовать виртуальные конструкторы копий.

С учетом сказанного выше появляется хороший повод определить собственную функцию клонирования, которая позволит сделать именно это:

```
class Fish
{
public:
    virtual Fish* Clone() const = 0; // чистая 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 этого
    }
};
```

Таким образом, виртуальная функция `Clone()` моделирует виртуальный конструктор копий, который должен быть вызван явно, как представлено в листинге 11.9.

ЛИСТИНГ 11.9. Классы `Tuna` и `Carp` с функцией `Clone()`, моделирующей виртуальный конструктор копий

```
 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: };
 9:
10: class Tuna: public Fish
11: {
12: public:
13:     Fish* Clone()
14:     {
15:         return new Tuna (*this);
16:     }
17:
18:     void Swim()
19:     {
20:         cout << "Tuna swims fast in the sea" << endl;
21:     }
22: };
23:
24: class Carp: public Fish
25: {
26:     Fish* Clone()
27:     {
28:         return new Carp(*this);
```

```
29:     }
30:     void Swim()
31:     {
32:         cout << "Carp swims slow in the lake" << endl;
33:     }
34: };
35:
36: int main()
37: {
38:     const int ARRAY_SIZE = 4;
39:
40:     Fish* myFishes[ARRAY_SIZE] = {NULL};
41:     myFishes[0] = new Tuna();
42:     myFishes[1] = new Carp();
43:     myFishes[2] = new Tuna();
44:     myFishes[3] = new Carp();
45:
46:     Fish* myNewFishes[ARRAY_SIZE];
47:     for (int Index = 0; Index < ARRAY_SIZE; ++Index)
48:         myNewFishes[Index] = myFishes[Index]->Clone();
49:
50:     // вызов виртуального метода для проверки
51:     for (int Index = 0; Index < ARRAY_SIZE; ++Index)
52:         myNewFishes[Index]->Swim();
53:
54:     // очистка памяти
55:     for (int Index = 0; Index < ARRAY_SIZE; ++Index)
56:     {
57:         delete myFishes[Index];
58:         delete myNewFishes[Index];
59:     }
60:
61:     return 0;
62: }
```

Результат

```
Tuna swims fast in the sea
Carp swims slow in the lake
Tuna swims fast in the sea
Carp swims slow in the lake
```

Анализ

Строки 40–44 в функции `main()` демонстрируют объявление статического массива указателей на базовый класс `Fish*` и индивидуальное присвоение его элементам вновь созданных объектов класса `Tuna`, `Carp`, `Tuna` 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 этого недостаточно — замечательно, вы можете копировать в новый массив `myNewFishes` типа `Fish*` при помощи вызова в цикле виртуальной функции

`Fish::Clone()`, как показано в строке 48. Обратите внимание, что массив очень мал: только четыре элемента. Он может быть много больше, хотя это и не будет иметь большого значения для логики копирования, а только потребует коррекции условия завершения цикла. Строка 52 фактически является проверкой, где вы вызываете виртуальную функцию `Fish::Swim()` для каждого хранимого в новом массиве элемента, чтобы проверить, скопировала ли функция `Clone()` объект класса `Tuna` как `Tuna`, а не только как `Fish`. Вывод демонстрирует, что все скопировано правильно.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Отмечайте виртуальными те функции базового класса, которые должны быть переопределены в производных классах	**Не забывайте** оснащать базовый класс виртуальным деструктором
Помните, что чистые виртуальные функции делают класс абстрактным, а сами эти функции должны быть реализованы в производном классе	**Не забывайте**, что компилятор не позволит создать экземпляр абстрактного класса
Учитывайте возможность использования виртуального наследования	**Не забывайте**, что виртуальное наследование гарантирует общий базовый класс от проблемы ромба и позволит создать только один его экземпляр
	Не путайте назначение ключевого слова `virtual` при использовании в создаваемой иерархии наследования с тем же словом в объявлении функций базового класса

Резюме

На этом занятии мы изучили мощь иерархий наследования в коде C++ при использовании полиморфизма. Вы научились объявлять и создавать виртуальные функции в базовом классе, а также узнали то, что они гарантируют их переопределение и реализацию в производном классе, даже если для вызова виртуального метода используется экземпляр базового класса. Было показано, что чистые виртуальные функции являются специальным типом виртуальных функций, гарантирующим невозможность создания экземпляра базового класса самого по себе, что делает их прекрасным местом для определения интерфейсов, которые должны реализовать производные классы. И наконец, вы ознакомились с проблемой ромба, вызванной множественным наследованием, и тем, как виртуальное наследование помогает решить ее.

Вопросы и ответы

- **Зачем использовать ключевое слово `virtual` в определении функции базового класса, когда код компилируется и без этого?**

  Без ключевого слова `virtual` вы не в состоянии гарантировать, что если некто вызовет функцию `objBase.Function()`, то она не будет переадресована функции `Derived::Function()`. В конце концов, компиляция кода — это не мера его качества.

- **Зачем компилятор составляет таблицу виртуальной функции?**
  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 сделает так:
  ```
  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 отдельно?**
  Абстрактный класс и не предназначен для создания автономных объектов; его задача быть унаследованным. Он содержит чистые виртуальные функции, определяющие набор функций, которые должны реализовать производные классы, выполняя таким образом роль интерфейса.
- **Должен ли я использовать в иерархии наследования ключевое слово `virtual` в объявлениях всех виртуальной функции или только в базовом классе?**
  Достаточно иметь только одно ключевое слово `virtual` в объявлении функции, и оно должно быть в базовом классе.
- **Могу ли я определить функции-члены и переменные-члены в абстрактном классе?**
  Конечно можете. Помните, что нельзя создать экземпляр абстрактного класса, поскольку у него есть по крайней мере одна чистая 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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "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 их классов обязательно реализовал функции `Area()` и `Print()`. Как это сделать?
2. Для всех ли классов компилятор составляет таблицу виртуальной функции?
3. У моего класса `Fish` есть два открытых метода, одна чистая виртуальная функция и несколько переменных-членов. Класс все еще остается абстрактным?

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. Каков порядок выполнения конструкторов и деструкторов в неисправленном коде упражнения 2, если экземпляр класса `Car` создается и удаляется так:

```
Vehicle* pMyRacer = new Car;
delete pMyRacer;
```

ЗАНЯТИЕ 12

Типы операторов и их перегрузка

Ключевое слово `class` позволяет инкапсулировать не только данные и методы, но и операторы, которые облегчают работу с объектами этого класса. Вы можете использовать эти операторы для выполнения таких операций, как присвоение или сложение объектов класса, как с целыми числами, которые мы рассмотрели на занятии 5, "Команды, выражения и операторы". Подобно функциям, операторы также могут быть перегружены.

На сегодняшнем занятии.

- Применение ключевого слова `operator`.
- Унарные и бинарные операторы.
- Операторы преобразования.
- Операторы присваивания при перемещении языка C++11.
- Операторы, которые не могут быть переопределены.

Что такое операторы C++

На синтаксическом уровне оператор от функции отличает очень немногое — лишь использование ключевого слова `operator`. Объявление оператора очень похоже на объявление функции:

```
тип_возвращаемого_значения operator символ_оператора (...список параметров...);
```

В данном случае *символ_оператора* может быть любым оператором, который захочет определить программист. Это может быть символ + (сложение) или && (логическое И) и т.д. Операнды помогают компилятору отличить один оператор от другого. Так почему же язык C++ предоставляет операторы, когда уже поддерживаются функции?

Рассмотрим вспомогательный класс `Date`, инкапсулирующий день, месяц и год:

```
Date Holiday (25, 12, 2011); // инициализация датой 25 декабря 2011 года
```

Если теперь понадобится сменить дату на следующий день, 26 декабря, то что из следующего оказалось бы удобней и интуитивно понятней?

- Возможность 1. Использовать оператор:

```
++ Holiday;
```

- Возможность 2. Использовать функцию `Date::Increment()`:

```
Holiday.Increment(); // 26 декабря 2011 года
```

Конечно, первая возможность понятней и проще, чем метод `Increment()`. Выражения на базе операторов легче в использовании и интуитивно понятней. Реализация оператора меньше (<) в классе `Date` позволила бы сравнить две даты так:

```
if(Date1 < Date2)
{
    // Сделать нечто
}
else
{
    // Сделать нечто другое
}
```

Операторы применяются вне классов, например, при работе с датами. Представьте оператор суммы (+), который обеспечивает простую конкатенацию строк во вспомогательном строковом классе, таком, как `MyString` (см. листинг 9.9):

```
MyString sayHello ("Hello ");
MyString sayWorld ("world");
MyString sumThem (sayHello + sayWorld); // использование оператора +
                                        // (недоступного в листинге 9.9)
```

ПРИМЕЧАНИЕ

Дополнительные усилия по реализации подходящих операторов окупается легкостью использования вашего класса.

В самом общем смысле операторы C++ можно подразделить на два типа: унарные и бинарные операторы.

Унарные операторы

Как и предполагает их название, *унарные операторы* (unary operator) работают с одиночным операндом. Вот типичное определение унарного оператора, реализованного как глобальная функция или статическая функция-член:

```
тип_возвращаемого_значения operator тип_оператора (тип_параметра)
{
    // ... реализация
}
Унарный оператор, являющийся членом класса, определяется так:
тип_возвращаемого_значения operator тип_оператора ()
{
    // ... 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
*	Обращение к значению указателя
->	Косвенное обращение к члену класса
!	Логическое NOT
&	Обращение к адресу
~	Дополнение
+	Унарная сумма
-	Унарное вычитание
Операторы преобразования	Операторы преобразования

Создание 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;
}
```

У постфиксного оператора инкремента (++), напротив, возвращаемое значение отличается от входного параметра (который используется не всегда):

```
Date operator ++ (int)
{
    // Сохранить копию текущего состояния объекта, прежде чем увеличить
    // значение дня
    Date Copy (*this);

    // код реализации оператора (осуществляющий инкремент этого объекта)

    // Возвратить состояние прежде, чем будет выполнен инкремент
    return Copy;
}
```

Синтаксис у префиксных и постфиксных операторов декремента такой же, как и у операторов инкремента, только объявление содержит -- там, где у инкремента ++. Листинг 12.1 демонстрирует простой класс Date, позволяющий увеличивать даты, используя оператор (++).

ЛИСТИНГ 12.1. Календарный класс, содержащий день, месяц и год, а также допускающий инкремент и декремент

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5:     private:
 6:     int Day; // Диапазон: 1 - 30 (Давайте предположим, что у всех
                 // месяцев по 30 дней)
 7:     int Month;
 8:     int Year;
 9:
10: public:
11:     // Конструктор, инициализирующий объект днем, месяцем и годом
12:     Date (int InputDay, int InputMonth, int InputYear)
13:         : Day (InputDay), Month (InputMonth), Year (InputYear) {};
14:
15:     // Унарный оператор инкремента (префиксный)
16:     Date& operator ++ ()
17:     {
18:         ++Day;
19:         return *this;
20:     }
21:
22:     // Унарный оператор декремента (префиксный)
23:     Date& operator -- ()
24:     {
25:         --Day;
26:         return *this;
27:     }
28:
29:     void DisplayDate ()
30:     {
31:         cout << Day << " / " << Month << " / " << Year << endl;
32:     }
```

```
33: };
34:
35: int main ()
36: {
37:     // Создать экземпляр и инициализировать его
        // датой 25 декабря 2011 года
38:     Date Holiday (25, 12, 2011);
39:
40:     cout << "The date object is initialized to: ";
41:     Holiday.DisplayDate ();
42:
43:     // Применение префиксного оператора
44:     ++ Holiday;
45:
46:     cout << "Date after prefix-increment is: ";
47:
48:     // Отображение даты после инкремента
49:     Holiday.DisplayDate ();
50:
51:     -- Holiday;
52:     -- Holiday;
53:
54:     cout << "Date after two prefix-decrements is: ";
55:     Holiday.DisplayDate ();
56:
57:     return 0;
58: }
```

Результат

```
The date object is initialized to: 25 / 12 / 2011
Date after prefix-increment is: 26 / 12 / 2011
Date after two prefix-decrements is: 24 / 12 / 2011
```

Анализ

Представляющие интерес операторы находятся в строках 16–27 и обеспечивают инкремент и декремент объектов класса `Date` при добавлении и вычитании дня, как показано в строках 44, 51 и 52 функции `main()`. Префиксные операторы инкремента сначала выполняют приращение, а затем возвращают ссылку на тот же объект.

ПРИМЕЧАНИЕ

> У этой версии класса даты минимальная реализация, практически пустая, чтобы сократить количество строк при объяснении реализации префиксных операторов (++) и (--). При этом я подразумевал, что месяц имеет 30 дней, и не реализовал функциональные возможности перевода месяца и года.

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)
{
    // Сохранить копию текущего состояния объекта, прежде чем
    // увеличить день
    Date Copy (Day, Month, Year);

    ++Day;

    // Возвратить состояние прежде, чем выполнить инкремент
    return Copy;
}
// постфиксный оператор декремента
Date operator -- (int)
{
    Date Copy (Day, Month, Year);

    --Day;

    return Copy;
}
```

Когда ваша версия класса `Date` будет поддерживать префиксные и постфиксные операторы инкремента и декремента, вы будете в состоянии использовать объекты класса следующим образом:

```
Date Holiday (25, 12, 2011); // создание экземпляра
++ Holiday; // использование префиксного оператора инкремента ++
Holiday ++; // использование постфиксного оператора инкремента ++
-- Holiday; // использование префиксного оператора декремента --
Holiday --; // использование постфиксного оператора декремента --
```

ПРИМЕЧАНИЕ

Как демонстрирует реализация постфиксных операторов, перед операцией инкремента или декремента создается копия, содержащая текущее состояние объекта; она и будет возвращена.

Другими словами, если вам нужен *только* инкремент, выбирайте 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: `binary '<<' :` не найден оператор, получающий правый операнд типа `'Date'` (или нет приемлемого преобразования)). По существу, это сообщение означает, что оператор `cout` не знает, как интерпретировать экземпляр класса `Date`, поскольку он не поддерживает подходящих операторов.

Однако оператор cout вполне может работать с константной строкой типа const char*:

```
std::cout << "Hello world"; // const char* работает!
```

Поэтому, чтобы оператор cout работал с объектом класса Date, достаточно добавить оператор, который возвращает его версию типа const char*:

```
operator const char*()
{
    // реализация оператора, возвращающая char*
}
```

Листинг 12.2 демонстрирует пример реализации этого оператора.

ЛИСТИНГ 12.2. Реализация оператора преобразования в const char* для класса Date

```
 0: #include <iostream>
 1: #include <sstream>
 2: #include <string>
 3: using namespace std;
 4:
 5: class Date
 6: {
 7: private:
 8:     int Day; // Диапазон: 1 - 30 (Давайте предположим, что у всех
                 // месяцев по 30 дней)
 9:     int Month;
10:     int Year;
11:
12:     string DateInString;
13:
14: public:
15:
16:     // Конструктор, инициализирующий объект днем, месяцем и годом
17:     Date (int InputDay, int InputMonth, int InputYear)
18:         : Day (InputDay), Month (InputMonth), Year (InputYear) {};
19:
20:     operator const char*()
21:     {
22:         ostringstream formattedDate;
23:         formattedDate << Day << " / " << Month << " / " << Year;
24:
25:         DateInString = formattedDate.str();
26:         return DateInString.c_str();
27:     }
28: };
29:
30: int main ()
31: {
32:     // Создать экземпляр и инициализировать его
       // датой 25 декабря 2011 года
33:     Date Holiday (25, 12, 2011);
34:
35:     cout << "Holiday is on: " << Holiday << endl;
36:
37:     return 0;
38: }
```

Результат

```
Holiday is on: 25 / 12 / 2011
```

Анализ

Преимущество реализации оператора `const char*` (строки 20–27) проявляется в строке 35 функции `main()`. Теперь экземпляр класса `Date` может непосредственно использоваться в операторе `cout` благодаря тому факту, что он понимает тип `const char*`. Компилятор автоматически использует вывод подходящего (а в данном случае единственно доступного) оператора и передает его оператору `cout`, который отображает дату на экране. В нашей реализации оператора `const char*` использован оператор `std::ostringstream`, преобразующий целые числа члена класса в объект `std::string` (строки 23 и 25). Можно было бы сразу вернуть результат метода `formattedDate.str()`, но мы сохраняем его копию в закрытом члене `Date::DateInString` (строка 25), поскольку переменная `formattedDate` является локальной, она удаляется при выходе из оператора. Так, указатель, полученный от метода `str()`, после выхода будет недействителен.

Этот оператор открывает новые возможности использования класса `Date`. Теперь дату можно даже присвоить непосредственно строке:

```
string strHoliday (Holiday);      // OK! Компилятор вызывает
                                  // оператор const char*
strHoliday = Date(11, 11, 2011); // Также OK!
```

ПРИМЕЧАНИЕ

Создайте столько операторов, сколько может понадобиться классу при использовании. Если бы ваше приложение нуждалось в целочисленном представлении даты, то пригодился бы такой оператор:

```
operator int()
{
    // здесь код преобразования
}
```

Это позволило бы использовать экземпляр класса `Date` как целое число:

```
SomeFuncThatTakesInt(Date(25, 12, 2011));
```

Создание оператора обращения к значению (*) и оператора обращения к члену класса (->)

Оператор обращения к значению (*) и оператор обращения к члену класса (->) чаще всего используются при создании классов интеллектуального указателя. *Интеллектуальные указатели* (smart pointer) — это вспомогательные классы, являющиеся оболочками обычных указателей и облегчающие управление памятью (или ресурсом), решая проблемы собственности и копирования. В некоторых случаях они способны даже повысить производительность приложения. Подробно интеллектуальные указатели обсуждаются на занятии 26, "Понятие интеллектуальных указателей", а на этом занятии рассматривается лишь то, 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` для управления динамически распределяемой памятью экземпляра класса `Date`

```
 0: #include <iostream>
 1: #include <memory> // включите это, чтобы использовать std::unique_ptr
 2: using namespace std;
 3:
 4: class Date
 5: {
 6: private:
 7:     int Day;
 8:     int Month;
 9:     int Year;
10:
11:     string DateInString;
12:
13: public:
14:     // Конструктор, инициализирующий объект днем, месяцем и годом
15:     Date (int InputDay, int InputMonth, int InputYear)
16:         : Day (InputDay), Month (InputMonth), Year (InputYear) {};
17:
18:     void DisplayDate()
19:     {
20:         cout << Day << " / " << Month << " / " << Year << endl;
21:     }
22: };
23:
24: int main()
25: {
26:     unique_ptr<int> pDynamicAllocInteger(new int);
27:     *pDynamicAllocInteger = 42;
28:
29:     // Использование интеллектуального указателя как типа int*
30:     cout << "Integer value is: " << *pDynamicAllocInteger << endl;
31:
32:     unique_ptr<Date> pHoliday (new Date(25, 11, 2011));
33:     cout << "The new instance of date contains: ";
34:
35:     // Использование pHoliday точно как Date*
36:     pHoliday->DisplayDate();
37:
38:     // При использовании unique_ptr в следующем нет необходимости:
39:     // delete pDynamicAllocInteger;
40:     // delete pHoliday;
41:
42:     return 0;
43: }
```

Результат

```
Integer value is: 42
The new instance of date contains: 25 / 11 / 2011
```

Анализ

В строке 26 объявляется интеллектуальный указатель типа `int`. Эта строка демонстрирует синтаксис инициализации шаблона для класса интеллектуального указателя `unique_ptr`. Точно так же в строке 32 объявляется интеллектуальный указатель на экземпляр класса `Date`. Пока сосредоточьтесь на шаблоне и игнорируйте детали.

ПРИМЕЧАНИЕ

> Не волнуйтесь, если синтаксис шаблона кажется пока непонятным, поскольку шаблоны рассматриваются позже, на занятии 14, "Макросы и шаблоны".

Этот пример демонстрирует не только то, как интеллектуальный указатель позволяет использовать обычный синтаксис указателя (строки 30 и 36). В строке 30 вы в состоянии отобразить значение типа `int`, используя синтаксис `*pDynamicAllocInteger`, тогда как в строке 36 вы используете вызов `pHoliday->DisplayData()`, как будто этими двумя переменными были `int*` и `Date*` соответственно. Секрет — в классе интеллектуального указателя `std::unique_ptr`, реализующего операторы (*) и (->). Листинг 12.4 является реализацией простого класса интеллектуального указателя.

ЛИСТИНГ 12.4. Реализация операторов (*) и (->) в простом классе интеллектуального указателя

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: template <typename T>
 4: class smart_pointer
 5: {
 6: private:
 7:     T* m_pRawPointer;
 8: public:
 9:     smart_pointer (T* pData) : m_pRawPointer (pData) {}
                                                  // Конструктор
10:     ~smart_pointer () {delete m_pRawPointer ;} // Деструктор
11:
12:     T& operator* () const // оператор обращения к значению
13:     {
14:         return *(m_pRawPointer);
15:     }
16:
17:     T* operator-> () const // оператор обращения к члену класса
18:     {
19:         return m_pRawPointer;
20:     }
21: };
22:
23: class Date
24: {
```

```
15: private:
16:     int Day, Month, Year;
17:     string DateInString;
18:
19: public:
20:     // Конструктор, инициализирующий объект днем, месяцем и годом
21:     Date (int InputDay, int InputMonth, int InputYear)
22:         : Day (InputDay), Month (InputMonth), Year (InputYear) {};
23:
24:     void DisplayDate()
25:     {
26:         cout << Day << " / " << Month << " / " << Year << endl;
27:     }
28: };
29:
30: int main()
31: {
32:     smart_pointer<int> pDynamicInt(new int (42));
33:     cout << "Dynamically allocated integer value = " << *pDynamicInt;
34:
35:     smart_pointer<Date> pDate(new Date(25, 12, 2011));
36:     cout << "Date is = ";
37:     pDate->DisplayDate();
38:
39:     return 0;
40: }
```

Результат

```
Dynamically allocated integer value = 42
Date is = 25 / 12 / 2011
```

Анализ

Это версия листинга 12.3, в которой использован собственный класс `smart_pointer`, определенный в строках 3–24. Поскольку используется синтаксис объявления шаблона, интеллектуальный указатель можно настроить так, чтобы он указывал на любой тип, будь то `int`, как показано в строке 45, или `Date`, как показано в строке 48. Наш класс интеллектуального указателя содержит закрытый член типа, на который он указывает, объявленный на строке 7. По существу, класс интеллектуального указателя стремится автоматизировать управление ресурсом, на который указывает этот член класса, включая его автоматическое освобождение в деструкторе, как показано в строке 10. Этот деструктор гарантирует, что, даже после вызова оператора `new`, вы не обязаны вызывать оператор `delete` для освобождения ресурса памяти вручную, и это не приводит к утечке памяти. Сосредоточьтесь на реализации оператора `*` в строках 12–15, который возвращает тип `T&` (т.е. ссылку на тип, для которого этот шаблон специализируется). Реализация возвращает ссылку на экземпляр (строка 14). Аналогично оператор `->`, как показано в строках 17–20, возвращает тип `T*` (т.е. указатель типа, для которого этот шаблон специализируется). Реализация оператора `->` в строке 19 возвращает указатель на член класса. Вместе эти два оператора гарантируют, что класс `smart_pointer` абстрагирует управление памятью простого указателя и позволяет использовать функциональные возможности обычного указателя, делая его таким образом *интеллектуальным* указателем.

ПРИМЕЧАНИЕ

Классы интеллектуального указателя способны на много большее, чем имитация обычных указателей или освобождение памяти, когда они выходят из области видимости. Более подробная информация по этой теме приведена на занятии 26, "Понятие интеллектуальных указателей".

Если применение указателя `unique_ptr` в листинге 12.3 заинтересовало вас, то, чтобы понять его внутреннее устройство, найдите его реализацию в файле заголовка `<memory>`, предоставляемом вашим компилятором или интегрированной средой разработки.

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:

```
тип_возвращаемого_значения тип_оператора (параметр1, параметр2);
```

Определение бинарного оператора, реализованного как член класса, имеет следующий вид:

```
тип_возвращаемого_значения тип_оператора (параметр);
```

Бинарный оператор получает только один параметр в версии члена класса, потому что второй параметр обычно является атрибутом самого 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 присвоением
&	Побитовое AND
&&	Логическое AND
&=	Побитовое AND с присвоением
*	Умножение
*=	Умножение с присвоением
+	Сложение
+=	Сложение с присвоением
-	Вычитание
-=	Вычитание с присвоением
->*	Косвенное обращение к указателю на член класса

Окончание табл. 12.2

Оператор	Название
/	Деление
/=	Деление с присвоением
<	Меньше
<<	Сдвиг влево
<<=	Сдвиг влево с присвоением
<=	Меньше или равно
=	Присвоение, присвоение копии и присвоение перемещения
==	Равенство
>	Больше
>=	Больше или равно
>>	Сдвиг вправо
>>=	Сдвиг вправо с присвоением
^	Исключающее OR
^=	Исключающее OR с присвоением
|	Побитовое OR
|=	Побитовое OR с присвоением
||	Логическое OR
[]	Оператор индексирования

Создание бинарных операторов сложения (a + b) и вычитания (a - b)

Подобно операторам инкремента и декремента, бинарные операторы "плюс" и "минус", будучи определены, позволяют добавлять 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 еще не поддерживает возможность перевода, скажем, на пять дней вперед. Для этого необходимо реализовать бинарный оператор (+), как демонстрирует код листинга 12.5.

ЛИСТИНГ 12.5. 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:
 8: public:
 9:
10:     // Конструктор, инициализирующий объект днем, месяцем и годом
```

```
11:     Date (int InputDay, int InputMonth, int InputYear)
12:         : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:     // Бинарный оператор суммы
15:     Date operator + (int DaysToAdd)
16:     {
17:         Date newDate (Day + DaysToAdd, Month, Year);
18:         return newDate;
19:     }
20:
21:     // Бинарный оператор вычитания
22:     Date operator - (int DaysToSub)
23:     {
24:         return Date(Day - DaysToSub, Month, Year);
25:     }
26:
27:     void DisplayDate ()
28:     {
29:         cout << Day << " / " << Month << " / " << Year << endl;
30:     }
31: };
32:
33: int main()
34: {
35:     // Создать экземпляр и инициализировать его
        // датой 25 декабря 2011 года
36:     Date Holiday (25, 12, 2011);
37:
38:     cout << "Holiday on: ";
39:     Holiday.DisplayDate ();
40:
41:     Date PreviousHoliday (Holiday - 19);
42:     cout << "Previous holiday on: ";
43:     PreviousHoliday.DisplayDate();
44:
45:     Date NextHoliday(Holiday + 6);
46:     cout << "Next holiday on: ";
47:     NextHoliday.DisplayDate ();
48:
49:     return 0;
50: }
```

Результат

```
Holiday on: 25 / 12 / 2011
Previous holiday on: 6 / 12 / 2011
Next holiday on: 31 / 12 / 2011
```

Анализ

Строки 14–25 содержат реализации бинарных операторов (+) и (–), которые позволяют использовать синтаксис простого сложения и вычитания, как можно заметить в строках 41 и 45 функции `main()`.

Бинарный оператор суммы также был бы очень полезен в случае создания строкового класса. На занятии 9, "Классы и объекты", мы уже анализировали простой класс оболочки строки `MyString`, инкапсулирующий управление памятью, копирование и так далее для символьной строки стиля C (см. листинг 9.9). Но что не поддерживает этот класс, так это конкатенацию двух строк с использованием следующего синтаксиса:

```
MyString Hello("Hello ");
MyString World(" World");
MyString HelloWorld(Hello + World); // ошибка: оператор + не определен
```

Само собой разумеется, оператор (+) чрезвычайно упростил бы использование класса `MyString`, а следовательно, он стоит потраченных на него усилий:

```
MyString operator+ (const MyString& AddThis)
{
    MyString NewString;
    if (AddThis.Buffer != NULL)
    {
        NewString.Buffer = new char[GetLength() \
                        + strlen(AddThis.Buffer) + 1];
        strcpy(NewString.Buffer, Buffer);
        strcat(NewString.Buffer, AddThis.Buffer);
    }

    return NewString;
}
```

Чтобы получить возможность использовать синтаксис сложения, добавьте приведенный выше код в листинг 9.9 с закрытым стандартным конструктором `MyString()` и пустой реализацией. Вы можете увидеть версию класса `MyString` с оператором (+) среди прочих в листинге 12.12.

ПРИМЕЧАНИЕ

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 мало вероятно, такой оператор не нашел бы применения.

Реализация операторов сложения с присвоением (+=) и вычитания с присвоением (-=)

Операторы сложения с присвоением обеспечивают такой синтаксис, как "`a += b;`", позволяющий программисту увеличивать значение объекта `a` на значение `b`. Преимущество оператора сложения с присвоением в том, что он может быть перегружен так, чтобы получать параметры `b` различных типов. Приведенный ниже листинг 12.6 позволяет добавлять целочисленное значение к объекту `CDate`.

ЛИСТИНГ 12.6. Определение операторов (+=) и (-=) для добавления и вычитания введенных дней

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class Date
 4: {
 5:     private:
 6:     int Day, Month, Year;
 7:
 8: public:
 9:
10:     // Конструктор, инициализирующий объект днем, месяцем и годом
11:     Date (int InputDay, int InputMonth, int InputYear)
12:         : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:     // Бинарный оператор сложения с присвоением
15:     void operator+= (int DaysToAdd)
16:     {
17:         Day += DaysToAdd;
18:     }
19:
20:     // Binary subtraction assignment
21:     void operator-= (int DaysToSub)
22:     {
23:         Day -= DaysToSub;
24:     }
25:
26:     void DisplayDate ()
27:     {
28:         cout << Day << " / " << Month << " / " << Year << endl;
29:     }
30: };
31:
32: int main()
33: {
34:     // Создать экземпляр и инициализировать его
        // датой 25 декабря 2011 года
35:     Date Holiday (25, 12, 2011);
36:
37:     cout << "Holiday is on: ";
38:     Holiday.DisplayDate ();
39:
40:     cout << "Holiday -= 19 gives: ";
41:     Holiday -= 19;
42:     Holiday.DisplayDate();
43:
44:     cout << "Holiday += 25 gives: ";
45:     Holiday += 25;
46:     Holiday.DisplayDate ();
47:
48:     return 0;
49: }
```

Результат

```
Holiday is on: 25 / 12 / 2011
Holiday -= 19 gives: 6 / 12 / 2011
Holiday += 25 gives: 31 / 12 / 2011
```

Анализ

Представляют интерес операторы сложения и вычитания с присвоением, находящиеся в строках 14–24. Они обеспечивают добавление и вычитание целочисленных значений к количеству дней в функции `main()`, например, так:

```
41:     Holiday -= 19;
45:     Holiday += 25;
```

Теперь класс `Date` позволяет пользователям добавлять и вычитать дни, как будто это целые числа, используя операторы сложения и вычитания с присвоением, получающие параметр типа `int`. Вы можете даже предоставить перегруженную версию оператора сложения с присвоением (+=), получающую экземпляр некого класса `CDays`:

```
// Оператор сложения с присвоением, добавляющий CDays к существующей дате
void operator += (const CDays& mDaysToAdd)
{
    Day += mDaysToAdd.GetDays ();
}
```

ПРИМЕЧАНИЕ

Синтаксис операторов умножения с присвоением *=, деления с присвоением /=, деления по модулю с присвоением %=, вычитания с присвоением -=, сдвига влево с присвоением <<=, сдвига вправо с присвоением >>=, XOR с приовоением ^=, побитовое OR с присвоением |= и побитовое AND с присвоением &= подобен синтаксису оператора сложения с присвоением, показанному в листинге 12.6.

Хотя конечная цель перегрузки операторов – сделать класс простым и интуитивно понятным в использовании, есть множество ситуаций, когда реализация оператора могла бы не иметь смысла. Например, у нашего календарного класса `Date` нет абсолютно никакого смысла в использовании оператора побитового AND с присвоением &=. Никакому пользователю этого класса никогда не понадобится (даже трудно придумать зачем) результат таких, например, операций, как `greatDay &= 20;`.

Перегрузка операторов равенства (==) и неравенства (!=)

Если пользователю нужно сравнить один объект класса `Date` с другим, он ожидает возможности применить следующий синтаксис:

```
if (Date1 == Date2)
{
    // Сделать нечто
}
else
{
    // Сделать нечто другое
}
```

В отсутствии оператора равенства компилятор просто выполнит побитовое сравнение двух этих объектов и возвратит значение true, если они абсолютно идентичны. Это могло бы сработать в некоторых случаях (включая класс Date в нынешнем состоянии), но он, вероятней всего, не сработает, как вы ожидали, если у рассматриваемого класса есть нестатический строковый член, содержащий такое строковое значение (char*), как MyString, в листинге 9.9. В таком случае побитовое сравнение атрибутов класса фактически сравнит строковые указатели, которые не равны (даже если строки имеют идентичное содержимое), и всегда будет возвращать значение false.

Таким образом, имеет смысл определить операторы сравнения. В общем виде выражение оператора равенства имеет следующий синтаксис:

```
bool operator== (const ТипКласса& сравнитьС)
{
    // здесь код сравнения, возвращающий true при равенстве и false
    // в противном случае
}
```

Оператор неравенства может повторно использовать оператор равенства:

```
bool operator!= (const ТипКласса& сравнитьС)
{
    // здесь код сравнения, возвращающий true при неравенстве и false
    // в противном случае
}
```

Оператор неравенства может быть инверсией (логическое NOT) результата оператора равенства. Листинг 12.7 демонстрирует операторы сравнения, определенные в классе Date.

ЛИСТИНГ 12.7. 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:
10:     // Конструктор, инициализирующий объект днем, месяцем и годом
11:     Date (int InputDay, int InputMonth, int InputYear)
12:         : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:     bool operator== (const Date& compareTo)
15:     {
16:         return ((Day == compareTo.Day)
17:             && (Month == compareTo.Month)
18:             && (Year == compareTo.Year));
19:     }
20:
21:     bool operator!= (const Date& compareTo)
22:     {
```

```
23:         return !(this->operator==(compareTo));
24:     }
25:
26:     void DisplayDate ()
27:     {
28:         cout << Day << " / " << Month << " / " << Year << endl;
29:     }
30: };
31:
32: int main()
33: {
34:     Date Holiday1 (25, 12, 2011);
35:     Date Holiday2 (31, 12, 2011);
36:
37:     cout << "Holiday 1 is: ";
38:     Holiday1.DisplayDate();
39:     cout << "Holiday 2 is: ";
40:     Holiday2.DisplayDate();
41:
42:     if (Holiday1 == Holiday2)
43:         cout << "Equality operator: The two are on the same day"
                 << endl;
44:     else
45:         cout << "Equality operator: The two are on different days"
                 << endl;
46:
47:     if (Holiday1 != Holiday2)
48:       . cout << "Inequality operator: The two are on different days"
                 << endl;
49:     else
50:         cout << "Inequality operator: The two are on the same day"
                 << endl;
51:
52:     return 0;
53: }
```

Результат

```
Holiday 1 is: 25 / 12 / 2011
Holiday 2 is: 31 / 12 / 2011
Equality operator: The two are on different days
Inequality operator: The two are on different days
```

Анализ

Оператор равенства (==) является простой реализацией, которая возвращает истину, если день, месяц и год равны, как показано в строках 14–19. Оператор неравенства (!=) — это просто повторное использование кода оператора равенства, как представлено в строке 23. Присутствие этих операторов позволяет сравнить два объекта (`Holiday1` и `Holiday2`) класса `Date` в функции `main()` (строки 42 и 47).

Перегрузка операторов <, >, <= и >=

Код листинга 12.7 сделал класс Date достаточно интеллектуальным, чтобы быть в состоянии сказать, равны ли два объекта класса Date или нет. Но что если пользователю класса нужно выполнить такую проверку условия, как эта:

```
if (Date1 < Date2) {// Сделать нечто}
```

или эта:

```
if (Date1 <= Date2) {// Сделать нечто}
```

или эта:

```
if (Date1 > Date2) {// Сделать нечто}
```

или эта:

```
if (greatDay >= Date2) {// Сделать нечто}
```

Пользователь календарного класса определенно нашел бы очень полезным, если бы мог просто сравнить две даты, чтобы узнать, предшествует ли этот день настоящему или следует за ним. Разработчик класса должен реализовать эти операторы сравнения, чтобы сделать использование своего класса столь же дружественным к пользователю и интуитивно понятным, насколько возможно, как демонстрирует код листинга 12.8.

ЛИСТИНГ 12.8. 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:
10:     // Конструктор, инициализирующий объект днем, месяцем и годом
11:     Date (int InputDay, int InputMonth, int InputYear)
12:         : Day (InputDay), Month (InputMonth), Year (InputYear) {};
13:
14:     bool operator== (const Date& compareTo)
15:     {
16:         return ((Day == compareTo.Day)
17:             && (Month == compareTo.Month)
18:             && (Year == compareTo.Year));
19:     }
20:
21:     bool operator< (const Date& compareTo)
22:     {
23:         if (Year < compareTo.Year)
24:             return true;
25:         else if (Month < compareTo.Month)
26:             return true;
27:         else if (Day < compareTo.Day)
```

```
28:             return true;
29:         else
30:             return false;
31:     }
32:
33:     bool operator<= (const Date& compareTo)
34:     {
35:         if (this->operator== (compareTo))
36:             return true;
37:         else
38:             return this->operator< (compareTo);
39:     }
40:
41:     bool operator > (const Date& compareTo)
42:     {
43:         return !(this->operator<= (compareTo));
44:     }
45:
46:     bool operator>= (const Date& compareTo)
47:     {
48:         if(this->operator== (compareTo))
49:             return true;
50:         else
51:             return this->operator> (compareTo);
52:     }
53:
54:     bool operator!= (const Date& compareTo)
55:     {
56:         return !(this->operator==(compareTo));
57:     }
58:
59:     void DisplayDate ()
60:     {
61:         cout << Day << " / " << Month << " / " << Year << endl;
62:     }
63: };
64:
65: int main()
66: {
67:     Date Holiday1 (25, 12, 2011);
68:     Date Holiday2 (31, 12, 2011);
69:
70:     cout << "Holiday 1 is: ";
71:     Holiday1.DisplayDate();
72:     cout << "Holiday 2 is: ";
73:     Holiday2.DisplayDate();
74:
75:     if (Holiday1 < Holiday2)
76:     cout << "operator<: Holiday1 happens first" << endl;
77:
78:     if (Holiday2 > Holiday1)
79:     cout << "operator>: Holiday2 happens later" << endl;
80:
81:     if (Holiday1 <= Holiday2)
```

```
82:     cout << "operator<=: Holiday1 happens on or before Holiday2"
             << endl;
83:
84:     if (Holiday2 >= Holiday1)
85:     cout << "operator>=: Holiday2 happens on or after Holiday1"
             << endl;
86:
87:     return 0;
88: }
```

Результат

```
Holiday 1 is: 25 / 12 / 2011
Holiday 2 is: 31 / 12 / 2011
operator<: Holiday1 happens first
operator>: Holiday2 happens later
operator<=: Holiday1 happens on or before Holiday2
operator>=: Holiday2 happens on or after Holiday1
```

Анализ

Представляющие интерес операторы реализованы в строках 21–52 и частично повторно используют оператор == из листинга 12.7. Обратите внимание, как эти операторы были реализованы: в основном за счет повторного использования одного или другого.

Применение этих операторов в строках 75–84 функции `main()` демонстрирует, насколько реализация этих операторов делает использование класса `Date` простым и интуитивно понятным.

Перегрузка оператора присвоения копии (=)

Нередко содержимое экземпляра класса необходимо присвоить другому экземпляру так:

```
Date Holiday(25, 12, 2011);
Date AnotherHoliday(1, 1, 2010);
AnotherHoliday = Holiday; // использование оператора присвоения копий
```

Это приведет к вызову стандартного оператора присвоения копий, который компилятор встроит в ваш класс, если вы не предоставите таковой. В зависимости от характера вашего класса стандартный конструктор копий может оказаться неадекватным, особенно если ваш класс задействует ресурс, который не будет скопирован. Чтобы гарантировать более глубокое копирование, как с конструктором копий, необходимо определить собственный оператор присвоения копий:

```
ClassType& operator= (const ClassType& CopySource)
{
    if(this != &copySource) // защита от копирования в себя самого
    {
        // Реализация оператора присвоения
    }
    return *this;
}
```

Глубокое копирование важно, если ваш класс инкапсулирует простой указатель, такой, как у класса `MyString`, представленного в листинге 9.9. В отсутствии оператора присвоения стандартный оператор присвоения копий, предоставляемый компилятором, просто копирует адрес, содержавшийся в указателе `char* Buffer`, из оригинала в копию, без глубокого копирования области памяти, на которую он указывает. Это тот же случай, что и при отсутствии конструктора копий. Чтобы гарантировать глубокое копирование во время присвоения, определите оператор присвоения копии, как показано в листинге 12.9.

ЛИСТИНГ 12.9. Улучшенный класс `MyString` из листинга 9.9 с оператором присвоения копии

```
#include <iostream>
using namespace std;

class MyString
{
private:
    char* Buffer;

public:
    // Конструктор
    MyString(const char* InitialInput)
    {
        if(InitialInput != NULL)
        {
            Buffer = new char [strlen(InitialInput) + 1];
            strcpy(Buffer, InitialInput);
        }
        else
            Buffer = NULL;
    }

    // вставить конструктор копий из листинга 9.9
    MyString(const MyString& CopySource);

    // Оператор присвоения копии
    MyString& operator= (const MyString& CopySource)
    {
        if ((this != &CopySource) && (CopySource.Buffer != NULL))
        {
            if (Buffer != NULL)
                delete[] Buffer;

            // гарантирует глубокую копию с предварительным
            // резервированием собственного буфера
            Buffer = new char [strlen(CopySource.Buffer) + 1];

            // копирование оригинала в локальный буфер
            strcpy(Buffer, CopySource.Buffer);
        }
        return *this;
    }

```

```
41:     // Деструктор
42:     ~MyString()
43:     {
44:         if (Buffer != NULL)
45:             delete [] Buffer;
46:     }
47:
48:     int GetLength()
49:     {
50:         return strlen(Buffer);
51:     }
52:
53:     operator const char*()
54:     {
55:         return Buffer;
56:     }
57: };
58:
59: int main()
60: {
61:     MyString String1("Hello ");
62:     MyString String2(" World");
63:
64:     cout << "Before assignment: " << endl;
65:     cout << String1 << String2 << endl;
66:     String2 = String1;
67:     cout << "After assignment String2 = String1: " << endl;
68:     cout << String1 << String2 << endl;
69:
70:     return 0;
71: }
```

Результат

```
Before assignment:
Hello World
After assignment String2 = String1:
Hello Hello
```

Анализ

Я преднамеренно пропустил конструктор копий в этом примере, чтобы сократить объем кода (но при создании подобного класса обязательно добавьте его; см. листинг 9.9). Оператор присвоения копии реализован в строках 25–39. Это очень похоже на конструктор копий, но с предварительной проверкой, гарантирующей, что оригинал и копия не являются тем же объектом. После успешной проверки оператор присвоения копии для класса `MyString` освобождает сначала свой внутренний буфер, затем повторно резервирует место для текста копии, а потом использует функцию `strcpy()` для копирования, как показано в строке 36.

ПРИМЕЧАНИЕ

Еще одно незначительное различие между листингами 12.9 и 9.9 в том, что функция `GetString()` заменена оператором `const char*`, как демонстрируют строки 53-56. Этот оператор облегчает использование класса `MyString`, как показано в строке 68, где один оператор `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 т.д., всегда следует реализовать (или рассмотреть необходимость реализовать) конструктор копий и оператор присвоения копии в дополнение к конструктору и деструктору.

Если вы не решаете проблему собственности ресурса явно, когда объект вашего класса копируется, ваш класс неполон и даже опасен для использования.

СОВЕТ

Чтобы создать класс, который не может быть скопирован, объявите конструктор копий и оператор присвоения копии как закрытый. Объявления (даже не реализации) вполне достаточно для компилятора, чтобы передать сообщение об ошибке при любых попытках копирования этого класса при передаче в функцию по значению или при присвоении одного экземпляра другому.

Оператор индексирования ([])

Оператор `[]`, позволяющий обращаться к классу в стиле массива, называется *оператором индексирования* (subscript operator). Типичный синтаксис оператора индексирования таков:

```
тип_возвращаемого_значения& operator [] (тип_индекса& индекс);
```

Так, при создании такого класса, как `MyString`, инкапсулирующего класс динамического массива символов `char* Buffer`, оператор индексирования существенно облегчит произвольный доступ к отдельным символам в буфере:

```
class MyString
{
    // ... другие члены класса
public:
    /*const*/ char& operator [] (int Index) /*const*/
    {
        // возвратить из буфера символ по позиции индекса
    }
};
```

Пример в листинге 12.10 демонстрирует, как оператор индексирования (`[]`) позволяет пользователю перебирать символы, содержавшиеся в экземпляре класса `MyString`, с использованием обычной семантики массива.

ЛИСТИНГ 12.10. Реализация оператора индексирования ([]) в классе `MyString`, обеспечивающего произвольный доступ к символам в буфере `MyString::Buffer`

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class MyString
 5: {
 6: private:
 7:     char* Buffer;
 8:
 9:     // закрытый стандартный конструктор
10:     MyString() {}
11:
12: public:
13:     // Конструктор
14:     MyString(const char* InitialInput)
15:     {
16:         if(InitialInput != NULL)
17:         {
18:             Buffer = new char [strlen(InitialInput) + 1];
19:             strcpy(Buffer, InitialInput);
20:         }
21:         else
22:             Buffer = NULL;
23:     }
24:
25:     // Конструктор копий: вставить из листинга 9.9
26:     MyString(const MyString& CopySource);
27:
28:     // Оператор присвоения копии: вставить из листинга 12.9
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 != NULL)
41:             delete [] Buffer;
42:     }
43:
44:     int GetLength() const
45:     {
46:         return strlen(Buffer);
47:     }
48:
49:     operator const char*()
50:     {
51:         return Buffer;
```

```
32:     }
33: };
34:
35: int main()
36: {
37:     cout << "Type a statement: ";
38:     string strInput;
39:     getline(cin, strInput);
40:
41:     MyString youSaid(strInput.c_str());
42:
43:     cout << "Using operator[] for displaying your input: " << endl;
44:     for(int Index = 0; Index < youSaid.GetLength(); ++Index)
45:         cout << youSaid[Index] << " ";
46:     cout << endl;
47:
48:     cout << "Enter index 0 - " << youSaid.GetLength() - 1 << ": ";
49:     int InIndex = 0;
50:     cin >> InIndex;
51:     cout << "Input character at zero-based position: " << InIndex;
52:     cout << " is: "<< youSaid[InIndex] << endl;
53:
54:     return 0;
55: }
```

Результат

```
Type a statement: Hey subscript operators[] are fabulous
Using operator[] for displaying your input:
H e y   s u b s c r i p t   o p e r a t o r s [ ]   a r e   f a b u l o u s
Enter index 0 - 37: 2
Input character at zero-based position: 2 is: y
```

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 индексирования (`[]`) 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`. .

Реализовав оператор индексирования, вы можете улучшить версию, представленную в листинге 12.10. Это реализация одного оператора индексирования, который позволяет читать, и второго, позволяющего записывать значения в элемент динамического массива.

Вы можете реализовать два оператора индексирования: 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 доступа
                                      // к символу по индексу
```

ВНИМАНИЕ!

Константность операторов

При создании операторов важно использовать ключевое слово `const`. Обратите внимание, как листинг 12.10 ограничил возвращаемое значение оператора индексирования (`[]`) типом `const char&`. Программа работает и компилируется даже без ключевых слов `const`, но причина их применения в том, чтобы избежать подобного кода:

```
MyString sayHello("Hello World");
sayHello[2] = 'k' // ошибка: operator[] константный
```

При использовании ключевого слова `const` вы защищаете внутренний член класса `MyString::Buffer` от прямых модификаций извне при помощи оператора `[]`. Кроме объявления возвращаемого значения как `const`, следует также ограничить тип функции оператора как `const`, чтобы обеспечить неспособность этого оператора изменять атрибуты класса.

Как правило, желательно почаще использовать ограничение `const`, чтобы избежать непреднамеренных изменений данных и повысить защиту атрибутов класса.

Компилятор достаточно интеллектуален, чтобы вызвать константную версию функции для операций чтения и не константную для операций записи в объект `MyString`. Таким образом, вы можете (если хотите) иметь отдельные возможности в двух функциях индексирования. Например, одна функция регистрации записывает данные в контейнер, а другая читает их. Существуют и другие бинарные операторы (см. в табл. 12.2), которые могут быть переопределены или перегружены, но это не обсуждается далее. Однако их реализация подобна той, которая уже обсуждалась.

Другие операторы, такие как логические и побитовые операторы, 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 понадобятся часто.

Помните о цели вашего класса и способе его использования, принимая решение о перегрузке операторов или создании новых.

Оператор функции ()

Оператор `()`, заставляющий объекты вести себя, как функции, называется *оператором функции* (function operator). Они применяются в стандартной библиотеке шаблонов (STL) и, как правило, используются в алгоритмах STL. Они применимы при принятии решений; такие объекты функций обычно называются унарным или бинарным *предикатом* (predicate), в зависимости от количества операндов. Листинг 12.11 анализирует простой объект функции, позволяя понять, что дает ему такое интригующее имя!

ЛИСТИНГ 12.11. Объект функции, созданный с использованием оператора `()`

```
1: #include <iostream>
2: #include <string>
3:using namespace std;
4:
5: class CDisplay
```

```
 6: {
 7: public:
 8:     void operator () (string Input) const
 9:     {
10:         cout << Input << endl;
11:     }
12:};
13:
14:int main ()
15:{
16:     CDisplay mDisplayFuncObject;
17:
18:     // эквивалент
       // mDisplayFuncObject.operator () ("Display this string!");
19:     mDisplayFuncObject ("Display this string!");
20:
21:     return 0;
22: }
```

Результат

```
Display this string!
```

Анализ

Строки 8–11 реализуют оператор `()`, который затем используется в строке 18 функции `main()`. Обратите внимание, что компилятор может использовать объект `mDisplayFuncObject` как функцию (строка 18) при неявном преобразовании того, чему вызов оператора `()` придает вид вызова функции.

Следовательно, этот оператор также называется оператором функции `()`, и объект `CDisplay` также называется объектом функции, или *функтором* (functor). Более подробная информация по этой теме приведена на занятии 21, "Понятие объектов функций".

C++11

Конструктор перемещения и оператор присваивания при перемещении для высокоэффективного программирования

Конструктор перемещения и оператор присваивания при перемещении — это средства оптимизации производительности, которые стали частью стандарта C++11, гарантирующие, что временные значения (r-значения, которые не существуют вне выражения) избегут необязательного копирования. Это особенно полезно при работе класса, который управляет динамически распределяемым ресурсом, таким, как динамический массив или строка.

Проблема нежелательных этапов копирования

Обратите внимание на оператор суммы, реализованный в листинге 12.5, — фактически он создает копию и возвращает ее. Это же справедливо для оператора вычитания. Теперь рассмотрим, что произойдет при создании нового экземпляра класса `MyString`, использующего следующий синтаксис:

```
MyString Hello("Hello ");
MyString World("World");
MyString CPP(" of C++");
MyString sayHello(Hello + World + CPP); // оператор +, конструктор копий
MyString sayHelloAgain ("overwrite this");
sayHelloAgain = Hello + World + CPP;     // оператор +, конструктор копий,
                                         // оператор присвоения копии =
```

Это простая интуитивно понятная конструкция, конкатенирующая три строки с использованием бинарного оператора суммы (+), который создается так:

```
MyString operator+ (const MyString& AddThis)
{
    MyString NewString;

    if (AddThis.Buffer != NULL)
    {
        // копирование в NewString
    }
    return NewString; // возвращение копии по значению, вызов
                      // конструктора копий
}
```

У этого оператора суммы (+), облегчающего конкатенацию за счет использования интуитивно понятных выражений, также есть потенциальная возможность создать проблемы производительности. Создание объекта `sayHello` требует дважды выполнить оператор суммы; в результате выполнения каждого оператора + создается временная копия, поскольку класс `MyString` возвращается по значению, вызывая таким образом выполнение конструктора копий. Конструктор копий осуществляет глубокое копирование во временное значение, которое не существует после завершения выражения. Таким образом, это выражение приводит к созданию временных копий (r-значений), которые не нужны после завершения выражения, а следовательно, являются узким местом производительности, созданным особенностями языка C++. По крайней мере, до недавнего времени.

Теперь эта проблема наконец решена. Компилятор языка C++11 распознает временные объекты и использует конструктор перемещения или оператор присваивания при перемещении, если вы предоставите их.

Объявление конструктора перемещения и оператора присваивания при перемещении

Конструктор перемещения имеет следующий синтаксис:

```
Class MyClass
{
private:
    Type* PtrResource;

public:
    MyClass(); // стандартный конструктор
    MyClass(const MyClass& CopySource); // конструктор копий
    MyClass& operator= (const MyClass& CopySource); // Оператор
                                                    // присвоения копии
    MyClass(MyClass&& MoveSource) // Конструктор перемещения,
                                  // обратите внимание на &&
```

```
    {
        PtrResource = MoveSource.PtrResource; // взять в собственность,
                                              // начало перемещения
        MoveSource.PtrResource = NULL;
    }

    MyClass& operator= (MyClass&& MoveSource) // оператор присваивания
                           //  при перемещении, обратите внимание на &&
    {
        if(this != &MoveSource)
        {
            delete [] PtrResource; // освободить собственный ресурс
            PtrResource = MoveSource.PtrResource; // взять в
                                  // собственность,начало перемещения
            MoveSource.PtrResource = NULL; // освободить источник
                                        // перемещения от собственности
        }
    }
};
```

Таким образом, объявление конструктора перемещения и оператора присвоения отличается от обычного конструктора копий и оператора присвоения копии тем, что входной параметр имеет тип `MyClass&&`. Кроме того, поскольку входной параметр является источником перемещения, он не может быть константным, ведь он изменяется. Возвращаемые значения остаются теми же, поскольку это перегруженные версии конструктора и оператора присвоения соответственно.

Совместимые со стандартом C++11 компиляторы гарантируют, что для временных объектов (r-значений) используется конструктор перемещения, а не конструктор копий, и оператор присваивания при перемещении вместо оператора присвоения копии. В нашей реализации мы обеспечим просто перемещение ресурса из источника получателю вместо копирования. Листинг 12.12 демонстрирует эффективность этих двух нововведений C++11 для оптимизации класса `MyString`.

ЛИСТИНГ 12.12. Класс `MyString` с конструктором перемещения и оператором присваивания при перемещении в дополнение к конструктору копий и оператору присвоения копии

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: class MyString
 4: {
 5: private:
 6:     char* Buffer;
 7:
 8:     // закрытый стандартный конструктор
 9:     MyString(): Buffer(NULL)
10:     {
11:         cout << "Default constructor called" << endl;
12:     }
13:
14: public:
15:     // Деструктор
16:     ~MyString()
```

```
17:     {
18:         if (Buffer != NULL)
19:             delete [] Buffer;
20:     }
21:
22:     int GetLength()
23:     {
24:         return strlen(Buffer);
25:     }
26:
27:     operator const char*()
28:     {
29:         return Buffer;
30:     }
31:
32:     MyString operator+ (const MyString& AddThis)
33:     {
34:         cout << "operator+ called: " << endl;
35:         MyString NewString;
36:
37:         if (AddThis.Buffer != NULL)
38:         {
39:             NewString.Buffer = new char[GetLength() \
                        + strlen(AddThis.Buffer) + 1];
40:             strcpy(NewString.Buffer, Buffer);
41:             strcat(NewString.Buffer, AddThis.Buffer);
42:         }
43:
44:         return NewString;
45:     }
46:
47:     // Конструктор
48:     MyString(const char* InitialInput)
49:     {
50:         cout << "Constructor called for: " << InitialInput << endl;
51:         if(InitialInput != NULL)
52:         {
53:             Buffer = new char [strlen(InitialInput) + 1];
54:             strcpy(Buffer, InitialInput);
55:         }
56:         else
57:             Buffer = NULL;
58:     }
59:
60:     // конструктор копий
61:     MyString(const MyString& CopySource)
62:     {
63:         cout<<"Copy constructor to copy from: "<< CopySource.Buffer
                << endl;
64:         if(CopySource.Buffer != NULL)
65:         {
66:             // гарантировать глубокое копирование, зарезервировав
                // предварительно собственный буфер
67:             Buffer = new char [strlen(CopySource.Buffer) + 1];
```

```
68:
69:             // копировать из оригинала в локальный буфер
70:             strcpy(Buffer, CopySource.Buffer);
71:         }
72:         else
73:             Buffer = NULL;
74:     }
75:
76:     // Оператор присвоения копии
77:     MyString& operator= (const MyString& CopySource)
78:     {
79:         cout<<"Copy assignment operator to copy from: "
                << CopySource.Buffer << endl;
80:         if ((this != &CopySource) && (CopySource.Buffer != NULL))
81:         {
82:             if (Buffer != NULL)
83:                 delete[] Buffer;
84:
85:             // гарантировать глубокое копирование, зарезервировав
                // предварительно собственный буфер
86:             Buffer = new char [strlen(CopySource.Buffer) + 1];
87:
88:             // копировать из оригинала в локальный буфер
89:             strcpy(Buffer, CopySource.Buffer);
90:         }
91:
92:         return *this;
93:     }
94:
95:     // Конструктор перемещения
96:     MyString(MyString&& MoveSource)
97:     {
98:         cout << "Move constructor to move from: "
                 << MoveSource.Buffer << endl;
99:         if(MoveSource.Buffer != NULL)
100:        {
101:            Buffer = MoveSource.Buffer; // взять собственность
                                            // т.е. 'переместить'
102:            MoveSource.Buffer = NULL;   // освободить источник
                                            // перемещения
103:        }
104:    }
105:
106:    // Оператор присваивания при перемещении
107:    MyString& operator= (MyString&& MoveSource)
108:    {
109:        cout<<"Move assignment operator to move from: "
                << MoveSource.Buffer << endl;
110:        if((MoveSource.Buffer != NULL) && (this != &MoveSource))
111:        {
112:            delete Buffer; // освободить собственный буфер
113:
114:            Buffer = MoveSource.Buffer; // взять собственность
                                            // т.е. 'переместить'
```

```
115:             MoveSource.Buffer = NULL; // освободить источник
                                           // перемещения
116:         }
117:
118:         return *this;
119:     }
120: };
121:
122: int main()
123: {
124:     MyString Hello("Hello ");
125:     MyString World("World");
126:     MyString CPP(" of C++");
127:
128:     MyString sayHelloAgain ("overwrite this");
129:     sayHelloAgain = Hello + World + CPP;
130:
131:     return 0;
132: }
```

Результат

Вывод без конструктора перемещения и оператора присваивания при перемещении (при закомментированных строках 95–119):

```
Constructor called for: Hello
Constructor called for: World
Constructor called for: of C++
Constructor called for: overwrite this
operator+ called:
Default constructor called
Copy constructor to copy from: Hello World
operator+ called:
Default constructor called
Copy constructor to copy from: Hello World of C++
Copy assignment operator to copy from: Hello World of C++
```

Вывод с конструктором перемещения и оператором присваивания при перемещении:

```
Constructor called for: Hello
Constructor called for: World
Constructor called for: of C++
Constructor called for: overwrite this
operator+ called:
Default constructor called
Move constructor to move from: Hello World
operator+ called:
Default constructor called
Move constructor to move from: Hello World of C++
Move assignment operator to move from: Hello World of C++
```

Анализ

Пример кода получился действительно длинным, но большая его часть уже была представлена в предыдущих примерах и занятиях. Самая важная часть этого листинга находится в строках 95–119, где реализованы конструктор перемещения и оператор присваивания при перемещении соответственно. Те части вывода, на которые воздействуют 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 конструктора перемещения и оператора присваивания при перемещении, то можно заметить, что семантика перемещения по существу реализуется за счет принятия принадлежности ресурсов от источника перемещения (строка 101 в конструкторе перемещения и строка 114 в операторе присваивания при перемещении). Непосредственно за этим следует присвоение значения NULL указателю источника (строки 102 и 115). Таким образом, даже когда источник перемещения удаляется, вызываемый через деструктор в оператор `delete` (строки 16–20) по существу ничего не делает, поскольку собственность была передана объекту получателя. Обратите внимание, что в отсутствии конструктора перемещения вызывается конструктор копий, который осуществляет глубокое копирование строки. Таким образом, конструктор перемещения существенно экономит на продолжительности обработки и сокращает количество нежелательных операций резервирования памяти и этапов копирования.

Создание конструктора перемещения и оператора присваивания при перемещении совершенно необязательно. В отличие от конструктора копий и оператора присвоения копии, компилятор не добавляет его стандартную реализацию сам.

Используйте эти средства C++11 для оптимизации работы классов, которые указывают на динамически распределяемые ресурсы, которые в противном случае требовали бы глубокого копирования даже в тех случаях, где они требуются только временно.

Операторы, которые не могут быть перегружены

Со всей той гибкостью, которую предоставляет язык C++ в настройке поведения операторов и классов, он все же не разрешает переделывать или изменять поведение некоторых операторов, которые при любых ситуациях выполняются однозначно. 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
`::`	Область видимости
`? :`	Условный троичный оператор
`sizeof`	Размер объекта типа или класса

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Создавайте столько операторов, сколько необходимо для упрощения использования класса, но не больше	**Не забывайте**, что компилятор предоставит стандартный оператор присвоения копий и конструктор копий, если вы не снабдите его ими, но они не будут гарантировать глубокого копирования простых указателей, содержавшихся в пределах класса
Всегда создавайте оператор присвоения копии (с конструктором копий и деструктором) для класса, членом которого является простой указатель	**Не забывайте**, что если вы не предоставите оператор присваивания при перемещении или конструктора перемещения, компилятор не добавит их сам, а вернется вместо этого к обычному оператору присвоения копии и конструктору копий
Всегда создавайте оператор присваивания при перемещении (и конструктор перемещения) для классов, которые управляют динамически распределенными ресурсами, такими, как массив данных, при использовании компилятора, совместимого со стандартом C++11	**Не забывайте**, что создание операторов не является обязательным, но все же они повышают удобство и простоту использования вашего класса
	Не забывайте, что ваш класс интеллектуального указателя еще не указатель, пока вы не реализовали операторы (*) и (->), и он не достаточно интеллектуален, пока вы не реализовали деструктор и не обдумали случаи присвоения копии и создания копии

Резюме

Вы узнали, как создание операторов может существенно повлиять на легкость использования вашего класса. При разработке класса, который управляет ресурсами, например динамическим массивом или строкой, в дополнение к деструктору необходимо предоставить конструктор копий и оператор присвоения копии как минимум. Вспомогательный класс, который управляет динамическим массивом, неплохо укомплектовать конструктором перемещения и оператором присваивания при перемещении, который гарантирует, что содержавшийся ресурс не будет копироваться глубоко для временных объектов. И наконец, вы узнали, что такие операторы, как `.`, `.*`, `::`, `?:` и `sizeof`, не могут быть переопределены.

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 присвоение, или косвенно, при передаче функции по значению. Как правило, вы реализуете конструктор копий, оператор присвоения копии и деструктор. В определенных случаях вы также

реализуете конструктор перемещения и оператор присваивания при перемещении, если хотите повысить производительность класса.

- **У меня есть экземпляр `object` класса. Я хочу обеспечить такой синтаксис: `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 заключается в том, чтобы определить оператор `char*()`, как мы уже сделали 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 реализуете операторы (`*`) и (`->`). Кроме того, чтобы он был интеллектуальным, вам нужно также позаботиться об автоматическом освобождении ресурсов, предоставив соответственно деструктор, а также четко определиться с тем, как осуществляется копирование и присвоение, реализовав конструктор копий и оператор присвоения копии либо запретив их, объявив последние два закрытыми.

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 Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Может ли мой оператор индексирования `[]` возвращать константный и не константный варианты типов возвращаемого значения?
   ```
   const Type& operator[](int Index);
   Type& operator[](int Index); // это нормально?
   ```
2. Объявляли бы вы конструктор копий или оператор присвоения копии как `private`?
3. Имеет ли смысл определять конструктор перемещения и оператор присваивания при перемещении для нашего класса `Date`?

Упражнения

1. Создайте оператор преобразования для класса `Date`, который преобразует содержащуюся в нем дату в целое число.
2. Создайте конструктор перемещения и оператор присваивания при перемещении для класса `DynIntegers`, который инкапсулирует динамически распределенный массив в форме закрытого члена `int*`.

ЗАНЯТИЕ 13
Операторы приведения

Приведение типов (casting) — это механизм, позволяющий программисту изменить интерпретацию объекта компилятором. Но это не подразумевает изменение самого объекта, изменяется только его интерпретация. Операторы, которые изменяют интерпретацию объекта, называются *операторами приведения* (casting operator).

На сегодняшнем занятии.

- Потребность в операторах приведения.
- Почему приведение в стиле C не нравится некоторым программистам C++.
- Четыре оператора приведения типов C++.
- Концепции приведения вверх и приведения вниз.
- Почему операторы приведения типов C++ не всегда наилучший выбор.

Потребность в приведении типов

В абсолютно строго типизированном мире хорошо продуманных приложений C++ не должно быть никакой потребности в приведении типов и в операторах приведения. Однако мы живем в реальном мире, где программы разрабатывают по частям множество разных людей и исполнителей, а нередко необходимо взаимодействие различных систем. Для этого очень часто приходится заставлять компилятор интерпретировать данные такими способами, чтобы приложения компилировались и работали правильно.

Возьмем реальную ситуацию: хотя некоторые компиляторы C++ могут поддерживать тип `bool` как базовый, но большинство 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. Так, тип `bool` на этих компиляторах выглядит примерно следующим образом:

```
typedef unsigned short BOOL;
```

Функция, возвращающая логические данные, была бы объявлена как

```
BOOL IsX ();
```

Теперь, если такая библиотека должна использоваться с новым приложением, созданным для последней версии компилятора C++, у разработчика есть способ сделать данные логического типа `BOOL`, понятные библиотеке, понятными функциям компилятора C++. Для этого используется приведение типов:

```
bool bCPPResult = (bool)IsX (); // приведение в стиле C
```

Развитие языка C++ привело к появлению новых операторов приведения, и это разделило мнение сообщества разработчиков C++ по данному поводу: одни продолжают использовать приведения в стиле C, а другие неукоснительно придерживаются применения ключевых слов приведения типов, введенных компиляторами C++. Аргумент первой группы: приведения в стиле C++ громоздки для использования и иногда немного отличаются по функциональным возможностям таковых в C, что, впрочем, имеет только теоретическое значение. Вторая группа, которая очевидно состоит из фанатиков синтаксиса C++, указывает на недостаточность приведения в стиле C для всех случаев. 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 осуществлять даже это:

```
char* pszString = "Hello World!";
int* pBuf = pszString; // ошибка: нельзя преобразовать char* в int*
```

...причем, вполне резонно!

Современные компиляторы C++ все еще 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 назначение как тип, который очень удобен по мнению разработчика, но именно он в данном случае не потрудился подумать о том, что компилятор имел серьезные основания, сообщая о ошибке, и просто заткнул ему рот, вынуждая повиноваться. Конечно, это не станет поперек горла тем разработчикам C++, которые, видя угрозу безопасности типа, созданную приведениями, проталкивают что-нибудь свое.

Операторы приведения C++

Несмотря на недостатки приведения типов, от самой их концепции отказываться нельзя. Во многих ситуациях приведение — единственная возможность решения важных проблем совместимости. Кроме того, язык C++ предоставляет новый оператор приведения, специфический для случаев наследования, которых не существовало в языке C.

Четыре оператора приведения C++:

- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`
- `const_cast`

Синтаксис их применения одинаков:

```
тип_назначения результат = тип_приведения <тип_назначения> (приводимый_объект);
```

Использование оператора `static_cast`

Оператор `static_cast` применяется для преобразования указателей связанных типов и явного преобразования для стандартных типов данных, которые в противном случае осуществлялись бы автоматически или неявно. Поскольку речь идет об указателях, оператор `static_cast` реализует простую проверку времени компиляции приводимости указателя к связанному типу. Это усовершенствование по сравнению с приведением в стиле C, которое позволяет указателю на один объект быть приведенным к абсолютно несвязанному типу без всяких жалоб. Используя оператор `static_cast`, указатель можно привести вверх к базовому классу или вниз к производному, как демонстрирует следующий пример:

```
Base* pBase = new Derived ();          // создать объект класса Derived
Derived* pDerived = static_cast<Derived*>(pBase); // ok!

// CUnrelated никак не связан с классом Base в иерархии наследования
CUnrelated* pUnrelated = static_cast<CUnrelated*>(pBase); // Ошибка
// Приведение вверх не разрешается, поскольку типы не связаны
```

ПРИМЕЧАНИЕ

Приведение производного типа к базовому называется *приведением вверх* (upcast) и может быть осуществлено без явного оператора приведения:

```
Derived objDerived;
Base* pBase = &objDerived; // ok!
```

Приведение базового типа к производному называется *приведением вниз* (downcast) и не может быть осуществлено без применения явных операторов приведения:

```
Derived objDerived;
Base* pBase = &objDerived; // Приведение вверх -> ok!
Derived* pDerived = pBase; // Ошибка: приведение вниз
осуществляется явно
```

Но обратите внимание, что оператор `static_cast` проверяет только то, что ссылочные типы связаны. Он *не выполняет* проверок времени выполнения. Так, с оператором `static_cast` разработчик вполне может совершить следующую ошибку:

```
Base* pBase = new Base ();
Derived* pDerived = static_cast<Derived*>(pBase); // Все еще
                                                  // никакой ошибки!
```

Здесь указатель `pDerived` фактически указывает на частичный объект `Derived`, поскольку объект, на который он указывает, фактически имеет тип `Base()`. Так как оператор `static_cast` выполняет проверку только во время компиляции, подтверждая связанность рассматриваемых типов, и не выполняет проверку времени выполнения, вызов `pDerived->SomeDerivedClassFunction()` будет откомпилирован, но, вероятно, приведет к неожиданным последствиям во время выполнения.

Кроме помощи в приведении вверх или вниз, оператор `static_cast` может помочь сделать неявные приведения явными и привлечь к ним внимание разработчика или читателя:

```
double dPi = 3.14159265;
int Num = static_cast<int>(dPi); // Сделать неявное приведение явным
```

В приведенном выше коде выражение `Num = dPi` 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.

Использование оператора `dynamic_cast` и идентификация типа времени выполнения

Динамическое приведение типов, как и предполагает его название, является противоположностью статического приведения типов и фактически выполняет приведение в исполняющей среде (времени выполнения) — то есть во время выполнения приложения. Результат выполнения оператора `dynamic_cast` можно проверить и выяснить, была ли успешной попытка приведения типов. Вот типичный синтаксис применения оператора `dynamic_cast`:

```
тип_назначения* рНазн = dynamic_cast <тип_класса*> (рИсточн);
if (рНазн) // Проверить успех операции приведения типов прежде,
           // чем использовать указатель
    рНазн->ВызовФунк();
```

Например:

```
Base* pBase = new Derived();

// Осуществить приведение вниз
Derived* pDerived = dynamic_cast <Derived*> (pBase);

if (pDerived) // Проверить успех приведения
    pDerived->CallDerivedClassFunction ();
```

Как показано в приведенном выше коротком примере, имея указатель на объект базового класса, разработчик может прибегнуть к оператору `dynamic_cast`, чтобы проверить тип объекта назначения, прежде чем перейти к использованию указателя на него. Обратите внимание: во фрагменте кода очевидно, что объект назначения имеет тип `Derived`. Это пример лишь для демонстрации. Но так бывает не всегда, например, когда указатель типа `Derived*` передается функции, получающей тип `Base*`. Функция может применить оператор `dynamic_cast` к переданному указателю типа базового класса, чтобы выяснить его тип, а затем выполнить операции, специфические для выясненного типа. Таким образом, оператор `dynamic_cast` позволяет определить тип времени выполнения и использовать приведенный указатель, когда это безопасно. Листинг 13.1 использует знакомую иерархию классов `Tuna` и `Carp`, связанных базовым классом `Fish`, где функция `DetectFishType()` динамически обнаруживает, имеет ли указатель типа `Fish*` на самом деле тип `Tuna*` или `Carp*`.

ПРИМЕЧАНИЕ

Поэтому данный механизм идентификации типа объекта во время выполнения и называется *идентификацией типа времени выполнения* (runtime type identification), или *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 << "Fish swims in water" << endl;
 9:     }
10:
11:     // базовый класс всегда должен иметь виртуальный деструктор
12:     virtual ~Fish() {}
13: };
14:
15: class Tuna: public Fish
```

```
16: {
17: public:
18:     void Swim()
19:     {
20:         cout << "Tuna swims real fast in the sea" << endl;
21:     }
22:
23:     void BecomeDinner()
24:     {
25:         cout << "Tuna became dinner in Sushi" << endl;
26:     }
27: };
28:
29: class Carp: public Fish
30: {
31: public:
32:     void Swim()
33:     {
34:         cout << "Carp swims real slow in the lake" << endl;
35:     }
36:
37:     void Talk()
38:     {
39:         cout << "Carp talked crap" << endl;
40:     }
41: };
42:
43: void DetectFishType(Fish* InputFish)
44: {
45:     Tuna* pIsTuna = dynamic_cast <Tuna*>(InputFish);
46:     if (pIsTuna)
47:     {
48:         cout << "Detected Tuna. Making Tuna dinner: " << endl;
49:         pIsTuna->BecomeDinner(); // вызов Tuna::BecomeDinner
50:     }
51:
52:     Carp* pIsCarp = dynamic_cast <Carp*>(InputFish);
53:     if(pIsCarp)
54:     {
55:         cout << "Detected Carp. Making carp talk: " << endl;
56:         pIsCarp->Talk(); // вызов Carp::Talk
57:     }
58:
59:     cout << "Verifying type using virtual Fish::Swim: " << endl;
60:     InputFish->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

```
Detected Tuna. Making Tuna dinner:
Tuna became dinner in Sushi
Verifying type using virtual Fish::Swim:
Tuna swims real fast in the sea

Detected Carp. Making carp talk:
Carp talked crap
Verifying type using virtual Fish::Swim:
Carp swims real slow in the lake
```

Анализ

Это иерархия классов `Tuna`, `Carp` и `Fish`, рассматривавшаяся на занятии 10, "Реализация наследования". Напомню, что эти два производных класса не только реализуют виртуальную функцию `Swim()`, но и содержат функцию, специфическую для каждого из типов, а именно `Tuna::BecomeDinner()` и `Carp::Talk()`. Особенным в этом примере является то, что, имея экземпляр базового класса `Fish*`, вы можете динамически обнаружить, указывает ли этот указатель на объект класса `Tuna` или `Carp`. Это 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 характера входного указателя базового класса типа `Fish*` на тип `Tuna*`. Если это указатель на тип `Tuna`, оператор возвращает допустимый адрес, в противном случае — значение NULL. Следовательно, результат оператора `dynamic_cast` всегда должен проверяться на допустимость. После проверки успешности в строке 46 вы знаете, что указатель `pIsTuna` указывает на допустимый объект класса `Tuna`, и в состоянии использовать его для вызова функции `Tuna::BecomeDinner()`, как демонстрирует строка 49. С карпом вы используете указатель для вызова функции `Carp::Talk()`, как демонстрирует строка 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 допустимость. Когда приведение недопустимо, возвращается значение NULL.

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, т.е. он интерпретирует тип с использованием следующего синтаксиса:

```
Base * pBase = new Base ();
CUnrelated * pUnrelated = reinterpret_cast<CUnrelated*>(pBase);
// Такой код плох, даже когда он компилируется!
```

Это приведение фактически вынуждает компилятор принять ситуации, которые обычно не разрешил бы оператор `static_cast`. Он находит применение в определенных низкоуровневых приложениях (таких, как драйверы, например), где данные должны быть преобразованы в простой тип, который может принять интерфейс API (например, некоторые функции API работают только с байтовыми потоками, т.е. `unsigned char*`):

```
SomeClass* pObject = new SomeClass ();
// Необходимо передать объект как поток байтов...
unsigned char* pBytes = reinterpret_cast <unsigned char*>(pObject);
```

Приведение, используемое в коде выше, не изменило двоичного представления исходного объекта и фактически обмануло компилятор, разрешив разработчику просмотреть отдельные байты, содержащиеся в объекте типа `SomeClass`. Поскольку никакой другой оператор приведения C++ не позволил бы такого преобразования, оператор `reinterpret_cast` явно предупреждает разработчика о потенциально небезопасном преобразовании.

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 для проекта, и для реализации.

Использование оператора `const_cast`

Оператор `const_cast` позволяет отключить модификатор доступа `const` объекта. Если вы задаетесь вопросом, зачем это приведение нужно вообще, то, вероятно, правы. В идеальной ситуации, когда разработчики пишут свои классы правильно, они не забывают использовать ключевое слово `const` часто и в правильных местах. На практике все, к сожалению, совсем не так, и код наподобие следующего весьма распространен:

```
class SomeClass
{
public:
    // ...
    void DisplayMembers (); // функция представления должна
                            // быть константной
};
```

Когда вы создаете такую функцию, как эта, происходит отказ:

```
void DisplayAllData (const SomeClass& mData)
{
    mData.DisplayMembers (); // Отказ компиляции
    // причина отказа: вызов не константного члена
    // класса с использованием константной ссылки
}
```

Вы, вероятно, мимоходом исправили передачу объекта mData на константную ссылку. В конце концов, функция отображения предназначена только для чтения и не должна позволять вызывать непостоянные функции-члены, т.е. она не должна позволять вызывать функции, способные изменить состояние объекта. Однако реализация функции DisplayMembers(), которая также должна быть константой, к сожалению, таковой не является. Пока 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 принадлежит библиотеке стороннего производителя, и внести в нее изменения невозможно. В подобных ситуациях на выручку приходит оператор const_cast.

Синтаксис его применения для функции DisplayMembers() будет таким:

```
void DisplayAllData (const SomeClass& mData)
{
    SomeClass& refData = const_cast <SomeClass&>(mData);
    refData.DisplayMembers(); // Разрешено!
}
```

Обратите внимание на то, что применение оператора const_cast для вызова не константных функций должно быть последним средством. Имейте в виду, что использование оператора const_cast для изменения константного объекта может привести к непредвиденным последствиям.

Обратите также внимание на то, что оператор const_cast применяется с указателями:

```
void DisplayAllData (const SomeClass* pData)
{
    // pData->DisplayMembers(); // Ошибка: попытка вызвать
                                // не константную функцию!
    SomeClass* pCastedData = const_cast <SomeClass*>(pData);
    pCastedData->DisplayMembers(); // Разрешено!
}
```

Проблемы с операторами приведения C++

Не все довольны всеми операторами приведения типов C++, даже те, кто их предпочитает. Причины самые разнообразные — от слишком громоздкого и не интуитивно понятного синтаксиса до избыточности.

Сравним следующий код:

```
double dPi = 3.14159265;

// Приведение в стиле C++: static_cast
int Num = static_cast <int>(dPi); // результат: nNum будет 3

// Приведение в стиле C
int Num2 = (int)dPi; // результат: Num2 будет 3

// оставить приведение типов компилятору
int Num3 = dPi; // результат: Num3 будет 3. Никакой ошибки!
```

Во всех трех случаях достигнут тот же результат. В практических случаях вторая версия, вероятно, является наиболее распространенной, затем следует третья. Немногие

решатся использовать первую возможность. В любом случае, компилятор достаточно интеллектуален, чтобы преобразовывать такие типы правильно. Это создает впечатление, будто синтаксис приведения затрудняет чтение кода.

Аналогично, другой случай применения оператора `static_cast` также хорошо заменяется приведениями в стиле C, которые, по общему мнению, выглядят проще:

```
// использование static_cast
Derived* pDerived = static_cast <Derived*>(pBase);
// Но это работает точно так же...
Derived* pDerivedSimple = (Derived*)pBase;
```

Таким образом, преимущества использования оператора `static_cast` зачастую омрачаются неуклюжестью его синтаксиса. Собственные слова *Бьярне Страуструпа* (Bjarne Stroustrup) точно выражают ситуацию: "Возможно, поскольку оператор `static_cast` настолько неуклюж и относительно труден при вводе, вы, вероятно, подумаете дважды, прежде чем использовать его. Это и к лучшему, поскольку в современном языке C++ приведений, как правило, действительно можно избежать". (См. раздел "frequently asked questions about C++ style and technique" на сайте Бьярне Страуструпа по адресу `http://www.research.att.com/~bs/bs_faq2.html`.)

Рассмотрим другие операторы. Оператор `reinterpret_cast` позволяет протолкнуть ваш способ, когда оператор `static_cast` не работает; точно так же применяется оператор `const_cast`, изменяя модификатор доступа `const`. Таким образом, операторы приведения C++, кроме `dynamic_cast`, вполне преодолимы в современных приложениях C++. Применение других операторов приведения становится уместным только тогда, когда решено использовать устаревшие приложения. В таких случаях предпочтительно использование приведений в стиле C, а использование операторов приведения C++ зачастую является делом вкуса. Однако важней всего в максимально возможной степени избегать приведений, а когда это не удается, необходимо, по крайней мере, точно знать, что при этом происходит внутренне.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Помните, что приведение типа `Derived*` к `Base*` называется приведением вверх, и оно безопасно	**Не забывайте** проверять указатель на допустимость после использования оператора `dynamic_cast`
Помните, что приведение типа `Base*` непосредственно к типу `Derived*` называется приведением вниз, и оно может быть небезопасно, если вы не используете оператор `dynamic_cast`	**Не создавайте** свое приложение на базе RTTI с использованием оператора `dynamic_cast`
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	

Резюме

На сегодняшнем занятии рассматривались различные операторы приведения C++, аргументы за их применение и против. Вы также узнали, что вообще применения приведений следует избегать.

Вопросы и ответы

- **Нормально ли изменять содержимое константного объекта при приведении типа указателя или ссылки на него с использованием оператора `const_cast`?**

  Как правило, определенно нет. Результат такой операции не предсказуем и определенно не желателен.

- **Мне нужен указатель `Bird*`, а имеется `Dog*`. Компилятор не позволяет мне использовать указатель на объект `Dog` как `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`. Это хорошо?**

  Тоже 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, а возможно приведет и к отказу приложения.

- **У меня есть объект класса `Derived` и указатель на него `pBase`, типа `Base*`. Я уверен, что указатель `pBase` указывает на объект класса `Derived`, так должен ли я использовать оператор `dynamic_cast`?**

  Поскольку вы уверены, что типом объекта, на который он указывает, является `Derived`, можете сэкономить ресурсы исполняющей среды на использовании оператора `static_cast`.

- **Язык C++ предоставляет операторы приведения, и все же мне настоятельно советуют не использовать их. Почему?**

  Вы храните дома аспирин, но не едите его ложкой каждый день только потому, что он есть, правильно? Используйте их, только когда на самом деле необходимо.

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 Г, "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 `pBase` на объект базового класса. Какой оператор приведения использовать, чтобы выяснить его тип: `Derived1` или `Derived2`?

2. Вы имеете константную ссылку на объект и пытаетесь вызвать открытую функцию-член, написанную вами. Компилятор не позволяет этого, поскольку рассматриваемая функция — не константный член класса. Вы бы исправили функцию или использовали оператор `const_cast`?
3. Оператор `reinterpret_cast` следует использовать только тогда, когда оператор `static_cast` не работает, а приведение неизбежно и заранее известно, что оно безопасно. Правда ли это?
4. Правда ли, что большинство преобразований, выполняемых оператором `static_cast`, особенно между простыми типами данных, компилятор C++ способен выполнить автоматически?

Упражнения

1. **Отладка:** Что неправильно в следующем коде?

```
void DoSomething(Base* pBase)
{
    Derived* pDerived = dynamic_cast <Derived*>(pBase);
    pDerived->DerivedClassMethod();
}
```

2. У вас есть указатель `pFish*`, указывающий на объект класса `Tuna`.

```
Fish* pFish = new Tuna;
Tuna* pTuna = <what cast?>pFish;
```

Какой оператор приведения следует использовать, чтобы получить указатель `Tuna*` на этот объект типа `Tuna`? Представьте использующий его код.

ЗАНЯТИЕ 14
Макросы и шаблоны

К настоящему моменту у вас должно быть четкое понимание основ синтаксиса языка C++. Программы, написанные на языке C++, должны быть уже понятны, и вы готовы изучать те средства языка, которые помогут писать приложения эффективней.

На сегодняшнем занятии.

- Введение в препроцессор.
- Ключевое слово `#define` и макросы.
- Введение в шаблоны.
- Как писать шаблоны функций и классов.
- Различие между макросами и шаблонами.
- Как C++11 помогает разрабатывать проверки времени компиляции, используя ключевое слово `static_assert`.

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 инструкций препроцессор фактически решает, что будет компилироваться. Все *директивы препроцессора* (preprocessor directive) начинаются со знака #. Например:

```
// указать препроцессору вставить здесь содержимое заголовка iostream
#include <iostream>

// определить константу макроса
#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, представленных во фрагменте кода выше; 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 заменяет некий идентификатор другим текстом.

Использование директивы #define для определения константы

Синтаксис применения директивы `#define` для определения константы очень прост:

```
#define идентификатор значение
```

Например, константа `ARRAY_LENGTH`, заменяемая значением 25, была бы объявлена так:

```
#define ARRAY_LENGTH 25
```

Теперь текст идентификатора `ARRAY_LENGTH` заменяется текстом `25` везде, где препроцессор встретит его:

```
int MyNumbers [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 MyNumbers [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 такой цикл `for`, как этот:

```
for(int Index = 0; Index < ARRAY_LENGTH; ++Index)
    MyNumbers[Index] = Index;
```

Этот цикл `for` видим компилятору так:

```
for(int Index = 0; Index < 25; ++Index)
    MyNumbers[Index] = Index;
```

Чтобы увидеть все это в действии, рассмотрим листинг 14.1.

ЛИСТИНГ 14.1. Объявление и использование макроса, определяющего 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 MyNumbers [ARRAY_LENGTH] = {0};
12:     cout << "Array's length: " << sizeof(MyNumbers) / sizeof(int)
        << endl;
13:
14:     cout << "Enter a radius: ";
15:     MY_DOUBLE Radius = 0;
16:     cin >> Radius;
17:     cout << "Area is: " << PI * Radius * Radius << endl;
18:
19:     string FavoriteWhisky (FAV_WHISKY);
20:     cout << "My favorite drink is: " << FAV_WHISKY << endl;
21:
22:     return 0;
23: }
```

Результат

```
Array's length: 25
Enter a radius: 2.1569
Area is: 14.7154
My favorite drink is: Jack Daniels
```

Анализ

ARRAY_LENGTH, PI, MY_DOUBLE и FAV_WHISKY являются четырьмя макро-константами, определенными в строках 3–7 соответственно. Как можно заметить, первая используется

при определении длины массива в строке 11, которая проверяется при помощи оператора `sizeof()` в строке 12. Константа `MY_DOUBLE` используется при объявлении переменный `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 используется в операторе `cout` (строка 20). Все эти случаи демонстрируют, что препроцессор просто осуществляет текстовую замену.

У такой "тупой" текстовой замены, которая, кажется, нашла повсеместное применение в листинге 14.1, есть также и недостатки.

СОВЕТ

Поскольку препроцессор делает лишь текстовую подстановку, он позволяет вам совершать ошибки (которые компилятор не всегда выявляет). Вы могли бы определить константу `FAV_WHISKY` в строке 7 листинга 14.1 так:

```
#define FAV_WHISKY 42 // "Jack Daniels"
```

Это может закончиться ошибкой компиляции в строке 19 при создании экземпляра класса `std::string`, а может и нет, тогда компилятор продолжит работу и выведет следующее:

```
My favorite drink is: 42
```

Это, конечно, не имело бы смысла, но важней всего то, что осталось бы необнаруженным. Кроме того, вы не имеете особого контроля над определением такой константы `PI`: будет ли она иметь тип `double` или `float`? Ответ: ни тот ни другой. `PI` для препроцессора – это 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; // использовать 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 *файлами заголовка* (header file). Соответствующие функции определяются в файлах с расширением `.cpp`, в которые включают файлы заголовка, используя директиву препроцессора `#include <заголовок>`. Если один файл заголовка (назовем его `class1.h`) объявляет класс, членом которого является другой класс, объявленный в заголовке `class2.h`, то файл `class1.h` должен включать файл `class2.h`. Если бы проект был достаточно сложен и другой класс требовал бы также предыдущего, то файл заголовка `class2.h` включал бы также файл `class1.h`!

Но для препроцессора два файла заголовка, которые включают друг друга, является проблемой рекурсивного характера. Чтобы избежать этой проблемы, можно использовать макрос вместе с директивами препроцессора `#ifndef` и `#endif`.

Заголовок `header1.h`, включающий заголовок `header2.h`, выглядит так:

```
#ifndef HEADER1_H_  // защита от множественного включения:
#define HEADER1_H_  // препроцессор будет читать эту и последующие
                    // строки только однажды
#include <header2.h>

class Class1
{
    // члены класса
};
#endif // конец header1.h
```

Заголовок `header2.h` выглядит похоже, но с другим определением и включает заголовок `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 прочитать как "если не определено". Это директива условного выражения, инструктирующая препроцессор продолжить выполнение, только если идентификатор не был определен. Директива `#endif` отмечает конец этой условной инструкции препроцессора.

Таким образом, если препроцессор встречает первым заголовок `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 выполнить этот файл закончится в первой строке, содержащей директиву `#ifndef`, поскольку это условие теперь ложно. То же справедливо и для заголовка `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 просто заменять текстовые элементы, идентифицируемые макросом, позволяет писать простые функции, 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 таких очень простых вычислений. Они обеспечивают преимущество обычного вызова функций, в которые они разворачиваются и встраиваются перед компиляцией, а следовательно, способны помочь улучшить производительность кода в определенных ситуациях. 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 << "Enter an integer: ";
13:     int Input1 = 0;
14:     cin >> Input1;
15:
16:     cout << "SQUARE(" << Input1 << ") = " << SQUARE(Input1) << endl;
17:     cout << "Area of a circle with radius " << Input1 << " is: ";
18:     cout << AREA_CIRCLE(Input1) << endl;
19:
20:     cout << "Enter another integer: ";
21:     int Input2 = 0;
22:     cin >> Input2;
23:
24:     cout << "MIN(" << Input1 << ", " << Input2 << ") = ";
25:     cout << MIN (Input1, Input2) << endl;
26:
27:     cout << "MAX(" << Input1 << ", " << Input2 << ") = ";
28:     cout << MAX (Input1, Input2) << endl;
29:
30:     return 0;
31: }
```

Результат

```
Enter an integer: 36
SQUARE(36) = 1296
Area of a circle with radius 36 is: 4071.51
Enter another integer: -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, использует константу PI, свидетельствуя таким образом, что один макрос может многократно использовать другой. В конце концов, это только команды препроцессора для простой замены оного текста другим. Давайте проанализируем строку 25, где используется макрофункция MIN:

```
cout << MIN (Input1, Input2) << endl;
```

После разворачивания макроса по месту эта строка, по существу, передается компилятору в следующем виде:

```
cout << (((Input1) < (Input2)) ? (Input1) : (Input2)) << endl;
```

ВНИМАНИЕ!

Макрофункции не учитывают тип, а следовательно, могут быть опасны. Например, макрофункция AREA_CIRCLE должна в идеале возвращать тип double, чтобы вы были уверенны в точности возвращаемого значения вычисленной площади и ее независимости от характера введенного радиуса.

Зачем все эти скобки?

Снова обратите внимание на макрос вычисления площади круга:

```
#define AREA_CIRCLE(r) (PI*(r)*(r))
```

У этого вычисления странный синтаксис с множеством скобок. Сравните его с функцией Area() из листинга 7.1 занятия 7, "Организация кода при помощи функций".

```
// Определения функций (реализации)
double Area(double InputRadius)
{
    return Pi * InputRadius * InputRadius; // видите, никаких скобок
}
```

Так почему же для макроса мы переусердствовали со скобками, когда та же формула в функции значительно отличается? Причина в способе, которым препроцессор обрабатывает макрос, т.е. в механизме текстовой подстановки.

Рассмотрим макрос без множества скобок:

```
#define AREA_CIRCLE(r) (PI*r*r)
```

Что будет при вызове этого макроса в таком выражении:

```
cout << AREA_CIRCLE (4+6);
```

Он был бы развернут препроцессором в такой код:

```
cout << (PI*4+6*4+6); // совсем не то, что и PI*10*10
```

Таким образом, по правилам 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, принято просматривать и проверять каждый путь выполнения кода непосредственно после его написания, это зачастую оказывается физически невозможно. Но можно вставить в код проверки, которые проверяют значения выражений или переменных.

Макрос `assert` позволяет сделать именно это. Чтобы использовать макрос `assert`, необходимо включить заголовок `assert.h`, а синтаксис его использования следующий:

```
assert (выражение, возвращающее true или false);
```

Вот пример использования макроса `assert()`, проверяющего содержимое указателя:

```
#include <assert.h>
int main()
{
    char* sayHello = new char [25];
    assert(sayHello != NULL); // сообщить, если указатель пуст

    // другой код

    delete [] sayHello;
    return 0;
}
```

Макрос `assert()` гарантирует уведомление, если указатель окажется недопустим. Для проверки я инициализировал указатель `sayHello` значением `NULL` и при запуске в режиме отладки Visual Studio немедленно получил на экране окно, которое вы видите на рис. 14.1.

Таким образом, макрос `assert()`, как реализовано в 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 проверить входные параметры функций. Это настоятельно рекомендуется и позволяет надолго улучшить качество вашего кода.

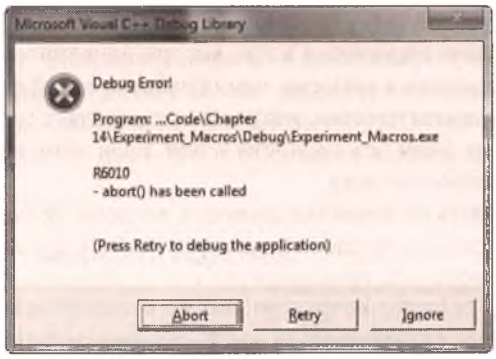


РИС. 14.1. Что происходит, когда макрос `assert` обнаруживает недопустимость указателя

ПРИМЕЧАНИЕ

При выпуске макрос `assert()` обычно отключают, а сообщения об ошибках и соответствующую информацию он предоставляет только в режиме отладки большинства сред разработки.

Кроме того, некоторые среды реализуют его как функцию, а не как макрос.

ВНИМАНИЕ!

Поскольку макрос `assert` не работает в финальном выпуске, выполнение критически важных для функционирования приложения проверок (например, возвращаемое значение оператора `dynamic_cast`) следует обеспечить отдельно, используя оператор `if`. Макрос `assert` позволяет обнаруживать проблемы, но это вовсе не замена проверки указателя, необходимая в коде.

Преимущества и недостатки использования макрофункций

Макросы позволяют многократно 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, 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`. Эта оптимизация и сокращение строк кода являются небольшим преимуществом и соблазняют некоторых программистов на использование макроса для определения простых вспомогательных функций. Эти макрофункции разворачиваются и встраиваются в код перед компиляцией, а следовательно, производительность простого макроса выше, чем у обычного вызова

функции, решающего ту же задачу. Это связано с тем, что вызов функции требует создания стека вызовов, передачи аргументов и т.д., так что дополнительные затраты зачастую занимают больше процессорного времени, чем обработка самой функции MIN.

Несмотря на все эти преимущества, макросы представляет серьезную проблему: они не поддерживают никаких форм безопасности типов. Если этого недостаточно, то отладка макроса также весьма непростое дело.

Если необходимо создать обобщенные функции, которые не зависят от типа, но все же обеспечивают их безопасность, лучше использовать шаблон функции вместо макрофункции. Если необходимо увеличить производительность, объявите функцию встраиваемой (inline).

Вы уже познакомились со встраиваемыми функциями и использованием ключевого слова inline в листинге 7.10 на занятии 7, "Организация кода при помощи функций".

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Создавайте собственные макрофункции по возможности реже	**Не забывайте** заключать в скобки каждую переменную в определении макрофункции
Используйте вместо макроса, если возможно, константы	**Не забывайте** использовать в ваших файлах заголовков защиту от множественного включения, используя директивы #ifndef, #define и #endif
Помните, что макрос не обеспечивает безопасности типов и препроцессор никак не контролирует соответствия типов	**Не забывайте** щедро снабжать свой код макросами assert() — в финальной версии они не делают ничего, но способны улучшить качество вашего кода

Пришло время изучать практики обобщенного программирования с использованием шаблонов!

Введение в шаблоны

Шаблоны (template) — это, вероятно, одно из самых мощных средств языка C++, на которое зачастую не обращают внимания или не понимают. Прежде чем заняться этой темой, сначала посмотрим, как определяется термин "шаблон" в словаре Вебстера:

Произношение: \'tem-plət\

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) a: 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 звучит 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 `objectType`, делая его знакоместом 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 m_Obj1;
    T2 m_Obj2;
public:
    T1 GetObj1() {return m_Obj1; }
    // ... 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`.

Различные типы объявлений шаблона

Объявление шаблона может быть таким:

- 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.

Шаблон функции

Вообразите функцию, которая сама приспосабливается к параметрам различных типов. Такая функция 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 objectType>
const objectType& GetMax (const objectType& value1,
                          const objectType& value2)
{
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

Пример применения:

```
int Integer1 = 25;
int Integer2 = 40;
int MaxValue = GetMax <int> (Integer1, Integer2);
double Double1 = 1.1;
double Double2 = 1.001;
double MaxValue = GetMax <double> (Double1, Double2);
```

Обратите внимание на фрагмент <int>, используемый в вызове функции GetMax(). Это фактически определение параметра шаблона objectType как типа int. Приведенный выше код инструктирует 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 шаблоны функции не обязательно нуждаются в соответствующем спецификаторе типа. Так, следующий вызов функции сработает отлично:

```
int MaxValue = GetMax(Integer1, Integer2);
```

Компиляторы достаточно интеллектуальны, чтобы понять, что шаблон функции вызывается для целочисленного типа. Для шаблона класса, однако, необходимо явно указать тип, как представлено в листинге 14.3.

ЛИСТИНГ 14.3. Шаблон функции GetMax, позволяющей выявить наибольшее 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 Int1 = -101, Int2 = 2011;
23:     DisplayComparison(Int1, Int2);
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 в функции `DisplayComparison()` (строки 13–18). Строки 23, 26 и 29 функции `main()` демонстрируют многократное использование того же шаблона функции для совершенно разных типов данных: `int`, `double` и `std::string`. Мало того, что этот шаблон функции используется повторно (точно так же, как и его аналог-макрос), его проще создать и поддерживать, а кроме того, он обеспечивает безопасность типов! Обратите внимание: вполне возможно вызвать функцию `DisplayComparison()` с явным указанием типа:

```
23:     DisplayComparison<int>(Int1, Int2);
```

Однако при вызове шаблона функции это не является обязательным. Вы не обязаны определять тип(ы) параметров шаблона, поскольку компилятор в состоянии вывести его самостоятельно. При применении шаблонов класса, тем не менее, это необходимо.

Шаблоны и безопасность типов

Шаблоны функции `DisplayComparison()` и `GetMax()`, представленные в листинге 14.3, обеспечивают безопасность типов. Это значит, что они не позволят такой, например, бессмысленный вызов:

```
DisplayComparison(Integer, "Some string");
```

Это немедленно привело бы к отказу компиляции.

Шаблон класса

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 реальное представление класса — это его объект. Так, например, `Tom` (Том) можно считать объектом класса `Human` (Человек) с атрибутом `Age` (Возраст), содержащим значение 15. Но что, если вы хотите хранить возраст в атрибуте типа `long long` для тех людей, которые, как ожидается, будут долгожителями, и типа `short` для других? Вот где могли бы пригодиться шаблоны класса. *Шаблон класса* (template class) — это шаблонная версии классов C++. Фактически это чертежи чертежей. При использовании шаблона класса появляется возможность определить *тип*, для которого вы специализируете класс. Это позволяет создать одни объекты класса `Human` с параметром шаблона `Age` типа `long long`, другие — типа `int`, а некоторые — типа `short`.

Простой пример шаблона класса с одним параметром шаблона `T` может быть написан так:

```
template <typename T>
class MyFirstTemplateClass
{
public:
    void SetValue (const T& newValue) { Value = newValue; }
    const T& GetValue() const {return Value;}
private:
    T Value;
};
```

Класс `MyFirstTemplateClass` был разработан для хранения значения переменной типа `T` — типа, который применяется во время использования шаблона. Рассмотрим типичное применение этого шаблона класса:

```
MyFirstTemplateClass <int> HoldInteger; // Создание экземпляра шаблона
HoldInteger.SetValue(5);
std::cout << "The value stored is: " << HoldInteger.GetValue()
          << std::endl;
```

Мы использовали этот шаблон класса для содержания и возвращения объекта типа `int`; т.е. экземпляр шаблона класса создается для параметра шаблона типа `int`. Точно так же можно использовать этот же класс для работы с символьными строками подобным способом:

```
MyFirstTemplateClass <char*> HoldString;
HoldString.SetValue("Sample string");
```

```
std::cout << "The value stored is: " << HoldString.GetValue()
          << std::endl;
```

Таким образом, класс определяет шаблон и многократно использует его для применения того же шаблона, который он реализует для различных типов данных. Настраиваемый класс Human, позволяющий выбрать тип параметра Age, будет выглядеть следующим образом:

```
template <typename T>
class CustomizableHuman
{
public:
    void SetAge (const T& newValue) { Age = newValue; }
    const T& GetAge() const {return Age;}

private:
    T Age; // T — это тип, выбираемый для настройки данного шаблона!
};
```

При использовании этого шаблона вы задаете тип при помощи следующего синтаксиса создания экземпляра шаблона:

```
CustomizableHuman<int> NormalLifeSpan;      // создание экземпляра для
                                            // типа int
    NormalLifeSpan.SetAge(80);
CustomizableHuman<long long> LongLifeSpan; // создание экземпляра для
                                            // типа long long
    LongLifeSpan.SetAge(3147483647);
CustomizableHuman<short> ShortLifeSpan;     // создание экземпляра для
                                            // типа short
    ShortLifeSpan.SetAge(40);
```

Создание и специализация экземпляра шаблона

Когда дело доходит до шаблонов, терминология немного изменяется. Термин *создание экземпляра* (instantiation) при использовании в контексте классов обычно подразумевает получение *объектов классов* (object of class).

В случае шаблонов, однако, *создание экземпляра* — это действие или *процесс* создания специфического типа из объявления шаблона и одного или нескольких аргументов шаблона.

Рассмотрим следующее объявление шаблона:

```
template <typename T>
class TemplateClass
{
    T m_member;
};
```

При использовании этого шаблона код будет выглядеть так:

```
TemplateClass <int> IntTemplate;
```

Получение специфического типа в результате этого создания экземпляра называется *специализацией* (specialization).

Объявление шаблонов с несколькими параметрами

Список параметров шаблона может быть расширен для объявления нескольких параметров, отделенных запятой. Так, если вы хотите объявить обобщенный класс, содержащий два объекта, типы которых могут отличаться, можете использовать конструкции. показанные в следующем примере (где 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& value1, const T2& value2)
    {
        Value1 = value1;
        Value2 = value2;
    };
    // ... Другие объявления функций
};
```

Здесь класс `HoldsPair` получает два параметра шаблона по имени `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 с заданными по умолчанию параметрами

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

```
// Объявление типов по умолчанию для параметров.
// Первый int, второй float
template <typename T1=int, typename T2=double>
class HoldsPair
{
private:
    T1 Value1;
    T2 Value2;
public:
    // Конструктор, инициализирующий переменные-члены
    HoldsPair (const T1& value1, const T2& value2)
    {
        Value1 = value1;
        Value2 = value2;
    };

    // Функции доступа
    const T1 & GetFirstValue () const
    {
        return Value1;
    };

    const T2& GetSecondValue () const
    {
        return Value2;
    };
};

#include <iostream>
using namespace std;

int main ()
{
    // Создание двух экземпляров шаблона HoldsPair
    HoldsPair <> mIntFloatPair (300, 10.09);
    HoldsPair <short,char*> mShortStringPair(25, \
                                    "Learn templates, love C++");

    // Вывод значений, содержащихся в первом объекте...
    cout << "The first object contains -" << endl;
    cout << "Value 1: " << mIntFloatPair.GetFirstValue () << endl;
    cout << "Value 2: " << mIntFloatPair.GetSecondValue () << endl;

    // Вывод значений, содержащихся во втором объекте...
    cout << "The second object contains -" << endl;
    cout << "Value 1: " << mShortStringPair.GetFirstValue () << endl;
    cout << "Value 2: " << mShortStringPair.GetSecondValue ()
         << endl;

    return 0;
}
```

Результат

```
The first object contains -
Value 1: 300
Value 2: 10.09
The second object contains -
Value 1: 25
Value 2: Learn templates, love C++
```

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()` адаптируются для возвращения объектов соответствующих типов на основании синтаксиса создания экземпляра шаблона. Таким образом, удалось определить шаблон `HoldsPair`, который можно многократно использовать для предоставления одинаковой логики обработки переменных различных типов. Следовательно, шаблоны обеспечивают повторное использование кода.

Шаблоны классов и статические члены

Как уже упоминалось, шаблоны — это чертежи классов, которые в свою очередь являются чертежами объектов. Но как функционировали бы в пределах шаблона класса статические атрибуты? На занятии 9, "Классы и объекты", вы узнали, что объявление члена класса статическим позволяет совместно использовать его всем экземплярам этого класса. Это очень похоже на происходящее в шаблоне класса, но его статический член совместно используется всеми экземплярами шаблона класса с той же специализацией. Так, статический член `X` в пределах шаблона класса `T` является статическим в пределах всех экземпляров `T`, специализированных для типа `int`. Подобным же образом член `X` является статическим в пределах всех экземпляров `T`, специализированных для типа `double`, независимо от других, специализированных для типа `int`. Другими словами, вы можете представить это как создание компилятором двух версий: `X_int` — для первого и `X_double` — для второго случая (листинг 14.5).

ЛИСТИНГ 14.5. Результат применения статических переменных в шаблоне класса и его экземплярах

```
0: #include <iostream>
1: using namespace std;
2:
3: template <typename T>
4: class TestStatic
5: {
6: public:
7:     static int StaticValue;
8: };
```

```
 9:
10: // инициализация статического члена
11: template<typename T> int TestStatic<T>::StaticValue;
12:
13: int main()
14: {
15:     TestStatic<int> Int_Year;
16:     cout << "Setting StaticValue for Int_Year to 2011" << endl;
17:     Int_Year.StaticValue = 2011;
18:     TestStatic<int> Int_2;
19:
20:     TestStatic<double> Double_1;
21:     TestStatic<double> Double_2;
22:     cout << "Setting StaticValue for Double_2 to 1011" << endl;
23:     Double_2.StaticValue = 1011;
24:
25:     cout << "Int_2.StaticValue = " << Int_2.StaticValue << endl;
26:     cout << "Double_1.StaticValue = " << Double_1.StaticValue
          << endl;
27:
28:     return 0;
29: }
```

Результат

```
Setting StaticValue for Int_Year to 2011
Setting StaticValue for Double_2 to 1011
Int_2.StaticValue = 2011
Double_1.StaticValue = 1011
```

Анализ

В строках 17 и 21 устанавливаются значения члена `StaticValue` экземпляров шаблона для типов `int` и `double` соответственно. В строках 25 и 26 функции `main()` это значение читается с использованием членов других экземпляров — `Int_2` и `Double_1`. Вывод демонстрирует, что были получены два разных значения `StaticValue` — 2011 установлено с использованием другого экземпляра, специализированного для типа `int`, а значение 1011 установлено с использованием другого экземпляра, специализированного для типа `double`.

Таким образом, компилятор гарантировал неизменность поведения статической переменной при специализации класса для типа. Каждая специализация шаблона класса фактически получает собственную статическую переменную.

ПРИМЕЧАНИЕ

Обратите внимание на синтаксис создания экземпляра статического члена для шаблона класса в строке 11 листинга 14.5.

```
template<typename T> int TestStatic<T>::StaticValue;
```

Он имеет такой формат:

```
template<параметры шаблона> СтатическийТип ИмяКласса<Аргументы Шаблона>::ИмяСтатическойПеременной;
```

C++11

Использование макроса static_assert для проверки во время компиляции

Язык C++11 допускает блочную компиляцию, если определенные проверки не выполняются. Звучит фантастически, но может оказаться весьма полезным с шаблонами классов. Вы могли бы, например, захотеть гарантировать, что экземпляр вашего шаблона класса не будет создан для типа int! Макрос static_assert позволяет отобразить во время компиляции специальное сообщение в вашей среде разработки (или на консоли):

```
static_assert(проверяемое выражение, "Сообщение об ошибке, когда проверка терпит неудачу");
```

Чтобы предотвратить создание экземпляра вашего шаблона класса для типа int, можно использовать макрос static_assert() с оператором sizeof(T), сравнивая его результат с результатом оператора sizeof(int) и отображая сообщение об ошибке, если проверка неравенства терпит неудачу:

```
static_assert(sizeof(T) != sizeof(int), "No int please!");
```

Такой шаблон класса, использующий макрос static_assert для блокирования компиляции при создании экземпляров для определенных типов, представлен в листинге 14.6.

ЛИСТИНГ 14.6. Привередливый шаблон класса, возражающий против создания экземпляра для типа int

```
 0: template <typename T>
 1: class EverythingButInt
 2: {
 3: public:
 4:     EverythingButInt()
 5:     {
 6:         static_assert(sizeof(T) != sizeof(int), "No int please!");
 7:     }
 8: };
 9:
10: int main()
11: {
12:     EverythingButInt<int> test; // создание экземпляра шаблона
                                    // для типа int.
13:     return 0;
14: }
```

Результат

Никакого вывода нет, при ошибке компиляции отображается заданное вами сообщение:

```
error: No int please!
```

Анализ

Возражение, заданное в строке 6, регистрируется компилятором. Таким образом, макрос `static_assert` — это способ, предоставляемый языком C++11 для защиты вашего кода шаблона от нежелательного создания экземпляра.

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. Эти шаблоны классов библиотеки STL позволяют реализовать динамические массивы, списки и контейнеры пар "ключ–значение", в то время как алгоритмы, такие как сортировка, работают с этими контейнерами и обрабатывают данные, которые они содержат.

Знание синтаксиса шаблонов, который вы изучили здесь, очень поможет в использовании контейнеров и функций STL, подробности которых рассматриваются на следующих занятиях. Хорошее понимание контейнеров и алгоритмов библиотеки STL, в свою очередь, поможет вам создавать эффективные приложения C++ с использованием проверенной и надежной реализации библиотеки STL, а также избежать долгих часов разбирательства в деталях шаблонов.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте шаблоны для реализации обобщенных концепций	**Не забывайте** использовать принципы константности при разработке шаблонов функций и классов
Предпочитайте шаблоны макросам	**Не забывайте**, что статический член, содержащийся в шаблоне класса, является статическим для каждой специализации типа класса

Резюме

На сегодняшнем занятии представлено больше подробностей о работе с препроцессором. Каждый раз, когда вы запускаете компилятор, сначала запускается препроцессор, преобразующий такие директивы, как `#define`.

Препроцессор осуществляет лишь текстовую подстановку, хотя при использовании макроса они могут быть достаточно сложными. Макрофункции обеспечивают сложную текстовую подстановку на основании аргументов, переданных макросу во время компиляции. Каждый аргумент в макросе следует помещать в круглые скобки, чтобы гарантировать правильность подстановки.

Шаблоны помогают обеспечить многократное использование кода, который применим для множества типов данных. Они также служат обеспечивающей безопасность типов заменой для макросов. Со знанием шаблонов, полученным на этом занятии, вы готовы к изучению библиотеки STL!

Вопросы и ответы

- **Почему я должен использовать защиту от включения в своих файлах заголовка?**
  Защита от включения с использованием директив `#ifndef`, `#define` и `#endif` защищает ваш заголовок от ошибок множественного или рекурсивного включения, а в некоторых случаях даже ускоряет компиляцию.
- **Почему я должен предпочитать макрофункции шаблонам, если рассматриваемые функциональные возможности могут быть реализованы обоими способами?**
  Как правило, желательно использовать шаблоны, поскольку они, обеспечивая обобщенную реализацию, учитывают безопасность типов. Макрос не учитывает безопасности типов, поэтому его лучше избегать.
- **Должен ли я определять аргументы шаблона при вызове шаблона функции?**
  Обычно нет, поскольку компилятор способен вывести их самостоятельно, по переданным при вызове аргументам функции.
- **Сколько экземпляров статических переменных существует для данного шаблона класса?**
  Все зависит от количества типов, для которых были созданы экземпляры шаблона класса. Так, если были созданы экземпляры для типов `int`, `string` и специального типа `X`, то будут доступны три экземпляра статической переменной — по одной для каждой специализации шаблона.

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 Г, "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 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 шаблон, аналогичный макросу из контрольного вопроса 4.
3. Реализуйте шаблон функции `swap()` для перестановки значений двух переменных.
4. **Отладка**: Как исправить следующий макрос, вычисляющий четверть исходного значения?

```
#define QUARTER(x) (x / 4)
```

5. Напишите простой шаблон класса, содержащий два массива типов, которые определены в списке параметров шаблона класса. Размер массива 10, у шаблона класса должны быть функции доступа, обеспечивающие манипулирование элементами массива.

ЧАСТЬ III

Знакомство со стандартной библиотекой шаблонов (STL)

ЗАНЯТИЕ 15
Введение в стандартную библиотеку шаблонов

Проще говоря, стандартная библиотека шаблонов (STL) — это набор шаблонов классов и функций, который предоставляет программистам следующее.

На сегодняшнем занятии.

- Контейнеры для хранения информации.
- Итераторы для обращения к хранимой информации.
- Алгоритмы для манипулирования содержимым контейнеров.

На этом занятии вы получите общее представление об этих трех основах библиотеки STL.

Контейнеры STL

Контейнеры (container) — это классы библиотеки STL, используемые 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 разновидностью контейнеров, которые имеют ограниченные функциональные возможности и предназначены для специфических целей.

Последовательные контейнеры

Как и подразумевает их название, *последовательные контейнеры* (sequential container) используются для содержания 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 вначале.
- `std::list`. Работает как двухсвязный список. Список похож на цепь, где каждый объект связан со следующим звеном. Вы можете добавить или удалить звено (т.е. объект) в любой позиции.
- `std::forward_list`. Подобен списку `std::list`, но односвязный список позволяет осуществлять перебор только в одном направлении.

Класс `vector` библиотеки STL сродни массиву и обеспечивает произвольный доступ к элементам, т.е. вы можете непосредственно обращаться к элементам вектора по позиции, используя оператор индексирования (`[]`), и манипулировать их данными. Кроме того, вектор STL является динамическим массивом, а потому может изменять свои размеры, чтобы приспособиться к требованиям приложения. Для обеспечения этой возможности при сохранении способности произвольного обращения к элементам массива по позиции большинство реализаций контейнера `vector` библиотеки STL сохраняет все элементы последовательно (т.е. в смежных областях). Поэтому вектор должен часто изменять свои размеры; он может существенно ухудшить производительность приложения в зависимости от типа объектов, которые содержит. Кратко вектор был представлен в листинге 4.4, а более подробно контейнер `vector` обсуждается 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 способен организовать элементы в разделах, состоящих из нескольких участков, несмежных

в 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 двухсвязного и односвязного списков библиотеки STL".

Ассоциативные контейнеры

Ассоциативные контейнеры (associative container), хранящие данные в отсортированном виде, сродни словарю. В результате вставка иногда осуществляется медленнее, но когда дело доходит до поиска, преимущества оказываются существенными.

Библиотека STL предоставляет следующие ассоциативные контейнеры.

- `std::set`. Хранит уникальные значения отсортированными по вставке в контейнере с *логарифмической сложностью*[1] (logarithmic complexity).
- `std::unordered_set`. Хранит уникальные значения отсортированными по вставке в контейнере с *близкой к постоянной сложностью* (near constant complexity). Доступен с версии C++11.
- `std::map`. Хранит пары "ключ–значение" отсортированными по их индивидуальным ключам в контейнере с логарифмической сложностью.
- `std::unordered_map`. Хранит пары "ключ–значение" отсортированными по их индивидуальным ключам в контейнере с близкой к постоянной сложностью. Доступен с версии C++11.
- `std::multiset`. Сродни контейнеру `set`. Дополнительно обеспечивает возможность хранить по несколько элементов с одинаковыми значениями; т.е. значение не обязательно должно быть уникальным.
- `std::unordered_multiset`. Сродни контейнеру `unordered_set`. Дополнительно обеспечивает возможность хранить по несколько элементов с одинаковыми значениями, т.е. значение не обязательно должно быть уникальным.
- `std::multimap`. Сродни контейнеру `map`. Дополнительно обеспечивает возможность хранить пары "ключ–значение" с не уникальными ключами.
- `std::unordered_multimap`. Сродни контейнеру `unordered_map`. Дополнительно обеспечивает возможность хранить пары "ключ–значение" с не уникальными ключами. Доступен с версии C++11.

Критерии сортировки контейнеров STL могут быть настроены программно при помощи функций предиката.

СОВЕТ

Некоторые реализации библиотеки STL предоставляют также такие ассоциативные контейнеры, как `hash_set`, `hash_multiset`, `hash_map` и `hash_multimap`. Они подобны контейнерам `unordered_*`, которые поддерживаются по стандарту. В некоторых сценариях варианты `hash_*` и `unordered_*` могут оказаться лучше при поиске элемента, поскольку они предоставляют операции с постоянным временем выполнения (независимые от количества элементов

[1] Имеется в виду продолжительность работы алгоритма в зависимости от количества данных, содержащихся в контейнере. — *Примеч. ред.*

в контейнере). Как правило, эти контейнеры предоставляют также открытые методы, идентичные таковым в их стандартных аналогах, а следовательно, столь же удобные.

Использование стандартных вариантов приводит к созданию кода, который проще переносить на другие платформы и компилировать, а следовательно, они предпочтительней. Возможно также, что логарифмическое сокращение производительности стандартных контейнеров может не затронуть значительно ваше приложение.

Выбор правильного контейнера

Конечно, требованиям вашего приложения могли бы удовлетворять контейнеры STL нескольких типов. Выбор следует сделать правильно, поскольку неправильный выбор может привести к потере производительности приложения.

Поэтому очень важно оценить все преимущества и недостатки контейнера, прежде чем выбрать его (табл. 15.1).

ТАБЛИЦА 15.1. Свойства контейнерных классов STL

Контейнер	Преимущества	Недостатки
`std::vector` (Последовательный контейнер)	Быстрая (постоянная по продолжительности) вставка в конец	Изменение размеров может привести к потере производительности
	Доступ, как у массива	Время поиска пропорционально количеству элементов в контейнере
`std::deque` (Последовательный контейнер)	Все преимущества контейнера `vector`, а также постоянная по продолжительности вставка в начало контейнера	Недостатки вектора по производительности и поиску относятся также к двухсторонней очереди
		В отличие от вектора, двухсторонняя очередь по спецификации не должна предоставлять функцию `reserve()`, которая резервирует область в памяти, используемую средствами, позволяющими избежать частого изменения размеров что улучшает производительность
`std::list` (Последовательный контейнер)	Постоянная по продолжительности вставка в начало, середину и конец списка	К элементам нельзя обращаться произвольно по индексу, как в массиве
	Постоянное по продолжительности удаление элементов из списка независимо от их позиции	Доступ к элементам может быть медленнее, чем у вектора, поскольку хранятся они не в смежных областях памяти
	Вставка и удаление элементов не влияют на итераторы, которые указывают на другие элементы в списке	Время поиска пропорционально количеству элементов в контейнере

Продолжение табл. 15.1

Контейнер	Преимущества	Недостатки
`std::forward_list` (Последовательный контейнер)	Класс односвязного списка, допускающий итерацию только в одном направлении	Допускает вставку только в начало списка при помощи метода `push_front()`
`std::set` (Ассоциативный контейнер)	Поиск не прямо пропорционален количеству элементов в контейнере, а скорее его логарифму, следовательно, зачастую осуществляется значительно быстрее, чем в последовательных контейнерах	Вставка элементов осуществляется медленнее, чем в последовательных дубликатах, поскольку элементы при вставке сортируются
`std::unordered_set` (Ассоциативный контейнер)	Поиск, вставка и удаление в контейнере этого типа почти не зависят от количества элементов	Поскольку элементы упорядочены слабо, на их относительную позицию в пределах контейнера положиться нельзя
`std::multiset` (Ассоциативный контейнер)	Используется, когда набор должен содержать не уникальные значения	Вставка может осуществляться медленнее, чем в последовательном контейнере, поскольку элементы (пары) сортируются при вставке
`std::unordered_multiset` (Ассоциативный контейнер)	Когда необходимо содержать не уникальные значения, предпочтительней использования контейнера `unordered_set` Производительность, как у контейнера `unordered_set`, а именно: постоянное среднее время поиска, вставки и удаления элементов не зависит от размера контейнера	Поскольку элементы упорядочены слабо, на их относительную позицию в пределах контейнера положиться нельзя
`std::map` (Ассоциативный контейнер)	Контейнер пар "ключ–значение" с повышенной производительностью поиска, пропорциональной логарифму количества элементов в контейнере, а следовательно, зачастую выполняющаяся значительно быстрее, чем в последовательном контейнере	Элементы (пары) сортируются при вставке, следовательно, вставка осуществляется медленнее, чем в последовательном контейнере пар
`std::unordered_map` (Ассоциативный контейнер)	Обеспечивает преимущество близкого к постоянному времени поиска, вставки и удаления элементов, независимое от размера контейнера	Поскольку элементы упорядочены слабо, не подходит для случаев, когда важен порядок
`std::multimap` (Ассоциативный контейнер)	Предпочтительней контейнера `std::map`, когда необходим контейнер пар "ключ–значение", способный содержать элементы с не уникальными ключами	Вставка элементов будет медленнее, чем в последовательном эквиваленте, поскольку элементы сортируются при вставке

Окончание табл. 15.1

Контейнер	Преимущества	Недостатки
`std::unordered_multimap` (Ассоциативный контейнер)	Предпочтительней контейнера `multimap`, когда необходим контейнер пар "ключ–значение", способный содержать элементы с не уникальными ключами Обеспечивает постоянное среднее время вставки, поиска и удаления элементов независимо от размера контейнера	Поскольку элементы упорядочены слабо, на их относительную позицию в пределах контейнера положиться нельзя

Адаптеры контейнеров

Адаптеры контейнеров (container adapter) — это разновидности последовательных и ассоциативных контейнеров с ограниченными функциональными возможностями и предназначенные для специфических целей. Основные классы адаптеров приведены ниже.

- `std::stack`. Элементы хранятся в порядке LIFO (Last-In-First-Out — последним вошел, первым вышел), позволяя вставлять и извлекать элементы с вершины стека.
- `std::queue`. Элементы хранятся 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, чтобы первым в очереди всегда был тот элемент, значение 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 осуществить инкремент этого указателя, и он будет указывать на следующий элемент, позволяя манипулировать элементом в данной области.

Итераторы библиотеки STL — это шаблоны классов, которые до некоторой степени являются обобщением указателей. Такие шаблоны классов предоставляют разработчикам средство, позволяющее работать с контейнерами STL и выполнять операции с ними. Но операции также могут быть алгоритмами STL, которые являются шаблонами функций. Итераторы — своего рода мост, позволяющий шаблонам функций единообразно работать с контейнерами, которые являются шаблонами классов.

Предоставляемые библиотекой STL итераторы можно в основном разделить так.

- *Итератор ввода* (input iterator). Обращение к его значению позволяет получить доступ к объекту. Объект может находиться в коллекции. Классический итератор ввода гарантирует доступ только для чтения.

- *Итератор вывода* (output iterator). Обеспечивает запись в коллекцию. Классический итератор вывода гарантирует доступ только для записи.

Основные типы итераторов, упомянутые в предыдущем списке, далее можно подразделить на следующие.

- *Прямой итератор* (forward iterator). Усовершенствованный итератор ввода и вывода, обеспечивающий как ввод, так и вывод. Прямые итераторы могут быть константными, обеспечивая доступ к объекту, на который указывает итератор, только для чтения, либо нет, обеспечивая операции чтения и записи. Как правило, прямой итератор используется в односвязном списке.
- *Двунаправленный итератор* (bidirectional iterator). Усовершенствованный прямой итератор, допускающий как инкремент, так и декремент, чтобы перемещаться назад. Двунаправленный итератор, как правило, используется в двухсвязном списке.
- *Итератор произвольного доступа* (random access iterator). Концептуально усовершенствованный двунаправленный итератор, допускающий смещение при добавлении и вычитании, а также вычитание одного итератора из другого для поиска разницы между ними или дистанции между двумя объектами в коллекции. Итератор произвольного доступа, как правило, используется в массиве.

ПРИМЕЧАНИЕ

На уровне реализации *усовершенствование* можно считать *наследованием* или *специализацией*.

Алгоритмы STL

Поиск, сортировка, изменение порядка на обратный и тому подобное являются стандартными операциями при программировании, реализацию которых разработчик не должен изобретать заново. Вот почему библиотека STL предоставляет эти функции в форме алгоритмов STL, которые, работая с контейнерами при помощи итераторов, помогают разработчику справиться с некоторыми из наиболее распространенных операций.

Ниже приведены некоторые из наиболее популярных алгоритмов STL.

- `std::find`. Позволяет найти значение в коллекции.
- `std::find_if`. Позволяет найти значение в коллекции на основании определяемого пользователем предиката.
- `std::reverse`. Обращает коллекцию.
- `std::remove_if`. Позволяет удалить элемент из коллекции на основании определяемого пользователем предиката.
- `std::transform`. Позволяет применить пользовательскую функцию преобразования к элементам в контейнере.

Эти алгоритмы представляют собой шаблоны функций из пространства имен `std`. Для применения требуется включить в код стандартный заголовок `<algorithm>`.

Взаимодействие контейнеров и алгоритмов с использованием итераторов

Рассмотрим на примере, как использование итераторов объединяет контейнеры и алгоритмы STL. Программа, представленная в листинге 15.1, использует последовательный контейнер STL `std::vector`, похожий на динамический массив, для сохранения нескольких целых чисел, а затем использует алгоритм `std::find` для поиска одного из них. Обратите внимание, как итераторы объединяют контейнеры и алгоритмы STL. Не обращайте внимания на сложности синтаксиса или функциональных возможностей. Контейнеры, такие как `std::vector`, и алгоритмы, такие как `std::find`, обсуждаются подробно на занятии 17, "Классы динамических массивов библиотеки STL" и занятии 23, "Алгоритмы библиотеки STL" соответственно. Если вы находите эту часть слишком сложной, то можете пропустить ее пока.

ЛИСТИНГ 15.1. Поиск элемента по его позиции в векторе

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main ()
{
    // Динамический массив целых чисел
    vector <int> vecIntegerArray;

    // Вставить примеры целых чисел в массив
    vecIntegerArray.push_back (50);
    vecIntegerArray.push_back (2991);
    vecIntegerArray.push_back (23);
    vecIntegerArray.push_back (9999);

    cout << "The contents of the vector are: " << endl;

    // Перебор вектора и чтение значений с помощью итератора
    vector <int>::iterator iArrayWalker = vecIntegerArray.begin ();

    while (iArrayWalker != vecIntegerArray.end ())
    {
        // Вывод значения на экран
        cout << *iArrayWalker << endl;

        // Инкремент итератора для доступа к следующему элементу
        ++ iArrayWalker;
    }

    // Поиск элемента (скажем, 2991) в массиве с использованием
    // алгоритма 'find'...
```

```
32:     vector <int>::iterator iElement = find (vecIntegerArray.begin()
33:                                       , vecIntegerArray.end (), 2991);
34:
35:     // Проверить, найдено ли значение
36:     if (iElement != vecIntegerArray.end ())
37:     {
38:         // Значение найдено... Определить позицию в массиве:
39:         int Position = distance (vecIntegerArray.begin (), iElement);
40:         cout << "Value "<< *iElement;
41:         cout << " found in the vector at position: " << Position
                 << endl;
42:     }
43:
44:     return 0;
45: }
```

Результат

```
The contents of the vector are:
50
2991
23
9999
Value 2991 found in the vector at position: 1
```

Анализ

В листинге 15.1 показано применение итераторов при переборе вектора и взаимодействие с ним, позволяющее применить такие алгоритмы, как `find`, к таким контейнерам, как `vector`. Объект итератора `iArrayWalker` объявляется в строке 20 и инициализируется началом контейнера, т.е. возвращаемым значением функции-члена `begin()` контейнера `vector`. Строки 22–29 демонстрируют использование этого итератора в цикле для поиска и отображения элементов вектора таким же способом, как и в статическом массиве. Итераторы используются совершенно одинаково со всеми контейнерами STL. Все они предоставляют функцию `begin()`, указывающую на первый элемент, и функцию `end()`, указывающую на конец контейнера *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 — это итератор, а успех ее выполнения проверяется при сравнении итератора с концом контейнера, как можно заметить в строке 36. Если элемент найден, то он может быть отображен при обращении к значению этого итератора (как у указателя). Алгоритм `distance()` применяется при вычислении позиции (смещения) найденного элемента.

Если не глядя заменить в листинге 15.1 все слова `'vector'` словом `'deque'`, его код все еще будет компилироваться и прекрасно работать. Это демонстрация простоты работы итераторов с алгоритмами и контейнерами.

C++11

Использование ключевого слова auto позволяет компилятору определить тип

В листинге 15.1 показано несколько объявлений итератора. Они выглядят подобно этому:

```
20:     vector <int>::iterator iArrayWalker = vecIntegerArray.begin ();
```

Определение типа итератора выглядит пугающе. Если вы используете компилятор, совместимый со стандартом C++11, то можете упростить эту строку до следующей:

```
20:     auto iArrayWalker = vecIntegerArray.begin (); // тип обнаруживает
                                                      // компилятор
```

Обратите внимание, что переменную, объявленную как тип auto, необходимо инициализировать (именно по этому инициализирующему значению компилятор может обнаружить тип).

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 манипуляций с простыми символьными строками.
- std::wstring. Специализация шаблона std::basic_string для типа wchar_t, используемая для манипуляций с широкими символьными строками.

Эти вспомогательные классы подробно обсуждается на занятии 16, "Классы строк библиотеки STL", где вы увидите, насколько они упрощают работу со строками.

Резюме

На сегодняшнем занятии рассматривались такие концепции библиотеки STL, как контейнеры, итераторы и базовые алгоритмы. Вы познакомились с шаблоном basic_string <T>, который подробно обсуждается на следующем занятии. Контейнеры, итераторы и алгоритмы — это одна из самых важных концепций библиотеки STL, и их полное понимание поможет вам эффективно использовать библиотеку STL в своем приложении. Более подробная информация по этим концепциям и применению их реализации приведена на занятиях 17–25.

Вопросы и ответы

- **Мне нужно использовать массив, но неизвестно количество элементов, которые он должен содержать. Какой контейнер STL использовать?**

  Контейнеры `std::vector` и `std::deque` отлично подойдут. Их самостоятельное управление памятью и динамическое масштабирование улучшат приложение.

- **В моем приложении поиск задействуется довольно часто. Какой контейнер выбрать?**

  Такие ассоциативные контейнеры, как `std::map` и `std::set`, или их неупорядоченные варианты лучше всего подойдут для частых поисков.

- **Я должен хранить пары "ключ–значение" для быстрого поиска. Но может случиться так, что ключи будут не уникальны. Какой контейнер выбрать?**

  Подойдет ассоциативный контейнер типа `std::multimap`. Контейнер `multimap` может содержать не уникальные пары "ключ–значение" и может обеспечить быстрый поиск, характерный для ассоциативных контейнеров.

- **Приложение предназначено для разных платформ и компиляторов. Необходим контейнер с быстрым поиском на основании ключа. Я должен использовать контейнер `std::map` или `std::hash_map`?**

  Переносимость — важный ограничивающий фактор, поэтому необходимо использовать стандартные контейнеры. 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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Какой бы контейнер вы выбрали, 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?
3. Необходимо хранить элементы в контейнере `std::set`, но необходима еще возможность изменения критериев поиска на основании условия, которое не обязательно связано со значением элементов. Возможно ли это?
4. Какая часть библиотеки STL позволяет объединить алгоритмы с контейнерами, чтобы они могли воздействовать на их элементы?
5. Выбрали бы вы контейнер `hash_set` для приложения с возможностью перенесения на различные платформы и компиляции на разных компиляторах C++?

ЗАНЯТИЕ 16
Классы строк библиотеки STL

Стандартная библиотека шаблонов (STL) предоставляет программистам контейнерный класс, облегчающий операции и манипулирование со строками. Класс `string` не только динамически изменяет свои размеры, чтобы удовлетворить 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 сегодняшнем 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 основанную на шаблоне реализацию строк библиотеки STL.

Потребность в классах обработки строк

Строка в языке C++ — это массив символов. Как вы видели на занятии 4, "Массивы и строки", простейший символьный массив может быть определен следующим образом:

```
char staticName [20];
```

Здесь объявляется символьный массив (называемый также строкой) фиксированной (а значит, статический) длины в 20 элементов. Как видно, этот массив может содержать строку ограниченной длины, он оказался бы переполнен при попытке сохранить в нем больше символов. Изменение размеров такого статического массива невозможно. Для преодоления этого ограничения язык C++ предоставляет динамическое распределение памяти для данных. Вот более динамическое представление строкового массива:

```
char* dynamicName = new char [ArrayLength];
```

Это динамически распределенный символьный массив, длина экземпляра которого может быть задана при создании значением переменной `ArrayLength`, определяемым во время выполнения, а следовательно, способный содержать данные переменной длины. Но если понадобится изменить длину массива во время выполнения, то придется сначала освободить распределенную память, а затем повторно зарезервировать ее для содержания необходимых данных.

Все усложняется, если такие символьные строки используются как атрибуты класса. В ситуациях, когда объект этого класса присваивается другому, при отсутствии грамотно созданного конструктора копий и оператора присваивания, эти два объекта будут содержать копии указателя, по существу указывающего на тот же строковый буфер. В результате два строковых указателя в двух объектах будут содержать одинаковый адрес, а следовательно, указывать на ту же область в памяти. В результате удаления первого объекта указатель в другом объекте оказывается недействительным и на горизонте вырисовывается приближающийся аварийный отказ.

Строковые классы решают эти проблемы самостоятельно. Строковый класс `std::string` библиотеки STL моделирует символьную строку, а класс `std::wstring` — широкую символьную строку, помогая вам следующими способами.

- Сокращает усилия по созданию и манипулированию строками.
- Увеличивает стабильность приложения за счет инкапсуляции подробностей распределения памяти.
- Встроенный конструктор копий и оператор присвоения 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 того же шаблона класса `std::basic_string<T>` для таких типов, как `char` и `wchar_t`, соответственно. Когда вы изучите его использование подробней, вы сможете использовать те же методы и операторы для других типов.

Давайте на примере класса `std::string` изучим некоторые из вспомогательных функций, предоставляемых строковыми классами библиотеки STL.

Работа с классами строк библиотеки 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 заголовок `<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, объект класса `std::string` можно просто инициализировать строкой или присвоить ему постоянный символьный строковый литерал:

```
const char* constCStyleString = "Hello String!";
std::string strFromConst (constCStyleString);
```

или

```
std::string strFromConst = constCStyleString;
```

Приведенное выше очень похоже на следующее:

```
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` сделал 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` принять только *n* первых символов передаваемой исходной строки:

```
// Инициализировать строку первыми 5 символами другой строки
std::string strPartialCopy (constCStyleString, 5);
```

А также инициализировать строку определенным количеством специфического символа:

```
// Инициализировать строку 10 символами 'a'
std::string strRepeatChars (10, 'a');
```

Листинг 16.1 анализирует некоторые наиболее популярные способы создания экземпляров класса `std::string` и копирования строк.

ЛИСТИНГ 16.1. Способы создания экземпляров строк STL и их копирования

```
 0: #include <string>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:     const char* constCStyleString = "Hello String!";
 7:     cout << "Constant string is: " << constCStyleString << endl;
 8:
 9:     std::string strFromConst (constCStyleString); // Конструктор
10:     cout << "strFromConst is: " << strFromConst << endl;
11:
12:     std::string str2 ("Hello String!");
13:     std::string str2Copy (str2);
14:     cout << "str2Copy is: " << str2Copy << endl;
15:
16:     // Инициализировать строку первыми 5 символами другой строки
17:     std::string strPartialCopy (constCStyleString, 5);
18:     cout << "strPartialCopy is: " << strPartialCopy << endl;
19:
20:     // Инициализировать строку 10 символами 'a'
21:     std::string strRepeatChars (10, 'a');
22:     cout << "strRepeatChars is: " << strRepeatChars << endl;
23:
24:     return 0;
25: }
```

Результат

```
Constant string is: Hello String!
strFromConst is: Hello String!
str2Copy is: Hello String!
strPartialCopy is: Hello
strRepeatChars is: 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 содержит пример значения, заданного в строке 6. Строка 9 демонстрирует, насколько просто конструктор класса `std::string` позволяет создать его копию. 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 оператор str2Copy. 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 конструкторы копий 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 будет таким:

```
const char* constCStyleString = "Hello World!";
// Чтобы создать копию, сначала зарезервировать память для нее...
char * pszCopy = new char [strlen (constCStyleString) + 1];
strcpy (pszCopy, constCStyleString); // Этап копирования

// Освобождение памяти после использования pszCopy
delete [] pszCopy;
```

Как можно заметить, здесь больше строк кода и выше вероятность ошибки. Кроме того, необходимо позаботиться об управлении памятью и ее освобождении. Классы строк библиотеки STL делает все это сами и еще больше!

Доступ к символу в строке класса std::string

К символьному содержимому строки STL можно обратиться при помощи итератора или подобного массиву синтаксиса, где смещение задается с использованием оператора индексирования []. Представление строки в стиле C может быть получено при помощи функции-члена c_str() (листинг 16.2).

ЛИСТИНГ 16.2. Два способа обращения к символу строки STL: оператор ([]) и итератор

```
 0: #include <string>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     // Пример строки
 8:     string strSTLString ("Hello String");
 9:
10:     // Доступ к содержимому строки с использованием
       // синтаксиса массива
11:     cout<<"Displaying the elements in the string using array-syntax:"
            << endl;
12:     for ( size_t nCharCounter = 0
13:         ; nCharCounter < strSTLString.length ()
14:         ; ++ nCharCounter )
15:     {
16:         cout << "Character [" << nCharCounter << "] is: ";
17:         cout << strSTLString [nCharCounter] << endl;
```

```
18:     }
19:     cout << endl;
20:
21:     // Доступ к содержимому строки с использованием итератора
22:     cout << "Displaying the contents of the string using iterators: "
             << endl;
23:     int charOffset = 0;
24:     string::const_iterator iCharacterLocator;
25:     for ( iCharacterLocator = strSTLString.begin ()
26:         ; iCharacterLocator != strSTLString.end ()
27:         ; ++ iCharacterLocator )
28:     {
29:         cout << "Character [" << charOffset ++ << "] is: ";
30:         cout << *iCharacterLocator << endl;
31:     }
32:     cout << endl;
33:
34:     // Обращение к содержимому строки в стиле C
35:     cout << "The char* representation of the string is: ";
36:     cout << strSTLString.c_str () << endl;
37:
38:     return 0;
39: }
```

Результат

```
Displaying the elements in the string using array-syntax:
Character [0] is: H
Character [1] is: e
Character [2] is: l
Character [3] is: l
Character [4] is: o
Character [5] is:
Character [6] is: S
Character [7] is: t
Character [8] is: r
Character [9] is: i
Character [10] is: n
Character [11] is: g

Displaying the contents of the string using iterators:
Character [0] is: H
Character [1] is: e
Character [2] is: l
Character [3] is: l
Character [4] is: o
Character [5] is:
Character [6] is: S
Character [7] is: t
Character [8] is: r
Character [9] is: i
Character [10] is: n
Character [11] is: g
The char* representation of the string is: Hello String
```

Анализ

Код отображает несколько способов обращения к содержимому строки. Итераторы важны в том смысле, что большинство функций-членов класса `string` возвращают свои результаты в форме итераторов. Строки 12–18 отображают символы строки с использованием реализованного классом `std::string` оператора индексирования `[]`, как у массива. Обратите внимание, что этому оператору нужно предоставить смещение, как можно заметить в строке 17. Поэтому очень важно не пересечь границы строки, т.е. вы не должны читать символы по смещению больше длины строки. Строки 25–31 также отображают содержимое посимвольной строки, но с использованием итератора.

Конкатенация строк

Конкатенация строк может быть осуществлена либо при помощи оператора +=, либо функции-члена `append()`:

```
string strSample1 ("Hello");
string strSample2 (" String!");
strSample1 += strSample2; // использование std::string::operator+=
// альтернативно используется std::string::append()
strSample1.append (strSample2); // (перегружен также для char*)
```

Листинг 16.3 демонстрирует применение этих двух вариантов.

ЛИСТИНГ 16.3. Конкатенация строк с использованием оператора сложения с присвоением (+=) или метода `append()`

```
 0: #include <string>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     string strSample1 ("Hello");
 8:     string strSample2 (" String!");
 9:
10:     // Конкатенация
11:     strSample1 += strSample2;
12:     cout << strSample1 << endl << endl;
13:
14:     string strSample3 (" Fun is not needing to use pointers!");
15:     strSample1.append (strSample3);
16:     cout << strSample1 << endl << endl;
17:
18:     const char* constCStyleString = " You however still can!";
19:     strSample1.append (constCStyleString);
20:     cout << strSample1 << endl;
21:
22:     return 0;
23: }
```

Результат

```
Hello String!

Hello String! Fun is not needing to use pointers!

Hello String! Fun is not needing to use pointers! You however still can!
```

Анализ

Строки 11, 15 и 19 отображают различные способы конкатенации строк STL. Обратите внимание на использование оператора += и возможность функции `append()`, у которой есть множество перегруженных версий, получать другие строковые объекты (как показано в строке 11) и символьные строки в стиле C.

Поиск символа или подстроки в строке

Класс `string` библиотеки STL предоставляет функцию-член `find()` в нескольких перегруженных версиях. Она позволяет найти символ или подстроку в данном объекте класса `string`.

```
// Найти подстроку "day" в строке strSample, начиная с позиции 0
size_t charPos = strSample.find ("day", 0);

// Удостовериться, что подстрока найдена, сравнив со string::npos
if (charPos != string::npos)
    cout << "First instance of \"day\" was found at position "
         << charPos;
else
    cout << "Substring not found." << endl;
```

Листинг 16.4 демонстрирует удобство 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 strSample ("Good day String! Today is beautiful!");
 8:     cout << "The sample string is: " << endl;
 9:     cout << strSample << endl << endl;
10:
11:     // Найти в ней подстроку "day"...
12:     size_t charPos = strSample.find ("day", 0);
13:
14:     // Удостовериться, что подстрока найдена...
15:     if (charPos != string::npos)
16:         cout << "First instance of \"day\" was found at position "
              << charPos;
```

```
17:     else
18:         cout << "Substring not found." << endl;
19:
20:     cout << endl << endl;
21:
22:     cout << "Locating all instances of substring \"day\"" << endl;
23:     size_t SubstringPos = strSample.find ("day", 0);
24:
25:     while (SubstringPos != string::npos)
26:     {
27:         cout << "\"day\" found at position " << SubstringPos << endl;
28:
29:         // Продолжить поиск вперед, от следующего символа
30:         size_t nSearchPosition = SubstringPos + 1;
31:
32:         SubstringPos = strSample.find ("day", nSearchPosition);
33:     }
34:
35:     cout << endl;
36:
37:     cout << "Locating all instances of character 'a'" << endl;
38:     const char charToSearch = 'a';
39:     charPos = strSample.find (charToSearch, 0);
40:
41:     while (charPos != string::npos)
42:     {
43:         cout << "'" << charToSearch << "' found";
44:         cout << " at position: " << charPos << endl;
45:
46:         // Продолжить поиск вперед, от следующего символа
47:         size_t charSearchPos = charPos + 1;
48:
49:         charPos = strSample.find (charToSearch, charSearchPos);
50:     }
51:
52:     return 0;
53: }
```

Результат

```
The sample string is:
Good day String! Today is beautiful!

First instance of "day" was found at position 5

Locating all instances of substring "day"
"day" found at position 5
"day" found at position 19

Locating all instances of character 'a'
'a' found at position: 6
'a' found at position: 20
'a' found at position: 28
```

Анализ

Строки 12–18 демонстрируют самый простой случай применения функции `find()` — поиск в строке специфической подстроки. Для этого результат метода `find()` сравнивается со значением `std::string::npos` (фактически -1) и выясняет, что разыскиваемый элемент не найден. Когда функция `find()` не возвращает `npos`, она возвращает смещение, указывающее позицию подстроки или символа в строке.

Код далее демонстрирует применение функции `find()` в цикле `while` для поиска всех экземпляров символа или подстроки в строке STL. Здесь используется перегруженная версия функции `find()`, получающая два параметра: искомую подстроку или символ и смещение поиска, означающее точку, начиная с которой осуществляется поиск.

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()`, обеспечивающие разработчику дополнительные возможности.

Усечение строк STL

Класс `string` библиотеки STL предоставляет функцию-член `erase()`, осуществляющую удаление:

- набора символов, когда даны позиция смещения и количество удаляемых символов.

```
string strSample ("Hello String! Wake up to a beautiful day!");
strSample.erase (13, 28); // Hello String!
```

- отдельного символа, когда дан указывающий на него итератор.

```
strSample.erase (iCharS); // итератор указывает на определенный символ
```

- набора символов, диапазон которых задан двумя итераторами.

```
// удалить от начала до конца
strSample.erase (strSample.begin(), strSample.end());
```

Пример в листинге 16.5 демонстрирует различные способы применения перегруженных версий функции `string::erase()`.

ЛИСТИНГ 16.5. Использование функции `string::erase()` для усечения строки, начиная с позиции, заданной смещением или итератором

```
 0: #include <string>
 1: #include <algorithm>
 2: #include <iostream>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:
 8:     string strSample ("Hello String! Wake up to a beautiful day!");
 9:     cout << "The original sample string is: " << endl;
10:     cout << strSample << endl << endl;
11:
```

```
12:     // Удалить из строки символы, заданные позицией и количеством
13:     cout << "Truncating the second sentence: " << endl;
14:     strSample.erase (13, 28);
15:     cout << strSample << endl << endl;
16:
17:     // Найти в строке символ 'S', используя алгоритм поиска STL
18:     string::iterator iCharS = find ( strSample.begin ()
19:                                      , strSample.end (), 'S');
20:
21:     // Если символ найден, удалить его
22:     cout << "Erasing character 'S' from the sample string:" << endl;
23:     if (iCharS != strSample.end ())
24:         strSample.erase (iCharS);
25:
26:     cout << strSample << endl << endl;
27:
28:     // Удалить диапазон символов, используя перегруженную
        // версию функции erase()
29:     cout << "Erasing a range between begin() and end(): " << endl;
30:     strSample.erase (strSample.begin (), strSample.end ());
31:
32:     // Проверить длину после операции erase() выше
33:     if (strSample.length () == 0)
34:         cout << "The string is empty" << endl;
35:
36:     return 0;
37: }
```

Результат

```
The original sample string is:
Hello String! Wake up to a beautiful day!

Truncating the second sentence:
Hello String!

Erasing character 'S' from the sample string:
Hello tring!

Erasing a range between begin() and end():
The string is empty
```

Анализ

Листинг демонстрирует три версии функции `erase()`. Одна версия удаляет набор символов, заданных начальным смещением и количеством, как показано в строке 14. Другая 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 предоставляют функции-члены `begin()` и `end()` класса `string`, который фактически включает все содержимое строки, вызов метода `erase()` для этого диапазона очищает содержимое объекта строки. Обратите внимание: класс `string` предоставляет также функцию `clear()`, которая фактически очищает внутренний буфер и переустанавливает объект класса `string`.

C++11

Упрощение объявления итератора с использованием ключевого слова auto

Стандарт C++11 позволяет упростить пространное объявление итератора, такое, как представлено в листинге 16.5:

```
18:     string::iterator iCharS = find ( strSample.begin ()
19:                                    , strSample.end (), 'S');
```

Чтобы сократить его, используйте ключевое слово `auto`, как было продемонстрировано на занятии 3, "Использование переменных, объявление констант":

```
auto iCharS = find ( strSample.begin ()
                   , strSample.end (), 'S');
```

Компилятор автоматически выводит тип переменной `iCharS`, получая информацию о типе возвращаемого значения от функции `std::find`.

Обращение строки

Иногда необходимо изменить содержимое строки на обратное. Предположим, необходимо определить, не является ли введенная пользователем строка палиндромом[1]. Один из способов сделать это подразумевает изменение копии содержимого строки на обратное и последующее сравнение с оригиналом. Обобщенный алгоритм `std::reverse()` библиотеки STL позволяет обратить содержимое строки:

```
string strSample ("Hello String! We will reverse you!");
reverse (strSample.begin (), strSample.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 strSample ("Hello String! We will reverse you!");
 9:     cout << "The original sample string is: " << endl;
10:     cout << strSample << endl << endl;
11:
12:     reverse (strSample.begin (), strSample.end ());
13:
14:     cout << "After applying the std::reverse algorithm: " << endl;
15:     cout << strSample << endl;
```

[1] А роза упала на лапу Азора. — *Примеч. ред.*

```
16:
17:     return 0;
18: }
```

Результат

```
The original sample string is:
Hello String! We will reverse you!

After applying the std::reverse algorithm:
!uoy esrever lliw eW !gnirtS olleH
```

Анализ

Алгоритм `std::reverse()`, используемый в строке 12, работает с границами контейнера, переданными двумя входными параметрами. В данном случае эти границы — начало и конец строкового объекта, что обращает содержимое всей строки. Строку можно обратить и по частям, задав соответствующие границы. Обратите внимание: границы никогда 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 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 << "Please enter a string for case-convertion:" << endl;
 9:     cout << "> ";
10:
11:     string strInput;
12:     getline (cin, strInput);
13:     cout << endl;
14:
15:     transform(strInput.begin(),strInput.end(),strInput.begin(),\
              toupper);
16:     cout << "The string converted to upper case is: " << endl;
17:     cout << strInput << endl << endl;
18:
19:     transform(strInput.begin(),strInput.end(),strInput.begin(),\
              tolower);
```

```
20:     cout << "The string converted to lower case is: " << endl;
21:     cout << strInput << endl << endl;
22:
23:     return 0;
24: }
```

Результат

```
Please enter a string for case-convertion:
> ConverT thIS StrINg!

The string converted to upper case is:
CONVERT THIS STRING!

The string converted to lower case is:
convert this string!
```

Анализ

Строки 15 и 19 демонстрируют, насколько эффективно применяется алгоритм `std::transform()` для изменения регистра содержимого строки.

Реализация строки на базе шаблона STL

Как уже упоминалось, класс `std::string` является фактически специализацией шаблона 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`. Следовательно, 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 так:

```
typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t> >
    string;
```

Таким образом, все возможности и средства строк, рассмотренные до сих пор, фактически предоставлены шаблоном `basic_string`, а потому имеются также у класса `wstring`.

СОВЕТ

Используйте класс `std::wstring` для приложений, которые должны поддерживать нелатинские символы, такие как в японском или китайском языке.

Резюме

На сегодняшнем занятии рассматривался класс `string` библиотеки STL. Это предоставляемый стандартной библиотекой шаблонов контейнер, обеспечивающий разработчику множество возможностей по обработке строк. Данный класс предоставляет вполне очевидные преимущества реализации управления памятью, сравнения строк и функций манипулирования строками.

Вопросы и ответы

- **Я должен обратить строку, используя алгоритм `std::reverse()`. Какой заголовок следует включить, чтобы использовать эту функцию?**

  Чтобы функция `std::reverse()` стала доступной, следует включить заголовок `<algorithm>`.
- **Какую роль играет алгоритм `std::transform()` в преобразовании символов строки в нижний регистр при использовании функции `tolower()`?**

  Функция `std::transform()` вызывает функцию `tolower()` для символов объекта класса `string` в пределах границ, переданных функции преобразования.
- **Почему классы `std::wstring` и `std::string` демонстрируют одинаковое поведение и функции-члены?**

  Оба они являются специализацией шаблона класса `std::basic_string`.
- **Чувствительны ли к регистру операторы сравнения строковых классов библиотеки STL?**

  Результаты сравнения зависят от регистра символов.

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 Г, "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. Если бы понадобилось выполнить независящее от регистра сравнение двух строк, то как это сделать?
3. Действительно ли строки STL и строки стиля C подобны?

Упражнения

1. Напишите программу проверки введенных пользователем слов на палиндром. Например: слово ATOYOTA — палиндром, поскольку при обращении оно не изменяется.
2. Напишите программу, сообщающую пользователю количество гласных в предложении.
3. Преобразуйте каждый символ строки в верхний регистр.
4. У вашей программы должно быть четыре строковых объекта, инициализированных как "I", "Love", "STL" и "String". Добавьте к ним промежуточные пробелы и отобразите предложение.

ЗАНЯТИЕ 17
Классы динамических массивов библиотеки STL

В отличие от статических массивов, динамические массивы предоставляют большую гибкость, позволяя хранить данные, даже если неизвестен их точный объем во время разработки приложения. Естественно, это очень распространенное требование, и стандартная библиотека шаблонов (STL) предоставляет готовое к применению решение в форме класса `std::vector`.

На сегодняшнем занятии.

- Характеристики класса `std::vector`.
- Типичные операции с вектором.
- Концепция размера вектора и его емкости.
- Класс `deque` библиотеки STL.

Характеристики класса std::vector

Шаблон класса vector предоставляет обобщенные функциональные возможности динамического массива и характеризуется следующим.

- Продолжительность операции добавления элемента в конец массива постоянна, т.е. она не зависит от размера массива. То же относится к извлечению элемента.
- Продолжительность операции вставки или удаления элементов в середину массива прямо пропорциональна количеству элементов позади этого элемента.
- Количество содержащихся элементов является динамическим, класс vector сам контролирует использование памяти.

Вектор (vector) — это динамический массив, который может быть представлен, как на рис. 17.1.

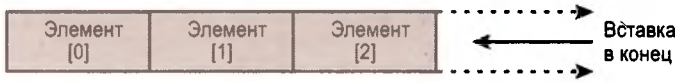


РИС. 17.1. Внутренняя организация вектора

СОВЕТ

Чтобы использовать класс std::vector, включите его заголовок:

```
#include <vector>
```

Типичные операции с вектором

Открытые методы и члены класса std::vector определены стандартом C++. Следовательно, операции с вектором, рассматриваемые на этом занятии, поддерживаются большинством платформ программирования C++, совместимых со стандартом.

Создание экземпляра вектора

Экземпляры шаблона класса vector создаются в соответствии с методиками, описанными на занятии 14, "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 так:

```
std::vector<int> vecDynamicIntegerArray; // вектор для целых чисел
std::vector<float> vecDynamicFloatArray; // вектор для floats
std::vector<Tuna> vecDynamicTunaArray;   // вектор для Tuna
```

Для объявления итератора, указывающего на элементы вектора, используется код

```
std::vector<int>::const_iterator iElementInSet;
```

Если необходим итератор для изменения значений или вызова не константных функций, используйте ключевое слово iterator вместо const_iterator.

Поскольку класс std::vector имеет несколько перегруженных конструкторов, его экземпляр можно создать, 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`: определение размера, исходных значений и копирование значения из другого вектора

```
1: #include <vector>
2:
3: int main ()
4: {
5:     std::vector <int> vecIntegers;
6:
7:     // Создание экземпляра вектора с 10 элементами (впоследствии
       // он может стать больше)
8:     std::vector <int> vecWithTenElements (10);
9:
10:    // Создание экземпляра вектора с 10 элементами, каждый из
       // которых инициализирован значением 90
11:    std::vector <int> vecWithTenInitializedElements (10, 90);
12:
13:    // Создание экземпляра одного вектора и инициализация его
       // содержимым другого
14:    std::vector <int> vecArrayCopy (vecWithTenInitializedElements);
15:
16:    // Использование итераторов для создания экземпляра вектора
       // из 5 элементов другого
17:    std::vector <int> vecSomeElementsCopied ( \
                     vecWithTenElements.cbegin ()
18:                , vecWithTenElements.cbegin () + 5 );
19:
20:    return 0;
21: }
```

Анализ

В приведенном выше коде используется специализация шаблона класса `vector` для типа `int`; другими словами, создается экземпляр вектора целых чисел. Этот вектор по имени `vecIntegers` использует стандартный конструктор, весьма полезный при неизвестном минимальном размере контейнера, т.е. когда вы не знаете, сколько целых чисел предстоит содержать в нем. Вторая и третья формы создания экземпляра вектора расположены в строках 10 и 13, когда разработчику известно, что он нуждается в векторе, способном содержать, по крайней мере, 10 элементов. Обратите внимание, что окончательный размер контейнера это не ограничивает, а только устанавливает начальный размер. Четвертая форма в строках 16 и 17 используется для создания экземпляра вектора из содержимого другого вектора, другими словами, чтобы создать объект класса `vector`, являющийся копией или частью другого. Эта конструкция работает со всеми контейнерами библиотеки STL. Последняя форма использует итераторы. Вектор `vecSomeElementsCopied` содержит первые пять элементов вектора `vecWithTenElements`.

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 бы один из них был вектором, скажем, для типа `float`, код не компилировался бы.

СОВЕТ

Не будет ли ошибки компиляции при использовании методов cbegin() и cend()?

Если вы попытаетесь откомпилировать эту программу, используя компилятор, не совместимый со стандартом C++11, задействуйте методы `begin()` и `end()` вместо `cbegin()` и `cend()` соответственно.

Методы `cbegin()` и `cend()` отличаются (в лучшую сторону) тем, что они возвращают итератор, но не поддерживаются устаревшими компиляторами.

Вставка элементов в конец с использованием метода 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> vecIntegers; // объявление вектора для типа int

// Вставка целых чисел в вектор:
vecIntegers.push_back (50);
vecIntegers.push_back (1);
```

Листинг 17.2 демонстрирует использование метода `push_back()` для динамического добавления элементов в вектор `std::vector`.

ЛИСТИНГ 17.2. Вставка элементов в вектор с использованием метода `push_back()`

```
0: #include <iostream>
1: #include <vector>
2: using namespace std;
3:
4: int main ()
5: {
6: vector <int> vecIntegers;
7:
8: // Вставка целых чисел в вектор:
9: vecIntegers.push_back (50);
10: vecIntegers.push_back (1);
11: vecIntegers.push_back (987);
12: vecIntegers.push_back (1001);
13:
14: cout << "The vector contains ";
15: cout << vecIntegers.size () << " Elements" << endl;
16:
17: return 0;
18: }
```

Результат

```
The vector contains 4 Elements
```

Анализ

Метод `push_back()`, строки 9–12, является открытым членом класса `vector`, вставляющим объекты в конец динамического массива. Обратите внимание на использование функции `size()`, которая возвращает количество элементов, содержащихся в векторе.

C++11

Списки инициализации

Язык C++11 предоставляет класс списков инициализации `std::initialize_list<>`, позволяющий создать экземпляр вектора и инициализировать его элементы, как будто это статический массив:

```
vector<int> vecIntegers = {50, 1, 987, 1001};
// альтернатива:
vector<int> vecMoreIntegers {50, 1, 987, 1001};
```

Этот синтаксис сократил бы три строки листинга 17.2. Но все же мы не использовали его, поскольку на момент написания этой книги компилятор Microsoft Visual C++ 2010 не поддерживал списки инициализации для реализации класса `std::vector`.

Вставка элементов в определенную позицию с использованием метода `insert()`

Метод `push_back()` позволяет вставить элементы в конец вектора. Но что если нужно вставить элемент в середину? 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
vecIntegers.insert (vecIntegers.begin (), 25);
```

Другая позволяет определить позицию и количество элементов со значением, которое должно быть вставлено:

```
// Вставить в конец 2 числа со значением 45
vecIntegers.insert (vecIntegers.end (), 2, 45);
```

Вы можете также вставить содержимое одного вектора в выбранную позицию другого:

```
// Другой вектор, содержащий два элемента со значением 30
vector <int> vecAnother (2, 30);

// Вставить два элемента из другого контейнера в позицию [1]
vecIntegers.insert (vecIntegers.begin () + 1,
                    vecAnother.begin (), vecAnother.end ());
```

Для указании функции `insert()` позиции вставки новых элементов, как правило, используется итератор, возвращаемый функцией `begin()` или `end()`.

СОВЕТ

Этот итератор может быть также возвращен таким алгоритмом STL, как `std::find()`, применяемым для поиска элемента и последующей вставки другого в эту позицию (вставка сдвинет найденный элемент).

Эти форма метода `vector::insert()` представлена в листинге 17.3.

ЛИСТИНГ 17.3. Использование функции `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>& vecInput)
 5: {
 6:     for (auto iElement = vecInput.cbegin() // auto и cbegin(): C++11
 7:             ; iElement != vecInput.cend() // cend() только C++11
 8:             ; ++ iElement )
 9:         cout << *iElement << ' ';
10:
11:     cout << endl;
12: }
13:
14: int main ()
15: {
16:     // Создать экземпляр вектора с 4 элементами,
       // инициализированными значением 90
17:     vector <int> vecIntegers (4, 90);
18:
19:     cout << "The initial contents of the vector: ";
20:     DisplayVector(vecIntegers);
21:
22:     // Вставить 25 в начало
23:     vecIntegers.insert (vecIntegers.begin (), 25);
24:
25:     // Вставить в конец 2 числа со значением 45
26:     vecIntegers.insert (vecIntegers.end (), 2, 45);
27:
28:     cout << "Vector after inserting elements at beginning and end: ";
29:     DisplayVector(vecIntegers);
30:
31:     // Другой вектор, содержащий два элемента со значением 30
32:     vector <int> vecAnother (2, 30);
33:
34:     // Вставить два элемента из другого контейнера в позицию [1]
35:     vecIntegers.insert (vecIntegers.begin () + 1,
36:         vecAnother.begin (), vecAnother.end ());
37:
38:     cout << "Vector after inserting contents from another vector: ";
39:     cout << "in the middle:" << endl;
40:     DisplayVector(vecIntegers);
41:
42:     return 0;
43: }
```

Результат

```
The initial contents of the vector: 90 90 90 90
Vector after inserting elements at beginning and end: 25 90 90 90 90 45 45
Vector after inserting contents from another vector: in the middle:
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 за отправную точку, 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 предпочтительней использовать метод `push_back()`.

Обратите внимание, что метод `insert()` — неэффективный способ 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, содержавшихся в последовательности, продолжительность этой операции сдвига может оказаться существенной с точки зрения вызова конструктора копий или оператора присвоения копии. В данном примере вектор содержит объекты типа `int`, перемещение которых осуществляется относительно быстро. Однако в других случаях все могло бы быть вовсе не так.

СОВЕТ

Если у вашего контейнера должны быть очень частые вставки в середину, имеет смысл использовать класс `std::list`, рассматриваемый на занятии 18, "Классы двухсвязного и односвязного списков библиотеки STL".

ВНИМАНИЕ!

Используете ли вы устаревший компилятор C++?

Функция `DisplayVector()` в листинге 17.3 использует ключевое слово C++11 `auto` для определения типа итератора в строке 6. В этом примере и далее для компиляции с использованием компилятора, не совместимого со стандартом C++11, необходимо заменить ключевое слово `auto` явным типом, в данном случае типом `vector<int>::const_iterator`.

Так, функция `DisplayVector()` для устаревшего компилятора должна быть изменена следующим образом:

```
// для старых компиляторов C++
void DisplayVector(const vector<int>& vecInput)
{
    for (vector<int>::const_iterator iElement = vecInput.begin
    ()
            ; iElement != vecInput.end ()
            ; ++ iElement )
        cout << *iElement << ' ';

    cout << endl;
}
```

Доступ к элементам вектора с использованием семантики массива

К элементам вектора можно обратиться, используя семантику массива с оператором индексирования ([ ]), функцию-член `at()` или итераторы.

В листинге 17.1 показано создание экземпляра вектора, который может быть инициализирован 10 элементами:

```
std::vector <int> vecArrayWithTenElements (10);
```

К индивидуальным элементам вектора можно обратиться, используя синтаксис, как у массива:

```
vecArrayWithTenElements[3] = 2011; // задать 4-й элемент
```

Листинг 17.4 демонстрирует обращение к элементам вектора с использованием оператора индексирования ([ ]).

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> vecIntegerArray;
 7:
 8:     // Вставить в вектор целые числа:
 9:     vecIntegerArray.push_back (50);
10:     vecIntegerArray.push_back (1);
11:     vecIntegerArray.push_back (987);
12:     vecIntegerArray.push_back (1001);
13:
14:     for (size_t Index = 0; Index < vecIntegerArray.size (); ++Index)
15:     {
16:         cout << "Element[" << Index << "] = " ;
17:         cout << vecIntegerArray[Index] << endl;
18:     }
19:
20:     // изменить 3-е число с 987 на 2011
21:     vecIntegerArray[2] = 2011;
22:     cout << "After replacement: " << endl;
23:     cout << "Element[2] = " << vecIntegerArray[2] << endl;
24:
25:     return 0;
26: }
```

Результат

```
Element[0] = 50
Element[1] = 1
Element[2] = 987
```

```
Element[3] = 1001
After replacement:
Element[2] = 2011
```

Анализ

В строках 17, 21 и 23 вектор используется для доступа к элементам таким же способом, как при использовании статического массива, — при помощи оператора индексирования ([]). Этот оператор получает отсчитываемый от нуля индекс элемента, как и в статическом массиве. Обратите внимание, как в строке 15 был задан цикл `for`. Он сравнивает его с возвращаемым значением метода `vector::size()`, чтобы индекс не пересек границы вектора.

ВНИМАНИЕ!

Доступ к элементам вектора с использованием оператора `[]` чреват теми же опасностями, что и при обращении к элементам массива; т.е. вы не должны пересекать границы контейнера. Если использовать оператор индексирования (`[]`) для доступа к элементу вектора в позиции за его границей, то результат операции будет непредсказуем (может случиться что угодно, возможно, даже нарушение прав доступа).

Безопасней использовать функцию-член `at()`:

```
// получить элемент в позиции 2
cout << vecIntegerArray.at (2);
// версия vector::at() кода строки 17 листинга 17.4:
cout << vecIntegerArray.at(Index);
```

Во время выполнения метод `at()` проверяет размер контейнера и передает исключение при пересечении его границ (что недопустимо).

Оператор индексирования (`[]`) безопасен, когда используется способом, гарантирующим непреодолимость границ, как в приведенном выше примере.

Доступ к элементам вектора с использованием семантики указателя

К элементам вектора можно также обратиться, используя подобную указателю семантику итераторов (листинг 17.5).

ЛИСТИНГ 17.5. Доступ к элементам вектора с использованием семантики указателя (итераторов)

```
0: #include <iostream>
1: #include <vector>
2:
3: int main ()
4: {
5:     using namespace std;
6:     vector <int> vecIntegers;
7:
8:     // Вставить в вектор целые числа:
9:     vecIntegers.push_back (50);
```

```
10:     vecIntegers.push_back (1);
11:     vecIntegers.push_back (987);
12:     vecIntegers.push_back (1001);
13:
14:     // Доступ к объектам в векторе с использованием итераторов:
15:     vector <int>::iterator iElementLocator = vecIntegers.begin ();
16:     // итератор, объявленный с использованием ключевого
        // слова C++11 auto (следующая закомментированная строка)
17:     // auto iElementLocator = vecIntegers.begin ();
18:
19:     while (iElementLocator != vecIntegers.end ())
20:     {
21:         size_t Index = distance (vecIntegers.begin (),
22:                                  iElementLocator);
23:
24:         cout << "Element at position ";
25:         cout << Index << " is: " << *iElementLocator << endl;
26:
27:         // перейти к следующему элементу
28:         ++ iElementLocator;
29:     }
30:
31:     return 0;
32: }
```

Результат

```
Element at position 0 is: 50
Element at position 1 is: 1
Element at position 2 is: 987
Element at position 3 is: 1001
```

Анализ

Итератор в этом примере ведет себя весьма похоже на указатель, и характер его применения сходен с арифметикой адресов в указателе, как можно заметить в строке 25, где к хранящемуся в векторе значению обращаются с использованием оператора обращения к значению (*), и в строке 29, где инкремент итератора при помощи оператора ++ переводит его на следующий элемент. Обратите внимание на использование в строке 21 метода `std::distance()` для вычисления отсчитываемой от нуля позиции смещения элемента в векторе (т.е. позиции относительно начала) по возвращаемому значению метода `begin()` и указывающему на элемент итератору.

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 при помощи метода `pop_back()` занимает постоянное время, т.е. время удаления 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>& vecInput)
 6: {
 7:     for (auto iElement = vecInput.cbegin() // auto и cbegin(): C++11
 8:             ; iElement != Input.cend()     // cend() только C++11
 9:             ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     vector <int> vecIntegers;
18:
19:     // Вставить в вектор целые числа:
20:     vecIntegers.push_back (50);
21:     vecIntegers.push_back (1);
22:     vecIntegers.push_back (987);
23:     vecIntegers.push_back (1001);
24:
25:     cout << "Vector contains " << vecIntegers.size ()
            << " elements: ";
26:     DisplayVector(vecIntegers);
27:
28:     // Удалить один элемент в конце
29:     vecIntegers.pop_back ();
30:
31:     cout << "After a call to pop_back()" << endl;
32:     cout << "Vector contains " << vecIntegers.size ()
            << " elements: ";
33:     DisplayVector(vecIntegers);
34:
35:     return 0;
36: }
```

Результат

```
Vector contains 4 elements: 50 1 987 1001
After a call to pop_back()
Vector contains 3 elements: 50 1 987
```

Анализ

Вывод указывает, что метод pop_back(), используемый в строке 29, сократил количество элементов в векторе, удалив последний вставленный в него элемент. В строке 32 следующий вызов метода size() демонстрирует, что количество элементов в векторе сократилось на одни, как свидетельствует вывод.

ПРИМЕЧАНИЕ

Функция `DisplayVector()` в строках 4-13 листинга 17.6 приняла форму шаблона по сравнению с листингом 17.3, где она получала только вектор целых чисел. Это позволит нам повторно использовать этот шаблон функции для вектора типа `float` (вместо типа `int`):

```
vector <float> vecFloats;
DisplayVector(vecFloats); // работает, поскольку это обобщение
    function
```

Теперь она поддерживает вектор любого класса, который предоставляет оператор `*`, возвращающий значение, понятное оператору `cout`.

Концепция 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 повторное резервирование памяти позволит сохранить больше элементов. Поэтому размер вектора меньше или равен его емкости.

Количество элементов в векторе можно выяснить, вызвав функцию `size()`:

```
cout << "Size: " << vecIntegers.size ();
```

Емкость возвращает функция `capacity()`:

```
cout << "Capacity: " << vecIntegers.capacity () << endl;
```

При частом повторном резервировании памяти для внутреннего динамического массива вектор может создать проблемы с производительностью. Эта проблема в значительной степени может быть решена при помощи функции-члена `reserve(`*число*`)`. По существу, она увеличивает объем памяти, резервируемой для внутреннего массива вектора, что позволяет сохранять больше элементов без повторных резервирований. В зависимости от типа хранимых в векторе объектов, сокращение количества повторных резервирований сокращает также время копирования объектов и экономит 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 экземпляра вектора, способного изначально
        // содержать 5 целых чисел
 8:     vector <int> vecIntegers (5);
 9:
10:     cout << "Vector of integers was instantiated with " << endl;
11:     cout << "Size: " << vecIntegers.size ();
```

```
12:     cout << ", Capacity: " << vecIntegers.capacity () << endl;
13:
14:     // Вставка в вектор 6-го элемента
15:     vecIntegers.push_back (666);
16:
17:     cout << "After inserting an additional element... " << endl;
18:     cout << "Size: " << vecIntegers.size ();
19:     cout << ", Capacity: " << vecIntegers.capacity () << endl;
20:
21:     // Вставка другого элемента
22:     vecIntegers.push_back (777);
23:
24:     cout << "After inserting yet another element... " << endl;
25:     cout << "Size: " << vecIntegers.size ();
26:     cout << ", Capacity: " << vecIntegers.capacity () << endl;
27:
28:     return 0;
29: }
```

Результат

```
Vector of integers was instantiated with
Size: 5, Capacity: 5
After inserting an additional element...
Size: 6, Capacity: 7
After inserting yet another element...
Size: 7, Capacity: 7
```

Анализ

В строке 8 показано создание экземпляра вектора целых чисел, способного изначально содержать пять целых чисел со значением по умолчанию (0). Строки 11 и 12 отображают размер и емкость вектора соответственно, свидетельствуя, что во время создания экземпляра они равны. В строке 9 в вектор вставляется шестой элемент. Поскольку емкость вектора до вставки была пять, во внутреннем буфере вектора нет места для сохранения этого нового элемента. Другими словами, для сохранения шестого элемента класс `vector` должен повторно зарезервировать свой внутренний буфер. Реализация логики повторного резервирования достаточно интеллектуальна, чтобы избежать очередного резервирования при вставке следующего элемента; она преимущественно резервирует емкость, превосходящую требования текущего момента.

Вывод демонстрирует это при вставке в вектор емкостью пять элементов шестого элемента — повторное резервирование увеличивает емкость до семи элементов. Метод `size()` всегда отображает количество элементов в векторе и имеет на данном этапе значение шесть. Добавление седьмого элемента в строке 22 не приводит к увеличению емкости — имеющейся памяти вполне достаточно. На данном этапе и размер, и емкость имеют одинаковое значение, указывая, что вектор заполнен полностью и вставка следующего элемента приведет к резервированию вектором нового внутреннего буфера, копированию в него существующих значений и вставке нового значения.

ПРИМЕЧАНИЕ

Избыточное увеличение емкости внутреннего буфера вектора не регулируется никакими стандартными директивами. Это зависит от разновидности используемой библиотеки STL.

Класс deque библиотеки STL

Класс deque является классом динамического массива библиотеки STL, очень похожим на класс vector, но обеспечивающим вставку и удаление элементов как в конец, так и в начало массива. Экземпляр класса deque для целых чисел можно создать так:

```
// Определение двухсторонней очереди целых чисел
deque <int> dqIntegers;
```

СОВЕТ

Чтобы использовать класс std::deque, включите его заголовок:

```
#include<deque>
```

Двухстороннюю очередь (deque) можно представить так, как показано на рис. 17.2.

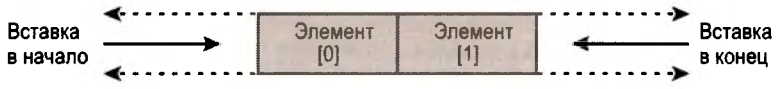


РИС. 17.2. Внутренняя организация двухсторонней очереди

Двухсторонняя очередь очень похожа на вектор, она обеспечивает вставку и извлечение элементов с конца при помощи методов push_back() и pop_back(). Подобно вектору, двухсторонняя очередь также обеспечивает доступ к элементам с использованием семантики массива и оператора индексирования ([]). Двухсторонняя очередь отличается от вектора тем, что позволяет также вставлять и извлекать элементы с начала, используя методы push_front() и pop_front() (листинг 17.8).

ЛИСТИНГ 17.8. Создание экземпляра двухсторонней очереди STL, а также применение методов push_front() и pop_front() для вставки и извлечения элементов с начала

```
 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> dqIntegers;
10:
11:     // Вставка целых чисел в конец массива
12:     dqIntegers.push_back (3);
13:     dqIntegers.push_back (4);
14:     dqIntegers.push_back (5);
```

```
15:
16:     // Вставка целых чисел в начало массива
17:     dqIntegers.push_front (2);
18:     dqIntegers.push_front (1);
19:     dqIntegers.push_front (0);
20:
21:     cout << "The contents of the deque after inserting elements ";
22:     cout << "at the top and bottom are:" << endl;
23:
24:     // Отображение содержимого на экране
25:     for ( size_t nCount = 0
26:           ; nCount < dqIntegers.size ()
27:           ; ++ nCount )
28:     {
29:         cout << "Element [" << nCount << "] = ";
30:         cout << dqIntegers [nCount] << endl;
31:     }
32:
33:     cout << endl;
34:
35:     // Извлечение элемента с начала
36:     dqIntegers.pop_front ();
37:
38:     // Извлечение элемента с конца
39:     dqIntegers.pop_back ();
40:
41:     cout << "The contents of the deque after erasing an element ";
42:     cout << "from the top and bottom are:" << endl;
43:
44:     // Отображает содержимое снова: на сей раз при помощи итераторов
45:     // При компиляции на устаревших компиляторах удалите ключевое
       // слово auto и снимите комментарий со следующей строки
46:     // deque <int>::iterator iElementLocator;
47:     for (auto iElementLocator = dqIntegers.begin ()
48:           ; iElementLocator != dqIntegers.end ()
49:           ; ++ iElementLocator )
50:     {
51:         size_t Offset = distance (dqIntegers.begin (),
                                      iElementLocator);
52:         cout << "Element [" << Offset << "] = " << *iElementLocator
                 << endl;
53:     }
54:
55:     return 0;
56: }
```

Результат

```
The contents of the deque after inserting elements at the top and bottom are:
Element [0] = 0
Element [1] = 1
Element [2] = 2
Element [3] = 3
```

```
Element [4] = 4
Element [5] = 5
The contents of the deque after erasing an element from the top and bottom are:
Element [0] = 1
Element [1] = 2
Element [2] = 3
Element [3] = 4
```

Анализ

В строке 10 создается экземпляр двухсторонней очереди целых чисел. Обратите внимание, насколько похож этот синтаксис на синтаксис создания экземпляра вектора целых чисел. Строки 11–14 демонстрируют применение функции-члена `push_back()` класса `deque`, а строки 16–19 — функции-члена `push_front()`. Последнее отличает двухстороннюю очередь от вектора. Применение метода `pop_front()` показано в строке 37. Первый механизм отображения содержимого двухсторонней очереди использует для доступа к элементам синтаксис, как у 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.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте динамический массив векторов или двухстороннюю очередь, когда неизвестно количество элементов, которые необходимо хранить	**Не забывайте**, что метод `pop_back()` извлекает последний элемент из коллекции
Помните, что вектор может расти только с конца, при помощи метода `push_back()`	**Не забывайте**, что метод `pop_front()` удаляет первый элемент из двухсторонней очереди
Помните, что двухсторонняя очередь может расти на обоих направлениях, при помощи методов `push_back()` и `push_front()`	**Не обращайтесь** к динамическому массиву вне его границ

Резюме

На сегодняшнем занятии рассматривались основы использования таких динамических массивов, как вектор и двухсторонняя очередь. Были объяснены концепции размера и емкости, а также изложено, что применение вектора может быть оптимизировано для уменьшения количества повторных резервирований содержащего объекты внутреннего буфера, копирование которого способно ухудшить производительность. Хоть вектор и является самым простым из контейнеров библиотеки STL, он все же самый популярный, а возможно, и самый эффективный.

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. Кстати, это справедливо для всех контейнеров STL.
- **Вставка и извлечение элементов из вектора занимают больше времени, если вектор содержит больше элементов?**
  Нет. Время выполнения операций вставки и извлечения элементов вектора не зависит от количества элементов в нем.
- **В чем преимущество использования функции-члена reserve()?**
  Она резервирует пространство для внутреннего буфера вектора, снижая частоту его повторных резервирований при вставке элементов в вектор. В зависимости от характера хранимых в векторе объектов резервирование нового буфера вектора и копирование в него элементов из прежнего буфера может существенно снизить производительность.
- **Есть ли различие между вектором и двухсторонней очередью, когда дело доходит до вставки элементов?**
  Пока речь идет о вставке в конец с постоянным временем выполнения и вставке в середину с пропорциональным временем выполнения, никакой разницы нет. Но вектор допускает вставку только в конец, а двухсторонняя очередь — в конец и в начало.

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 Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Элементы могут быть вставлены в середину или в начало вектора за постоянное время?
2. Метод `size()` моего вектора возвращает значение 10, а метод `capacity()` — 20. Сколько еще элементов я могу вставить в него до повторного резервирования буфера?
3. Что делает функция `pop_back()`?

4. Если `vector<int>` является динамическим массивом целых чисел, то динамическим массивом какого типа является вектор `vector<CMammal>`?
5. Можно ли произвольно обращаться к элементам вектора? Если да, то как?
6. Какой тип итератора обеспечивает произвольный доступ к элементам вектора?

Упражнения

1. Напишите автономную программу, которая получает 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, существует ли уже запрошенное 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.

ЗАНЯТИЕ 18

Классы двухсвязного и односвязного списков библиотеки STL

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 постоянное время. Начиная с версии C++11 вы можете также использовать *односвязный список* (singly linked list) в форме шаблона класса `std::forward_list`, который можно пройти только в одном направлении.

На сегодняшнем занятии.

- Как создать экземпляр классов `list` и `forward_list`.
- Использование классов списков библиотеки STL, включая вставку и извлечение.
- Как обратить 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 также указатель на следующий узел, т.е. каждый узел связан со следующим и с предыдущим, как показано на рис. 18.1.

РИС. 18.1. Визуальное представление двухсвязного списка

Реализация класса list библиотеки STL обеспечивает постоянную продолжительность вставки в начало, в конец и в середину списка.

СОВЕТ

Чтобы использовать класс std::list, включите его заголовок:

```
#include <list>
```

Основные операции со списком

Чтобы использовать класс list библиотеки STL, включите в код файл заголовка <list>. Шаблон класса list, расположенный в пространстве имен std, является обобщенной реализацией шаблона, экземпляр которой должен быть создан прежде, чем вы сможете использовать любую из его функций-членов.

Создание экземпляра класса std::list

При создании экземпляра шаблона класса std::list нужно определить тип объекта, который предполагается хранить в списке. Так, инициализация списка выглядит следующим образом:

```
std::list<int> listIntegers; // 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 iElementInSet;
```

Если необходим итератор для изменения значений или вызова не константных функций, используйте ключевое слово iterator вместо const_iterator.

Поскольку класс std::list имеет несколько перегруженных конструкторов, его экземпляр можно создать, указав начальное количество элементов и их исходные значения, как показано в листинге 18.1.

ЛИСТИНГ 18.1. Различные формы создания экземпляров класса `std::list`: определение количества элементов и их исходных значений

```
 0: #include <list>
 1: #include <vector>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     // создание экземпляра пустого списка
 8:     list <int> listIntegers;
 9:
10:     // создание экземпляра списка с 10 целыми числами
11:     list<int> listWith10Integers(10);
12:
13:     // создание экземпляра списка с 4 целыми числами,
        // инициализированными значением 99
14:     list<int> listWith4IntegerEach99 (10, 99);
15:
16:     // создание точной копии существующего списка
17:     list<int> listCopyAnother(listWith4IntegerEach99);
18:
19:     // вектор для 10 целых чисел со значением 2011 каждый
20:     vector<int> vecIntegers(10, 2011);
21:
22:     // создание экземпляра списка с использованием значений
        // из другого контейнера
23:     list<int> listContainsCopyOfAnother(vecIntegers.cbegin(),
24:                                         vecIntegers.cend());
25:
26:     return 0;
27: }
```

Анализ

Эта программа не имеет вывода, она 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 `listWith4IntegersEach99`, содержащий 4 целых числа, инициализированных 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 2011, а затем, в строке 23, создается экземпляр списка, содержащий элементы, скопированные из вектора с использованием константных итераторов, возвращенных методами `vector::cbegin()` и `vector::cend()` (новшество C++11). Листинг 18.1 демонстрирует также то, как итераторы позволяют отделить реализацию одного контейнера от другого и использовать их обобщенные функциональные возможности для создания экземпляра списка с использованием взятых из вектора значений (строки 23 и 24).

СОВЕТ

Не будет ли ошибки компиляции при использовании методов cbegin() и cend()?

Если вы попытаетесь откомпилировать эту программу, используя компилятор, не совместимый со стандартом C++11, задействуйте методы `begin()` и `end()` вместо `cbegin()` и `cend()` соответственно. Методы `cbegin()` и `cend()` доступны только в C++11, они возвращают константный итератор, который не может быть использован для изменения элементов.

ПРИМЕЧАНИЕ

Сравнив листинги 18.1 и 17.1, можно заметить шаблон и замечательное подобие способов создания экземпляров 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:

```
listIntegers.push_back (-1);
listIntegers.push_front (2001);
```

В листинге 18.2 показан результат использования этих двух методов в списке 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& Input)
 6: {
 7:     for (auto iElement = Input.cbegin() // auto и cbegin: C++11
 8:                     ; iElement != Input.cend()
 9:                     ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     std::list <int> listIntegers;
18:
19:     listIntegers.push_front (10);
20:     listIntegers.push_front (2011);
21:     listIntegers.push_back (-1);
```

```
12:     listIntegers.push_back (9999);
13:
14:     DisplayContents(listIntegers);
15:
16:     return 0;
17: }
```

Результат

```
2011 10 -1 9999
```

Анализ

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 (при помощи обобщенного шаблона функции `DisplayContents()`), демонстрируя, что элементы хранятся не в порядке вставки.

ВНИМАНИЕ!

Возникла ошибка компиляции при использовании ключевого слова auto?

Функция `DisplayContents()` в листинге 18.2 использует ключевое слово C++11 `auto` для определения типа итератора в строке 7. Кроме того, она использует возвращающие итератор `const_iterator` функции `cbegin()` и `cend()`, которые совместимы только со стандартом C++11.

В этом и последующих примерах для компиляции с использованием компилятора, не совместимого со стандартом C++11, необходимо заменить ключевое слово `auto` явным типом.

Так, функцию `DisplayContents()` для устаревшего компилятора следует изменить, как показано ниже.

```
template <typename T>
void DisplayContents (const T& Input)
{
    for (T::const_iterator iElement = Input.begin ()
            ; iElement != Input.end ()
            ; ++ iElement )
        cout << *iElement << ' ';
    cout << endl;
}
```

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 последний работал только для вектора, обобщение типа хранимых в нем элементов обеспечивает действительно обобщенную версию даже для контейнеров разных типов.

Вы можете вызвать версию метода `DisplayContents()` из листинга 18.2 с вектором или списком в качестве аргумента, и он будет работать прекрасно.

Вставка в середину списка

Контейнер `std::list` характеризуется возможностью вставки элементов в середину коллекции за постоянное время. Для этого применяется его функция-член `insert()`.

Функция `insert()` класса `list` доступна в трех формах.

- **Форма 1**

```
iterator insert(iterator pos, const T& x)
```

Здесь функция `insert()` получает позицию вставки как первый параметр и вставляемое значение как второй. Эта функция возвращает итератор, указывающий на элемент, только что вставленный в список.

- **Форма 2**

```
void insert(iterator pos, size_type n, const T& x)
```

Эта функция получает позицию вставки как первый параметр, вставляемое значение как последний параметр и количество элементов в переменной `n`.

- **Форма 3**

```
template <class InputIterator>
void insert(iterator pos, InputIterator f, InputIterator l)
```

Эта перегруженная версия — шаблон функции, который получает кроме позиции два итератора ввода, отмечающие границы вставляемой в список коллекции. Обратите внимание: входной тип, `InputIterator`, — параметрический тип шаблона, а потому может указать на границы любой коллекции, будь то массив, вектор или другой список.

В листинге 18.3 показано применение этих перегруженных версий функции `list::insert()`.

ЛИСТИНГ 18.3. Различные 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& Input)
 6: {
 7:     for (auto iElement = Input.cbegin() // auto и cbegin: C++11
 8:             ; iElement != Input.cend()
 9:             ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     list <int> listIntegers1;
18:
19:     // Вставка элементов в начало...
20:     listIntegers1.insert (listIntegers1.begin (), 2);
```

```
11:     listIntegers1.insert (listIntegers1.begin (), 1);
12:
13:     // Вставка элементов в конец...
14:     listIntegers1.insert (listIntegers1.end (), 3);
15:
16:     cout << "The contents of list 1 after inserting elements:"
             << endl;
17:     DisplayContents (listIntegers1);
18:
19:     list <int> listIntegers2;
20
21:     // Вставка 4 элементов с одинаковым значением 0...
22:     listIntegers2.insert (listIntegers2.begin (), 4, 0);
23:
24:     cout << "The contents of list 2 after inserting '";
25:     cout << listIntegers2.size () << "' elements of a value:"
             << endl;
26:     DisplayContents (listIntegers2);
27:
28:     list <int> listIntegers3;
29:
30:     // Вставка элементов из другого списка в начало...
31:     listIntegers3.insert (listIntegers3.begin (),
32:                   listIntegers1.begin (), listIntegers1.end ());
33:
34:     cout << "The contents of list 3 after inserting the contents of";
35:     cout << "list 1 at the beginning:" << endl;
36:     DisplayContents (listIntegers3);
37:
38:     // Вставка элементов из другого списка в конец...
39:     listIntegers3.insert (listIntegers3.end (),
50:                   listIntegers2.begin (), listIntegers2.end ());
51:
52:     cout << "The contents of list 3 after inserting ";
53:     cout << "the contents of list 2 at the beginning:" << endl;
54:     DisplayContents (listIntegers3);
55:
56:     return 0;
57: }
```

Результат

```
The contents of list 1 after inserting elements:
1 2 3
The contents of list 2 after inserting '4' elements of a value:
0 0 0 0
The contents of list 3 after inserting the contents of list 1 at the beginning:
1 2 3
The contents of list 3 after inserting the contents of list 2 at the beginning:
1 2 3 0 0 0 0
```

Анализ

Функции-члены `begin()` и `end()` возвращают итераторы, указывающие на начало и конец списка соответственно. Это справедливо для всех контейнеров STL, включая `std::list`. Его функция `insert()` получает итератор, отмечающий 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 помещенными в начало 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 вставляется список целых чисел в другой список, вставлен может быть также диапазон из вектора, определяемый методами `begin()` и `end()`, как в листинге 18.1, или обычного статического массива.

Удаление элементов из списка

Функция `erase()` класса `list` имеет две перегруженные версии: удаляющую один элемент по переданному итератору, указывающему на него, и удаляющую диапазон элементов из списка. В действии функцию `list::erase()` можно увидеть в листинге 18.4, где из списка удаляется один элемент или 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& Input)
 6: {
 7:     for(auto iElement = Input.cbegin() // auto и cbegin: C++11
 8:                 ; iElement != Input.cend()
 9:                 ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main()
16: {
17:     std::list <int> listIntegers;
18:
19:     // Вставка элементов в вначало и конец...
20:     listIntegers.push_back(4);
21:     listIntegers.push_front(3);
22:     listIntegers.push_back(5);
23:
24:     // Сохранить итератор, полученный при помощи функции insert()
25:     auto iValue2 = listIntegers.insert(listIntegers.begin(), 2);
26:
27:     cout << "Initial contents of the list:" << endl;
```

```
28:     DisplayContents(listIntegers);
29:
30:     listIntegers.erase(listIntegers.begin(), iValue2);
31:     cout << "Contents after erasing a range of elements:" << endl;
32:     DisplayContents(listIntegers);
33:
34:     cout << "After erasing element '"<< *iValue2 << "':" << endl;
35:     listIntegers.erase(iValue2);
36:     DisplayContents(listIntegers);
37:
38:     listIntegers.erase(listIntegers.begin(), listIntegers.end());
39:     cout << "Number of elements after erasing range: ";
40:     cout << listIntegers.size() << endl;
41:
42:     return 0;
43: }
```

Результат

```
Initial contents of the list:
2 3 4 5
Contents after erasing a range of elements:
2 3 4 5
After erasing element '2':
3 4 5
Number of elements after erasing range: 0
```

Анализ

Строки 20–25 функции `main()` используют различные методы для вставки в список целых чисел. Когда для вставки значений используется метод `insert()`, он возвращает итератор на вставленный элемент. В данном случае итератор указывает на элемент со значением 2. В строке 25 он сохраняется в переменной `iValue2` и используется позже при вызове функции `erase()` в строке 35 для удаления этого элемента из списка. Строки 30 и 38 демонстрируют применение метода `erase()` для удаления диапазона элементов. В первом случае удаляется диапазон от `begin()` до элемента, содержащего значение 2 (но не включая его). Во втором случае очищается диапазон от `begin()` до `end()`, — фактически удаляется весь список.

ПРИМЕЧАНИЕ

Строка 40 листинга 18.4 демонстрирует, что количество элементов списка может быть определено при помощи метода `size()` класса `std::list`, как и у вектора. Это применимо ко всем контейнерным классам библиотеки STL.

Обращение списка и сортировка его элементов

У списка есть одна особенность: указывающие на элементы в списке итераторы остаются допустимыми, несмотря на перестановку элементов или вставку новых. Для

обеспечения этой особенности класс `list` предоставляет методы `sort()` и `reverse()`. хотя библиотека STL предлагает такие же алгоритмы, вполне способные работать с классом `list`. Эти алгоритмы в версии членов класса гарантируют, что итераторы, указывающие на элементы в списке, останутся допустимыми при изменении относительной позиции элементов.

Обращение элементов списка с использованием метода `list::reverse()`

Функция-член `reverse()` класса `list` не получает никаких параметров и обращает порядок содержимого списка:

```
listIntegers.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& Input)
 6: {
 7:     for(auto iElement = Input.cbegin() // auto и cbegin: C++11
 8:            ; iElement != Input.cend()
 9:            ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main()
16: {
17:     std::list <int> listIntegers;
18:
19:     // Вставка элементов в начало и в конец
20:     listIntegers.push_front(4);
21:     listIntegers.push_front(3);
22:     listIntegers.push_front(2);
23:     listIntegers.push_front(1);
24:     listIntegers.push_front(0);
25:     listIntegers.push_back(5);
26:
27:     cout << "Initial contents of the list:" << endl;
28:     DisplayContents(listIntegers);
29:
30:     listIntegers.reverse();
31:
32:     cout << "Contents of the list after using reverse():" << endl;
33:     DisplayContents(listIntegers);
34:
35:     return 0;
36: }
```

Результат

```
Initial contents of the list:
0 1 2 3 4 5
Contents of the list after using reverse():
5 4 3 2 1 0
```

Анализ

Как показано в строке 30, метод `reverse()` просто обращает порядок элементов списка. Это простой вызов без параметров, гарантирующий, что указывающие на элементы итераторы, если они были сохранены, останутся допустимыми.

Сортировка элементов

Функция-член `sort()` класса `list` не доступна в версии без параметров:

```
listIntegers.sort(); // сортировка в порядке возрастания
```

Другая версия позволяет определять собственные приоритеты сортировки при помощи функции предиката, переданной как параметр:

```
bool SortPredicate_Descending (const int& lsh, const int& rsh)
{
    // определение критериев для метода list::sort: возвратить true
    // для желательного порядка
    return (lsh > rsh);
}
// Использование предиката для сортировки списка:
listIntegers.sort (SortPredicate_Descending);
```

Эти два варианта представлены 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& lsh, const int& rsh)
 5: {
 6:     // определение критериев для метода list::sort: возвратить true
        // для желательного порядка
 7:     return (lsh > rsh);
 8: }
 9:
10: template <typename T>
11: void DisplayContents (const T& Input)
12: {
13:     for (auto iElement = Input.cbegin() // auto и cbegin: C++11
14:             ; iElement != Input.cend()
15:             ; ++ iElement )
16:         cout << *iElement << ' ';
17:
```

```
18:     cout << endl;
19: }
20:
21: int main ()
22: {
23:     list <int> listIntegers;
24:
25:     // Вставка элементов в начало и конец
26:     listIntegers.push_front (444);
27:     listIntegers.push_front (2011);
28:     listIntegers.push_front (-1);
29:     listIntegers.push_front (0);
30:     listIntegers.push_back (-5);
31:
32:     cout << "Initial contents of the list are - " << endl;
33:     DisplayContents (listIntegers);
34:
35:     listIntegers.sort ();
36:
37:     cout << "Order of elements after sort():" << endl;
38:     DisplayContents (listIntegers);
39:
40:     listIntegers.sort (SortPredicate_Descending);
41:     cout << "Order of elements after sort() with a predicate:"
            << endl;
42:     DisplayContents (listIntegers);
43:
44:     return 0;
45: }
```

Результат

```
Initial contents of the list are -
0 -1 2011 444 -5
Order of elements after sort():
-5 -1 0 444 2011
Order of elements after sort() with a predicate:
2011 444 0 -1 -5
```

Анализ

Данный пример демонстрирует функциональные возможности сортировки в списке целых чисел. В нескольких первых строках кода создается объект списка и заполняется примерами значений. Строка 35 демонстрирует 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. Функция `SortPredicate_Descending()`, определенная в строках 4–8, является бинарным предикатом, который позволяет функции `sort()` списка решить, является ли один элемент меньше другого. В противном случае он меняет их позиции. Другими словами, вы указываете списку, что следует интерпретировать как "меньше" (в данном случае первый параметр

больше второго). Этот предикат передается как параметр функции `sort()` (строка 40). 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 позицию элементов.

Сортировка и удаление элементов из списка, который содержит объекты класса

Что, если у вас есть список объектов класса, а не таких простых встроенных типов, как `int`? Скажем, список записей адресной книги, где 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 содержат список, оператор <.
- Предоставьте некий *бинарный предикат* (binary predicate) — функцию, получающую на входе два значения и возвращающую логическое значение, указывающее, меньше ли первое значение второго.

Реальные приложения, задействующие контейнеры библиотеки 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 главным образом 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 DisplayContents (const T& Input)
 7: {
 8:     for(auto iElement = Input.cbegin() // auto, cbegin и cend: C++11
 9:           ; iElement != Input.cend()
10:           ; ++ iElement )
11:         cout << *iElement << endl;
12:
13:     cout << endl;
14: }
15:
16: struct ContactItem
17: {
18:     string strContactsName;
19:     string strPhoneNumber;
20:     string strDisplayRepresentation;
21:
22:     // Конструктор и деструктор
```

```
23:     ContactItem (const string& strName, const string & strNumber)
24:     {
25:         strContactsName = strName;
26:         strPhoneNumber = strNumber;
27:         strDisplayRepresentation = (strContactsName + ": "\
                                        + strPhoneNumber);
28:     }
29:
30:     // используется list::remove() для элемента списка контактов
31:     bool operator == (const ContactItem& itemToCompare) const
32:     {
33:         return (itemToCompare.strContactsName == \
                    this->strContactsName);
34:     }
35:
36:     // используется в list::sort() без параметров
37:     bool operator < (const ContactItem& itemToCompare) const
38:     {
39:         return (this->strContactsName < \
                        itemToCompare.strContactsName);
40:     }
41:
42:     // Используется в DisplayContents через cout
43:     operator const char*() const
44:     {
45:         return strDisplayRepresentation.c_str();
46:     }
47: };
48:
49: bool SortOnPhoneNumber (const ContactItem& item1,
50:                         const ContactItem& item2)
51: {
52:     return (item1.strPhoneNumber < item2.strPhoneNumber);
53: }
54:
55: int main ()
56: {
57:     list <ContactItem> Contacts;
58:     Contacts.push_back(ContactItem("Jack Welsch",
                                       "+1 7889 879 879"));
59:     Contacts.push_back(ContactItem("Bill Gates",
                                       "+1 97 7897 8799 8"));
60:     Contacts.push_back(ContactItem("Angela Merkel",
                                       "+49 23456 5466"));
61:     Contacts.push_back(ContactItem("Vladimir Putin",
                                       "+7 6645 4564 797"));
62:     Contacts.push_back(ContactItem("Manmohan Singh",
                                       "+91 234 4564 789"));
63:     Contacts.push_back(ContactItem("Barack Obama",
                                       "+1 745 641 314"));
64:
65:     cout << "List in initial order: " << endl;
66:     DisplayContents(Contacts);
```

```
67:
68:     Contacts.sort();
69:     cout << "After sorting in alphabetical order via operator<:"
             << endl;
70:     DisplayContents(Contacts);
71:
72:     Contacts.sort(SortOnPhoneNumber);
73:     cout << "After sorting in order of phone numbers via predicate:"
             << endl;
74:     DisplayContents(Contacts);
75:
76:     cout << "After erasing Putin from the list: ";
77:     Contacts.remove(ContactItem("Vladimir Putin", ""));
78:     DisplayContents(Contacts);
79:
80:     return 0;
81: }
```

Результат

```
List in initial order:
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Vladimir Putin: +7 6645 4564 797
Manmohan Singh: +91 234 4564 789
Barack Obama: +1 745 641 314

After sorting in alphabetical order via operator<:
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Jack Welsch: +1 7889 879 879
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797

After sorting in order of phone numbers via predicate:
Barack Obama: +1 745 641 314
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Vladimir Putin: +7 6645 4564 797
Manmohan Singh: +91 234 4564 789

After erasing Putin from the list:
Barack Obama: +1 745 641 314
Jack Welsch: +1 7889 879 879
Bill Gates: +1 97 7897 8799 8
Angela Merkel: +49 23456 5466
Manmohan Singh: +91 234 4564 789
```

Анализ

Сначала сосредоточимся на строках 57–81 функции `main()`. В строке 57 создается экземпляр списка элементов книги адресов типа `ContactItem`. В строках 58–63 этот список заполняется вымышленными номерами телефонов и именами нескольких знаменитостей и политических деятелей. В строке 66 осуществляется вывод этого списка на экран. Функция `list::sort` без предиката используется в строке 68. При отсутствии предиката эта функция сортировки ищет оператор `operator<` в классе `ContactItem`, где он и был определен в строках 37–40. Оператор `ContactItem::operator<` позволяет контейнеру `list::sort` сортировать свои элементы в алфавитном порядке хранимых имен (а не номеров телефонов или произвольно). Чтобы отсортировать тот же список по номерам телефонов, используйте функцию `list::sort()`, предоставив ей как аргумент бинарный предикат `SortOnPhoneNumber()` (строка 72). Эта функция, реализованная в строках 49–53, гарантирует, что входные аргументы типа `ContactItem` сравниваются с другими по номеру телефона, а не по имени. Таким образом, это позволяет функции `list::sort()` сортировать список знаменитостей на основе их номеров телефонов, как свидетельствует вывод. И наконец, в строке 77 используется метод `list::remove()` для удаления контакта из списка. Объект контакта передается как параметр. Метод `list::remove()` сравнивает этот объект с другими элементами списка с использованием оператора `ContactItem::operator=`, реализованного в строках 30–34. 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, предикатов.

C++11

Шаблон класса `std::forward_list`

В языке C++11 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).

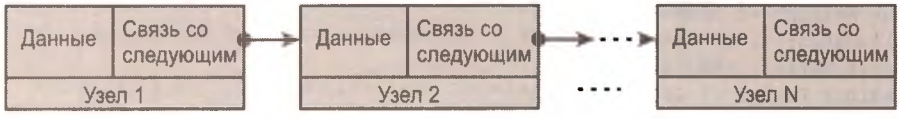


РИС. 18.2. Визуальное представление односвязного списка

СОВЕТ

Для использования шаблона класса `std::forward_list` необходимо включить заголовок `<forward_list>`:

```
#include<forward_list>
```

Класс forward_list используется очень похоже на класс list, но перемещение итераторов 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& Input)
 6: {
 7:     for (auto iElement = Input.cbegin() // auto и cbegin: C++11
 8:             ; iElement != Input.cend ()
 9:             ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main()
16: {
17:     forward_list<int> flistIntegers;
18:     flistIntegers.push_front(0);
19:     flistIntegers.push_front(2);
20:     flistIntegers.push_front(2);
21:     flistIntegers.push_front(4);
22:     flistIntegers.push_front(3);
23:     flistIntegers.push_front(1);
24:
25:     cout << "Contents of forward_list: " << endl;
26:     DisplayContents(flistIntegers);
27:
28:     flistIntegers.remove(2);
29:     flistIntegers.sort();
30:     cout << "Contents after removing 2 and sorting: " << endl;
31:     DisplayContents(flistIntegers);
32:
33:     return 0;
34: }
```

Результат

```
Contents of forward_list:
1 3 4 2 2 0
Contents after removing 2 and sorting:
0 1 3 4
```

Анализ

Как видно из примера, функционально класс `forward_list` весьма похож на класс `list`. Поскольку класс `forward_list` не поддерживает двунаправленный перебор, вы можете использовать для итератора оператор `operator++`, но не оператор `operator--`. Этот пример демонстрирует применение функции `remove(2)` для удаления всех элементов со значением 2 (строка 28). Строка 29 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 меньше памяти, чем класс `list` (поскольку элемент содержит ссылку только на следующий элемент, но не на предыдущий).

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте класс `std::list` вместо класса `std::vector`, когда необходимо частое удаление и вставка элементов, особенно в середину, поскольку вектор должен изменять размеры своего внутреннего буфера для обеспечения семантики, как у массива, что вызывает продолжительные операции копирования, а список только привязывает или отцепляет элементы	**Не используйте** класс `std::list` при нечастых вставках и удалениях в конец и отсутствии вставки и удаления в середину; классы `vector` и `deque` в этих случаях могут оказаться значительно быстрее
Помните, что методы `push_front()` и `push_back()` позволяют вставлять элементы в начало и в конец списка соответственно	**Не забывайте** предоставить функцию предиката, если хотите использовать метод `sort()` или `remove()` класса `list` с критериями, отличными от принятых по умолчанию
Помните о необходимости реализовать операторы `operator<` и `operator==` в классе, объекты которого будут храниться в таком контейнере STL, как `list`, чтобы предоставить предикат по умолчанию для сортировки и удаления	
Помните, что вы всегда можете определить количество элементов в списке, используя метод `list::size()`, как и в любом другом контейнере библиотеки STL	
Помните, что вы можете освободить список, используя метод `list::clear()`, как и в любом другом контейнере библиотеки STL	

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` библиотеки STL гарантирует, что указывающие на его элементы итераторы останутся допустимыми независимо от их позиции в списке. Хотя алгоритмы STL прекрасно работают и с классом `list`, эти функции-члены обеспечивают вышеупомянутую способность итераторов списка указывать на те же элементы даже после сортировки списка.
- **Вы используете список, элементы которого имеют тип `CAnimal`. Какие операторы должен определять класс `CAnimal`, чтобы функции-члены списка были в состоянии работать с ним правильно?**
  Вы должны предоставить заданные по умолчанию операторы сравнения == и < для любого класса, объекты которых хранятся в контейнерах библиотеки STL.
- **Зачем заменять ключевое слово `auto` явным описанием типа в следующей строке:**

```
list<int> listIntegers(10); // список из 10 целых чисел
auto iFirstElement = listIntegers.begin();
```

  Если вы используете устаревший компилятор, который не совместим со стандартом C++11, ключевое слово `auto` следует заменить явным описанием типа:

```
list<int> listIntegers(10); // список из 10 целых чисел
list<int>::iterator iFirstElement = listIntegers.begin();
```

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 Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Будет ли потеря производительности при вставке элементов в середину списка библиотеки STL, по сравнению со вставкой в его начало или в конец?
2. Два итератора указывают на два элемента в объекте списка библиотеки STL, затем между ними вставляется новый элемент. Сделает ли вставка эти итераторы недопустимыми?
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. На примере короткой программы продемонстрируйте, что итератор, указывающий на элемент в списке, продолжает оставаться допустимым, несмотря на то, что после или перед ним был вставлен другой элемент, изменив таким образом относительную позицию прежнего элемента.
3. Напишите программу, которая вставляет содержимое вектора в список, используя функцию вставки класса `list`.
4. Напишите программу, обращающую список строк.

ЗАНЯТИЕ 19
Классы наборов библиотеки STL

Стандартная библиотека шаблонов (STL) предоставляет разработчикам классы контейнеров, обеспечивающих частый и быстрый поиск. Классы `std::set` и `std::multiset` используются для хранения отсортированного набора элементов и предоставляют возможность поиска с логарифмической зависимостью его продолжительности от количества данных. Их неупорядоченные аналоги обеспечивают возможность вставки и поиска за постоянное время.

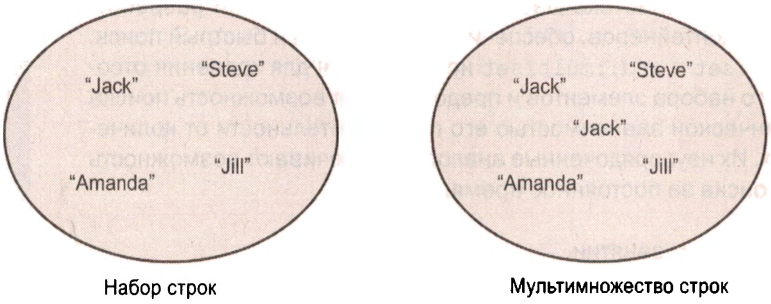
На сегодняшнем занятии.

- Чем могут быть полезны контейнеры `set`, `multiset`, `unordered_set` и `unordered_multiset` библиотеки STL.
- Вставка, извлечение и поиск элементов.
- Преимущества и недостатки использования этих контейнеров.

Введение в классы наборов библиотеки STL

Контейнеры `set` и `multiset` обеспечивают быстрый поиск ключей в хранящем их контейнере, т.е. значений ключей, содержащихся в одномерном контейнере. Различие между *набором* (set) и *мультимножеством* (multiset) в том, что последний допускает дубликаты, тогда как первый позволяет хранить только уникальные значения.

На рис. 19.1 показано, что набор имен содержит только уникальные имена, тогда как мультимножество допускает дубликаты. Будучи шаблонами классов, контейнеры библиотеки STL — это обобщения, применимые для хранения объектов любых типов, включая строки, целые числа, структуры или классы.



Набор строк Мультимножество строк

РИС. 19.1. Визуальное представление набора и мультимножества имен

Чтобы облегчить быстрый поиск, классы `set` и `multiset` библиотеки STL внутренне реализованы как двоичное дерево. Это означает, что для ускорения поиска элементы сортируются при вставке. Это также означает, что, в отличие от вектора, где элемент в определенной позиции может быть заменен другим, элемент набора в определенной позиции не может быть заменен новым элементом с другим значением. Дело в том, что класс `set` поместит его в другую область внутреннего дерева в соответствии с его значением.

СОВЕТ

Чтобы использовать классы `std::set` или `std::multiset`, включите в код их заголовок:

```
#include <set>
```

Простые операции с классами `set` и `multiset` библиотеки STL

Прежде чем вы сможете использовать любую из функций-членов шаблонов классов `set` и `multiset`, необходимо создать их экземпляр.

Создание экземпляра объекта `std::set`

Для создания экземпляра набора или мультимножества следует специализировать шаблон класса `std::set` или `std::multiset` для конкретного типа:

```
std::set <int> setIntegers;
std::multiset <int> msetIntegers;
```

Чтобы определить набор или мультимножество, содержащее объекты класса Tuna, используется следующий код:

```
std::set <Tuna> setIntegers;
std::multiset <Tuna> msetIntegers;
```

Чтобы объявить итератор, указывающий на элементы набора или мультимножества, используется следующий код:

```
std::set<int>::const_iterator iElementInSet;
std::multiset<int>::const_iterator iElementInMultiset;
```

Если необходим итератор для изменения значений или вызова не константных функций, используйте ключевое слово iterator вместо const_iterator.

Поскольку контейнеры set и multiset сортируют элементы при вставке, они используют заданный по умолчанию предикат std::less<>, если вы не предоставите иной критерий сортировки. Это гарантирует, что ваш набор будет содержать элементы отсортированными в порядке возрастания.

Вы создаете бинарный предикат сортировки, определяя класс с оператором operator(), который получает два содержащихся в наборе значения и возвращает значение true в зависимости от ваших критериев. Вот один из таких предикатов сортировки в порядке убывания:

```
// используется как параметр шаблона при создании экземпляра
// набора / мультимножества
template <typename T>
struct SortDescending
{
    bool operator()(const T& lhs, const T& rhs) const
    {
        return (lhs > rhs);
    }
};
```

Затем этот предикат используется при создании экземпляра набора или мультимножества:

```
// набор и мультимножество целых чисел (использующие предикат сортировки)
set <int, SortDescending<int> > setIntegers;
multiset <int, SortDescending<int> > msetIntegers;
```

Кроме этих вариантов, вы всегда сможете создать набор или мультимножество как полную или частичную копию другого контейнера (листинг 19.1).

ЛИСТИНГ 19.1. Различные способы создания экземпляра набора и мультимножества

```
0: #include <set>
1:
2: // используется как параметр шаблона при создании экземпляра
   // набора / мультимножества
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:     // набор или мультимножество целых чисел (использующие
        // предикат сортировки)
17:     set <int> setIntegers1;
18:     multiset <int> msetIntegers1;
19:
20:     // создание экземпляра набора и мультимножества с заданным
        // пользователем предикатом сортировки
21:     set<int, SortDescending<int> > setIntegers2;
22:     multiset<int, SortDescending<int> > msetIntegers2;
23:
24:     // создание набора из другого контейнера или его части
25:     set<int> setIntegers3(setIntegers1);
26:     multiset<int> msetIntegers3(setIntegers1.cbegin(),
                                    setIntegers1.cend());
27:
28:     return 0;
29: }
```

Анализ

Эта программа не имеет вывода, она демонстрирует применение различных способов создания экземпляров набора и мультимножества, специализированных для хранения элементов типа `int`. В строках 17 и 18 представлена самая простая форма, где 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, необходимо определить предикат, как это сделано в строках 3–10, и использовать его в функции `main()`, как в строках 21 и 22. Этот предикат обеспечивает сортировку по убыванию (стандартная по возрастанию). И наконец, строки 25 и 26 демонстрируют способы создания экземпляра набора как копии другого и экземпляра мультимножества из диапазона значений, взятых из набора (но это может быть вектор, список или любой другой контейнер библиотеки STL, возвращающий итераторы, которые описывают границы при помощи методов `cbegin()` и `cend()`).

СОВЕТ

Не будет ли ошибки компиляции при использовании методов `cbegin()` и `cend()`?

Если вы попытаетесь откомпилировать эту программу, используя компилятор, не совместимый со стандартом C++11, задействуйте методы `begin()` и `end()` вместо `cbegin()` и `cend()` соответственно. Методы `cbegin()` и `cend()` доступны только в C++11, они возвращают константный итератор, который не может быть использован для изменения элементов.

Вставка элементов в набор 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:

```
setIntegers.insert (-1);
msetIntegers.insert (setIntegers.begin (), setIntegers.end ());
```

В листинге 19.2 показана вставка элементов в эти контейнеры.

ЛИСТИНГ 19.2. Вставка элементов в набор и мультимножество библиотеки STL

```
 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 iElement = Input.cbegin () // auto, cebgin(): C++11
 8:            ; iElement != Input.cend ()  // cend(): C++11
 9:            ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: int main ()
16: {
17:     set <int> setIntegers;
19:
20:     setIntegers.insert (60);
21:     setIntegers.insert (-1);
22:     setIntegers.insert (3000);
23:     cout << "Writing the contents of the set to the screen" << endl;
24:     DisplayContents (setIntegers);
25:
26:     msetIntegers.insert (setIntegers.begin (), setIntegers.end ());
27:     msetIntegers.insert (3000);
28:
29:     cout << "Writing the contents of the multiset to the screen"
            << endl;
30:     DisplayContents (msetIntegers);
31:
32:     cout << "Number of instances of '3000' in the multiset are: '";
33:     cout << msetIntegers.count (3000) << "'" << endl;
34:
35:     return 0;
36: }
```

Результат

```
Writing the contents of the set to the screen
-1 60 3000
Writing the contents of the multiset to the screen
-1 60 3000 3000
Number of instances of '3000' in the multiset are: '2'
```

Анализ

Строки 4–13 содержат обобщенный шаблон функции `DisplayContents()`, который вы уже видели на занятиях 17 и 18, предназначенной для вывода содержимого контейнера STL на консоль или экран. Строки 17 и 18, как вы уже знаете, определяют объекты классов `set` и `multiset`. Строки 20–22 вставляют значения в набор, используя функцию-член `insert()`. Строка 26 демонстрирует применение функции `insert()` для вставки содержимого набора в мультимножество (в данном случае содержимого набора `setIntegers` в мультимножество `msetIntegers`). В строке 27 к мультимножеству добавляется элемент со значением 3000, которое уже существует в нем. Вывод демонстрирует, что мультимножество в состоянии содержать несколько одинаковых значений. Строки 32 и 33 демонстрируют удобство функции-члена `multiset::count()`, возвращающей количество элементов в мультимножестве, содержащем указанное значение.

СОВЕТ

Для поиска в мультимножестве количества элементов с одинаковым значением, переданным как аргумент, используйте функцию `multiset::count()`.

СОВЕТ

Возникла ошибка компиляции при использовании ключевого слова `auto`?

Функция `DisplayContents()` в листинге 19.2 использует ключевое слово C++11 `auto` для определения типа итератора в строке 7. Кроме того, она использует возвращающие итератор `const_iterator` функции `cbegin()` и `cend()`, которые совместимы только со стандартом C++11.

В этом и последующих примерах для компиляции с использованием компилятора, не совместимого со стандартом C++11, необходимо заменить ключевое слово `auto` явным типом.

Так, функцию `DisplayContents()` для устаревшего компилятора следует изменить следующим образом:

```
template <typename T>
void DisplayContents (const T& Input)
{
    for (T::const_iterator iElement = Input.begin () //
явный тип
                          ; iElement != Input.end ()
                          ; ++ iElement )
        cout << *iElement << ' ';
    cout << endl;
}
```

Поиск элементов в наборе и мультимножестве

Ассоциативные контейнеры, такие как `set`, `multiset`, `map` и `multimap`, предоставляют функцию-член `find()`, позволяющую находить значение по ключу:

```
auto iElementFound = setIntegers.find (-1);

// Проверить, если найдено...
if (iElementFound != setIntegers.end ())
    cout << "Element " << *iElementFound << " found!" << endl;
else
    cout << "Element not found in set!" << endl;
```

Использование функции `find()` представлено в листинге 19.3. В случае мультимножества, допускающего несколько элементов с одинаковым значением, эта функция находит первое, соответствующее 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> setIntegers;
 7:
 8:     // Вставить произвольные значения
 9:     setIntegers.insert (43);
10:     setIntegers.insert (78);
11:     setIntegers.insert (-1);
12:     setIntegers.insert (124);
13:
14:     // Вывод содержимого набора на экран
15:     for (auto iElement = setIntegers.cbegin ()
16:             ; iElement != setIntegers.cend ()
17:             ; ++ iElement )
18:         cout << *iElement << endl;
19:
20:     // Попытаться найти элемент
21:     auto iElementFound = setIntegers.find (-1);
22:
23:     // Проверить, если найдено...
24:     if (iElementFound != setIntegers.end ())
25:         cout << "Element " << *iElementFound << " found!" << endl;
26:     else
27:         cout << "Element not found in set!" << endl;
28:
29:     // Попытаться найти другой элемент
30:     auto iAnotherFind = setIntegers.find (12345);
31:
32:     // Проверить, если найдено...
33:     if (iAnotherFind != setIntegers.end ())
34:         cout << "Element " << *iAnotherFind << " found!" << endl;
35:     else
```

```
36:         cout << "Element 12345 not found in set!" << endl;
37:
38:     return 0;
39: }
```

Результат

```
-1
43
78
124
Element -1 found!
Element 12345 not found in set!
```

Анализ

Строки 21–27 демонстрируют применение функции-члена `find()`. Она возвращает итератор, сравнение которого с результатом функции `end()`, как показано в строке 24, позволяет проверить, был ли найден элемент. Если итератор допустим, можно обратиться к значению, на которое указывает `*iElementFound`.

ПРИМЕЧАНИЕ Пример в листинге 19.3 сработает правильно и для мультимножества, т.е. если в строке 6 `set` заменить на `multiset`, то код продолжит работать правильно.

Удаление элементов в наборе и мультимножестве

Ассоциативные контейнеры, такие как `set`, `multiset`, `map` и `multimap`, предоставляют функцию-член `erase()`, позволяющую удалять значение по ключу:

```
setObject.erase (key);
```

Другая форма функции `erase()` позволяет удалить определенный элемент, заданный указывающим на него итератором:

```
setObject.erase (iElement);
```

Вы можете стереть диапазон элементов набора или мультимножества, используя итераторы, задающие границы диапазона:

```
setObject.erase (iLowerBound, iUpperBound);
```

Пример в листинге 19.4 демонстрирует использование метода `erase()` для удаления элементов из набора или мультимножества.

ЛИСТИНГ 19.4. Использование функции-члена `erase()` для 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 iElement = Input.cbegin () // auto, cebgin(): C++11
 8:            ; iElement != Input.cend ()  // cend(): C++11
 9:            ; ++ iElement )
10:         cout << *iElement << ' ';
11:
12:     cout << endl;
13: }
14:
15: typedef multiset <int> MSETINT;
16:
17: int main ()
18: {
19:     MSETINT msetIntegers;
20:
21:     // Вставить произвольные значения
22:     msetIntegers.insert (43);
23:     msetIntegers.insert (78);
24:     msetIntegers.insert (78); // Совпадение
25:     msetIntegers.insert (-1);
26:     msetIntegers.insert (124);
27:
28:     cout << "multiset contains " << msetIntegers.size ()
            << " elements.";
29:     cout << " These are: " << endl;
30:
31:     // Вывод содержимого мультимножества на экран
32:     DisplayContents(msetIntegers);
33:
34:     cout << "Please enter a number to be erased from the set"
            << endl;
35:     int nNumberToErase = 0;
36:     cin >> nNumberToErase;
37:
38:     cout << "Erasing " << msetIntegers.count (nNumberToErase);
39:     cout << " instances of value " << nNumberToErase << endl;
40:
41:     // Попытаться найти элемент
42:     msetIntegers.erase (nNumberToErase);
43:
44:     cout << "multiset contains " << msetIntegers.size ()
            << " elements.";
45:     cout << " These are: " << endl;
46:     DisplayContents(msetIntegers);
47:
48:     return 0;
49: }
```

Результат

```
multiset contains 5 elements. These are:
-1 43 78 78 124
Please enter a number to be erased from the set
```

```
78
Erasing 2 instances of value 78
multiset contains 3 elements. These are:
-1 43 124
```

Анализ

Обратите внимание на использование в строке 15 ключевого слова `typedef`. Строка 38 демонстрирует применение функции `count()` для выяснения количества элементов с определенным значением. Фактическое удаление осуществляется в строке 42, где удаляются все элементы, которые соответствуют определенному числу.

Обратите внимание на то, что функция `erase()` имеет несколько перегруженных версий. Ее можно вызвать для итератора, возвращенного, скажем, в результате поиска, чтобы удалить один элемент с найденным значением, как показано ниже.

```
MSETINT::iterator iElementFound = msetIntegers.find (nNumberToErase);
if (iElementFound != msetIntegers.end ())
    msetIntegers.erase (iElementFound);
else
    cout << "Element not found!" << endl;
```

Точно так же вы можете использовать функцию `erase()` для удаления из мультимножества диапазона значений:

```
MSETINT::iterator iElementFound = msetIntegers.find (nValue);

if (iElementFound != msetIntegers.end ())
    msetIntegers.erase (msetIntegers.begin (), iElementFound);
```

Приведенный выше фрагмент удаляет все элементы от начала до элемента со значением `nValue`, не включая последнего. И набор, и мультимножество могут быть освобождены от своего содержимого при помощи функции-члена `clear()`.

Теперь, после краткого обзора базовых функций набора 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 их все.

ЛИСТИНГ 19.5. Телефонный справочник, демонстрирующий возможности класса `set` библиотеки STL

```
 0: #include <set>
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents (const T& Input)
 7: {
 8:     for(auto iElement = Input.cbegin () // auto, cebgin(): C++11
 9:           ; iElement != Input.cend ()  // cend(): C++11
10:           ; ++ iElement )
11:         cout << *iElement << endl;
12:
```

```
13:     cout << endl;
14: }
15:
16: struct ContactItem
17: {
18:     string strContactsName;
19:     string strPhoneNumber;
20:     string strDisplayRepresentation;
21:
22:     // Конструктор и деструктор
23:     ContactItem (const string& strName, const string & strNumber)
24:     {
25:         strContactsName = strName;
26:         strPhoneNumber = strNumber;
27:         strDisplayRepresentation = (strContactsName + ": " \
                                    + strPhoneNumber);
28:     }
29:
30:     // используется в set::find()
31:     bool operator == (const ContactItem& itemToCompare) const
32:     {
33:         return (itemToCompare.strContactsName == \
                 this->strContactsName);
34:     }
35:
36:     // используется как предикат сортировки
37:     bool operator < (const ContactItem& itemToCompare) const
38:     {
39:         return (this->strContactsName \
                       < itemToCompare.strContactsName);
40:     }
41:
42:     // Используется в DisplayContents для cout
43:     operator const char*() const
44:     {
45:         return strDisplayRepresentation.c_str();
46:     }
47: };
48:
49: int main ()
50: {
51:     set<ContactItem> setContacts;
52:     setContacts.insert(ContactItem("Jack Welsch",
                                       "+1 7889 879 879"));
53:     setContacts.insert(ContactItem("Bill Gates",
                                       "+1 97 7897 8799 8"));
54:     setContacts.insert(ContactItem("Angela Merkel",
                                       "+49 23456 5466"));
55:     setContacts.insert(ContactItem("Vladimir Putin",
                                       "+7 6645 4564 797"));
56:     setContacts.insert(ContactItem("Manmohan Singh",
                                       "+91 234 4564 789"));
57:     setContacts.insert(ContactItem("Barack Obama",
                                       "+1 745 641 314"));
```

```
58:     DisplayContents(setContacts);
59:
60:     cout << "Enter a person whose number you wish to delete: ";
61:     string NameInput;
62:     getline(cin, NameInput);
63:
64:     auto iContactFound = setContacts.find(ContactItem(NameInput,
                                                  ""));
65:     if(iContactFound != setContacts.end())
66:     {
67:         // Удалить контакт, найденный в наборе
68:         setContacts.erase(iContactFound);
69:         cout << "Displaying contents after erasing: " << NameInput
                 << endl;
70:         DisplayContents(setContacts);
71:     }
72:     else
73:         cout << "Contact not found" << endl;
74:
75:     return 0;
76: }
```

Результат

```
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Jack Welsch: +1 7889 879 879
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797

Enter a person whose number you wish to delete: Jack Welsch
Displaying contents after erasing: Jack Welsch
Angela Merkel: +49 23456 5466
Barack Obama: +1 745 641 314
Bill Gates: +1 97 7897 8799 8
Manmohan Singh: +91 234 4564 789
Vladimir Putin: +7 6645 4564 797
```

Анализ

Это очень похоже на листинг 18.7, где список был отсортирован в алфавитном порядке, но в данном случае сортировка набора осуществляется по вставке. Как демонстрирует вывод, не нужно вызывать никаких функций, чтобы гарантировать сортировку элементов в наборе, поскольку они сортируются на вставке. Вы позволяете пользователю удалить запись, и в строке 64 демонстрируется вызов функции `find()`, позволяющей найти ту запись, которая удаляется в строке 68 при помощи метода `erase()`.

СОВЕТ

Эта реализация телефонного справочника основана на классе `set` библиотеки STL, а потому она не позволяет содержать несколько записей с одинаковым значением. Если необходима реализация справочника, позволяющая хранить две записи с одинаковым именем (скажем, Том), то выбирайте класс `multiset` библиотеки STL. Если контейнер `setContacts` станет мультимножеством, то приведенный выше код продолжит работать правильно. Чтобы развить возможность мультимножества хранить несколько записей с одинаковым значением, используйте функцию-член `count()`, чтобы знать количество элементов, содержащих определенное значение. Это представлено в предыдущем примере кода. Подобные элементы помещаются в мультимножестве рядом, и функция `find()` возвращает итератор первого найденного значения. Приращение этого итератора позволит обратиться к следующим найденным элементам.

Преимущества и недостатки использования наборов и мультимножеств

Классы `set` и `multiset` библиотеки STL предоставляют существенные преимущества приложениям, нуждающимся в частых поисках. Поскольку их содержимое отсортировано, поиск осуществляется быстрей. Но, чтобы предоставить это преимущество, контейнер должен сортировать элементы во время вставки. Таким образом, при вставке элементов есть дополнительные затраты на их сортировку, являющиеся необходимой платой за возможность часто использовать такие функции, как `find()`.

Функция `find()` использует внутреннюю двоичную древовидную структуру. Эта отсортированная двоичная древовидная структура является причиной другого неявного недостатка 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 набора элементы сортируются согласно их значениям, а потому никогда не следует допускать перезаписи элемента с помощью итератора, даже если бы программно это было возможно.

C++11

Реализация хеш-набора библиотеки STL — классы `std::unordered_set` и `std::unordered_multiset`

Контейнеры `std::set` и `std::multiset` библиотеки STL сортируют элементы (которые одновременно являются ключами) на основании предиката `std::less<T>` или предиката, предоставленного пользователем. Поиск в отсортированном контейнере быстрее, чем в не отсортированном, таком, как вектор, а метод `sort()` обеспечивает логарифмическую сложность. Это означает, что время, потраченное на поиск элемента в наборе, не прямо пропорционально количеству элементов в нем, а скорее его логарифму. Таким образом, поиск среди 10 000 элементов набора осуществляется вдвое дольше, чем среди 100 элементов (так, 100^2 = 10000 или log(10000) = 2×log(100)).

Однако даже этого существенного увеличения производительности по сравнению с не отсортированным контейнером (где продолжительность поиска прямо пропорциональна количеству элементов) иногда недостаточно. Программисты и математики упорно ищут способ осуществления вставки и сортировки за постоянное время, и одним из них является реализация на базе хеша, где хеш-функция используется для определения индекса сортировки. Добавляемые в хеш-набор элементы сначала обрабатываются хеш-функцией, которая создает уникальный индекс ячейки, в которой они размещаются. Библиотека STL предоставляет свой вариант хеш-набора в виде класса контейнера `std::unordered_set`.

СОВЕТ

Чтобы использовать классы контейнеров `std::unordered_set` или `std::unordered_multiset`, включите в код их заголовок:

```
#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);

// find():
auto iPairThousand = usetInt.find(1000);

if (iPairThousand != usetInt.end())
    cout << *iPairThousand << endl;
```

Но все же одной из важнейших особенностей контейнера `unordered_map` является доступность хеш-функции, отвечающей за решение о порядке сортировки:

```
unordered_set<int>::hasher HFn = usetInt.hash_function();
```

В листинге 19.6 показано применение некоторых из наиболее популярных действий, поддерживаемых классом `std::hash_set`.

ЛИСТИНГ 19.6. Применение методов `insert()`, `find()`, `size()`, `max_bucket_count()`, `load_factor()` и `max_load_factor()` 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& Input)
 6: {
 7:     cout << "Number of elements, size() = " << Input.size() << endl;
 8:     cout << "Max bucket count = " << Input.max_bucket_count()
          << endl;
 9:     cout << "Load factor: " << Input.load_factor() << endl;
10:     cout << "Max load factor = " << Input.max_load_factor() << endl;
11:     cout << "Unordered set contains: " << endl;
12:
```

```
13:     for(auto iElement = Input.cbegin() // auto, cbegin: C++11
14:             ; iElement != Input.cend()  // cend: C++11
15:             ; ++ iElement )
16:         cout<< *iElement << ' ';
17:
18:     cout<< endl;
19: }
20:
21: int main()
22: {
23:     // создание экземпляра контейнера unordered_set:
24:     unordered_set<int> usetInt;
25:
26:     usetInt.insert(1000);
27:     usetInt.insert(-3);
28:     usetInt.insert(2011);
29:     usetInt.insert(300);
30:     usetInt.insert(-1000);
31:     usetInt.insert(989);
32:     usetInt.insert(-300);
33:     usetInt.insert(111);
34:     DisplayContents(usetInt);
35:     usetInt.insert(999);
36:     DisplayContents(usetInt);
37:
38:     // find():
39:     cout << "Enter int you want to check for existence in set: ";
40:     int Key = 0;
41:     cin >> Key;
42:     auto iPairThousand = usetInt.find(Key);
43:
44:     if (iPairThousand != usetInt.end())
45:         cout << *iPairThousand << " found in set" << endl;
46:     else
47:         cout << Key << " not available in set" << endl;
48:
49:     return 0;
50: }
```

Результат

```
Number of elements, size() = 8
Max bucket count = 8
Load factor: 1
Max load factor = 1
Unordered set contains:
1000 -3 2011 300 -1000 -300 989 111
Number of elements, size() = 9
Max bucket count = 64
Load factor: 0.140625
Max load factor = 1
Unordered set contains:
1000 -3 2011 300 -1000 -300 989 999 111
Enter int you want to check for existence in set: -1000
-1000 found in set
```

Анализ

Здесь создается контейнер `unordered_set` для целых чисел; в него вставляется восемь значений, а затем содержимое отображается на экране, включая поставляемую методами `max_bucket_count()`, `load_factor()` и `max_load_factor()` статистику, как показано в строках 8–10. Вывод свидетельствует о том, что счет ячеек начинается с восьми, с восьмью элементами в контейнере и коэффициентом загрузки 1, который является максимальным. Когда в контейнер `unordered_set` вставляется девятый элемент, он реорганизует себя, создаст 64 ячейки и воссоздаст хеш-таблицу, а коэффициент загрузки уменьшится. Остальная часть кода в `main()` демонстрирует, что синтаксис поиска элементов в контейнере `unordered_set` подобен таковому в контейнере `set`. Метод `find()` возвращает итератор, успех выполнения которого должен быть проверен, как показано в строке 42, 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 карт библиотеки STL".

Контейнер `std::unordered_map` является реализацией хеш-таблицы, появившейся в C++11.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Помните, что контейнеры `set` и `multiset` библиотеки STL оптимизированы для ситуаций с частым поиском	**Не забывайте** определять операторы `operator<` и `operator==` для классов, объекты которых могут храниться в таких контейнерах, как `set` и `multiset`. Первый становится предикатом сортировки, а последний используется для таких функций, как `set::find()`
Помните, что контейнер `std::multiset` допускает несколько одинаковых элементов (ключей), а контейнер `std::set` разрешает хранить только уникальные значения	**Не используйте** контейнеры `std::set` и `std::multiset` в случаях с частыми вставками и нечастыми поисками. Для этого обычно лучше подходят такие контейнеры, как `std::vector` и `std::list`
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	

Резюме

На сегодняшнем занятии рассматривались контейнеры `set` и `multiset` библиотеки STL, их основные функции-члены и характеристики. Вы также видели их применение для разработки простого меню телефонного справочника, реализующего также функции поиска и удаления.

Вопросы и ответы

- **Как мне объявить набор целых чисел, отсортированных и хранящихся в порядке убывания величин?**

  Шаблон класса `set<int>` определяет набор 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 набор как `set<int, greater<int>>`.

- **Что будет, если вставить строку "Jack" в набор строк дважды?**

  Набор не предназначен для хранения совпадающих значений. Реализация класса `std::set` не позволила бы вставить второе значение.

- **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 наборе и имею теперь указывающий на него итератор. Как мне использовать этот итератор для изменения значения, на которое он указывает?**

  Никак. Некоторые реализации библиотеки STL могли бы позволить пользователю изменить значение элемента в наборе при помощи итератора, возвращенного, например, функцией `find()`. Но так поступать неправильно. Итератор на элемент набора должен использоваться как константный, даже если реализация библиотеки STL не препятствует этому.

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 Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Вы объявляете набор целых чисел так `set<int>`. Какая функция предоставит критерий сортировки?
2. Как совпадающие элементы располагаются в контейнере `multiset`?
3. Какая функция контейнеров `set` и `multiset` возвращает количество элементов в нем?

Упражнения

1. Дополните пример телефонного справочника этого занятия поиском имени человека по данному номеру телефона без изменения структуры `ContactItem`. (Подсказка: определите набор с бинарным предикатом, сортирующим элементы по номеру, переопределяя таким образом сортировку по умолчанию на основании оператора <.)
2. Определите мультимножество для хранения введенных слов и их значений, т.е. создайте мультимножество как словарь. (Подсказка: мультимножество должно хранить объекты структуры, которая содержит две строки: слово и его значение.)
3. Представьте простую программу, демонстрирующую, что набор не может хранить совпадающие записи, а мультимножество может.

ЗАНЯТИЕ 20
Классы карт библиотеки STL

Стандартная библиотека шаблонов (STL) предоставляет разработчикам классы контейнеров для приложений, которым требуются частые и быстрые поиски.

На сегодняшнем занятии.

- Чем могут быть полезны классы `map`, `multimap`, `unordered_map` и `unordered_multimap` библиотеки STL.
- Вставка, удаление и поиск элементов.
- Предоставление специального предиката сортировки.
- Основы работы хеш-таблиц.

Введение в классы карт библиотеки STL

Классы `map` и `multimap` — контейнеры пар "ключ–значение", допускающие поиск на основе ключа, как показано на рис. 20.1.

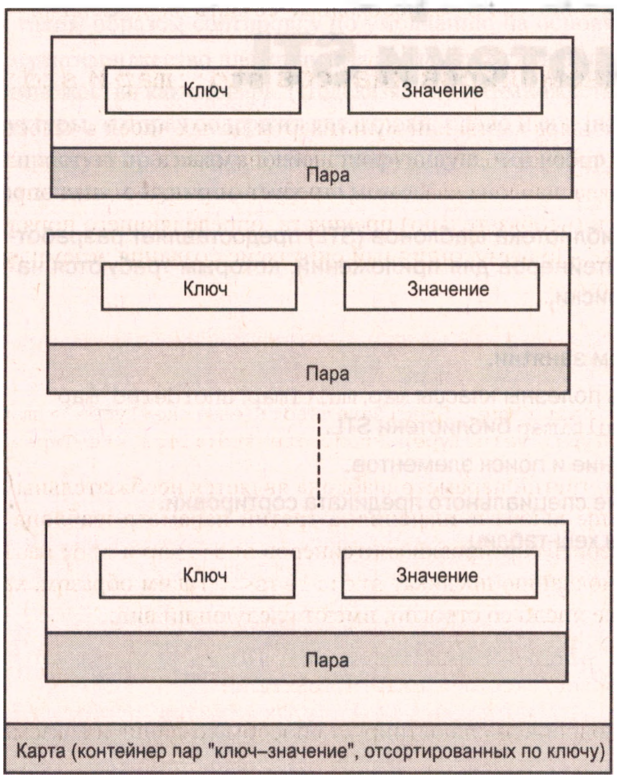


РИС. 20.1. Визуальное представление контейнеров пар, содержащих ключи и значения

Различие между картой (`map`) и мультикартой (`multimap`) в том, что только последняя допускает совпадающие элементы, а первая позволяет хранить только уникальные ключи.

Чтобы облегчить быстрый поиск, реализации классов `map` и `multimap` библиотеки STL внутренне выглядят как двоичные деревья. Это значит, что элементы, вставленные в карту или мультикарту, сортируются по порядку вставки. Это также означает, что в отличие от вектора, где элементы в определенной позиции могут быть заменены другими, элементы карты в определенной позиции не могут быть заменены новым элементом с другим значением. Дело в том, что карта поместит его в другую область внутреннего дерева, в соответствии со значением относительно других.

СОВЕТ

Чтобы использовать класс `std::map` или `std::multimap`, включите их заголовок:

```
#include<map>
```

Простые операции с классами std::map и std::multimap библиотеки STL

Прежде чем вы сможете использовать любую из функций-членов шаблонов классов map и multimap библиотеки STL, необходимо создать их экземпляр.

Создание экземпляров классов std::map и std::multimap

Создание экземпляра карты или мультикарты целых чисел в качестве ключа и строк в качестве значений требует специализации шаблона класса std::map или std::multimap. Создание экземпляра шаблона класса map требует от разработчика определения типа ключа, типа значения и (необязательно) предиката, определяющего порядок сортировки элементов при вставке. Поэтому типичный синтаксис создания экземпляра карты 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, а третий параметр шаблона игнорируется, для определения критериев сортировки контейнеров std::map и std::multimap используется заданный по умолчанию предикат std::less<>. Таким образом, карта и мультикарта, соотносящие целое число со строкой, имеют следующий вид:

```
std::map<int, string> mapIntToString;
std::multimap<int, string> mmapIntToString;
```

Листинг 20.1 подробней иллюстрирует способы создания их экземпляров.

ЛИСТИНГ 20.1. Создание экземпляров объектов map и multimap, соотносящих целочисленные ключи со строковыми значениями

```
 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:     // карта и мультикарта ключей типа int со значениями типа string
```

```
17:     map<int, string> mapIntToString1;
18:     multimap<int, string> mmapIntToString1;
19:
20:     // карта и мультикарта создаются как копия другого контейнера
21:     map<int, string> mapIntToString2(mapIntToString1);
22:     multimap<int, string> mmapIntToString2(mmapIntToString1);
23:
24:     // карта и мультикарта создаются как часть другого контейнера
25:     map<int, string> mapIntToString3(mapIntToString1.cbegin(),
26:                                      mapIntToString1.cend());
27:
28:     multimap<int, string> mmapIntToString3(mmapIntToString1.cbegin(),
29:                                            mmapIntToString1.cend());
30:
31:     // карта и мультикарта с предикатом сортировки в обратном порядке
32:     map<int, string, ReverseSort<int> > mapIntToString4
33:         (mapIntToString1.cbegin(), mapIntToString1.cend());
34:
35:     multimap<int, string, ReverseSort<int> > mmapIntToString4
36:         (mapIntToString1.cbegin(), mapIntToString1.cend());
37:
38:     return 0;
39: }
```

Анализ

Для начала сосредоточимся на строках 12–39 функции `main()`. Простейшие карта и мультикарта целочисленных ключей и строковых значений создаются в строках 21 и 22. Строки 25–28 демонстрируют создание карты или мультикарты, инициализированных диапазоном значений из других контейнеров. Строки 31–36 демонстрируют создание экземпляров карты и мультикарты с собственным критерием сортировки. Обратите внимание, что сортировка по умолчанию (в предыдущих экземплярах) использует предикат `std::less<T>`, который сортировал бы элементы в порядке возрастания. Если вы хотите изменить это поведение, предоставьте предикат, который является классом или структурой, реализующей оператор `operator()`. Такая структура предиката `ReverseSort` находится в строках 4–10 и используется при создании экземпляра карты в строке 32 и мультикарты в строке 35.

СОВЕТ

Не будет ли ошибки компиляции при использовании методов `cbegin()` и `cend()`?

Если вы попытаетесь откомпилировать эту программу, используя компилятор не совместимый со стандартом C++11, задействуйте методы `begin()` и `end()` вместо `cbegin()` и `cend()` соответственно. Методы `cbegin()` и `cend()` доступны только в C++11, они возвращают константный итератор, который не может быть использован для изменения элементов.

Вставка элементов в карту или мультикарту библиотеки STL

Большинство функций карты и мультикарты 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> mapIntToString1;
// вставить пару ключа и значения с использованием функции make_pair()
mapIntToString.insert (make_pair (-1, "Minus One"));
```

Поскольку элементы этих двух контейнеров содержат пары "ключ–значение", вы можете также непосредственно вставлять инициализированные пары `std::pair`:

```
mapIntToString.insert (pair <int, string>(1000, "One Thousand"));
```

В качестве альтернативы можете использовать для вставки синтаксис, как у массива, который привычен пользователю и поддерживается оператором индексирования `operator[]`:

```
mapIntToString [1000000] = "One Million";
```

Вы можете также создать экземпляр мультикарты как копию карты:

```
std::multimap<int, std::string> mmapIntToString(mapIntToString.cbegin(),
mapIntToString.cend());
```

В листинге 20.2 приведены различные методы создания экземпляра.

ЛИСТИНГ 20.2. Вставка элементов в карту и мультикарту с использованием перегруженного метода `insert()` и семантики массива с применением оператора `operator[]`

```
 0: #include <map>
 1: #include <iostream>
 2: #include<string>
 3:
 4: using namespace std;
 5:
 6: // определение типа карты и мультикарты 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& Input)
12: {
13:     for(auto iElement = Input.cbegin() // auto and cbegin(): C++11
14:           ; iElement != Input.cend()  // cend(): C++11
15:           ; ++ iElement )
16:         cout << iElement->first << " -> " << iElement->second
                << endl;
17:
18:     cout << endl;
19: }
20:
21: int main ()
22: {
23:     MAP_INT_STRING mapIntToString;
24:
25:     // Вставка пары "ключ-значение" в карту с использованием
       // ключевого слова value_type
```

```
26:     mapIntToString.insert (MAP_INT_STRING::value_type (3, "Three"));
27:
28:     // Вставка пары с использованием функции make_pair()
29:     mapIntToString.insert (make_pair (-1, "Minus One"));
30:
31:     // Вставка объекта пары непосредственно
32:     mapIntToString.insert (pair <int, string>(1000, "One Thousand"));
33:
34:     // Вставка пары "ключ-значение" с использованием синтаксиса
        // массива
35:     mapIntToString [1000000] = "One Million";
36:
37:     cout << "The map contains " << mapIntToString.size ();
38:     cout << " key-value pairs. They are: " << endl;
39:     DisplayContents(mapIntToString);
40:
41:     // Создание экземпляра мультикарты, являющейся копией карты
42:     MMAP_INT_STRING mmapIntToString(mapIntToString.cbegin(),
43:                                     mapIntToString.cend());
44:
45:     // Функция insert() работает так же, как у мультикарты
46:     // Мультикарта может хранить дубликаты. Вставка дубликата
47:     mmapIntToString.insert (make_pair (1000, "Thousand"));
48:
49:     cout << endl << "The multimap contains "
             << mmapIntToString.size ();
50:     cout << " key-value pairs. They are: " << endl;
51:     cout << "The elements in the multimap are: " << endl;
52:     DisplayContents(mmapIntToString);
53:
54:     // Мультикарта способна возвратить количество пар с тем же ключом
55:     cout << \
        "The number of pairs in the multimap with 1000 as their key:"
56:     << mmapIntToString.count (1000) << endl;
57:
58:     return 0;
59: }
```

Результат

```
The map contains 4 key-value pairs. They are:
-1 -> Minus One
3 -> Three
1000 -> One Thousand
1000000 -> One Million

The multimap contains 5 key-value pairs. They are:
The elements in the multimap are:
-1 -> Minus One
3 -> Three
1000 -> One Thousand
1000 -> Thousand
1000000 -> One Million

The number of pairs in the multimap with 1000 as their key: 2
```

Анализ

Обратите внимание на определение типа для создания экземпляров шаблонов `map` и `multimap` в строках 7 и 8. В результате код будет выглядеть немного проще (и сократит синтаксис шаблона). Строки 10–19 содержат версию функции `DisplayContents()`, адаптированную для карты и мультикарты, в которой для доступа используется итератор `first`, указывающий на ключ, и `second`, указывающий на значение. Строки 26–32 содержат различные способы вставки пар "ключ–значение" в карту с использованием перегруженной версии метода `insert()`. Строка 35 демонстрирует возможность использования семантики массива при помощи оператора `operator[]` для вставки элементов в карту. Обратите внимание на то, что эти механизмы вставки работают также для мультикарты, которая представлена в строке 47, где в нее вставляется дубликат. Интересно, что мультикарта инициализируется как копия карты (строки 42 и 43). Вывод показывает, что эти два контейнера автоматически сортируют вставляемые пары "ключ–значение" в порядке возрастания ключей. Вывод также демонстрирует, что мультикарта способна хранить пары с одинаковым ключом (в данном случае 1000). Строка 56 демонстрирует применение метода `multimap::count()`, сообщающего количество элементов в контейнере с указанным ключом.

СОВЕТ

Возникла ошибка компиляции при использовании ключевого слова `auto`?

Функция `DisplayContents()` в листинге 20.2 использует ключевое слово C++11 `auto` для определения типа итератора в строке 13. Кроме того, она использует возвращающие итератор `const_iterator` функции `cbegin()` и `cend()`, которые совместимы только со стандартом C++11.

В этом и последующих примерах для компиляции с использованием компилятора, не совместимого со стандартом C++11, необходимо заменить ключевое слово `auto` явным типом.

Так, функцию `DisplayContents()` для устаревшего компилятора следует изменить следующим образом:

```
template <typename T>
void DisplayContents (const T& Input)
{
    for (T::const_iterator iElement = Input.begin ()
                         ; iElement != Input.end ()
                         ; ++ iElement )
        cout << iElement->first << " -> " << iElement->second
             << endl;
    cout << endl;
}
```

Поиск элементов в карте STL

Ассоциативные контейнеры, такие как `map` и `multimap`, предоставляют функцию-член `find()`, позволяющую находить значения по данному ключу. Результат операции поиска всегда итератор:

```
multimap <int, string>::const_iterator iPairFound = mapIntToString.
find(Key);
```

Но прежде чем использовать этот итератор для доступа к найденному значению, необходимо проверить успех выполнения метода `find()`:

```
if (iPairFound != mapIntToString.end())
{
    cout << "Key " << iPairFound->first << " points to Value: ";
    cout << iPairFound->second << endl;
}
else
    cout << "Sorry, pair with key " << Key << " not in map" << endl;
```

СОВЕТ

> Если вы используете компилятор, совместимый со стандартом C++11, при объявлении итератора очень удобно использовать ключевое слово `auto`:
>
> `auto iPairFound = mapIntToString.find(Key);`
>
> Компилятор определяет тип итератора автоматически, выводя его из заявленного возвращаемого значения функции `map::find()`.

Пример в листинге 20.3 демонстрирует применение функции `multimap::find()`.

ЛИСТИНГ 20.3. Использование функции-члена `find()` для поиска в карте пары "ключ–значение"

```
 0: #include <map>
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: template <typename T>
 6: void DisplayContents (const T& Input)
 7: {
 8:     for(auto iElement = Input.cbegin () // auto and cbegin(): C++11
 9:           ; iElement != Input.cend()   // cend(): C++11
10:           ; ++ iElement )
11:         cout << iElement->first << " -> " << iElement->second
                 << endl;
12:
13:     cout << endl;
14: }
15:
16: int main()
17: {
18:     map<int, string> mapIntToString;
19:
20:     mapIntToString.insert(make_pair(3, "Three"));
21:     mapIntToString.insert(make_pair(45, "Forty Five"));
22:     mapIntToString.insert(make_pair(-1, "Minus One"));
23:     mapIntToString.insert(make_pair(1000, "Thousand"));
24:
25:     cout << "The multimap contains " << mapIntToString.size();
```

```
26:     cout << " key-value pairs. They are: " << endl;
27:
28:     // Вывод содержимого карты на экран
29:     DisplayContents(mapIntToString);
30:
31:     cout << "Enter the key you wish to find: ";
32:     int Key = 0;
33:     cin >> Key;
34:
35:     auto iPairFound = mapIntToString.find(Key);
36:     if (iPairFound != mapIntToString.end())
37:     {
38:         cout << "Key " << iPairFound->first << " points to Value: ";
39:         cout << iPairFound->second << endl;
40:     }
41:     else
42:         cout << "Sorry, pair with key " << Key << " not in map"
                 << endl;
43:
44:     return 0;
45: }
```

Результат

```
The multimap contains 4 key-value pairs. They are:
-1 -> Minus One
3 -> Three
45 -> Forty Five
1000 -> Thousand

Enter the key you wish to find: 45
Key 45 points to Value: Forty Five
Следующий запуск (где функция find() не находит соответствующего значения):
The multimap contains 4 key-value pairs. They are:
-1 -> Minus One
3 -> Three
45 -> Forty Five
1000 -> Thousand

Enter the key you wish to find: 2011
Sorry, pair with key 2011 not in map
```

Анализ

Строки 20–23 функции `main()` заполняют карту примерами пар, каждая из которых соотносит целочисленный ключ со строковым значением. Когда пользователь вводит ключ, функция `find()` в строке 35 выполняет его поиск в карте. Функция `map::find()` всегда возвращает итератор, который всегда имеет смысл проверять, сравнивая с итератором, возвращаемым методом `end()`, как показано в строке 36. Если итератор действительно допустим, для доступа к значению используется член класса `second` (строка 39). При втором запуске вы вводите ключ `2011`, которого нет в карте, и пользователю отображается сообщение об ошибке.

ВНИМАНИЕ!

Никогда не используйте результат функции `find()` непосредственно, не проверив возвращенный итератор.

Поиск элементов в мультикарте STL

Если бы в листинге 20.3 использовалась мультикарта, появилась бы возможность хранить в контейнере несколько пар с одинаковым ключом и необходимость поиска в значениях, ключи которых совпадают. Следовательно, в случае мультимножества вы использовали бы метод `multiset::count()` для поиска количества значений, соответствующих ключу, а приращение итератора позволило бы получить доступ к следующим значениям:

```
auto iPairFound = mmapIntToString.find(Key);

// Проверка успеха поиска
if(iPairFound != mmapIntToString.end())
{
    // Найти количество пар с тем же предоставленным ключом
    size_t nNumPairsInMap = mmapIntToString.count(1000);

    for( size_t nValuesCounter = 0
              ; nValuesCounter < nNumPairsInMap // оставаться в границах
              ; ++ nValuesCounter )
    {
        cout << "Key: " << iPairFound->first; // ключ
        cout << ", Value [" << nValuesCounter << "] = ";
        cout << iPairFound->second << endl;   // значение

        ++ iPairFound;
    }
}
else
    cout << "Element not found in the multimap";
```

Стирание элементов из карты или мультикарты STL

Карта и мультикарта предоставляют функцию-член `erase()`, которая удаляет элементы из контейнера. Для удаления всех пар с определенным ключом его следует передать функции `erase()` как параметр:

```
mapObject.erase (key);
```

Другая форма функции `erase()` позволяет удалить определенный элемент по указывающему на него итератору:

```
mapObject.erase (iElement);
```

Вы можете удалить диапазон элементов из карты или мультикарты, используя итераторы, обозначающие их границы:

```
mapObject.erase (iLowerBound, iUpperBound);
```

В листинге 20.4 показано применение функции `erase()`.

ЛИСТИНГ 20.4. Удаление элементов из мультикарты

```
 0: #include<map>
 1: #include<iostream>
 2: #include<string>
 3: using namespace std;
 4:
 5: template<typename T>
 6: void DisplayContents(const T& Input)
 7: {
 8:     for(auto iElement = Input.cbegin () // auto и cbegin(): C++11
 9:            ; iElement != Input.cend()   // cend(): C++11
10:            ; ++ iElement )
11:         cout<< iElement->first<< " -> "<< iElement->second<< endl;
12:
13:     cout<< endl;
14: }
15:
16: int main()
17: {
18:     multimap<int, string> mmapIntToString;
19:
20:     // Вставка пар "ключ-значение" в мультикарту
21:     mmapIntToString.insert(make_pair(3, "Three"));
22:     mmapIntToString.insert(make_pair(45, "Forty Five"));
23:     mmapIntToString.insert(make_pair(-1, "Minus One"));
24:     mmapIntToString.insert(make_pair(1000, "Thousand"));
25:
26:     // Вставка дубликатов в мультикарту
27:     mmapIntToString.insert(make_pair(-1, "Minus One"));
28:     mmapIntToString.insert(make_pair(1000, "Thousand"));
29:
30:     cout<< "The multimap contains "<< mmapIntToString.size();
31:     cout<< " key-value pairs. "<< "They are: "<< endl;
32:     DisplayContents(mmapIntToString);
33:
34:     // Удаление элемента с ключом -1 из мультикарты
35:     auto NumPairsErased = mmapIntToString.erase(-1);
36:     cout<< "Erased " << NumPairsErased << " pairs with -1 as key."
            << endl;
37:
38:     // Удаление из мультикарты элемента по данному итератору
39:     auto iPairLocator = mmapIntToString.find(45);
40:     if(iPairLocator != mmapIntToString.end())
41:     {
42:         mmapIntToString.erase(iPairLocator);
43:         cout<< "Erased a pair with 45 as key using an iterator"
                << endl;
44:     }
45:
46:     // Удаление из мультикарты диапазона...
47:     cout<< "Erasing the range of pairs with 1000 as key."<< endl;
48:     mmapIntToString.erase( mmapIntToString.lower_bound(1000)
49:                          , mmapIntToString.upper_bound(1000) );
50:
```

```
51:     cout<< "The multimap now contains "<< mmapIntToString.size();
52:     cout<< " key-value pair(s)."<< "They are: "<< endl;
53:     DisplayContents(mmapIntToString);
54:
55:     return 0;
56: }
```

Результат

```
The multimap contains 6 key-value pairs. They are:
-1 -> Minus One
-1 -> Minus One
3 -> Three
45 -> Forty Five
1000 -> Thousand
1000 -> Thousand

Erased 2 pairs with -1 as key.
Erased a pair with 45 as key using an iterator
Erasing the range of pairs with 1000 as key.
The multimap now contains 1 key-value pair(s).They are:
3 -> Three
```

Анализ

Код строк 21–28 вставляет в мультикарту примеры значений, некоторые из которых являются дубликатами (поскольку мультикарта, в отличие от карты, допускает вставку совпадающих элементов). После того как пары были вставлены в мультикарту, код удаляет элементы при помощи той версии функции `erase()`, которая получает ключ и удаляет все элементы с этим ключом (-1), как показано в строке 35. Функция `map::erase(Key)` возвращает количество удаленных элементов, которое и отображается на экране. В строке 39 вы используете итератор, возвращенный функцией `find(45)`, для удаления из карты пары с ключом 45. Строки 48 и 49 демонстрируют удаление диапазона пар, определенного методами `lower_bound()` и `upper_bound()`.

Предоставление специального предиката сортировки

Определения шаблонов `map` и `multimap` включают третий параметр, позволяющий передать предикат сортировки для их правильного функционирования. Если не предоставить этот третий параметр (как в примерах выше), применяется критерий сортировки по умолчанию, обеспечиваемый предикатом `std::less<>`, который, по существу, сравнивает два объекта, используя оператор `operator<`.

Чтобы предоставить другой критерий сортировки, предоставьте бинарный предикат в форме класса или структуры, реализующей оператор `operator()`:

```
template<typename ТипКлюча>
struct Predicate
{
    bool operator()(const ТипКлюча& key1, const ТипКлюча& key2)
```

```
    {
        // здесь ваша логика приоритета сортировки
    }
};
```

Карта, содержащая ключ типа `std::string`, имеет по умолчанию критерий сортировки на основании оператора <, определенного в классе `std::string` и применяемого заданным по умолчанию предикатом сортировки `std::less<T>`, а потому чувствительным к регистру. Это позволяет многим приложениям, таким как телефонный справочник, обеспечить вставку и поиск, которые не чувствительны к регистру. Один из способов решения этой задачи заключается в предоставлении карте предиката сортировки, возвращающего значение `true` или `false` в зависимости от результата сравнения без учета регистра:

```
map <типКлюча, типЗначения, Предикат> объектКарты;
```

В листинге 20.5 это продемонстрировано подробно.

ЛИСТИНГ 20.5. Предоставление специального предиката сортировки — телефонный справочник

```
 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& Input)
 8: {
 9:     for(auto iElement = Input.cbegin () // auto и cbegin(): C++11
10:           ; iElement != Input.cend()   // cend(): C++11
11:           ; ++ iElement )
12:         cout << iElement->first << " -> " << iElement->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:         std::transform(str1.begin(), str1.end(), str1NoCase.begin(),
                           tolower);
23:         std::transform(str2.begin(), str2.end(), str2NoCase.begin(),
                           tolower);
24:
25:         return(str1NoCase< str2NoCase);
26:     };
27: };
28:
29: typedef map<string, string> DIRECTORY_WITHCASE;
30: typedef map<string, string, PredIgnoreCase> DIRECTORY_NOCASE;
31:
```

```
32: int main()
33: {
34:     // Не зависящий от регистра каталог: регистр строкового ключа
        // не имеет значения
35:     DIRECTORY_NOCASE dirCaseInsensitive;
36:
37:     dirCaseInsensitive.insert(make_pair("John", "2345764"));
38:     dirCaseInsensitive.insert(make_pair("JOHN", "2345764"));
39:     dirCaseInsensitive.insert(make_pair("Sara", "42367236"));
40:     dirCaseInsensitive.insert(make_pair("Jack", "32435348"));
41:
42:     cout << "Displaying contents of the case-insensitive map:"
            << endl;
43:     DisplayContents(dirCaseInsensitive);
44:
45:     // Зависящая от регистра карта: регистр строкового ключа
        // влияет на вставку и поиск
46:     DIRECTORY_WITHCASE dirCaseSensitive(dirCaseInsensitive.begin()
47:                                       , dirCaseInsensitive.end());
48:
49:     cout << "Displaying contents of the case-sensitive map:"<< endl;
50:     DisplayContents(dirCaseSensitive);
51:
52:     // Поиск по имени в двух картах и отображение результата
53:     cout << "Please enter a name to search: "<< endl<< "> ";
54:     string strNameInput;
55:     cin >> strNameInput;
56:
57:     // поиск в карте...
58:     auto iPairInNoCaseDir = dirCaseInsensitive.find(strNameInput);
59:     if(iPairInNoCaseDir != dirCaseInsensitive.end())
60:     {
61:         cout << iPairInNoCaseDir->first
                << "'s number in the case-insensitive";
62:         cout << " directory is: "<< iPairInNoCaseDir->second<< endl;
63:     }
64:     else
65:     {
66:         cout << strNameInput<< "'s number not found ";
67:         cout << "in the case-insensitive directory"<< endl;
68:     }
69:
70:     // поиск в зависящей от регистра карте...
71:     auto iPairInCaseSensDir = dirCaseSensitive.find(strNameInput);
72:     if(iPairInCaseSensDir != dirCaseSensitive.end())
73:     {
74:         cout <<iPairInCaseSensDir->first
                <<"'s number in the case-sensitive";
75:         cout << " directory is: "<< iPairInCaseSensDir->second
                << endl;
76:     }
77:     else
78:     {
```

```
79:         cout << strNameInput<< "'s number was not found ";
80:         cout << "in the case-sensitive directory"<< endl;
81:     }
82:
83:     return 0;
84:}
```

Результат

```
Displaying contents of the case-insensitive map:
Jack -> 32435348
John -> 2345764
Sara -> 42367236

Displaying contents of the case-sensitive map:
Jack -> 32435348
John -> 2345764
Sara -> 42367236

Please enter a name to search:
> sara
Sara's number in the case-insensitive directory is: 42367236
sara's number was not found in the case-sensitive directory
```

Анализ

Рассматриваемый код содержит два каталога с одинаковым содержимым: один был создан с заданным по умолчанию предикатом сортировки `std::less<T>` и зависящим от регистра оператором `std::string::operator<`, а другой — со структурой предиката `PredIgnoreCase` (строки 17–27), сравнивающего две строки после преобразования их символов в нижний регистр. Вывод указывает, что при поиске в двух картах слова `'sara'` в независящей от регистра карте будет найдена запись `Sara`, тогда как карта с предикатом по умолчанию неспособна найти эту запись.

ПРИМЕЧАНИЕ

В листинге 20.5 структура `PredIgnoreCase` может также быть классом, если для оператора `operator()` вы добавите ключевое слово `public`. Для компилятора C++ структура родственна классу с открытыми по умолчанию членами и открытым наследованием.

Этот пример демонстрирует возможность использования предикатов для настройки поведения карты, а также то, что потенциально ключ может иметь любой тип, а программист способен предоставить предикат, определяющий поведение карты для этого типа. Обратите внимание на то, что предикат был структурой, реализовавшей оператор `operator()`. Но это вполне может быть класс. Такие объекты называются также *объектами функций* (function object) или *функторами* (functor). Более подробная информация по этой теме приведена на занятии 21, "Понятие объектов функций".

ПРИМЕЧАНИЕ

Контейнер `std::map` хорошо подходит для хранения пар "ключ-значение", позволяя искать значение, заданное по ключу. Карта действительно гарантирует лучшую производительность, чем вектор или список, когда дело доходит до поиска. Но все же он замедляется при увеличении количества элементов. Оперативная производительность карты, как говорят, имеет логарифмический характер, т.е. она пропорциональна логарифму количества помещенных в карту элементов.

Проще говоря, логарифмическая сложность означает, что при 10 000 элементах поиск в таком контейнере, как `std::map` или `std::set`, осуществляется вдвое медленней, чем при 100 элементах.

Несортированный вектор имеет линейную сложность при поиске, т.е. для 10 000 элементов он в 100 раз медленнее, чем для 100.

C++11

Контейнер `std::unordered_map` библиотеки STL на базе хеш-таблиц ключей и значений

Начиная с версии C++11 библиотека STL предоставляет хеш-карту в форме класса `std::unordered_map`. Для использования этого шаблона класса включите его заголовок:

```
#include<unordered_map>
```

Контейнер `unordered_map` предоставляет преимущество постоянной продолжительности вставки, удаления и поиска произвольных элементов 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), попытаемся разобраться с тем, что делают *хеш-таблицы* (hash table).

Хеш-таблицу можно рассматривать как коллекцию пар "ключ–значение", где по данному ключу таблица может найти значение. Различие между хеш-таблицей и простой картой 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 ключ:

```
Индекс = Хеш-функция(Ключ, РазмерТаблицы);
```

При выполнении функции `find()` для данного ключа *Хеш-функция*`()` используется еще раз, чтобы определить позицию элемента и возвратить из таблицы значение по позиции, как возвратил бы хранимый элемент массива. В тех случаях, когда *Хеш-функция*`()` не определена, тот же *Индекс* может быть у нескольких элементов, расположенных в той же ячейке, которая внутренне станет списком элементов. В таких случаях, называемых *конфликтом* (collision), поиск осуществляется медленнее и не имеет больше постоянной продолжительности.

Использование хеш-таблиц C++11: unordered_map и unordered_multimap

С точки зрения применения эти два контейнера не слишком отличаются от контейнеров std::map и std::multimap соответственно. Они обеспечивают создание экземпляра, вставку и поиск согласно общему шаблону:

```
// создание экземпляра unordered_map для int и string:
unordered_map<int, string> umapIntToString;

// insert()
umapIntToString.insert(make_pair(1000, "Thousand"));

// find():
auto iPairThousand = umapIntToString.find(1000);
cout << iPairThousand->first << " -> " << iPairThousand->second << endl;

// поиск значения с использованием семантики массива:
cout << "umapIntToString[1000] = " << umapIntToString[1000] << endl;
```

Все же у контейнера unordered_map есть одна очень важная особенность — доступность хеш-функции, ответственной за решение о порядке сортировки:

```
unordered_map<int, string>::hasher HFn =
          umapIntToString.hash_function();
```

Вызвав хеш-функцию для ключа, можно просмотреть приоритет присвоения ключей:

```
size_t HashingValue1000 = HFn(1000);
```

Поскольку контейнер unordered_map хранит пары "ключ–значение" в ячейках, он обеспечивает автоматическую балансировку нагрузки, когда количество элементов в карте достигает или начинает достигать количества ячеек в нем:

```
cout << "Load factor: " << umapIntToString.load_factor() << endl;
cout << "Max load factor = " << umapIntToString.max_load_factor()
     << endl;
cout << "Max bucket count = " << umapIntToString.max_bucket_count()
     << endl;
```

Метод load_factor() отображает коэффициент загрузки, т.е. степень заполнения ячеек контейнера unordered_map. Когда в ходе вставки значение load_factor() превышает значение max_load_factor(), карта реорганизует себя так, чтобы увеличить количество доступных ячеек, и перестраивает хеш-таблицу, как представлено в листинге 20.6.

СОВЕТ

Контейнер std::unordered_multimap подобен контейнеру unordered_map, но допускает наличие нескольких пар с одинаковым ключом.

Контейнер std::unordered_multimap используется очень похоже на контейнер std::multimap, но с определенными функциями хеш-таблицы, как показано в листинге 20.6.

ЛИСТИНГ 20.6. Создание экземпляра реализации хеш-таблицы `unordered_map`, а также использование методов `insert()`, `find()`, `size()`, `max_bucket_count()`, `load_factor()` и `max_load_factor()`

```
 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>& Input)
 7: {
 8:     cout << "Number of pairs, size(): " << Input.size() << endl;
 9:     cout << "Max bucket count = " << Input.max_bucket_count()
          << endl;
10:     cout << "Load factor: " << Input.load_factor() << endl;
11:     cout << "Max load factor = " << Input.max_load_factor() << endl;
12:     cout << "Unordered Map contains: " << endl;
13:
14:     for(auto iElement = Input.cbegin() // auto, cbegin: C++11
15:           ; iElement != Input.cend()  // cend(): C++11
16:           ; ++ iElement )
17:         cout<< iElement->first<< " -> "<< iElement->second<< endl;
18: }
19:
20: int main()
21: {
22:     unordered_map<int, string> umapIntToString;
23:     umapIntToString.insert(make_pair(1, "One"));
24:     umapIntToString.insert(make_pair(45, "Forty Five"));
25:     umapIntToString.insert(make_pair(1001, "Thousand One"));
26:     umapIntToString.insert(make_pair(-2, "Minus Two"));
27:     umapIntToString.insert(make_pair(-1000, "Minus One Thousand"));
28:     umapIntToString.insert(make_pair(100, "One Hundred"));
29:     umapIntToString.insert(make_pair(12, "Twelve"));
30:     umapIntToString.insert(make_pair(-100, "Minus One Hundred"));
31:
32:     DisplayUnorderedMap<int, string>(umapIntToString);
33:
34:     cout << "Inserting one more element" << endl;
35:     umapIntToString.insert(make_pair(300, "Three Hundred"));
36:     DisplayUnorderedMap<int, string>(umapIntToString);
37:
38:     cout << "Enter key to find for: ";
39:     int Key = 0;
40:     cin >> Key;
41:
42:     auto iElementFound = umapIntToString.find(Key);
43:     if (iElementFound != umapIntToString.end())
44:     {
45:         cout << "Found! Key " << iElementFound->first
              << " points to value ";
46:         cout << iElementFound->second << endl;
47:     }
```

```
48:     else
49:         cout << "Key has no corresponding value in unordered map!"
                 << endl;
50:
51:     return 0;
52: }
```

Результат

```
Number of pairs, size(): 8
Max bucket count = 8
Load factor: 1
Max load factor = 1
Unordered Map contains:
-1000 -> Minus One Thousand
1001 -> Thousand One
1 -> One
-100 -> Minus One Hundred
45 -> Forty Five
-2 -> Minus Two
12 -> Twelve
100 -> One Hundred
Inserting one more element
Number of pairs, size(): 9
Max bucket count = 64
Load factor: 0.140625
Max load factor = 1
Unordered Map contains:
1 -> One
-1000 -> Minus One Thousand
1001 -> Thousand One
-100 -> Minus One Hundred
45 -> Forty Five
-2 -> Minus Two
300 -> Three Hundred
12 -> Twelve
100 -> One Hundred
100 -> One Hundred
Enter key to find for: 300
Found! Key 300 points to value Three Hundred
```

Анализ

Просмотрите вывод и обратите внимание на то, как контейнер `unordered_map`, изначально насчитывающий восемь ячеек и заполненный восьмью парами, изменяет свои размеры при вставке девяти пар. Вот когда количество ячеек увеличивается до 64. Обратите внимание на применение методов `max_bucket_count()`, `load_factor()` и `max_load_factor()` в строках 9–11, а также на то, что остальная часть кода почти не отличается от такового по сравнению с контейнером `std::map`. Это же относится и к применению метода `find()` в строке 42, который возвращает итератор, как и у контейнера `std::map`, его также следует сравнить с результатом метода `end()`, чтобы удостовериться в успехе операции.

ВНИМАНИЕ!

Не следует полагаться на порядок элементов в контейнере `unordered_map` независимо от ключа. Порядок элементов относительно других элементов в карте зависит от многих факторов, включая их ключ, порядок вставки, количество ячеек и т.д.

Эти контейнеры оптимизированы для производительности поиска, поэтому не стоит полагаться на порядок элементов при их переборе.

ПРИМЕЧАНИЕ

Продолжительность вставки и поиска в контейнере `std::unordered_map` практически постоянна (при отсутствии конфликтов) и не зависит от количества содержащихся в нем элементов. Но это не обязательно делает контейнер `std::unordered_map` предпочтительней контейнера `std::map`, который обеспечивает логарифмическую сложность во всех ситуациях. Постоянная продолжительность может оказаться намного больше логарифмической, когда количество содержащихся элементов не слишком велико.

При выборе типа контейнера имеет смысл провести испытания, моделирующие их применение в реальном сценарии.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте карту, если необходимо хранить пары "ключ-значение" с уникальными ключами	**Не забывайте**, что метод `multimap::count(`*Ключ*`)` сообщает количество доступных в контейнере пар, индексированных при помощи ключа *Ключ*
Используйте мультикарту, если необходимо хранить пары "ключ-значение", ключи которых могут повторяться (например, телефонный справочник)	**Не забывайте** проверять результат метода `find()`, сравнив его с результатом метода `end()`
Помните, что и карта, и мультикарта, как и другие контейнеры библиотеки STL, предоставляют метод `size()`, сообщающий количество хранимых пар	
Используйте контейнеры `unordered_map` или `unordered_multimap`, когда абсолютно необходима постоянная продолжительность вставки и поиска (обычно, когда количество элементов очень высоко)	

Резюме

На сегодняшнем занятии рассматривались шаблоны классов `map` и `multimap` библиотеки STL, их важнейшие функции-члены и характеристики. Вы также узнали, что у этих контейнеров логарифмическая сложность и что библиотека STL предоставляет хеш-таблицы в форме контейнеров `unordered_map` и `unordered_multimap`. Они демонстрируют высокую производительность операций вставки и поиска, которая не зависит от размера контейнера. Вы также узнали о важности возможности настройки критериев сортировки с использованием предиката, как представлено в приложении из листинга 20.5.

Вопросы и ответы

- **Как мне объявить контейнер map для хранения целых чисел, отсортированных в порядке убывания?**

  Карту целых чисел определяет шаблон класса map<int>. Он имеет заданный по умолчанию предикат сортировки std::less<T>, располагающий элементы в порядке возрастания. Его можно выразить как map<int, less <int>>. Для сортировки в порядке убывания определите карту как map<int, greater<int>>.

- **Что будет при вставке в карту для строк строки "Jack" дважды?**

  Карта не предназначена для хранения повторяющихся значений. Поэтому реализация класса std::map не позволит вставить второе значение.

- **Что нужно изменить в предыдущем примере, чтобы все-таки вставить две строки "Jack"?**

  Реализация класса map позволяет хранить только уникальные значения. 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 элемент в карте и имею теперь указывающий на него итератор. Как мне использовать этот итератор для изменения значения, на которое он указывает?**

  Никак. Некоторые реализации библиотеки STL могли бы позволить пользователю изменить значение элемента карты при помощи итератора, возвращенного, например, функцией find(). Но так поступать неправильно. Итератор на элемент карты должен использоваться как константный, даже когда реализация библиотеки STL не препятствует этому.

- **Я использую устаревший компилятор, который не поддерживает ключевое слово auto. Как мне объявить переменную, которая содержит возвращаемое значение метода map::find()?**

  Итератор всегда определяется с использованием такого синтаксиса:

  контейнер<Тип>::iterator *имяПеременной;*

  Таким образом, объявление итератора для карты целых чисел будет следующим:

```
std::map<int>::iterator iPairFound = mapIntegers.find(1000);
if (iPairFound != mapIntegers.end())
    ; // Сделать нечто
```

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 Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Вы объявляете карту целых чисел так `map<int>`. Какая функция предоставит критерий сортировки?
2. Как расположены двойные элементы в мультикарте?
3. Какая функция контейнеров `map` и `multimap` возвращает количество элементов в нем?
4. Как расположены двойные элементы в карте?

Упражнения

1. Необходимо написать приложение, работающее как телефонный справочник, где имена людей не должны быть уникальными. Какой контейнер выбрать? Напишите определение контейнера.
2. Вот определение шаблона карты приложения словаря:

   ```
   map <wordProperty, string, fPredicate> mapWordDefinition;
   ```

   где каждое слово такая структура:

   ```
   struct wordProperty
   {
       string strWord;
       bool bIsFromLatin;
   };
   ```

3. Определите бинарный предикат `fPredicate`, который позволяет карте сортировать ключи типа `wordProperty` согласно строковому атрибуту, который он содержит.
4. Продемонстрируйте на примере простой программы, что карта не может хранить совпадающие записи, а мультикарта может.

ЧАСТЬ IV
Подробней о библиотеке STL

ЗАНЯТИЕ 21

Понятие объектов функций

Объекты функций (function object), или *функторы* (functor), могли бы показаться чем-то экзотическим или пугающим, но в сущности вы, вероятно, их уже видели, а возможно, и использовали, еще не понимая этого.

На сегодняшнем занятии.

- Концепция объектов функций.
- Использование объектов функций как предикатов.
- Как реализуются унарные и бинарные предикаты с использованием объектов функций.

Концепция объектов функций и предикатов

На концептуальном уровне объекты функций — это объекты, работающие как функции. Однако на уровне реализации объекты функций — это объекты класса, реализующего оператор `operator()`. Хотя функции и указатели на функцию также могут быть классифицированы как объекты функций, здесь речь идет об объекте класса, реализующего оператор `operator()` для хранения его состояния (т.е. значения в атрибутах класса), что делает его применимым с алгоритмами стандартной библиотекой шаблонов (STL).

Объекты функции, как правило, используются при работе с библиотекой STL и подразделяются на следующие типы.

- *Унарная функция* (unary function). Функция вызывается с одним аргументом, например `f(x)`. Когда унарная функция возвращает значение типа `bool`, она называется *предикатом* (predicate).
- *Бинарная функция* (binary function). Функция вызывается с двумя аргументами, например `f(x, y)`. Когда бинарная функция возвращает значение типа `bool`, она называется *бинарным предикатом* (binary predicate).

Объекты функций, возвращающие значение типа `bool`, обычно используются в алгоритмах при принятии решений. Объект функции, объединяющий два объекта функции, называется *адаптивным объектом функции* (adaptive function object).

Типичные приложения объектов функций

Для объяснения объектов функций можно задействовать несколько страниц теории, а можно рассмотреть и понять их работу на примере небольшого приложения. Давайте применим практический подход и перейдем сразу к применению объектов функций, или функторов, при программировании на C++!

Унарные функции

Функции с одиночным параметром являются унарными. Унарная функция может делать нечто очень простое, например отобразить элемент на экране. Это может быть реализовано следующим образом:

```
// Унарная функция
template <typename elementType>
void FuncDisplayElement (const elementType & element)
{
    cout << element << ' ';
};
```

Функция `FuncDisplayElement()` получает один параметр шаблонного типа `elementType`, отображаемый на консоли с использованием оператора `std::cout`. Та же функция может иметь и другое представление, в котором реализация функции фактически содержится оператором `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` — это структура. Если бы это был класс, то оператор `operator()` должен был бы иметь модификатор доступа `public`. Структура сродни классу, но ее члены являются открытыми по умолчанию.

Любая из этих реализаций может применяться с алгоритмом библиотеки STL `for_each`, чтобы отобразить содержимое коллекции на экране, по одному элементу за раз, как представлено в листинге 21.1.

ЛИСТИНГ 21.1. Отображение содержимого коллекции на экране с использованием унарной функции

```
 0: #include <algorithm>
 1: #include <iostream>
 2: #include <vector>
 3: #include <list>
 4:
 5: using namespace std;
 6:
 7: // структура, ведущая себя как унарная функция
 8: template <typename elementType>
 9: struct DisplayElement
10: {
11:     void operator () (const elementType& element) const
12:     {
13:         cout << element << ' ';
14:     }
15: };
16:
17: int main ()
18: {
19:     vector <int> vecIntegers;
20:
21:     for (int nCount = 0; nCount < 10; ++ nCount)
22:         vecIntegers.push_back (nCount);
23:
24:     list <char> listChars;
25:
26:     for (char nChar = 'a'; nChar < 'k'; ++nChar)
27:         listChars.push_back (nChar);
28:
29:     cout << "Displaying the vector of integers: " << endl;
30:
31:     // Отобразить массив целых чисел
```

```
32:     for_each ( vecIntegers.begin () // Начало диапазона
33:              , vecIntegers.end ()   // Конец диапазона
34:              , DisplayElement <int> () ); // Объект унарной функции
35:
36:     cout << endl << endl;
37:     cout << "Displaying the list of characters: " << endl;
38:
39:     // Отобразить список символов
40:     for_each ( listChars.begin () // Начало диапазона
41:              , listChars.end ()   // Конец диапазона
42:              , DisplayElement <char> () ); // Объект унарной функции
43:
44:     return 0;
45: }
```

Результат

```
Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9

Displaying the list of characters:
a b c d e f g h i j
```

Анализ

Строки 8–15 содержат объект функции `DisplayElement`, реализующей оператор `operator()`. Этот объект функции используется с алгоритмом `std::for_each()` библиотеки STL в строках 32–34. Алгоритм `for_each()` получает три параметра: первый — начальный пункт диапазона, второй — конец диапазона, третий — функция, вызываемая для каждого элемента в этом диапазоне. Другими словами, код вызывает оператор `DisplayElement::operator()` для каждого элемента в векторе `vecIntegers`. Обратите внимание, что вместо структуры `DisplayElement` вы с тем же успехом можете использовать привычную функцию `FuncDisplayElement()`. Строки 40–42 демонстрируют те же возможности для списка символов.

СОВЕТ

Стандарт C++11 вводит лямбда-выражения, являющиеся безымянными объектами функций.

Версия лямбда-выражения структуры `DisplayElement<T>` из листинга 21.1 делает весь код компактней, включая определение структуры и ее применение в трех строках функции `main()`, если заменить строки 32-34:

```
// Отобразить массив целых чисел, используя лямбда-выражения
for_each ( vecIntegers.begin ()  // Начало диапазона
         , vecIntegers.end ()    // Конец диапазона
         , [](int& Element) {cout << element << ' '; } );
                                 // Лямбда-выражение
```

Таким образом, лямбда-выражения – это фантастическое преимущество C++, и вы непременно должны изучать 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 объекта функции, как в листинге 21.1.

Реальное преимущество использования объекта функции, реализованного в структуре, становится очевидным, когда вы в состоянии использовать объект структуры для хранения информации. Это нечто, чего функция `FuncDisplayElement()` не может сделать в отличие от структуры, поскольку структура способна хранить атрибуты, кроме оператора `operator()`. Вот несколько измененная версия, в которой используются атрибуты:

```
template <typename elementType>
struct DisplayElementKeepCount
{
    int Count;

    DisplayElementKeepCount () // Конструктор
    {
        Count = 0;
    }

    void operator () (const elementType& element)
    {
        ++ Count;
        cout << element << ' ';
    }
};
```

В приведенном выше фрагменте структура `DisplayElementKeepCount` немного модифицирована по сравнению с предыдущей версией. Оператор `operator()` больше не является константной функцией-членом, поскольку он осуществляет инкремент (а следовательно, изменяет) значения переменной-члена `Count`, используемой для хранения количества ее вызовов и применяемой для отображения данных. Этот подсчет возможен благодаря открытому атрибуту `Count`. Преимущество использования таких объектов функции, способных хранить состояние, показано в листинге 21.2.

ЛИСТИНГ 21.2. Использование объекта функции для хранения состояния

```
 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:     // Конструктор
11:     DisplayElementKeepCount() : Count(0) {}
12:
13:     // Отобразить элемент, хранящий количество!
14:     void operator()(const elementType& element)
15:     {
16:         ++ Count;
17:         cout << element<< ' ';
18:     }
19: };
20:
```

```
21: int main()
22: {
23:     vector<int> vecIntegers;
24:     for(int nCount = 0; nCount< 10; ++ nCount)
25:         vecIntegers.push_back(nCount);
26:
27:     cout << "Displaying the vector of integers: "<< endl;
28:
29:     // Отобразить массив целых чисел
30:     DisplayElementKeepCount<int> Result;
31:     Result = for_each( vecIntegers.begin()  // Начало диапазона
32:              , vecIntegers.end()            // Конец диапазона
33:              , DisplayElementKeepCount<int>() ); // объект функции
34:
35:     cout << endl<< endl;
36:
37:     // Использование хранилища состояния при возвращении значения!
38:     cout << "'" << Result.Count << "' elements were displayed!"
            << endl;
39:
40:     return 0;
41: }
```

Результат

```
Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9

'10' elements were displayed!
```

Анализ

Самое большое различие между этим примером и листингом 21.1 заключается в применении структуры `DisplayElementKeepCount` как возвращаемого значения алгоритма `for_each()`. Оператор `operator()`, реализованный в структуре `DisplayElementKeepCount`, вызывается алгоритмом `for_each()` для каждого элемента в контейнере. Это отображает элемент и увеличивает внутренний счетчик, хранившийся в переменной `Count`. После завершения работы алгоритма `for_each()` вы используете объект в строке 38 для отображения количества отображенных элементов. Обратите внимание, что использование в этом случае обычной функции вместо функции, реализованной в структуре, не позволило бы использовать это средство таким непосредственным способом.

Унарный предикат

Унарная функция, которая возвращает значение типа `bool`, является предикатом. Такая функция 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 число кратным другому

```
0: // Структура как унарный предикат
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:         // Проверить, кратен ли делитель делимому
14:         return ((element % Divisor) == 0);
15:     }
16: };
```

Анализ

Здесь оператор `operator()` возвращает тип `bool` и может работать как унарный предикат. Структура имеет конструктор и инициализируется значением делителя. Это значение хранится в объекте, а затем используется для определения, делится ли на него переданный для сравнения элемент. Как можно заметить, в реализации оператора `operator()` используется оператор деления по модулю `%`, который возвращает остаток от операции деления. Предикат сравнивает этот остаток с нулем, чтобы выявить кратность чисел.

В листинге 21.4 предикат используется, как и в листинге 21.3, для определения кратности чисел, вводимых в коллекцию, заданному пользователем делителю.

ЛИСТИНГ 21.4. Использование унарного предиката `IsMultiple` с алгоритмом `std::find_if()` для поиска в векторе элемента, кратного введенному пользователем делителю

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3: using namespace std;
 4: // Сюда вставить структуру IsMultiple из листинга 21.3
 5: int main ()
 6: {
 7:     vector <int> vecIntegers;
 8:     cout << "The vector contains the following sample values: ";
 9:
10:     // Вставить примеры значений: 25 - 31
11:     for (int nCount = 25; nCount < 32; ++ nCount)
12:     {
13:         vecIntegers.push_back (nCount);
14:         cout << nCount << ' ';
15:     }
16:     cout << endl << "Enter divisor (> 0): ";
17:     int Divisor = 2;
18:     cin >> Divisor;
19:
20:     // Найти в коллекции первый элемент, кратный 4
21:     auto iElement = find_if ( vecIntegers.begin ()
22:                             , vecIntegers.end ()
23:                             , IsMultiple<int>(Divisor) );
```

```
24:
25:     if (iElement != vecIntegers.end ())
26:     {
27:         cout << "First element in vector divisible by " << Divisor;
28:         cout << ": " << *iElement << endl;
29:     }
30:
31:     return 0;
32: }
```

Результат

```
The vector contains the following sample values: 25 26 27 28 29 30 31
The first element in the vector that is divisible by 4 is: 28
```

Анализ

Пример начинается с простого контейнера — вектора целых чисел. В строках 11–15 он заполняется примерами чисел. Применение унарного предиката осуществляется в алгоритме поиска `find_if()`, как показано в строке 23. Объект функции `IsMultiple()` инициализируется предоставляемым пользователем значением делителя, которое сохраняется в переменной `Divisor`. Алгоритм `find_if()` вызывает оператор `IsMultiple::operator()` унарного предиката для каждого элемента в определенном диапазоне. Когда оператор `operator()` возвращает для элемента значение `true` (что происходит, когда он делится на 4 без остатка), алгоритм `find_if()` возвращает итератор `iElement` на этот элемент. Результат вызова `find_if()` сравнивается с результатом вызова метода `end()` контейнера, чтобы удостовериться в успехе поиска элемента (строка 25). Затем итератор `iElement` используется для отображения значения, как показано в строке 28.

СОВЕТ

> Чтобы увидеть, как применение лямбда-выражений повышает компактность программы, представленной в листинге 21.4, обратите внимание на листинг 22.3 занятия 22.

Унарные предикаты применяются в большинстве алгоритмов STL, таких как `std::partition()`, позволяющих разделить диапазон, с использованием предиката `stable_partition`, который делает то же самое при сохранении относительного порядка разделенных элементов, а также таких функций поиска, как `std::find_if()`, и таких функций, как `std::remove_if()`, позволяющих удалять удовлетворяющие предикату элементы в определенном диапазоне.

Бинарные функции

Функции типа `f(x, y)` особенно полезны, когда они возвращают значение на основании введенных значений. Такие бинарные функции применяются для хранения арифметического действия с двумя операндами, такого как сложение, умножение, вычитание и т.д. Типичная бинарная функция, возвращающая произведение входных аргументов, может быть написана так:

```
template <typename elementType>
class Multiply
{
public:
    elementType operator () (const elementType& elem1,
                             const elementType& elem2)
    {
        return (elem1 * elem2);
    }
};
```

Представляющая интерес реализация снова находится в операторе `operator()`, который получает два аргумента и возвращает их произведение. Подобные бинарные функции используются в таких алгоритмах, как `std::transform()`, где вы можете использовать его для умножения содержимого двух контейнеров. В листинге 21.5 показано применение таких бинарных функций в алгоритме `std::transform()`.

ЛИСТИНГ 21.5. Использование бинарной функции 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:     // Создание двух векторов целых чисел по 10 элементов каждый
20:     vector <int> vecMultiplicand, vecMultiplier;
21:
22:     // Вставить примеры значений от 0 до 9
23:     for (int nCount1 = 0; nCount1 < 10; ++ nCount1)
24:     vecMultiplicand.push_back (nCount1);
25:
26:     // Вставить примеры значений от 100 до 109
27:     for (int nCount2 = 100; nCount2 < 110; ++ nCount2)
28:         vecMultiplier.push_back (nCount2);
29:
30:     // Третий контейнер содержит результат умножения
31:     vector <int> vecResult;
32:
33:     // Создать пространство для результата умножения
34:     vecResult.resize (10);
```

```
35:     transform ( vecMultiplicand.begin (), // диапазон множителей
36:         vecMultiplicand.end (),           // конец диапазона
37:         vecMultiplier.begin (),           // значения множителей
38:         vecResult.begin (),  // диапазон, содержащий результат
39:         Multiply <int> () ); // умножающая функция
40:
41:     cout << "The contents of the first vector are: " << endl;
42:     for (size_t nIndex1 = 0; nIndex1 < vecMultiplicand.size ();
                                 ++ nIndex1)
43:         cout << vecMultiplicand [nIndex1] << ' ';
44:     cout << endl;
45:
46:     cout << "The contents of the second vector are: " << endl;
47:     for (size_t nIndex2 = 0; nIndex2 < vecMultiplier.size ();
                                 ++nIndex2)
48:         cout << vecMultiplier [nIndex2] << ' ';
49:     cout << endl << endl;
50:
51:     cout << "The result of the multiplication is: " << endl;
52:     for (size_t nIndex = 0; nIndex < vecResult.size (); ++ nIndex)
53:         cout << vecResult [nIndex] << ' ';
54:
55:     return 0;
56: }
```

Результат

```
The contents of the first vector are:
0 1 2 3 4 5 6 7 8 9
The contents of the second vector are:
100 101 102 103 104 105 106 107 108 109

The result of the multiplication held in the third vector is:
0 101 204 309 416 525 636 749 864 981
```

Анализ

Строки 5–13 содержат класс `Multiply`. В данном примере алгоритм `std::transform()` используется для умножения содержимого двух диапазонов и сохранения результата в третьем. В данном случае рассматриваемые диапазоны содержатся в объектах `vecMultiplicand`, `vecMultiplier` и `vecResult` класса `std::vector`. Другими словами, функция `std::transform()` в строках 35–39 используется для умножения каждого элемента вектора `vecMultiplicand` на соответствующий элемент вектора `vecMultiplier` и сохраняет результат умножения в векторе `vecResult`. Само умножение осуществляется бинарной функцией `CMultiple::operator()`, которая вызывается для каждого элемента в исходных и результирующих диапазонах векторов. Возвращаемое значение оператора `operator()` сохраняется в векторе `vecResult`.

Таким образом, этот пример демонстрирует применение бинарных функций для выполнения арифметических операций с элементами в контейнерах STL.

Бинарный предикат

Функция, которая получает два аргумента и возвращает значение типа bool, является бинарным предикатом. Такие функции находят применение в таких алгоритмах библиотеки STL, как std::sort(). Листинг 21.6 демонстрирует применение бинарного предиката, который сравнивает две строки после перевода их в нижний регистр. Такой предикат применяется при выполнении независящей от регистра сортировки вектора строк, например.

ЛИСТИНГ 21.6. Бинарный предикат для сортировки строк, независящей 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:         // Зарезервировать пространство
12:         str1LowerCase.resize (str1.size ());
13:
14:         // Преобразовать каждый символ 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 операторе 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 operator< для возвращения результата сравнения.

Вы можете использовать этот двоичный предикат с алгоритмом std::sort() для сортировки динамического массива, содержащегося в векторе строк, как представлено в листинге 21.7.

ЛИСТИНГ 21.7. Использование объекта функции класса CompareStringNoCase для независящей 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& Input)
 6: {
 7:     for(auto iElement = Input.cbegin() // auto, cbegin и cend: C++11
 8:           ; iElement != Input.cend ()
 9:           ; ++ iElement )
10:         cout << *iElement << endl;
11: }
12:
13: int main ()
14: {
15:     // Определить вектор строк для имен
16:     vector <string> vecNames;
17:
18:     // Вставить в вектор несколько примеров имен
19:     vecNames.push_back ("jim");
20:     vecNames.push_back ("Jack");
21:     vecNames.push_back ("Sam");
22:     vecNames.push_back ("Anna");
23:
24:     cout << "The names in vector in order of insertion: " << endl;
25:     DisplayContents(vecNames);
26:
27:     cout << "Names after sorting using default std::less<>: "
          << endl;
28:     sort(vecNames.begin(), vecNames.end());
29:     DisplayContents(vecNames);
30:
31:     cout << "Names after sorting using predicate that ignores case:"
          << endl;
32:     sort(vecNames.begin(), vecNames.end(), CompareStringNoCase());
33:     DisplayContents(vecNames);
34:
35:     return 0;
36: }
```

Результат

```
The names in vector in order of insertion:
jim
Jack
Sam
Anna
Names after sorting using default std::less<>:
Anna
Jack
Sam
jim
Names after sorting using predicate that ignores case:
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 версия использует класс предиката сортировки `CompareStringNoCase<>` в строке 32 (он реализован в листинге 21.6), гарантирующего, 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 алгоритме `std::sort()`, осуществляющем сортировку, в алгоритме `std::stable_sort()`, осуществляющем сортировку в относительном порядке, в алгоритме `std::transform()`, позволяющем выполнить операцию с двумя диапазонами, и во многих других алгоритмах библиотеки STL, нуждающихся в бинарном предикате.

Резюме

На этом занятии вы познакомились с функторами (или объектами функций). Вы узнали, что объекты функций, реализованные в структуре или классе, полезней простых функций, поскольку они могут использоваться также для содержания информации состояния. Вы получили понятие о предикатах, которые являются специальным классом объектов функции, и ознакомились с некоторыми практическими примерами, демонстрирующими их удобство.

Вопросы и ответы

- **Предикат — это специальная категория объектов функций. Что делает его таким особенным?**

  Предикаты всегда возвращают логическое значение.
- **Какой объект функции использовать при вызове такой функции, как `remove_if()`?**

  Вы должны использовать унарный предикат, получающий через конструктор подлежащее обработке значение как начальное состояние.
- **Какой объект функции я должен использовать для карты?**

  Бинарный предикат.
- **Действительно ли простая функция без возвращаемого значения может быть использована как предикат?**

  Да. Функция без возвращаемых значений вполне может сделать нечто полезное. Например, она может отобразить входные данные.

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 Г, "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 объекта функции, который не изменяет данные и не возвращает значения типа `bool`? Можете объяснить ответ на примере?
3. Каково определение термина *объекты функций* (function object)?

Упражнения

1. Напишите унарную функцию, которая применяется в алгоритме `std::for_each()` для отображения входного параметра типа `double`.
2. Дополните этот предикат так, чтобы отображать количество раз его использования.
3. Напишите бинарный предикат, обеспечивающий сортировку в порядке возрастания.

ЗАНЯТИЕ 22
Лямбда-выражения языка C++11

Лямбда-выражения (lambda expressions) — это компактный способ определения и создания объектов функций без имени. Эти выражения — нововведение C++11.

На сегодняшнем занятии.

- Как создать лямбда-выражение.
- Использование лямбда-выражений в качестве предикатов.
- Как создать лямбда-выражение, способное содержать состояние и манипулировать им.

Что такое лямбда-выражение

Лямбда-выражение (или лямбда) можно считать компактной версией безымянной структуры (или класса) с открытым оператором `operator()`. В этом смысле лямбда-выражение — это объект функции, подобный представленному на занятии 21, "Понятие объектов функций". Прежде чем переходить к анализу разработки лямбда-функций, возьмем, к примеру, объект функции из листинга 21.1:

```
// структура, ведущая себя как унарная функция
template <typename elementType>
struct DisplayElement
{
    void operator () (const elementType& element) const
    {
        cout << element << ' ';
    }
};
```

Этот объект функции отображает на экране 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> () ); // Объект унарной функции
```

Лямбда-выражение уплотняет весь код, включая определение объекта функции, до трех строк:

```
// Отобразить массив целых чисел, используя лямбда-выражения
for_each ( vecIntegers.begin () // Начало диапазона
         , vecIntegers.end ()   // Конец диапазона
         , [](int& Element) {cout << element << ' '; } );
                                                  // Лямбда-выражение
```

Когда компилятор встречает лямбда-выражение, в данном случае такое:

```
[](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. Они

сопровождаются списком параметров, являющимся тем же списком параметров, 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) { // здесь код лямбда-выражения; }
```

Обратите внимание, что по мере необходимости параметр можно передать по ссылке:

```
[](Type& paramName) { // здесь код лямбда-выражения; }
```

Листинг 22.1 поможет изучить применение лямбда-функции для отображения содержимого контейнера стандартной библиотеки шаблонов (STL) с использованием алгоритма `for_each()`.

ЛИСТИНГ 22.1. Отображение элементов контейнера при помощи алгоритма `for_each()`, который вызывается лямбда-выражением, а не объектом функции

```
 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> vecIntegers;
10:
11:     for (int nCount = 0; nCount < 10; ++ nCount)
12:         vecIntegers.push_back (nCount);
13:
14:     list <char> listChars;
15:     for (char nChar = 'a'; nChar < 'k'; ++nChar)
16:         listChars.push_back (nChar);
17:
18:     cout << "Displaying vector of integers using a lambda: " << endl;
19:
20:     // Отобразить массив целых чисел
21:     for_each ( vecIntegers.begin () // Начало диапазона
22:              , vecIntegers.end ()   // Конец диапазона
23:              , [](int& element) {cout << element << ' '; } );
                                        // лямбда
24:
25:     cout << endl << endl;
26:     cout << "Displaying list of characters using a lambda: " << endl;
27:
28:     // Отобразить список символов
```

```
29:     for_each ( listChars.begin () // Начало диапазона
30:              , listChars.end ()   // Конец диапазона
31:              , [](char& element) {cout << element << ' '; } );
                                      // лямбда
32:
33:     cout << endl;
34:
35:     return 0;
36: }
```

Результат

```
Displaying vector of integers using a lambda:
0 1 2 3 4 5 6 7 8 9

Displaying list of characters using a lambda:
a b c d e f g h i j
```

Анализ

Интерес представляют два лямбда-выражения в строках 23 и 31. Они очень похожи, если бы не тип входного параметра, поскольку они были приспособлены к характеру элементов в этих двух контейнерах. Первое получает один параметр типа `int`, поскольку оно используется для отображения одного элемента за раз из вектора целых чисел, тогда как второе получает параметр типа `char`, поскольку предназначен для отображения элементов типа `char`, хранящихся в контейнере `std::list`.

ПРИМЕЧАНИЕ

Вывод листинга 22.1 вовсе не случайно совпадает с выводом листинга 21.1. Фактически эта программа – лямбда-версия листинга 21.1, где был использован объект функции `DisplayElement<T>`.

Сравнивая эти два листинга, можно придти к выводу, что у лямбда-функций есть серьезный потенциал, позволяющий сделать код C++ проще и компактней.

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, следующее лямбда-выражение возвращает значение `true` для четных чисел:

```
[](int& Num) {return ((Num % 2) == 0); }
```

Характер возвращаемого значения в данном случае указывает компилятору, что лямбда-выражение возвращает тип `bool`.

Вы можете использовать лямбда-выражение, являющееся унарным предикатом, в таких алгоритмах, как `std::find_if()`, для поиска четных чисел в коллекции. Пример приведен в листинге 22.2.

ЛИСТИНГ 22.2. Поиск четных чисел в коллекции с использованием лямбда-выражения, унарного предиката и алгоритма `std::find_if()`

```
 0: #include<algorithm>
 1: #include<vector>
 2: #include<iostream>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     vector<int> vecNums;
 8:     vecNums.push_back(25);
 9:     vecNums.push_back(101);
10:     vecNums.push_back(2011);
11:     vecNums.push_back(-50);
12:
13:     auto iEvenNum = find_if( vecNums.cbegin()
14:                     , vecNums.cend()          // диапазон для поиска
15:                     , [](const int& Num){return ((Num % 2) == 0); } );
16:
17:     if (iEvenNum != vecNums.cend())
18:         cout << "Even number in collection is: " << *iEvenNum
                 << endl;
19:
20:     return 0;
21: }
```

Результат

```
Even number in collection is: -50
```

Анализ

Лямбда-функция, работающая как унарный предикат, представлена в строке 15. Алгоритм `find_if()` вызывает унарный предикат для каждого элемента в диапазоне. Когда предикат возвращает значение `true`, алгоритм `find_if()` сообщает о находке возвращением итератора на этот элемент. В данном случае предикат (лямбда-выражение) возвращает значение `true`, когда алгоритм `find_if()` встречает четное целое число (т.е. результат операции деления по модулю на 2 равен нулю).

ПРИМЕЧАНИЕ

Листинг 22.2 не только демонстрирует лямбда-выражение как унарный предикат, но также и использование ключевого слова `const` в пределах лямбда-выражения.

Не забывайте использовать его для входных параметров, особенно если это ссылки.

Лямбда-выражение с состоянием и списки захвата [...]

В листинге 22.2 был создан унарный предикат, который возвращал значение true, если целое число было делимым на 2, т.е. целое четное число. Но что если нужна более обобщенная функция, которая возвращает значение true, если число делимо на предоставленный пользователем делитель? Делитель необходимо содержать в выражении как "состояние":

```
int Divisor = 2; // Исходное значение
...
auto iElement = find_if ( begin of a range
        , end of a range
        , [Divisor](int dividend){return (dividend % Divisor) == 0; } );
```

Список аргументов передается как *переменная состояния* (state variable) в квадратных скобках ([...]) и называется также *списком захвата* (capture list) лямбда-выражения.

ПРИМЕЧАНИЕ

Такое лямбда-выражение – это однострочный эквивалент 16 строк кода в листинге 21.3, определяющего структуру унарного предиката IsMultiple<>.

Таким образом, лямбды стремительно повышают эффективность программирования на C++11!

Листинг 22.3 демонстрирует применение унарного предиката с заданной переменной состояния для поиска в коллекции числа, кратного предоставляемому пользователем делителю.

ЛИСТИНГ 22.3. Использование лямбда-выражений, хранящих состояние для проверки кратности одного числа другому

```
 0: #include <algorithm>
 1: #include <vector>
 2: #include <iostream>
 3: using namespace std;
 4:
 5: int main ()
 6: {
 7:     vector <int> vecIntegers;
 8:     cout << "The vector contains the following sample values: ";
 9:
10:     // Вставить примеры значений от 25 до 31
11:     for (int nCount = 25; nCount < 32; ++ nCount)
12:     {
13:         vecIntegers.push_back (nCount);
14:         cout << nCount << ' ';
15:     }
16:     cout << endl << "Enter divisor (> 0): ";
17:     int Divisor = 2;
18:     cin >> Divisor;
19:
20:     // Найти первый элемент, кратный делителю
```

```
21:     vector <int>::iterator iElement;
22:     iElement = find_if ( vecIntegers.begin ()
23:      , vecIntegers.end ()
24:      , [Divisor](int dividend){return (dividend % Divisor) == 0; } );
25:
26:     if (iElement != vecIntegers.end ())
27:     {
28:         cout << "First element in vector divisible by " << Divisor;
29:         cout << ": " << *iElement << endl;
30:     }
31:
32:     return 0;
33: }
```

Результат

```
The vector contains the following sample values: 25 26 27 28 29 30 31
Enter divisor (> 0): 4
First element in vector divisible by 4: 28
```

Анализ

Лямбда-выражение, хранящее состояние и работающее как предикат, представлено в строке 24. Переменная состояния `Divisor` аналогична переменной `IsMultiple::Divisor`, которую вы видели в листинге 21.3. Следовательно, переменные состояния сродни членам класса объекта функции, который вы использовали до C++11. Таким образом, теперь вы можете передать состояние в лямбда-функцию и настроить ее применение на его основании.

ПРИМЕЧАНИЕ Листинг 22.3 использует лямбда-эквивалент объекта функции из листинга 21.4, но без класса. Одно средство C++11 сэкономило 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 `[...]`:

```
[StateVar1, StateVar2](Type& param) { // здесь код лямбды; }
```

Если эти переменные состояния должны изменяться в пределах лямбды, добавьте ключевое слово `mutable`:

```
[StateVar1, StateVar2](Type& param) mutable { // здесь код лямбды; }
```

Обратите внимание, что здесь передаваемые в списке захвата `[]` переменные допускают изменение значений в пределах лямбды, но вовне эти изменения не передаются.

Если необходимо, чтобы внесенные в пределах лямбды изменения переменных распространялись также во вне, используйте ссылки:

```
[&StateVar1, &StateVar2](Type& param) { // здесь код лямбды; }
```

Лямбда-выражения могут получать по нескольку входных параметров, отделяемых запятыми:

```
[StateVar1, StateVar2](Type1& var1, Type2& var2) { // здесь код лямбды; }
```

Если хотите указать тип возвращаемого значения и не создавать неоднозначности для компилятора, используйте оператор -> так:

```
[State1, State2](Type1 var1, Type2 var2) -> ReturnType
{ return (// здесь значение или выражение); }
```

И наконец, составной оператор { } может содержать несколько операторов, разделенных точкой с запятой (;), как показано ниже:

```
[State1, State2](Type1 var1, Type2 var2) -> ReturnType
{ Statement 1; Statement 2; return (// здесь значение или выражение); }
```

ПРИМЕЧАНИЕ

В случае, если лямбда-выражение распространяется на несколько строк, тип возвращаемого значения следует указать явно.

В листинге 22.5 показана лямбда-функция, определяющая тип возвращаемого значения и охватывающая несколько строк.

Таким образом, лямбда-функция — это компактная, полнофункциональная замена такого объекта функции, как следующий:

```
template<typename Type1, typename Type2>
struct IsNowTooLong
{
    // переменные состояния
    Type1 var1;
    Type2 var2;

    // Конструктор
    IsNowTooLong(const Type1& in1, Type2& in2): var1(in1), var2(in2) {};

    // фактическая цель
    ReturnType operator()
    {
        Statement 1;
        Statement 2;
        return (value or expression);
    }
};
```

Лямбда-выражение для бинарной функции

Бинарная функция получает два параметра и (не обязательно) возвращает значение. Эквивалентное лямбда-выражение выглядит так:

```
[...](Type1& param1Name, Type2& param2Name) { // здесь код лямбды; }
```

В листинге 22.4 представлена лямбда-функция, перемножающая элемент за элементом два вектора равных размеров. Она использует алгоритм `std:: transform()` и сохраняет результат в третьем векторе.

ЛИСТИНГ 22.4. Лямбда-выражение как бинарная функция, перемножающая элементы двух контейнеров и сохраняющая результат в третьем

```
 0: #include <vector>
 1: #include <iostream>
 2: #include <algorithm>
 3:
 4: int main ()
 5: {
 6:     using namespace std;
 7:
 8:     // Создание двух векторов целых чисел по 10 элементов каждый
 9:     vector <int> vecMultiplicand, vecMultiplier;
10:
11:     // Вставить примеры значений от 0 до 9
12:     for (int nCount1 = 0; nCount1 < 10; ++ nCount1)
13:         vecMultiplicand.push_back (nCount1);
14:
15:     // Вставить примеры значений от 100 до 109
16:     for (int nCount2 = 100; nCount2 < 110; ++ nCount2)
17:         vecMultiplier.push_back (nCount2);
18:
19:     // Третий контейнер содержит результат умножения
20:     vector <int> vecResult;
21:
22:     // Создать пространство для результата умножения
23:     vecResult.resize (10);
24:
25:     transform ( vecMultiplicand.begin (), // диапазон множителей
26:                 vecMultiplicand.end (),   // конец диапазона
27:                 vecMultiplier.begin (),   // значения множителей
28:                 vecResult.begin (), // диапазон, содержащий результат
29:                 [](int a, int b) {return a * b; } ); // лямбда
30:
31:     cout << "The contents of the first vector are: " << endl;
32:     for (size_t nIndex1 = 0; nIndex1 < vecMultiplicand.size ();
                                 ++ nIndex1)
33:         cout << vecMultiplicand [nIndex1] << ' ';
34:     cout << endl;
35:
36:     cout << "The contents of the second vector are: " << endl;
```

```
37:     for (size_t nIndex2 = 0; nIndex2 < vecMultiplier.size ();
                                 ++nIndex2)
38:         cout << vecMultiplier [nIndex2] << ' ';
39:     cout << endl << endl;
40:
41:     cout << "The result of the multiplication is: " << endl;
42:     for (size_t nIndex = 0; nIndex < vecResult.size (); ++ nIndex)
43:         cout << vecResult [nIndex] << ' ';
44:     cout << endl;
45:
46:     return 0;
47: }
```

Результат

```
The contents of the first vector are:
0 1 2 3 4 5 6 7 8 9
The contents of the second vector are:
100 101 102 103 104 105 106 107 108 109

The result of the multiplication is:
0 101 204 309 416 525 636 749 864 981
```

Анализ

Рассматриваемое лямбда-выражение используется в строке 29 как параметр для алгоритма `std::transform()`. Этот алгоритм принимает два диапазона и применяет преобразование, содержащееся в бинарной функции. Возвращаемое значение бинарной функции сохраняется в выходном контейнере. Эта бинарная функция — лямбда-выражение, принимающее два целых числа и возвращающее результат умножения как возвращаемое значение. Полученное возвращаемое значение сохраняется алгоритмом `std::transform()` в векторе `vecResult`. Вывод демонстрирует содержимое двух контейнеров и результат умножения их элементов.

ПРИМЕЧАНИЕ Листинг 22.4 демонстрирует лямбда-эквивалент объекта функции класса `Multiply<>` из листинга 21.5.

Лямбда-выражение для бинарного предиката

Бинарная функция, возвращающая значение `true` или `false`, позволяет принять решение при вызове бинарного предиката. Эти предикаты применяются в алгоритмах сортировки, таких как `std::sort()`, которые вызывают бинарный предикат для любых двух значений в контейнере, чтобы узнать, какой из них должен располагаться перед другим. Вот обобщенный синтаксис бинарного предиката:

```
[...](Type1& param1Name, Type2& param2Name) { // выражение, возвращающее
логическое значение; }
```

В листинге 22.5 показано лямбда-выражение, используемое при сортировке.

ЛИСТИНГ 22.5. Лямбда-выражение как бинарный предикат алгоритма `std::sort()`, обеспечивающий независящую от регистра сортировку

```
 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 iElement = Input.cbegin() // auto, cbegin и cend: C++11
10:           ; iElement != Input.cend ()
11:           ; ++ iElement )
12:         cout << *iElement << endl;
13: }
14:
15: int main ()
16: {
17:     // Определить вектор строк для содержания имен
18:     vector <string> vecNames;
19:
20:     // Вставить в вектор несколько примеров имен
21:     vecNames.push_back ("jim");
22:     vecNames.push_back ("Jack");
23:     vecNames.push_back ("Sam");
24:     vecNames.push_back ("Anna");
25:
26:     cout << "The names in vector in order of insertion: " << endl;
27:     DisplayContents(vecNames);
28:
29:     cout << "Names after sorting using default std::less<>: "
             << endl;
30:     sort(vecNames.begin(), vecNames.end());
31:     DisplayContents(vecNames);
32:
33:     cout << "Names after sorting using predicate that ignores case:"
             << endl;
34:     sort(vecNames.begin(), vecNames.end(),
35:             [](const string& str1, const string& str2) -> bool
                                                          // лямбда
36:         {
37:             string str1LowerCase;
38:
39:             // Зарезервировать пространство
40:             str1LowerCase.resize (str1.size ());
41:
42:             // Преобразовать каждый символ в нижний регистр
43:             transform(str1.begin(), str1.end(),
                          str1LowerCase.begin(), tolower);
44:
45:             string str2LowerCase;
46:             str2LowerCase.resize (str2.size ());
47:             transform (str2.begin (), str2.end (),
                           str2LowerCase.begin (),
48:             tolower);
```

```
49:
50:             return (str1LowerCase < str2LowerCase);
51:         } // конец лямбды
52:     ); // конец сортировки
53:     DisplayContents(vecNames);
54:
55:     return 0;
56: }
```

Результат

```
The names in vector in order of insertion:
jim
Jack
Sam
Anna
Names after sorting using default std::less<>:
Anna
Jack
Sam
jim
Names after sorting using predicate that ignores case:
Anna
Jack
jim
Sam
```

Анализ

Здесь приведена большая лямбда-функция, охватывающая строки 35–51, как третий параметр алгоритма `std::sort()`, который начинается в строке 34 и заканчивается в строке 52! Это показывает, что лямбда-функция может состоять из нескольких операторов и предваряться явным определением типа возвращаемого значения (`bool`), как показано в строке 35. Вывод демонстрирует содержимое вектора после вставки значений, где `jim` следует перед `Jack`, содержимое вектора после сортировки по умолчанию без предоставленных лямбды и предиката (строка 30), где `jim` следует после `Sam` благодаря чувствительному к регистру оператору `string::operator<`, и наконец, версию, использующую независящее от регистра лямбда-выражение (строки 34–52), где `jim` следует после `Jack`, как и ожидал бы обычно пользователь. Кроме того, при этой сортировке используется лямбда-выражение, охватывающее несколько строк.

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 лямбды, поскольку в данном случае объект функции допускает повторное использование в нескольких выражениях `std::sort()`, если нужно, а также в других алгоритмах, которые нуждаются в бинарном предикате.

Поэтому лямбды необходимо использовать тогда, когда они коротки, изящны и эффективны.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Помните, что лямбда-выражения всегда начинаются с квадратных скобок ([]) или ([state1, state2...])	**Не забывайте**, что лямбда-выражения – это безымянные представления класса или структуры с оператором `operator()`
Помните, что если не указано иное, переменные состояния, предоставляемые в пределах списка захвата [], не допускают изменения. Для возможности изменения используйте ключевое слово `mutable`	**Не забывайте** использовать константные параметры при создании лямбда-выражений `[](const T& value) { // лямбда-выражение ; }`
	Не забывайте явно указывать тип возвращаемого значения, когда лямбда-выражение включает несколько операторов в пределах блока {}
	Не предпочитайте лямбда-выражения объекту функции, когда лямбда становится чрезвычайно длинной и охватывает несколько операторов, поскольку они переопределяются при каждом использовании и не позволяют использовать код повторно

Резюме

На сегодняшнем занятии рассматривалось очень важное средство языка C++11 — лямбда-функции. Вы узнали, что лямбды — это безымянные объекты функций, способные получать параметры, хранить состояние, возвращать значения и быть многострочными. Вы научились использовать лямбды вместо объектов функций в таких алгоритмах STL, как `find()`, `sort()` или `transform()`. Лямбды делают программирование на C++ быстрым и эффективным, поэтому постарайтесь использовать их по возможности чаще.

Вопросы и ответы

- **Всегда ли я должен предпочитать лямбды объектам функций?**
  Лямбды, охватывающие несколько строк (листинг 22.5), не помогут увеличить эффективность программы по сравнению с объектом функции, который легко можно использовать многократно.
- **Как передаются параметры состояния лямбды, по значению или по ссылке?**
  Лямбда создается со списком захвата так:

```
[Var1, Var2, ... N](Type& Param1, ... ) { ...expression ;}
```

- Параметры состояния `Var1` и `Var2` копируются (а не передаются по ссылке). Если хотите иметь их как ссылочные параметры, используйте такой синтаксис:

```
[&Var1, &Var2, ... &N](Type& Param1, ... ) { ...expression ;}
```

  В данном случае нужна осторожность, поскольку модификация переменных состояния, переданных в пределах списка захвата, распространяется вне лямбды.
- **Могу ли я использовать локальные переменные в лямбде как в функции?**

Вы можете передать локальные переменные в списке захвата:

```
[Var1, Var2, ... N](Type& Param1, ... ) { ...expression ;}
```

Если хотите захватить все переменные, используйте такой синтаксис:

```
[=](Type& Param1, ... ) { ...expression ;}
```

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 Г, "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 вы передали бы переменные состояния лямбда-функции?
3. Если необходимо передать возвращаемое значение из лямбды, то как это сделать?

Упражнения

1. Напишите бинарный предикат на базе лямбды, который обеспечил бы сортировку в порядке убывания.
2. Напишите лямбда-функцию, которая, будучи использована в алгоритме `for_each()`, добавит заданное пользователем значение к значению элемента такого контейнера, как вектор.

ЗАНЯТИЕ 23
Алгоритмы библиотеки STL

Важнейшей частью стандартной библиотеки шаблонов (STL) является набор обобщенных функций, предоставляемых заголовком `<algorithm>` для помощи в работе с содержимым контейнера.

На сегодняшнем занятии.

- Как использовать алгоритмы библиотеки STL для сокращения шаблонного кода.
- Обобщенные функции, помогающие при подсчете, поиске, удалении и других операциях с контейнерами STL.

Что такое алгоритмы STL

Поиск, нахождение, удаление и подсчет являются наиболее популярными обобщенными алгоритмами, которые находят применение в широком диапазоне программ. Библиотека STL решает эти и многие другие задачи в форме обобщенных шаблонов функций, которые воздействуют на контейнеры через итераторы. Чтобы использовать алгоритмы STL, программист сначала должен включить в код заголовок `<algorithm>`.

ПРИМЕЧАНИЕ

Хотя большинство алгоритмов работают с контейнерами через итераторы, не все они воздействуют обязательно на контейнеры, а следовательно, не все алгоритмы нуждаются в итераторах. Некоторые из них, такие как `swap()`, просто получают пару значений для обмена. Подобно алгоритму `swap()`, алгоритм `max()` также воздействует непосредственно на значения.

Классификация алгоритмов STL

Алгоритмы STL можно подразделить на два типа: изменяющие и не изменяющие.

Не изменяющие алгоритмы

Алгоритмы, которые не изменяют ни порядок, ни содержимое контейнера, называются *не изменяющими алгоритмами* (non-mutating algorithm). Некоторые из не изменяющих алгоритмов представлены в табл. 23.1.

ТАБЛИЦА 23.1. Краткий перечень не изменяющих алгоритмов

Алгоритм	Описание
Алгоритмы подсчета	
`count`	Находит в диапазоне все элементы, значения которых соответствуют предоставленному значению
`count_if`	Находит в диапазоне все элементы, значения которых удовлетворяют предоставленному условию
Алгоритмы поиска	
`search`	Поиск в диапазоне первого вхождения заданной последовательности на основании равенства элемента (т.е. оператора ==) или указанного бинарного предиката
`search_n`	Поиск в диапазоне первого вхождения *n* элементов с заданным значением или удовлетворяющих заданному предикату
`find`	Поиски в диапазоне первого элемента, соответствующего заданному значению
`find_if`	Поиски в диапазоне первого элемента, удовлетворяющего заданному условию
`find_end`	Поиски в диапазоне последнего вхождения заданного поддиапазона

Окончание табл. 23.1

Алгоритм	Описание
`find_first_of`	Поиски в диапазоне первого вхождения любого элемента целевого диапазона или (в перегруженной версии) первого вхождения элемента, удовлетворяющего заданному критерию поиска
`adjacent_find`	Поиски в коллекции двух элементов, которые равны или удовлетворяют заданному условию
Алгоритмы сравнения	
`equal`	Сравнивает два элемента на равенство или использует заданный предикат для той же цели
`mismatch`	Находит первую позицию различия в двух диапазонах элементов, используя заданный бинарный предикат
`lexicographical_compare`	Сравнивает элементы двух последовательностей, чтобы определить, которая из двух 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
Алгоритмы инициализации	
`fill`	Присваивает заданное значение каждому элементу в указанном диапазоне
`fill_n`	Присваивает заданное значение первым n элементам в указанном диапазоне
`generate`	Присваивает возвращаемое значение заданного объекта функции каждому элементу в указанном диапазоне
`generate_n`	Присваивает созданное функцией значение определенному количеству элементу в указанном диапазоне
Алгоритмы изменения	
`for_each`	Выполняет операцию с каждым из элементов диапазона. Когда определенный аргумент изменяет диапазон, алгоритм `for_each()` становится изменяющим
`transform`	Применяет определенную унарную функцию к каждому элементу в указанном диапазоне
Алгоритмы копирования	
`copy`	Копирует один диапазон в другой
`copy_backward`	Копирует один диапазон в другой, упорядочивая его элементы в обратном порядке
Алгоритмы удаления	
`remove`	Удаляет элемент заданного значения из указанного диапазона

Окончание табл. 23.2

Алгоритм	Описание
`remove_if`	Удаляет элемент, удовлетворяющий заданному унарному предикату из указанного диапазона
`remove_copy`	Копирует все элементы исходного диапазона в результирующий, кроме таковых с определенным значением
`remove_copy_if`	Копирует все элементы исходного диапазона в результирующий, кроме удовлетворяющих заданному унарному предикату
`unique`	Сравнивает смежные элементы в диапазоне и удаляет соседствующие дубликаты. Перегруженная версия использует бинарный предикат
`unique_copy`	Копирует все смежные двойные элементы из исходного диапазона в результирующий
Алгоритмы замены	
`replace`	Заменяет в диапазоне каждый элемент, соответствующий указанному значению, заданным значением
`replace_if`	Заменяет в диапазоне каждый элемент, удовлетворяющий указанному условию, заданным значением
Алгоритмы сортировки	
`sort`	Сортирует элементы диапазона, используя заданный критерий сортировки, являющийся бинарным предикатом. Способен изменить относительные позиции эквивалентных элементов
`stable_sort`	Сортировка подобно алгоритму `sort()`, но с сохранением порядка
`partial_sort`	Сортирует указанное количество элементов в диапазоне
`partial_sort_copy`	Копирует элементы исходного диапазона в результирующий с сортировкой
Алгоритмы разделения	
`partition`	Разделяет заданный диапазон на два набора: значения элементов первого удовлетворяют унарному предикату, а остальные следуют после. Может не поддерживать относительный порядок элементов в наборе
`stable_partition`	Разделяет заданный диапазон на два набора, как и алгоритм `partition()`, но обеспечивает относительный порядок
Алгоритмы для работы с отсортированными контейнерами	
`binary_search`	Используется для определения наличия элемента в отсортированной коллекции
`lower_bound`	Возвращает итератор, указывающий на первую позицию, куда потенциально может быть вставлен элемент отсортированной коллекции на основании его значения или заданного бинарного предиката
`upper_bound`	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

При наличии такого контейнера, как вектор, алгоритмы `find()` и `find_if()` библиотеки STL помогают найти элемент, который соответствует значению или удовлетворяет условию соответственно. Применение алгоритма `find()` осуществляется по следующему шаблону:

```
auto iElementFound = find ( vecIntegers.cbegin () // Начало диапазона
                          , vecIntegers.cend ()   // Конец диапазона
                          , NumToFind );          // Искомый элемент
// Проверить успех поиска
if ( iElementFound != vecIntegers.cend ())
    cout << "Result: Value found!" << endl;
```

Алгоритм `find_if()` похож на `find()`, но требует предоставления унарного предиката (унарной функции, возвращающей значение `true` или `false`) как третьего параметра.

```
auto iEvenNumber = find_if ( vecIntegers.cbegin () // Начало диапазона
                           , vecIntegers.cend ()   // Конец диапазона
                , [](int element) { return (element % 2) == 0; } );

if (iEvenNumber != vecIntegers.cend ())
cout << "Result: Value found!" << endl;
```

Таким образом, обе функции поиска возвращают итератор, который необходимо сравнить с результатом метода `end()` или `cend()` контейнера, чтобы проверить успех операции поиска. Если проверка прошла успешно, можете использовать этот итератор далее. 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 функции `find()` для поиска значения в векторе и функции `find_if()` для поиска первого четного числа по заданному унарному предикату в лямбда-выражении

```
 0: #include <iostream>
 1: #include <algorithm>
 2: #include <vector>
 3:
 4: int main()
 5: {
 6:     using namespace std;
 7:     vector<int> vecIntegers;
 8:
 9:     // Вставка примеров значений от -9 до 9
10:     for (int SampleValue = -9; SampleValue < 10; ++ SampleValue)
11:         vecIntegers.push_back (SampleValue);
12:
```

```
13:     cout << "Enter number to find in collection: ";
14:     int NumToFind = 0;
15:     cin >> NumToFind;
16:
17:     auto iElementFound = find ( vecIntegers.cbegin () // Начало
                                                          // диапазона
18:                               , vecIntegers.cend ()   // Конец
                                                          // диапазона
19:                               , NumToFind );    // Искомый элемент
20:
21:     // Проверить успех поиска
22:     if ( iElementFound != vecIntegers.cend ())
23:         cout << "Result: Value " << *iElementFound << " found!"
                 << endl;
24:     else
25:         cout << "Result: No element contains value " << NumToFind
                 << endl;
26:
27:     cout << "Finding the first even number using find_if: " << endl;
28:
29:     auto iEvenNumber = find_if ( vecIntegers.cbegin () // Начало
                                                           // диапазона
30:                                , vecIntegers.cend ()   // Конец
                                                           // диапазона
31:                 , [](int element) { return (element % 2) == 0; } );
32:
33:     if (iEvenNumber != vecIntegers.cend ())
34:     {
35:         cout << "Number '" << *iEvenNumber
                 << "' found at position [";
36:         cout << distance (vecIntegers.cbegin (), iEvenNumber);
37:         cout << "]" << endl;
38:     }
39:
40:     return 0;
41: }
```

Результат

```
Enter number to find in collection: 7
Result: Value 7 found!
Finding the first even number using find_if:
Number '-8' found at position [1]
Следующий запуск:
Enter number to find in collection: 2011
Result: No element contains value 2011
Finding the first even number using find_if:
Number '-8' found at position [1]
```

Анализ

Функция `main()` начинается с создания вектора целых чисел, инициализированного значениями в диапазоне от –9 до 9. Алгоритм `find()` в строках 17–19 используется для

поиска введенного пользователем числа. Использование алгоритма `find_if()` для поиска первого четного числа в заданном диапазоне представлено в строках 29–31. Строка 31 является унарным предикатом, созданным как лямбда-выражение. Это лямбда-выражение возвращает значение `true`, когда элемент делится на 2.

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.

СОВЕТ

> В листинге 17.5 (см. занятие 17, "Классы динамических массивов библиотеки STL") также показано использование алгоритма `std::distance()` для определения смещения найденного элемента от начала вектора (строка 21).

Подсчет элементов по заданному значению или условию

Алгоритмы `std::count()` и `count_if()` помогают при подсчете элементов заданного диапазона. Алгоритм `std::find()` позволяет подсчитать количество элементов, которые соответствуют заданному значению (для проверки используется оператор равенства `operator==`):

```
size_t nNumZeroes = count (vecIntegers.begin (),vecIntegers.end (),0);
cout << "Number of instances of '0': " << nNumZeroes << endl << endl;
```

Алгоритм `std::count_if()` позволяет подсчитать количество элементов, которые удовлетворяют унарному предикату, переданному как параметр (это может быть объект функции или лямбда-выражение):

```
// Унарный предикат:
template <typename elementType>
bool IsEven (const elementType& number)
{
    return ((number % 2) == 0); // true, если четное
}
...
// Использование алгоритма count_if с унарным предикатом IsEven:
size_t nNumEvenElements = count_if (vecIntegers.begin (),
                                    vecIntegers.end (), IsEven <int> );
cout << "Number of even elements: " << nNumEvenElements << endl;
```

Применение этих функций показано в коде листинга 23.2.

ЛИСТИНГ 23.2. Применение функции `std::count()` для определения количества элементов с указанным значением и функции `count_if()` для определения количества элементов, удовлетворяющих условию

```
0: #include <algorithm>
1: #include <vector>
2: #include <iostream>
3:
```

```
 4: // Унарный предикат для функций *_if
 5: template <typename elementType>
 6: bool IsEven (const elementType& number)
 7: {
 8:     return ((number % 2) == 0); // true, если четное
 9: }
10:
11: int main ()
12: {
13:     using namespace std;
14:     vector <int> vecIntegers;
15:
16:     cout << "Populating a vector<int> with values from -9 to 9"
          << endl;
17:     for (int nNum = -9; nNum < 10; ++ nNum)
18:         vecIntegers.push_back (nNum);
19:
20:     // Использование алгоритма count для определения количества '0'
21:     size_t nNumZeroes = count (vecIntegers.begin (),
                                   vecIntegers.end (),0);
22:     cout << "Number of instances of '0': " << nNumZeroes << endl
          << endl;
23:
24:     // Использование алгоритма count_if с унарным предикатом IsEven:
25:     size_t nNumEvenElements = count_if (vecIntegers.begin (),
26:                                         vecIntegers.end (),
                                            IsEven <int> );
27:
28:     cout << "Number of even elements: " << nNumEvenElements << endl;
29:     cout << "Number of odd elements: ";
30:     cout << vecIntegers.size () - nNumEvenElements << endl;
31:
32:     return 0;
33: }
```

Результат

```
Populating a vector<int> with values from -9 to 9
Number of instances of '0': 1

Number of even elements: 9
Number of odd elements: 10
```

Анализ

В строке 21 алгоритм `count()` использован для определения количества значений 0 в векторе `vector<int>`. Точно так же в строке 25 алгоритм `count_if()` использован для определения количества четных чисел в векторе. Обратите внимание на третий параметр, которым является унарный предикат `IsEven()`, определенный в строках 6–9. Количество элементов с нечетными значениями в векторе вычисляется при вычитании возвращаемого значения функции `count_if()` из общего количества содержавшихся в векторе элементов, возвращаемого функцией `size()`.

ПРИМЕЧАНИЕ

В листинге 23.2 алгоритм `count_if()` использует функцию предиката `IsEven()`, тогда как в листинге 23.1 используется 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 как в алгоритме `find_if()`, так и в `count_if()`, обеспечивая возможность многократного использования.

Поиск элемента или диапазона в коллекции

В листинге 23.1 продемонстрирована возможность поиска элемента в контейнере. Но иногда необходимо найти диапазон значений или шаблон. В таких ситуациях следует использовать алгоритм `search()` или `search_n()`. Алгоритм `search()` применяется для проверки наличия одного диапазона в другом:

```
auto iRange = search ( vecIntegers.begin ()    // Начало диапазона
                     , vecIntegers.end ()      // Конец диапазона
                                               // для поиска
                     , listIntegers.begin ()   // Начало диапазона
                                               // для поиска
                     , listIntegers.end () ); // Конец диапазона
                                               // для поиска
```

Алгоритм `search_n()` применяется для проверки наличия в контейнере *n* экземпляров значения, расположенных последовательно:

```
auto iPartialRange = search_n ( vecIntegers.begin () // Начало диапазона
                              , vecIntegers.end ()   // Конец диапазона
                              , 3    // Количество искомых элементов
                              , 9 ); // Искомый элемент
```

Обе функции возвращают итератор на первый экземпляр 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& Input)
 8: {
 9:     for(auto iElement = Input.cbegin() // auto, cbegin и cend: C++11
10:           ; iElement != Input.cend()
11:           ; ++ iElement )
12:         cout << *iElement << ' ';
13:
```

```
14:     cout << endl;
15: }
16:
17: int main ()
18: {
19:     // Пример контейнера (вектор целых чисел, содержащий
        // значения от -9 до 9)
20:     vector <int> vecIntegers;
21:     for (int nNum = -9; nNum < 10; ++ nNum)
22:         vecIntegers.push_back (nNum);
23:
24:     // Вставить в вектор еще несколько примеров значений
25:     vecIntegers.push_back (9);
26:     vecIntegers.push_back (9);
27:
28:     // Еще один пример контейнера (список целых чисел от -4 до 4)
29:     list <int> listIntegers;
30:     for (int nNum = -4; nNum < 5; ++ nNum)
31:         listIntegers.push_back (nNum);
32:
33:     cout << "The contents of the sample vector are: " << endl;
34:     DisplayContents (vecIntegers);
35:
36:     cout << "The contents of the sample list are: " << endl;
37:     DisplayContents (listIntegers);
38:
39:     cout << "search() for the contents of list in vector:" << endl;
40:     auto iRange = search ( vecIntegers.begin ()   // Начало диапазона
41:                          , vecIntegers.end ()     // Конец диапазона
                                                      // для поиска
42:                          , listIntegers.begin ()  // Начало диапазона
                                                      // для поиска
43:                          , listIntegers.end () ); // Конец диапазона
                                                      // для поиска
44:
45:     // Проверка успеха поиска
46:     if (iRange != vecIntegers.end ())
47:     {
48:         cout << "Sequence in list found in vector at position: ";
49:         cout << distance (vecIntegers.begin (), iRange) << endl;
50:     }
51:
52:     cout << "Searching {9, 9, 9} in vector using search_n(): "
             << endl;
53:     auto iPartialRange = search_n ( vecIntegers.begin () // Начало
                                                            // диапазона
54:                                   , vecIntegers.end () // Конец диапазона
55:                                   , 3     // Количество искомых элементов
56:                                   , 9 );  // Искомый элемент
57:
58:     if (iPartialRange != vecIntegers.end ())
59:     {
60:         cout << "Sequence {9, 9, 9} found in vector at position: ";
61:         cout << distance (vecIntegers.begin (), iPartialRange)
```

```
                << endl;
62:     }
63:
64:     return 0;
65: }
```

Результат

```
The contents of the sample vector are:
-9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 9 9
The contents of the sample list are:
-4 -3 -2 -1 0 1 2 3 4
search() for the contents of list in vector:
Sequence in list found in vector at position: 5
Searching {9, 9, 9} in vector using search_n():
```

Анализ

Листинг начинается с двух примеров контейнеров, вектора и списка, которые первоначально заполнены примерами целочисленных значений. Алгоритм `search()` используется для выявления наличия содержимого списка в векторе, как показано в строке 40. Поскольку вы хотите искать содержимое всего списка во всем векторе, диапазон задается итераторами, возвращенными методами `begin()` и `end()` классов этих двух контейнеров. Это фактически демонстрирует то, как хорошо итераторы приспособлены к алгоритмам и контейнерам. Физические характеристики контейнеров, предоставляющие 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 53 для поиска первого вхождения прогрессии `{9, 9, 9}` в векторе.

Инициализация элементов в контейнере заданным значением

Алгоритмы `fill()` и `fill_n()` библиотеки STL позволяют заполнить содержимое заданного диапазона определенным значением. Алгоритм `fill()` используется для перезаписи значений элементов в диапазоне, заданном его границами, указанным значением:

```
vector <int> vecIntegers (3);

// заполнить все элементы контейнера значением 9
fill (vecIntegers.begin (), vecIntegers.end (), 9);
```

Как и предполагает его название, алгоритм `fill_n()` устанавливает значения `n` элементов. Он нуждается в исходной позиции, количестве и значении для заполнения:

```
fill_n (vecIntegers.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:     // Инициализировать пример вектора из 3 элементов
 9:     vector <int> vecIntegers (3);
10:
11:     // Заполнить все элементы контейнера значением 9
12:     fill (vecIntegers.begin (), vecIntegers.end (), 9);
13:
14:     // Увеличить размер вектора до 6 элементов
15:     vecIntegers.resize (6);
16:
17:     // Заполнить эти три элемента значением -9, начиная с позиции 3
18:     fill_n (vecIntegers.begin () + 3, 3, -9);
19:
20:     cout << "Contents of the vector are: " << endl;
21:     for (size_t nIndex = 0; nIndex < vecIntegers.size (); ++ nIndex)
22:     {
23:         cout << "Element [" << nIndex << "] = ";
24:         cout << vecIntegers [nIndex] << endl;
25:     }
26:
27:     return 0;
28: }
```

Результат

```
Contents of the vector are:
Element [0] = 9
Element [1] = 9
Element [2] = 9
Element [3] = -9
Element [4] = -9
Element [5] = -9
```

Анализ

Листинг 23.4 использует функции `fill()` и `fill_n()` 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, это создает элементы, которые впоследствии будут заполнены значениями. Алгоритм `fill()` воздействует на весь диапазон, а алгоритм `fill_n()` способен воздействовать на часть диапазона.

Использование алгоритма std::generate() для инициализации элементов значениями, созданными во время выполнения

Подобно тому, как функции fill() и fill_n() заполняют коллекцию определенным значением, такие алгоритмы библиотеки STL, как generate() и generate_n(), инициализируют коллекции значениями, возвращаемыми унарной функцией.

Вы можете использовать функцию generate() для заполнения диапазона с использованием возвращаемого значения функции-генератора:

```
generate ( vecIntegers.begin (), vecIntegers.end () // диапазон
         , rand ); // вызов функции-генератора
```

Алгоритм generate_n() подобен алгоритму generate() за исключением того, что необходимо указать количество элементов, которым будут присвоены значения, а не границы диапазона:

```
generate_n (listIntegers.begin (), 5, rand);
```

Таким образом, вы можете использовать эти два алгоритма для инициализации содержимого контейнера содержимым файла, например, 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:
 5: int main ()
 6: {
 7:     using namespace std;
 8:
 9:     vector <int> vecIntegers (10);
10:     generate ( vecIntegers.begin (), vecIntegers.end () // диапазон
11:              , rand ); // вызов функции-генератора
12:
13:     cout << "Elements in the vector of size " << vecIntegers.size ();
14:     cout << " assigned by 'generate' are: " << endl << "{";
15:     for (size_t nCount = 0; nCount < vecIntegers.size (); ++ nCount)
16:         cout << vecIntegers [nCount] << " ";
17:
18:     cout << "}" << endl << endl;
19:
20:     list <int> listIntegers (10);
21:     generate_n (listIntegers.begin (), 5, rand);
22:
23:     cout << "Elements in the list of size: " << listIntegers.size ();
24:     cout << " assigned by 'generate_n' are: " << endl << "{";
25:     list <int>::const_iterator iElementLocator;
26:     for ( iElementLocator = listIntegers.begin ()
```

```
27:         ; iElementLocator != listIntegers.end ()
28:         ; ++ iElementLocator )
29:     cout << *iElementLocator << ' ';
30:
31:     cout << "}" << endl;
32:
33:     return 0;
34: }
```

Результат

```
Elements in the vector of size 10 assigned by 'generate' are:
{41 18467 6334 26500 19169 15724 11478 29358 26962 24464 }

Elements in the list of size: 10 assigned by 'generate_n' are:
{5705 28145 23281 16827 9961 0 0 0 0 0 }
```

Анализ

Листинг 23.5 использует функцию `generate()` для заполнения всех элементов вектора случайными значениями, предоставляемыми функцией `rand()`. Обратите внимание, что функция `generate()` получает диапазон, а следовательно, вызывает определенный объект функции `rand()` для каждого его элемента. Функция `generate_n()`, напротив, получает только исходную позицию. Затем вызывается объект указанной функции `rand()` такое количество раз, которое задано параметром `count`. В результате содержимое заданного количества элементов будет перезаписано. Элементы контейнера вне заданной последовательности 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:

```
unaryFunctionObjectType mReturn = for_each ( start_of_range
                                           , end_of_range
                                           , unaryFunctionObject );
```

Объект унарной функции может быть также лямбда-выражением, которое получает один параметр. Возвращаемое значение свидетельствует о том, что функция `for_each()` возвращает объект функции (называемый также функтором), который обрабатывает каждый элемент в заданном диапазоне. Преимущество такой конструкции в том, что при использовании структуры или класса для создания объекта функции можно хранить информацию о состоянии, к которой можно обратиться впоследствии, по завершении функции `for_each()`. Это показано в листинге 23.6, код которого использует объект функции для отображения элементов в диапазоне, а также посчитывает количество отображенных элементов.

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: // Тип объекта унарной функции, вызываемой алгоритмом for_each
 7: template <typename elementType>
 8: struct DisplayElementKeepCount
 9: {
10:     int Count;
11:
12:     // Конструктор
13:     DisplayElementKeepCount (): Count (0) {}
14:
15:     void operator () (const elementType& element)
16:     {
17:         ++ Count;
18:         cout << element << ' ';
19:     }
20: };
21:
22: int main ()
23: {
24:     vector <int> vecIntegers;
25:     for (int nCount = 0; nCount < 10; ++ nCount)
26:         vecIntegers.push_back (nCount);
27:
28:     cout << "Displaying the vector of integers: " << endl;
29:
30:     // Отобразить массив целых чисел
31:     DisplayElementKeepCount<int> Functor =
32:     for_each ( vecIntegers.begin () // Начало диапазона
33:              , vecIntegers.end ()   // Конец диапазона
34:              , DisplayElementKeepCount<int> () );// Функтор
34:
35:     cout << endl;
36:
37:     // Использование состояния, хранимого в возвращаемом значении
       // алгоритма for_each!
38:     cout << "'" << Functor.Count << "' elements were displayed"
            << endl;
39:
40:     string Sample ("for_each and strings!");
41:     cout << "String: " << Sample << ", length: " << Sample.length()
            << endl;
42:
43:     cout << "String displayed using lambda:" << endl;
44:     int NumChars = 0;
45:     for_each ( Sample.begin()
46:              , Sample.end ()
47:              , [&NumChars](char c) { cout << c << ' '; ++NumChars; } );
```

```
48:
49:     cout << endl;
50:     cout << "'" << NumChars << "' characters were displayed" << endl;
51:
52:     return 0;
53: }
```

Результат

```
Displaying the vector of integers:
0 1 2 3 4 5 6 7 8 9
'10' elements were displayed
String: for_each and strings!, length: 21
String displayed using lambda:
f o r _ e a c h a n d s t r i n g s !
'21' characters were displayed
```

Анализ

Пример кода демонстрирует удобство алгоритма `for_each()`, а также его способность возвратить объект функции `Result`, разработанный так, чтобы содержать такую информацию, как количество его вызовов. В коде используются два примера диапазонов: один, содержавшийся в векторе целых чисел `vecIntegers`, а другой, `Sample`, — объект класса `std::string`. Код вызывает функцию `for_each()` для этих диапазонов в строках 32 и 45 соответственно. В первый раз с использованием унарного предиката `DisplayElementKeepCount`, а второй — лямбда-выражения. 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. Когда функция `for_each()` завершает работу, возвращается объект функции, а ее член `Count` сообщает количество использований объекта. Этот способ хранения информации (или состояния) в объекте, который возвращается алгоритмом, может быть весьма полезен в практических ситуациях. Алгоритм `for_each()` в строке 45 делает для объекта класса `std::string` то же самое, что и его аналог в строке 32, но с использованием лямбда-выражения вместо объекта функции.

Выполнение преобразований в диапазоне с использованием алгоритма `std::transform()`

Алгоритмы `std::for_each()` и `std::transform()` очень похожи в том, что оба они вызывают объект функции для каждого элемента в исходном диапазоне. Однако у алгоритма `std::transform()` есть две версии. Первая версия получает унарную функцию и обычно используется для преобразования символов строки в верхний или нижний регистр с использованием функции `toupper()` или `tolower()`:

```
string Sample ("THIS is a TEst string!");
transform ( Sample.begin ()           // начало исходного диапазона
          , Sample.end ()             // конец исходного диапазона
          , strLowerCaseCopy.begin () // начало диапазона назначения
          , tolower );                // унарная функция
```

Вторая версия получает бинарную функцию, позволяющую алгоритму `transform()` обработать пару элементов, взятых из двух разных диапазонов:

```
// сложить элементы из двух диапазонов и сохранить результат в третьем
transform ( vecIntegers1.begin ()      // начало исходного диапазона 1
          , vecIntegers1.end ()        // конец исходного диапазона 1
          , vecIntegers2.begin ()      // начало исходного диапазона 2
          , dqResultAddition.begin () // сохранить результат в
                                       // двухсторонней очереди
          , plus <int> () );           // бинарная функция plus
```

Обе версии алгоритма `transform()` всегда присваивают результат определенной функции преобразования предоставленному диапазону назначения, в отличие от алгоритма `for_each()`, который воздействует только на один диапазон. 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 Sample ("THIS is a TEst string!");
12:     cout << "The sample string is: " << Sample << endl;
13:
14:     string strLowerCaseCopy;
15:     strLowerCaseCopy.resize (Sample.size ());
16:
17:     transform ( Sample.begin ()          // начало исходного диапазона
18:               , Sample.end ()            // конец исходного диапазона
19:               , strLowerCaseCopy.begin () // начало диапазона
                                              // назначения
20:               , tolower );               // унарная функция
21:
22:     cout << "Result of 'transform' on the string with 'tolower':"
             << endl;
23:     cout << "\"" << strLowerCaseCopy << "\"" << endl << endl;
24:
25:     // Два примера векторов целых чисел...
26:     vector <int> vecIntegers1, vecIntegers2;
27:     for (int nNum = 0; nNum < 10; ++ nNum)
28:     {
29:         vecIntegers1.push_back (nNum);
30:         vecIntegers2.push_back (10 - nNum);
31:     }
32:
33:     // Диапазон назначения для содержания результата сложения
```

```
34:     deque <int> dqResultAddition (vecIntegers1.size ());
35:
36:     transform ( vecIntegers1.begin () // начало исходного диапазона 1
37:               , vecIntegers1.end ()   // конец исходного диапазона 1
38:               , vecIntegers2.begin () // начало исходного диапазона 2
39:               , dqResultAddition.begin () // начало диапазона
                                              // назначения
40:               , plus <int> () );          // бинарная функция
41:
42:     cout << "Result of 'transform' using binary function 'plus': "
             << endl;
43:     cout <<endl << "Index Vector1 + Vector2 = Result (in Deque)"
             << endl;
44:     for (size_t nIndex = 0; nIndex < vecIntegers1.size (); ++ nIndex)
45:     {
46:         cout << nIndex << " \t " << vecIntegers1 [nIndex] << "\t+ ";
47:         cout << vecIntegers2 [nIndex] << " \t = ";
48:
49:         cout << dqResultAddition [nIndex] << endl;
50:     }
51:
52:     return 0;
53: }
```

Результат

```
The sample string is: THIS is a TEst string!
Result of using 'transform' with unary function 'tolower' on the string:
"this is a test string!"

Result of 'transform' using binary function 'plus':
Index  Vector1 + Vector2 = Result (in Deque)
0       0      +  10     =  10
1       1      +  9      =  10
2       2      +  8      =  10
3       3      +  7      =  10
4       4      +  6      =  10
5       5      +  5      =  10
6       6      +  4      =  10
7       7      +  3      =  10
8       8      +  2      =  10
9       9      +  1      =  10
```

Анализ

Пример демонстрирует обе версии алгоритма `std::transform()`: ту, которая воздействует на один диапазон с использованием унарной функции `tolower()`, как показано в строке 20, и вторую, которая воздействует на два диапазона с использованием бинарной функции `plus()`, как показано в строке 40. 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 36–40, воздействует на элементы, взятые

из двух исходных диапазонов (в данном случае два вектора), и использует бинарный предикат в форме функции plus() библиотеки STL (определена в заголовке <functional>) для их суммирования. Функция std::transform() получает одну пару за один раз, передавая их бинарной функции plus(), и присваивает результат элементу в диапазоне назначения, который в данном случае принадлежит контейнеру класса std:deque. Обратите внимание, что результат в отдельном контейнере сохраняется в демонстрационных целях. Это показывает, насколько хорошо итераторы абстрагируют контейнеры и их реализацию от алгоритмов STL; функция transform(), будучи алгоритмом, работает с диапазонами и действительно не обязана знать подробности о контейнере, который реализует эти диапазоны. Так, исходные диапазоны могут быть в векторе, а диапазоны назначения — в двухсторонней очереди, и все будет работать прекрасно, пока допустимы определяющие диапазон границы (предоставляемые как входные параметры функции transform()).

Операции копирования и удаления

Библиотека STL предоставляет три очевидных функции копирования: copy(), copy_if() и copy_backward(). Функция copy() способна присвоить содержимое исходного диапазона диапазону назначения в текущем порядке:

```
auto iLastPos = copy ( listIntegers.begin ()    // начало исходного
                                                // диапазона
                     , listIntegers.end ()      // конец исходного
                                                // диапазона
                     , vecIntegers.begin () ); // начало диапазона
                                                // назначения
```

Функция copy_if() копирует элемент, только если предоставленный вами унарный предикат возвращает значение true:

```
// скопировать нечетные числа из списка в вектор
copy_if ( listIntegers.begin(), listIntegers.end()
        , iLastPos
        , [](int element){return ((element % 2) == 1);});
```

ПРИМЕЧАНИЕ

Начиная с версии C++11 алгоритм copy_if() находится в пространстве имен std. Если вы используете старый компилятор или несовместимый со стандартом C++11, с его использованием могут возникнуть проблемы.

Функция copy_backward() присваивает содержимое диапазону назначения в обратном порядке:

```
copy_backward ( listIntegers.begin ()
              , listIntegers.end ()
              , vecIntegers.end () );
```

Функция remove(), напротив, удаляет из контейнера элементы, соответствующие определенному значению:

```
// Удалить все экземпляры '0' и изменить размер вектора,
// используя erase()
auto iNewEnd = remove (vecIntegers.begin (), vecIntegers.end (), 0);
vecIntegers.erase (iNewEnd, vecIntegers.end ());
```

Функция `remove_if()` использует унарный предикат и удаляет из контейнера те элементы, для которых предикат возвращает значение `true`:

```
// Удалить все нечетные числа из вектора, используя remove_if()
iNewEnd = remove_if (vecIntegers.begin (), vecIntegers.end (),
          [](int element) {return ((element % 2) == 1);} ); // предикат

vecIntegers.erase (iNewEnd , vecIntegers.end ());    // изменение
                                                     // размера
```

Применение функций удаления и копирования показаны в листинге 23.8.

ЛИСТИНГ 23.8. Функции `copy()`, `copy_if()`, `remove()` и `remove_if()` для копирования списка в вектор, а также удаления четных и нулевых чисел

```
 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& Input)
 8: {
 9:     for ( auto iElement = Input.cbegin() // auto, cbegin: C++11
10:             ; iElement != Input.cend()  // cend(): C++11
11:             ; ++ iElement)
12:         cout << *iElement << ' ';
13:
14:     cout << "| Number of elements: " << Input.size() << endl;
15: }
16: int main ()
17: {
18:     list <int> listIntegers;
19:     for (int nCount = 0; nCount < 10; ++ nCount)
20:         listIntegers.push_back (nCount);
21:
22:     cout << "Source (list) contains:" << endl;
23:     DisplayContents(listIntegers);
24:
25:     // Инициализировать вектор так, чтобы он содержал вдвое больше
       // элементов, чем список
26:     vector <int> vecIntegers (listIntegers.size () * 2);
27:
28:     auto iLastPos = copy ( listIntegers.begin ()  // начало исходного
                                                      // диапазона
29:                          , listIntegers.end ()    // конец исходного
                                                      // диапазона
30:                          , vecIntegers.begin () ); // начало
                                                  // диапазона назначения
31:
32:     // скопировать нечетные числа из списка в вектор
33:     copy_if ( listIntegers.begin(), listIntegers.end()
34:             , iLastPos
35:             , [](int element){return ((element % 2) == 1);});
36:
37:     cout << "Destination (vector) after copy and copy_if:" << endl;
```

```
38:     DisplayContents(vecIntegers);
39:
40:     // Удалить все экземпляры '0' и изменить размер вектора,
        // используя erase()
41:     auto iNewEnd = remove (vecIntegers.begin (),
                               vecIntegers.end (), 0);
42:     vecIntegers.erase (iNewEnd, vecIntegers.end ());
43:
44:     // Удалить все нечетные числа из вектора, используя remove_if
45:     iNewEnd = remove_if (vecIntegers.begin (), vecIntegers.end (),
46:              [](int element) {return ((element % 2) == 1);} );
                                                        // предикат
47:
48:     vecIntegers.erase (iNewEnd , vecIntegers.end ()); // изменение
                                                        // размера
49:
50:     cout << "Destination (vector) after remove, remove_if, erase:"
             << endl;
51:     DisplayContents(vecIntegers);
52:
53:     return 0;
54: }
```

Результат

```
Source (list) contains:
0 1 2 3 4 5 6 7 8 9 | Number of elements: 10
Destination (vector) after copy and copy_if:
0 1 2 3 4 5 6 7 8 9 1 3 5 7 9 0 0 0 0 0 | Number of elements: 20
Destination (vector) after remove, remove_if, erase:
2 4 6 8 | Number of elements: 4
```

Анализ

Применение функции `copy()` представлено в строке 28, где содержимое списка копируется в вектор. Функция `copy_if()` используется в строке 33, где она копирует все четные числа из исходного диапазона списка `listIntegers` в диапазон назначения вектора `vecIntegers`, начиная с позиции, указанной итератором `iLastPos`, возвращенным функцией `copy()`. Функция `remove()` представлена в строке 41. Она используется для избавления вектора `vecIntegers` от всех экземпляров со значением 0. Функция `remove_if()` используется в строке 45 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 контейнер `vecIntegers` еще не был изменен. Элементы были удалены алгоритмами удаления, и другие элементы были сдвинуты вперед, однако размер вектора остался неизменным, т.е. значения в конце остались. Чтобы изменить размеры контейнера (и это очень важно, иначе в конце останутся нежелательные значения), необходимо использовать итератор, возвращенный функцией `remove()` или `remove_if()` в последующем вызове метода `erase()`, как показано в строках 42 и 48.

Замена значений и элементов по заданному условию

Алгоритмы replace() и replace_if() библиотеки STL позволяют заменить в коллекции элементы, которые соответствуют определенному значению или удовлетворяют заданному условию соответственно. Функция replace() заменяет элементы на основании значения, возвращаемого оператором сравнения (==):

```
cout << "Using 'std::replace' to replace value 5 by 8" << endl;
replace (vecIntegers.begin (), vecIntegers.end (), 5, 8);
```

Функция replace_if() ожидает определенный пользователем унарный предикат, который возвращает значение true для каждого значения, подлежащего замене:

```
cout << "Using 'std::replace_if' to replace even values by -1" << endl;
replace_if (vecIntegers.begin (), vecIntegers.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& Input)
 7: {
 8:     for ( auto iElement = Input.cbegin() // auto, cbegin: C++11
 9:             ; iElement != Input.cend()  // cend(): C++11
10:             ; ++ iElement)
11:         cout << *iElement << ' ';
12:
13:     cout << "| Number of elements: " << Input.size() << endl;
14: }
15: int main ()
16: {
17:     vector <int> vecIntegers (6);
18:
19:     // заполнить сначала 3 элемента значением 8, а
        // последние 3 значением 5
20:     fill (vecIntegers.begin (), vecIntegers.begin () + 3, 8);
21:     fill_n (vecIntegers.begin () + 3, 3, 5);
22:
23:     // переупорядочить контейнер
24:     random_shuffle (vecIntegers.begin (), vecIntegers.end ());
25:
26:     cout << "The initial contents of the vector are: " << endl;
27:     DisplayContents(vecIntegers);
28:
29:     cout << endl << "Using 'std::replace' to replace value 5 by 8"
            << endl;
30:     replace (vecIntegers.begin (), vecIntegers.end (), 5, 8);
```

```
31:
32:     cout << "Using 'std::replace_if' to replace even values by -1"
             << endl;
33:     replace_if (vecIntegers.begin (), vecIntegers.end ()
34:               , [](int element) {return ((element % 2) == 0); }, -1);
35:
36:     cout << endl << "Contents of the vector after replacements:"
             << endl;
37:     DisplayContents(vecIntegers);
38:
39:     return 0;
40: }
```

Результат

```
The initial contents of the vector are:
5 8 5 8 8 5 | Number of elements: 6

Using 'std::replace' to replace value 5 by 8
Using 'std::replace_if' to replace even values by -1

Contents of the vector after replacements:
-1 -1 -1 -1 -1 -1 | Number of elements: 6
```

Анализ

Код заполняет вектор `vecIntegers` типа `vector<int>` примерами значений, а затем переупорядочивает его, используя алгоритм `std::random_shuffle()` библиотеки STL, как показано в строке 24. Строка 30 демонстрирует применение функции `replace()` для замены всех значений 5 и 8. В строке 33 функция `replace_if()` заменяет все четные числа значением –1. В результате получается, что у коллекции есть шесть элементов, содержащих идентичное значение –1, как показано в выводе.

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, у вас есть массив информации, которая должна быть отсортирована, скажем, в целях ее представления. Для сортировки контейнера можно использовать алгоритм `sort()` библиотеки STL:

```
sort (vecIntegers.begin (), vecIntegers.end ()); // порядок возрастания
```

Эта версия функции `sort()` применяет бинарный предикат `std::less<>`, использующий оператор `operator<`, реализованный содержащимся в векторе типом. Вы можете предоставить собственный предикат, чтобы изменить порядок сортировки, используя следующую перегруженную версию:

```
sort (vecIntegers.begin (), vecIntegers.end (),
      [](int lhs, int rsh) {return (lhs > rhs);} ); // порядок убывания
```

Перед отображением коллекции следует удалить дубликаты. Для удаления расположенных рядом повторившихся значений используется алгоритм unique():

```
auto iNewEnd = unique (vecIntegers.begin (), vecIntegers.end ());
vecIntegers.erase (iNewEnd, vecIntegers.end ()); // изменить размер
```

Для быстрого поиска библиотека STL предоставляет алгоритм binary_search(), который эффективен только в отсортированном контейнере:

```
bool bElementFound = binary_search (vecIntegers.begin (),
                                    vecIntegers.end (), 2011);

if (bElementFound)
    cout << "Element found in the vector!" << endl;
```

В листинге 23.10 показаны такие алгоритмы библиотеки STL, как std::sort(), который способен отсортировать диапазон std::binary_search(), обеспечивающий поиск в отсортированном диапазоне, и std::unique(), удаляющий расположенные рядом совпадающие элементы (которые 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& Input)
 8: {
 9:     for ( auto iElement = Input.cbegin() // auto, cbegin: C++11
10:             ; iElement != Input.cend()  // cend(): C++11
11:             ; ++ iElement)
12:         cout << *iElement << endl;
13: }
14: int main ()
15: {
16:     vector<string> vecNames;
17:     vecNames.push_back ("John Doe");
18:     vecNames.push_back ("Jack Nicholson");
19:     vecNames.push_back ("Sean Penn");
20:     vecNames.push_back ("Anna Hoover");
21:
22:     // вставка дубликатов в вектор
23:     vecNames.push_back ("Jack Nicholson");
24:
25:     cout << "The initial contents of the vector are:" << endl;
26:     DisplayContents(vecNames);
27:
28:     cout << "The sorted vector contains names in the order:" << endl;
29:     sort (vecNames.begin (), vecNames.end ());
30:     DisplayContents(vecNames);
31:
32:     cout << "Searching for \"John Doe\" using 'binary_search':"
```

```
            << endl;
33:     bool bElementFound = binary_search (vecNames.begin (),
                                            vecNames.end (),
34:                                         "John Doe");
35:
36:     if (bElementFound)
37:         cout << "Result: \"John Doe\" was found in the vector!"
                 << endl;
38:     else
39:         cout << "Element not found " << endl;
40:
41:     // Удаление смежных дубликатов
42:     auto iNewEnd = unique (vecNames.begin (), vecNames.end ());
43:     vecNames.erase (iNewEnd, vecNames.end ());
44:
45:     cout << "The contents of the vector after using 'unique':"
             << endl;
46:     DisplayContents(vecNames);
47:
48:     return 0;
49: }
```

Результат

```
The initial contents of the vector are:
John Doe
Jack Nicholson
Sean Penn
Anna Hoover
Jack Nicholson
The sorted vector contains names in the order:
Anna Hoover
Jack Nicholson
Jack Nicholson
John Doe
Sean Penn
Searching for "John Doe" using 'binary_search':
Result: "John Doe" was found in the vector!
The contents of the vector after using 'unique':
Anna Hoover
Jack Nicholson
John Doe
Sean Penn
```

Анализ

Приведенный выше код сначала сортирует вектор `vecNames` (строка 29), а затем (строка 33) использует алгоритм `binary_search()` для поиска в нем элемента `John Doe`. Точно так же в строке 42 используется алгоритм `std::unique()` для удаления смежных дубликатов. Обратите внимание, что функция `unique()`, как и `remove()`, не изменяет размер контейнера. Это приводит к сдвигу значений, но не сокращению общего количества элементов. Чтобы избавиться от нежелательных или неизвестных значений

в конце контейнера, после вызова функции `unique()` всегда следует вызвать функцию `vector::erase()`, используя итератор, возвращенный функцией `unique()`, как показано в строках 42 и 43.

ВНИМАНИЕ!

Такие алгоритмы, как `binary_search()`, эффективны только в отсортированных контейнерах. При использовании этого алгоритма с неотсортированным вектором могут возникнуть нежелательные последствия.

ПРИМЕЧАНИЕ

Функция `stable_sort()` используется точно так же, как функция `sort()`, которая была представлена ранее. Функция `stable_sort()` обеспечивает относительный порядок отсортированных элементов. Поддержка относительного порядка обходится потерей производительности, что следует учитывать, особенно если порядок смежных элементов не является необходимым.

Разделение диапазона

Функция `std::partition()` позволяет разделить исходный диапазон на два раздела: тот, который удовлетворяет унарному предикату, и другой, который не удовлетворяет:

```
bool IsEven (const int& nNumber) // унарный предикат
{
    return ((nNumber % 2) == 0);
}
...
partition (vecIntegers.begin(), vecIntegers.end(), IsEven);
```

Однако функция `std::partition()` не гарантирует относительный порядок элементов в пределах каждого раздела. Когда это важно, следует использовать функцию `std::stable_partition()`:

```
stable_partition (vecIntegers.begin(), vecIntegers.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& nNumber)
 6: {
 7:     return ((nNumber % 2) == 0);
 8: }
 9:
10: template <typename T>
11: void DisplayContents(const T& Input)
12: {
13:     for ( auto iElement = Input.cbegin() // auto, cbegin: C++11
```

```
14:              ; iElement != Input.cend()  // cend(): C++11
15:              ; ++ iElement)
16:         cout << *iElement << ' ';
17:
18:     cout << "| Number of elements: " << Input.size() << endl;
19: }
20: int main ()
21: {
22:     vector <int> vecIntegers;
23:
24:     for (int nNum = 0; nNum < 10; ++ nNum)
25:     vecIntegers.push_back (nNum);
26:
27:     cout << "The initial contents: " << endl;
28:     DisplayContents(vecIntegers);
29:
30:     vector <int> vecCopy (vecIntegers);
31:
32:     cout << "The effect of using partition():" << endl;
33:     partition (vecIntegers.begin (), vecIntegers.end (), IsEven);
34:     DisplayContents(vecIntegers);
35:
36:     cout << "The effect of using stable_partition():" << endl;
37:     stable_partition (vecCopy.begin (), vecCopy.end (), IsEven);
38:     DisplayContents(vecCopy);
39:
40:     return 0;
41: }
```

Результат

```
The initial contents:
0 1 2 3 4 5 6 7 8 9 | Number of elements: 10
The effect of using partition():
0 8 2 6 4 5 3 7 1 9 | Number of elements: 10
The effect of using stable_partition():
0 2 4 6 8 1 3 5 7 9 | Number of elements: 10
```

Анализ

Код делит диапазон целых чисел, содержащийся в векторе `vecIntegers`, на четные и нечетные значения. Сначала разделение осуществляется с использованием функции `std::partition()`, как показано в строке 33, а затем с использованием функции `stable_partition()` в строке 37. Для сравнения пример диапазона `vecIntegers` 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 выводе. Алгоритм `stable_partition()` обеспечивает относительный порядок элементов в каждом разделе. Обратите внимание, что поддержка этого порядка сказывается на производительности, которая может быть как незначительной, как в данном случае, так и существенной, в зависимости от типа содержавшихся 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 iMinInsertPos = lower_bound ( listNames.begin(), listNames.end()
                                   , "Brad Pitt" );
// альтернативно:
auto iMaxInsertPos = upper_bound ( listNames.begin(), listNames.end()
                                  , "Brad Pitt" );
```

Следовательно, функции `lower_bound()` и `upper_bound()` возвращают итераторы, указывающие на минимальную и максимальную позиции в отсортированном диапазоне, где может быть вставлен элемент 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& Input)
 8: {
 9: for ( auto iElement = Input.cbegin() // auto, cbegin: C++11
10:         ; iElement != Input.cend()   // cend(): C++11
11:         ; ++ iElement)
12:     cout << *iElement << endl;
13: }
14: int main ()
15: {
16:     list<string> listNames;
17:
18:     // Вставить примеры значений
19:     listNames.push_back ("John Doe");
20:     listNames.push_back ("Brad Pitt");
21:     listNames.push_back ("Jack Nicholson");
22:     listNames.push_back ("Sean Penn");
23:     listNames.push_back ("Anna Hoover");
24:
25:     cout << "The sorted contents of the list are: " << endl;
26:     listNames.sort ();
```

```
27:     DisplayContents(listNames);
28:
29:     cout <<
            "The lowest index where \"Brad Pitt\" can be inserted is: ";
30:     auto iMinInsertPos = lower_bound ( listNames.begin (),
                                           listNames.end ()
31:                                        , "Brad Pitt" );
32:     cout << distance (listNames.begin (), iMinInsertPos) << endl;
33:
34:     cout <<
            "The highest index where \"Brad Pitt\" can be inserted is: ";
35:     auto iMaxInsertPos = upper_bound ( listNames.begin (),
                                           listNames.end (),
36:                                        "Brad Pitt" );
37:     cout << distance (listNames.begin (), iMaxInsertPos) << endl;
38:
39:     cout << endl;
40:
41:     cout << "List after inserting Brad Pitt in sorted order: "
            << endl;
42:     listNames.insert (iMinInsertPos, "Brad Pitt");
43:
44:     DisplayContents(listNames);
45:     return 0;
46: }
```

Результат

```
The sorted contents of the list are:
Anna Hoover
Brad Pitt
Jack Nicholson
John Doe
Sean Penn
The lowest index where "Brad Pitt" can be inserted is: 1
The highest index where "Brad Pitt" can be inserted is: 2

List after inserting Brad Pitt in sorted order:
Anna Hoover
Brad Pitt
Brad Pitt
Jack Nicholson
John Doe
Sean Penn
```

Анализ

Элемент может быть вставлен в отсортированную коллекцию в двух потенциальных позициях: ближе к началу коллекции (итератор, возвращенной функцией `lower_bound()`) и ближе к концу коллекции (итератор, возвращенный функцией `upper_bound()`). В случае листинга 23.12, где в отсортированную коллекцию вставляется строка `"Brad Pitt"`, уже существующая в ней (вставлена в строке 20), верхняя и нижняя границы различаются

(в противном случае они бы совпадали). Применение этих функций представлено в строках 30 и 35 соответственно. Как демонстрирует вывод, при использовании для вставки строки в список итератора, возвращенного функцией `lower_bound()` (строка 42), список сохраняет отсортированное состояние. Таким образом, эти алгоритмы позволяют осуществить вставку в коллекцию, не нарушая порядок отсортированного содержимого. Итератор, возвращенный функцией `upper_bound()`, также сработал бы прекрасно.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Используйте метод `erase()` класса контейнера после использования алгоритмов `remove()`, `remove_if()` и `unique()` для изменения размера контейнера	**Не забывайте** сортировать содержимое контейнера при помощи метода `sort()`, прежде чем вызывать метод `unique()` для удаления повторяющихся смежных значений. Функция `sort()` гарантирует, что все элементы с совпадающими значениями расположатся рядом друг с другом, делая эффективной работу функции `unique()`
Проверяйте на допустимость итератор, возвращенный функциями `find()`, `find_if()`, `search()` и `search_n()`, прежде чем использовать его для сравнения с результатом метода `end()` контейнера	**Не забывайте**, что функцию `binary_search()` следует вызывать только для отсортированного контейнера
Предпочитайте использовать функцию `stable_partition()`, а не функцию `partition()` и функцию `stable_sort()`, а не функцию `sort()` только тогда, когда относительный порядок отсортированных элементов важен, поскольку версии `stable_*` могут снизить производительность приложения	

Резюме

На сегодняшнем занятии рассматривался один из самых важных аспектов библиотеки STL: алгоритмы. Вы изучили различные типы алгоритмов, а примеры помогли вам лучше понять применение алгоритмов.

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, то поступать так не нужно. Содержимое ассоциативного контейнера следует обрабатывать как константное. 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 в эффективности работы контейнера. Поэтому изменяющие алгоритмы, такие как `std::transform()`, не должны использоваться с наборами библиотеки 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 `copy_backward()` библиотеки STL 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()` в списке?**

  Алгоритм `std::sort()` применяется в списке таким же образом, как и в любом другом последовательном контейнере. Однако у списка есть специальное свойство: существующие итераторы остаются допустимыми при операциях со списком, а 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 такую функцию, как `lower_bound()` или `upper_bound()`, при вставке в отсортированный диапазон?**

  Эти функции предоставляют первую и последнюю позиции в отсортированной коллекции соответственно, куда 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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "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 заданному условию. Вы использовали бы функцию `std::remove_if()` или `list::remove_if()`?
2. У вас есть список типа `ContactItem`. Как функция `list::sort()` отсортирует элементы списка этого типа в отсутствии явно определенного бинарного предиката?
3. Как часто алгоритм `generate()` библиотеки STL вызывает функцию `generator()`?
4. Чем функция `std::transform()` отличается от функции `std::for_each()`?

Упражнения

1. Напишите бинарный предикат, получающий строки как входные аргументы и возвращающий значение на основании независящего от регистра сравнения.
2. Приведите пример того, как алгоритмы STL, такие как `std::copy()`, используют итераторы для выполнения своих задач, не нуждаясь в знании характера коллекции назначения при копировании двух последовательностей, содержащихся в двух несходных контейнерах.
3. Напишете приложение, которое записывает характеристики звезд, видимых на горизонте в порядке их восхождения. В астрономии размер звезды, а также информация об их относительной высоте и порядке важна. Если вы сортируете эту коллекцию звезд на основании их размеров, то использовали бы вы функцию `std::sort()` или `std::stable_sort()`?

ЗАНЯТИЕ 24

Адаптивные контейнеры: стек и очередь

Стандартная библиотека шаблонов (STL) предоставляет контейнеры, 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).

На сегодняшнем занятии.

- Поведенческие характеристики стеков и очередей.
- Использование контейнера `stack` библиотеки STL.
- Использование контейнера `queue` библиотеки STL.
- Использование контейнера `priority_queue` библиотеки STL.

Поведенческие характеристики стеков и очередей

Стеки и очереди очень похожи на массивы и списки, но с ограничениями на вставку элементов, доступ к ним и удаление. 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.

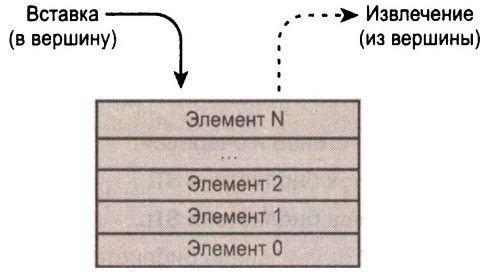


РИС. 24.1. Работа со стеком

Такое поведение стопки тарелок моделирует обобщенный контейнер `std::stack` библиотеки STL.

СОВЕТ

Чтобы использовать класс `std::stack`, включите его заголовок

```
#include <stack>
```

Очереди

Очередь (queue) — это структура в памяти, действующая по принципу *первым вошел, первым вышел* (First-In-First-Out — FIFO), элементы которой могут быть вставлены в основание контейнера и извлечены из вершины. Очередь можно представить как очередь ожидающих людей, в которой кто первым встал в очередь, тот раньше всех ее покинет. Этот способ организации элементов, подразумевающий "добавление в конец и извлечение из начала", представлен на рис. 24.2.

Такое поведение очереди моделирует обобщенный контейнер `std::queue` библиотеки STL.

РИС. 24.2. Работа с очередью

СОВЕТ

Чтобы использовать класс `std::queue`, включите его заголовок

```
#include <queue>
```

Использование класса `stack` библиотеки STL

Стек библиотеки STL является шаблоном класса `stack`, для использования которого необходимо включить в код заголовок `<stack>`. Это обобщенный класс, обеспечивающий вставку и извлечение элементов в его вершину и не разрешающий доступ или просмотр элементов в середине. В некотором смысле поведение класса `std::stack` очень похоже на стопку тарелок.

Создание экземпляра стека

Некоторые реализации библиотеки STL определяют шаблон класса `stack` так:

```
template <
    class typeЭлемента,
    class Контейнер = deque<Тип>
> class stack;
```

Параметр *typeЭлемента* задает тип объектов, 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 будет выглядеть так:

```
std::stack <int> stackInts;
```

Если необходимо создать стек объектов какого-нибудь иного типа, например класса `Tuna`, то можно использовать следующий синтаксис:

```
std::stack <Tuna> stackTunas;
```

Для создания стека на базе другого контейнера используйте следующий синтаксис:

```
std::stack <double, vector <double> > stackDoublesInVector;
```

Листинг 24.1 демонстрирует различные способы создания экземпляра стека.

ЛИСТИНГ 24.1. Создание экземпляра стека библиотеки STL

```
0: #include <stack>
1: #include <vector>
2:
3: int main ()
4: {
```

```
 5:     using namespace std;
 6:
 7:     // Стек целых чисел
 8:     stack <int> stackInts;
 9:
10:     // Стек чисел типа double
11:     stack <double> stackDoubles;
12:
13:     // Стек чисел типа double, содержащихся в векторе
14:     stack <double, vector <double> > stackDoublesInVector;
15:
16:     // Инициализация стека копией другого
17:     stack <int> stackIntsCopy(stackInts);
18:
19:     return 0;
20: }
```

Анализ

Пример ничего не выводит, он демонстрирует создание экземпляра шаблона стека библиотеки STL. В строках 8 и 11 создаются два экземпляра объекта класса `stack` для хранения элементов типа `int` и `double` соответственно. В строке 14 также создается экземпляр стека, но с использованием второго параметра шаблона — класса коллекции (`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 способ, которым элементы могут быть вставлены или извлечены для обеспечения поведения, которое ожидается от механизма, подобного стеку. 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
`push`	Вставляет элемент в вершину стека
	`stackInts.push (25);`
`pop`	Извлекает элемент из вершины стека
	`stackInts.pop ();`
`empty`	Проверяет, не пуст ли стек; возвращает значение типа `bool`
	`if (stackInts.empty ())` `  DoSomething ();`
`size`	Возвращает количество элементов в стеке
	`size_t nNumElements = stackInts.size ();`
`top`	Возвращает ссылку на верхний элемент в стеке
	`cout << "Element at the top = " << stackInts.top ();`

Как свидетельствует таблица, к открытым функциям-членам стека относятся только те методы, которые обеспечивают вставку и извлечение в позициях, которые согласуются с поведением стека. Таким образом, даже при том, что базовый контейнер мог бы быть двухсторонней очередью, вектором или списком, функциональные возможности этого контейнера не будут доступны, чтобы имитировать поведенческие характеристики стека.

Вставка и извлечение из вершины с использованием методов push() и pop()

Для вставки элементов используется метод `stack<T>::push()`:

```
stackInts.push (25); // вставить 25 в вершину стека
```

По определению стек разрешает доступ к элементу в вершине при помощи метода `top()`:

```
cout << stackInts.top() << endl;
```

Если необходимо извлечь верхний элемент, то можете использовать функцию `pop()`:

```
stackInts.pop (); // pop: имзвлекает верхний элемент
```

В листинге 24.2 показаны вставка элементов в стек с использованием метода `push()` и их извлечение с использованием 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> stackInts;
 7:
 8:     // push: вставить значения в вершину стека
 9:     cout << "Pushing {25, 10, -1, 5} on stack in that order:"
           << endl;
10:     stackInts.push (25);
11:     stackInts.push (10);
12:     stackInts.push (-1);
13:     stackInts.push (5);
14:
15:     cout << "Stack contains " << stackInts.size () << " elements"
           << endl;
16:     while (stackInts.size () != 0)
17:     {
18:         cout << "Popping topmost element: " << stackInts.top()
               << endl;
19:         stackInts.pop (); // pop: извлечь верхний элемент
20:     }
21:
22:     if (stackInts.empty ()) // true: благодаря предыдущему pop()
23:         cout << "Popping all elements empties stack!" << endl;
24:
25:     return 0;
26: }
```

Результат

```
Pushing {25, 10, -1, 5} on stack in that order:
Stack contains 4 elements
Popping topmost element: 5
Popping topmost element: -1
Popping topmost element: 10
Popping topmost element: 25
Popping all elements empties stack!
```

Анализ

Сначала в стек целых чисел `stackInts` вставляются значения с использованием метода `stack::push()`, как показано в строках 9–13, а затем они извлекаются с использованием метода `stack::pop()`. Стек разрешает доступ только к верхнему элементу, к нему 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 в выводе, те элементы, которые вставлены последними, извлекаются первыми, демонстрируя типичное 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 библиотеки STL, используемых как базовые контейнеры классом `stack`, не доступны как открытые функции-члены в классе `stack`. То же относится и к итераторам, позволяющим просмотреть все элементы, включая не расположенные в вершине контейнера. Все, что предоставляет стек, — это верхний элемент, ничего иного.

Использование класса `queue` библиотеки STL

Для применения шаблона класса `queue` библиотеки STL требуется включить его заголовок <queue>. Это обобщенный класс, обеспечивающий вставку элементов только в конец и 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 так:

```
template <
    class typeЭлемента,
    class Контейнер = deque<Тип>
> class queue;
```

Параметр *типЭлемента* 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 std::list, vector и deque. По умолчанию используется класс deque.

Самый простой экземпляр очереди целых чисел создается следующим образом:

```
std::queue <int> qIntegers;
```

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> > qDoublesInList;
```

Точно так же как стек, очередь может быть создана как копия другой очереди:

```
std::queue<int> qCopy(qIntegers);
```

В листинге 24.3 показаны различные способы создания экземпляра класса std::queue.

ЛИСТИНГ 24.3. Создание экземпляра очереди библиотеки STL

```
 0: #include <queue>
 1: #include <list>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     // Очередь целых чисел
 8:     queue <int> qIntegers;
 9:
10:     // Очередь чисел типа double
11:     queue <double> qDoubles;
12:
13:     // Очередь чисел типа double, содержащихся в списке
14:     queue <double, list <double> > qDoublesInList;
15:
16:     // Очередь создана как копия другой
17:     queue<int> qCopy(qIntegers);
18:
19:     return 0;
20: }
```

Анализ

Пример демонстрирует, как может быть создан экземпляр обобщенного класса queue библиотеки STL, чтобы получить очередь целых чисел (строка 8) и чисел типа double (строка 11). При создании экземпляра очереди qDoublesInList в строке 14 было явно определено во втором параметре шаблона, что базовым контейнером, адаптированным очередью для ее внутренней организации, будет класс std::list. При отсутствии второго параметра шаблона, как 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 библиотеки 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 qIntegers в листинге 24.3.

ТАБЛИЦА 24.2. Функции-члены класса std::queue

Функция	Описание
push	Вставляет элемент в конец очереди, т.е. в ее последнюю позицию `qIntegers.push (10);`
pop	Извлекает элемент из начала очереди, т.е. из ее первой позиции `qIntegers.pop ();`
front	Возвращает ссылку на элемент в начале очереди `cout << "Element at front: " << qIntegers.front ();`
back	Возвращает ссылку на элемент в конце очереди, т.е. на последний вставленный элемент `cout << "Element at back: " << qIntegers.back ();`

empty	Проверяет, не пуста ли очередь; возвращает значение типа bool
	`if (qIntegers.empty ())`
	`  cout << "The queue is empty!";`

size	Возвращает количество элементов в очереди `size_t nNumElements = qIntegers.size ();`

Класс queue библиотеки STL не предоставляет такие функции, как begin() и end(), хотя они доступны в большинстве контейнеров библиотеки STL, включая базовые классы deque, vector и list, лежащие в основе класса очереди. Это сделано намеренно, 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():

```
qIntegers.push (5); // элемент вставляется в конец
```

Извлечение, напротив, осуществляется с начала при помощи метода pop():

```
qIntegers.pop (); // извлечь элемент из начала
```

В отличие от стека, у очереди элементы доступны для просмотра с обоих концов контейнера при помощи методов front() и back():

```
cout << "Element at front: " << qIntegers.front() << endl;
cout << "Element at back: " << qIntegers.back() << endl;
```

Вставка, извлечение и просмотр представлены 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> qIntegers;
 7:
 8:     cout << "Inserting {10, 5, -1, 20} into queue" << endl;
 9:     qIntegers.push (10);
10:     qIntegers.push (5); // элемент вставляется в конец
11:     qIntegers.push (-1);
12:     qIntegers.push (20);
13:
14:     cout << "Queue contains " << qIntegers.size () << " elements"
          << endl;
15:     cout << "Element at front: " << qIntegers.front() << endl;
16:     cout << "Element at back: " << qIntegers.back () << endl;
17:
18:     while (qIntegers.size () != 0)
19:     {
20:         cout << "Deleting element: " << qIntegers.front () << endl;
21:         qIntegers.pop (); // извлечь элемент из начала
22:     }
23:
24:     if (qIntegers.empty ())  // true, поскольку все элементы
                                 // были извлечены
25:         cout << "The queue is now empty!" << endl;
26:
27:     return 0;
28: }
```

Результат

```
Inserting {10, 5, -1, 20} into queue
Queue contains 4 elements
Element at front: 10
Element at back: 20
Deleting element: 10
Deleting element: 5
Deleting element: -1
Deleting element: 20
```

Анализ

В строках 9–12 элементы были добавлены в конец очереди `qIntegers` 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 priority_queue библиотеки STL

Для применения шаблон класса priority_queue библиотеки STL требуется включить его заголовок <queue>. *Приоритетная очередь* (priority queue) отличается от очереди тем, что элемент с наивысшим значением (или значением, считающимся наивысшим согласно бинарному предикату) доступен в начале очереди, а работа с очередями ограничивается их началом.

Создание экземпляра приоритетной очереди

Шаблон класса std::priority_queue определен так:

```
template <
    class типЭлемента,
    class Контейнер = vector<Тип>,
    class Сравнение = less<typename Контейнер::value_type>
> class priority_queue
```

Параметр *типЭлемента* задает тип объектов, которые будут храниться в приоритетной очереди. Второй параметр шаблона, *Контейнер*, — это класс контейнера, используемого классом std::priority_queue для хранения данных, тогда как третий параметр позволяет программисту определить бинарный предикат для выявления располагающегося вверху элемента. Если бинарный предикат не определен, класс priority_queue использует заданный по умолчанию предикат std::less<>, который сравнивает объекты, используя оператор operator<.

Самый простой экземпляр приоритетной очереди целых чисел создается следующим образом:

```
std::priority_queue <int> pqIntegers;
```

Если необходимо создать приоритетную очередь, содержащую элементы типа double в контейнере std::deque, используйте следующий код:

```
priority_queue <int, deque <int>, greater <int> > pqIntegers_Inverse;
```

Подобно стеку, экземпляр очереди может быть создан как копия другой очереди:

```
std::priority_queue <int> pqCopy(pqIntegers);
```

Создание экземпляра класса priority_queue представлено в листинге 24.5.

ЛИСТИНГ 24.5. Создание экземпляра класса priority_queue

```
0: #include <queue>
1:
2: int main ()
3: {
4:     using namespace std;
5:
6:     // Приоритетная очередь целых чисел, отсортированных с
       // использованием предиката std::less<> (по умолчанию)
7:     priority_queue <int> pqIntegers;
```

```
 8:
 9:     // Приоритетная очередь чисел типа double
10:     priority_queue <double> pqDoubles;
11:
12:     // Приоритетная очередь целых чисел, отсортированных с
        // использованием предиката std::greater<>
13:     priority_queue <int, deque <int>,
                        greater <int> > pqIntegers_Inverse;
14:
15:     // Приоритетная очередь создана как копия другой
16:     priority_queue <int> pqCopy(pqIntegers);
17:
18:     return 0;
19: }
```

Анализ

Строки 7 и 10 демонстрируют создание двух экземпляров класса `priority_queue` для объектов типа `int` и `double` соответственно. В результате отсутствия всех остальных параметров шаблона по умолчанию используется класс `std::vector` как внутренний контейнер данных и `std::less<>` как критерий для сравнения. Поэтому приоритетом данной очереди будет целое число с наивысшим значением, оно и будет доступно в начале приоритетной очереди. Для экземпляра `pqIntegers_Inverse`, однако, в качестве внутреннего контейнера заданы двухсторонняя очередь и предикат `std::greater<>`. Этот предикат создает очередь, где в начале доступно наименьшее число.

Результат использования предиката `std::greater<T>` объясняется в листинге 24.7 далее на этом занятии.

Функции-члены класса `priority_queue`

Функции-члены `front()` и `back()`, доступные в классе `queue`, не доступны в классе `priority_queue`. Функции-члены класса `priority_queue` приведены в табл. 24.3.

ТАБЛИЦА 24.3. Функции-члены класса `std::priority_queue`

Функция	Описание
`push`	Вставляет элемент в приоритетную очередь `pqIntegers.push (10);`
`pop`	Извлекает элемент из начала очереди, т.е. наибольший элемент `pqIntegers.pop ();`

`top`	Возвращает ссылку на наибольший элемент в очереди, занимающий также верхнюю позицию
	`cout << "The largest element in the priority queue is: "`
	`<< pqIntegers.top ();`

`empty`	Проверяет, не пуста ли приоритетная очередь; возвращает значение типа `bool`
	`if (pqIntegers.empty ())`
	`cout << "The queue is empty!";`

`size`	Возвращает количество элементов в приоритетной очереди `size_t nNumElements = pqIntegers.size ();`

Как свидетельствует таблица, к членам приоритетной очереди можно обратиться, только используя метод top(), возвращающий элемент с самым высоким значением, согласно заданному пользователем предикату или предикату std::less<> при его отсутствии.

Вставка в конец и извлечение из начала приоритетной очереди с использованием методов push() и pop()

Для вставки элементов в приоритетную очередь используется метод push():

```
pqIntegers.push (5); // элементы помещаются в отсортированном порядке
```

Извлечение, напротив, осуществляется с начала при помощи метода pop():

```
pqIntegers.pop (); // извлечь элемент из начала
```

Использование методов класса priority_queue представлено в листинге 24.6.

ЛИСТИНГ 24.6. Работа с приоритетной очередью при помощи 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> pqIntegers;
 8:     cout << "Inserting {10, 5, -1, 20} into the priority_queue"
           << endl;
 9:     pqIntegers.push (10);
10:     pqIntegers.push (5); // помещаются в отсортированном порядке
11:     pqIntegers.push (-1);
12:     pqIntegers.push (20);
13:
14:     cout << "Deleting the " << pqIntegers.size () << " elements"
           << endl;
15:     while (!pqIntegers.empty ())
16:     {
17:         cout << "Deleting topmost element: " << pqIntegers.top ()
               << endl;
18:         pqIntegers.pop ();
19:     }
20:
21:     return 0;
22: }
```

Результат

```
Inserting {10, 5, -1, 20} into the priority_queue
Deleting the 4 elements
Deleting topmost element: 20
Deleting topmost element: 10
Deleting topmost element: 5
Deleting topmost element: -1
```

Анализ

Сначала код листинга 24.6 вставляет примеры целых чисел в приоритетную очередь (строки 9–12), а затем извлекает элементы из его вершины, используя метод `pop()`, как показано в строке 18. Вывод демонстрирует, что вверху очереди доступен элемент с самым большим значением. Поэтому применение метода `priority_queue::pop()` фактически удаляет элемент, значение которого считается самым большим среди всех элементов в контейнере. Его же предоставляет метод `top()` как значение, находящееся на самом верху (строка 17). Если предикат установки приоритетов не задан, очередь автоматически сортирует элементы в порядке убыванию (самое высокое значение вверху).

Следующий пример в листинге 24.7 демонстрирует создание экземпляра класса `priority_queue` с предикатом `std::greater<int>`. При этом предикате у очереди приоритет имеет наименьшее число, как элемент с самым высоким значением, которое и будет доступно при помощи метода `front()`.

ЛИСТИНГ 24.7. Создание экземпляра приоритетной очереди, хранящей наименьшее значение вверху

```
 0: #include <queue>
 1: #include <iostream>
 2:
 3: int main ()
 4: {
 5:     using namespace std;
 6:
 7:     // Определение объекта priority_queue с предикатом greater<int>
 8:     priority_queue <int, vector <int>, greater <int> > pqIntegers;
 9:
10:     cout << "Inserting {10, 5, -1, 20} into the priority queue"
            << endl;
11:     pqIntegers.push (10);
12:     pqIntegers.push (5);
13:     pqIntegers.push (-1);
14:     pqIntegers.push (20);
15:
16:     cout << "Deleting " << pqIntegers.size () << " elements" << endl;
17:     while (!pqIntegers.empty ())
18:     {
19:         cout << "Deleting topmost element " << pqIntegers.top ()
                << endl;
20:         pqIntegers.pop ();
21:     }
22:
23:     return 0;
24: }
```

Результат

```
Inserting {10, 5, -1, 20} into the priority queue
Deleting 4 elements
Deleting topmost element -1
Deleting topmost element 5
Deleting topmost element 10
Deleting topmost element 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. Но все же вывод демонстрирует, что эти две очереди ведут себя по-разному. Эта приоритетная очередь использует для сравнения элементов предикат `greater<int>`, как показано в строке 8. В результате целое число с самым низким значением считается больше других, а потому помещается в высшую позицию. Так, функция `top()`, использованная в строке 19, всегда отображала наименьшее целое число в приоритетной очереди, перед его извлечением с использованием метода `pop()` в строке 20.

Таким образом, когда элементы извлекаются из данной приоритетной очереди, целые числа располагаются в порядке увеличения значений.

Резюме

На этом занятии рассматривалось применение трех основных адаптивных контейнеров библиотеки STL: стека, очереди и приоритетной очереди. Они адаптируют последовательные контейнеры, хранящие их данные, к собственным требованиям, предоставляя лишь часть их функций-членов для моделирования поведенческих характеристик, которые делают стеки и очереди настолько уникальными.

Вопросы и ответы

- **Может ли быть изменен элемент в середине стека?**
  Нет, это противоречило бы поведению стека.
- **Могу ли я перебрать все элементы очереди?**
  Очередь не предоставляет итераторы на элементы очереди, обратиться можно только к ее концу.
- **Могут ли алгоритмы STL работать с адаптивными контейнерами?**
  Алгоритмы STL используют итераторы. Поскольку ни класс `stack`, ни класс `queue` не предоставляют итераторы, которые отмечают концы диапазонов, 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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "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 элемент с самым большим значением извлекался последним?
2. Есть приоритетная очередь объектов класса `Coins`. Какой оператор-член этого класса необходимо определить для класса приоритетной очереди, чтобы в верхней позиции оказались монеты с наибольшим номиналом?
3. Имеется стек элементов класса `CCoins`, содержащий шесть объектов. Можно ли обратиться к первому вставленному элементу или извлечь его?

Упражнения

1. Очередь людей (класс `CPerson`) выстроилась к почтовому отделению. Класс `CPerson` имеет атрибуты, содержащие возраст и пол. Он определяется так:

```
class CPerson
{
public:
    int Age;
    bool IsFemale;
};
```

   Напишите бинарный предикат для контейнера `priority_queue`, который позволит сотруднику обслужить сначала стариков и женщин.
2. Напишите программу, которая, используя класс `stack`, меняет порядок введенных пользователем строк 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 помочь в организации и манипулировании битовой информацией.

На сегодняшнем занятии.

- Класс `bitset`.
- Класс `vector<bool>`.

Класс bitset

Класс std::bitset (набор битов) библиотеки STL предназначен для обработки информации в виде битов и битовых флагов. Класс std::bitset не является контейнерным классом библиотеки STL, поскольку он 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, включите его заголовок

```
#include <bitset>
```

Создание экземпляра класса std::bitset

Этому шаблону класса требуется один параметр, содержащий количество битов, которым должен манипулировать экземпляр класса:

```
bitset <4> fourBits; // 4 бита, инициализированных 0000
```

Набор битов можно также инициализировать небольшой последовательностью в виде строкового литерала:

```
bitset <5> fiveBits ("10101"); // 5 битов 10101
```

Копирование из одного набора битов в другой при создании его экземпляра очень просто:

```
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:     // создание объекта класса bitset
 9:     bitset <4> fourBits; // 4 бита, инициализированных 0000
10:     cout << "Initial contents of fourBits: " << fourBits << endl;
11:
12:     bitset <5> fiveBits ("10101"); // 5 битов 10101
13:     cout << "Initial contents of fiveBits: " << fiveBits << endl;
14:
15:     bitset <8> eightbits (255); // 8 битов, инициализированных
                                    // значением 255 типа long int
```

```
16:     cout << "Initial contents of eightBits: " << eightbits << endl;
17:
18:     // создание экземпляра набора битов как копии другого
19:     bitset <8> eightBitsCopy(eightbits);
20:
21:     return 0;
22: }
```

Результат

```
Initial contents of fourBits: 0000
Initial contents of fiveBits: 10101
Initial contents of eightBits: 11111111
```

Анализ

Пример демонстрирует четыре способа создания объекта класса `bitset`. Стандартный конструктор инициализирует битовую последовательность нулями, как показано в строке 9. Строка в стиле C, содержащая строковое представление битовой последовательности, используется для инициализации в строке 12. Тип `unsigned long`, который содержит десятичное значение двоичной последовательности, используется в строке 15, а конструктор копий используется в строке 19. Обратите внимание на то, что в каждом из этих случаев вы вынуждены были указать как параметр шаблона количество битов, которые будет содержать набор битов. Это число фиксируется во время компиляции; оно не динамическое. Вы не можете вставить в набор больше битов, чем определено в коде, в отличие от вектора.

Использование класса `std::bitset` и его членов

Класс `bitset` предоставляет функции-члены, позволяющие осуществить вставку, установку и сброс содержимого, чтение битов и их запись в поток. Он предоставляет также операторы, позволяющие отображать содержимое набора битов, а также выполнять побитовые логические операции.

Вспомогательные операторы, предоставляемые классом `std::bitset`

Операторы рассматривались на занятии 12, "Типы операторов и их перегрузка", где 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, существенно облегчающих его использование. Примеры, объясняющие использование операторов, подразумевают набор битов `fourBits` из листинга 25.1.

ТАБЛИЦА 25.1. Операторы, поддерживаемые классом `std::bitset`

Оператор	Описание
`operator<<`	Передает текстовое представление битовой последовательности в поток вывода `Cout << fourBits;`
`operator>>`	Передает строку в объект класса `bitset` `"0101" >> fourBits;`
`operator&`	Выполняет побитовую операцию AND `bitset <4> result (fourBits1 & fourBits2);`
`operator|`	Выполняет побитовую операцию OR `bitwise <4> result (fourBits1 | fourBits2);`
`operator^`	Выполняет побитовую операцию XOR `bitwise <4> result (fourBits1 ^ fourBits2);`
`operator~`	Выполняет побитовую операцию NOT `bitwise <4> result (~fourBits1);`
`operator>>=`	Выполняет бинарный оператор сдвига вправо `fourBits >>= (2); // Сдвиг на два бита вправо`
`operator<<=`	Выполняет бинарный оператор сдвига влево `fourBits <<= (2); // Сдвиг на два бита влево`

`operator[N]`	Возвращает ссылку на бит номер *N* в последовательности
	`fourBits [2] = 0; // установить третий бит в 0`
	`bool bNum = fourBits [2]; // читать третий бит`

В дополнение к ним класс `std::bitset` предоставляет такие операторы, как `|=`, `&=`, `^=` и `~=`, позволяющие выполнять бинарные операции.

Методы класса `std::bitset`

Биты могут находиться в двух состояниях: они либо установлены (1), либо сброшены (0). Для манипулирования содержимым набора битов можно воспользоваться функциями-членами класса `bitset` (табл. 25.2), позволяющими работать с некоторыми или со всеми битами в наборе.

ТАБЛИЦА 25.2. Методы класса `std::bitset`

Оператор	Описание
`Set`	Устанавливает все биты последовательности в 1 `fourBits.set (); // теперь набор содержит: '1111'`
`set(N, val=1)`	Устанавливает в бит номер *N* значение `val` (по умолчанию 1) `fourBits.set (2, 0); // установить третий бит в 0`
`Reset`	Сбрасывает все биты последовательности в 0 `fourBits.reset (); // теперь набор содержит: '0000'`
`reset(N)`	Сбрасывает бит номер *N* `fourBits.reset (2); // теперь третий бит 0`

Окончание табл. 25.2

Оператор	Описание
Flip	Инвертирует все биты последовательности
	`fourBits.flip (); // 0101 измененилось на 1010`
Size	Возвращает количество битов последовательности
	`size_t NumBits = fourBits.size (); // возвращает 4`
Count	Возвращает количество установленных битов
	`size_t NumBitsSet = fourBits.count ();` `size_t NumBitsReset = fourBits.size () - fourBits.count ();`

Применение этих методов и операторов демонстрирует листинг 25.2.

ЛИСТИНГ 25.2. Логические операции с набором 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 << "Enter a 8-bit sequence: ";
 9:
10:     cin >> inputBits; // сохранить пользовательский
                          // ввод в наборе битов
11:
12:     cout << "Number of 1s you supplied: " << inputBits.count ()
             << endl;
13:     cout << "Number of 0s you supplied: ";
14:     cout << inputBits.size () - inputBits.count () << endl;
15:
16:     bitset <8> inputFlipped (inputBits); // копирование
17:     inputFlipped.flip ();                // инверсия битов
18:
19:     cout << "Flipped version is: " << inputFlipped << endl;
20:
21:     cout << "Result of AND, OR and XOR between the two:" << endl;
22:     cout << inputBits << " & " << inputFlipped << " = ";
23:     cout << (inputBits & inputFlipped) << endl; // побитовое AND
24:
25:     cout << inputBits << " | " << inputFlipped << " = ";
26:     cout << (inputBits | inputFlipped) << endl; // побитовое OR
27:
28:     cout << inputBits << " ^ " << inputFlipped << " = ";
29:     cout << (inputBits ^ inputFlipped) << endl; // побитовое XOR
30:
31:     return 0;
32: }
```

Результат

```
Enter a 8-bit sequence: 10110101
Number of 1s you supplied: 5
Number of 0s you supplied: 3
Flipped version is: 01001010
Result of AND, OR and XOR between the two:
10110101 & 01001010 = 00000000
10110101 | 01001010 = 11111111
10110101 ^ 01001010 = 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 записывать последовательности битов на экран и читать небольшие последовательности, введенные пользователем в командной строке. Набор битов `inputBits` получает введенную пользователем последовательность (строка 10). Используемый в строке 12 метод `count()` сообщает количество единиц в последовательности, а количество нулей вычисляется как разница между результатами выполнения метода `size()`, возвращающего количество битов в наборе битов, и метода `count()`, как показано в строке 14. Набор битов `inputFlipped`, изначально копия набора битов `inputBits`, впоследствии инвертируется с использованием метода `flip()` в строке 17. Теперь он содержит последовательность инвертированных битов, т.е. нули стали единицами, а единицы нулями. Остальная часть программы демонстрирует результат побитовых операций AND, OR и XOR между этими двумя наборами битов.

ПРИМЕЧАНИЕ

Одним из недостатков шаблона класса `bitset<>` библиотеки STL является его неспособность изменять свои размеры динамически. Вы можете использовать класс `bitset` только там, где количество хранимых в последовательности битов известно во время компиляции.

Библиотека STL снабжает программиста классом `vector<bool>` (называемым также `bit_vector` в некоторых реализациях библиотеки STL), 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 заботиться о количестве логических флагов во время компиляции.

СОВЕТ

Чтобы использовать класс `std::vector<bool>`, включите его заголовок:

```
#include <vector>
```

Создание экземпляра класса vector<bool>

Экземпляр класса vector<bool> создается подобно вектору:

```
vector <bool> vecBool1;
```

Например, можно создать вектор с 10 логическими значениями, для начала инициализированных значением 1 (т.е. true):

```
vector <bool> vecBool2 (10, true);
```

Вы можете также создать объект как копию другого:

```
vector <bool> vecBool2Copy (vecBool2);
```

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:     // Создать экземпляр объекта, используя стандартный конструктор
 7:     vector <bool> vecBool1;
 8:
 9:     // Инициализировать вектор из 10 элементов значением true
10:     vector <bool> vecBool2 (10, true);
11:
12:     // Создать экземпляр объекта как копию другого
13:     vector <bool> vecBool2Copy (vecBool2);
14:
15:     return 0;
16: }
```

Анализ

Этот пример демонстрирует некоторые из способов создания объекта класса vector<bool>. В строке 7 используется стандартный конструктор. 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> vecBoolFlags (3); // создать экземпляр
                                        // для 3 логических флагов
 7:     vecBoolFlags [0] = true;
 8:     vecBoolFlags [1] = true;
 9:     vecBoolFlags [2] = false;
10:
11:     vecBoolFlags.push_back (true); // вставить четвертый флаг в конец
12:
13:     cout << "The contents of the vector are: " << endl;
14:     for (size_t nIndex = 0; nIndex < vecBoolFlags.size (); ++ nIndex)
15:         cout << vecBoolFlags [nIndex] << ' ';
16:
17:     cout << endl;
18:     vecBoolFlags.flip ();
19:
20:     cout << "The contents of the vector are: " << endl;
21:     for (size_t nIndex = 0; nIndex < vecBoolFlags.size (); ++ nIndex)
22:         cout << vecBoolFlags [nIndex] << ' ';
23:
24:     cout << endl;
25:
26:     return 0;
27: }
```

Результат

```
The contents of the vector are:
1 1 0 1
The contents of the vector are:
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 наоборот. Обратите внимание на применение функции `push_back()` в строке 11. Хотя изначально вектор `vecBoolFlags` был создан для хранения трех флагов (строка 6), в строке 11 к нему удалось добавить еще один. Добавление большего количества флагов, чем было определено вначале, а также возможность выбрать их количество динамически уже после компиляции, отличает `vector<bool>` от класса `std::bitset`.

Резюме

На сегодняшнем занятии рассматривался весьма эффективный инструмент обработки битовых последовательностей и битовых флагов: класс `std::bitset`. Вы также узнали о классе `vector<bool>`, который также позволяет хранить логические флаги, количество которых, однако, не обязательно знать на момент компиляции.

Вопросы и ответы

- **В ситуации, где 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`, хранящий 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()` (полное количество хранимых битов), 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 он поддерживает итераторы.

- **Могу ли я во время компиляции задать количество элементов, которые будут храниться в объекте класса `vector<bool>`?**

  Да, либо указав 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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом сверьте полученные результаты с ответами в приложении Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Может ли набор битов расширить свой внутренний буфер для хранения переменного количества элементов?
2. Почему класс `bitset` не считается контейнерным классом библиотеки STL?
3. Использовали бы вы класс `std::vector` для хранения фиксированного количества битов, известного на момент компиляции?

Упражнения

1. Напишите пример, где набор битов содержит четыре бита. Инициализируйте его, отобразите результат и добавьте его к другому набору битов. (Предостережение: наборы битов не допускают такой синтаксис: `bitsetA = bitsetX + bitsetY`.)
2. Покажите, как бы вы инвертировали биты в наборе битов.

ЧАСТЬ V
Передовые концепции языка C++

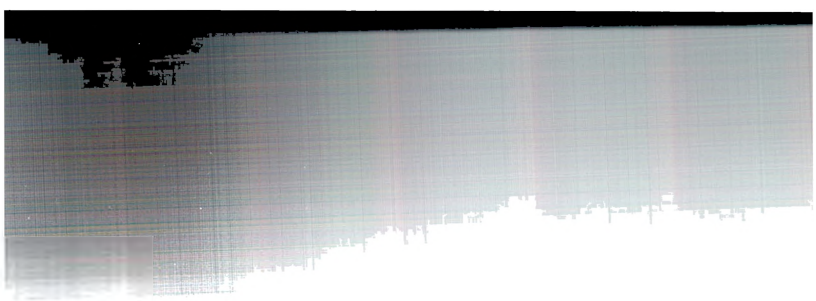
ЗАНЯТИЕ 26

Понятие интеллектуальных указателей

Программисты C++ не обязаны использовать простые ссылочные типы при управлении распределяемой памятью (или динамической памятью); они могут использовать интеллектуальные указатели.

На сегодняшнем занятии.

- Что такое интеллектуальный указатель и зачем он нужен.
- Как реализуются интеллектуальные указатели.
- Типы интеллектуальных указателей.
- Почему не стоит использовать устаревший тип `std::auto_ptr`.
- Интеллектуальный указатель `std::unique_ptr` стандартной библиотеки C++11.
- Популярные библиотеки интеллектуальных указателей.



ЗАНЯТИЕ 26
Понятие интеллектуальных указателей

Программисты C++ не обязаны использовать простые ссылочные типы при управлении распределяемой памятью (или динамической памятью); они могут использовать интеллектуальные указатели.

На сегодняшнем занятии.

- Что такое интеллектуальный указатель и зачем он нужен.
- Как реализуются интеллектуальные указатели.
- Типы интеллектуальных указателей.
- Почему не стоит использовать устаревший тип `std::auto_ptr`.
- Интеллектуальный указатель `std::unique_ptr` стандартной библиотеки C++11.
- Популярные библиотеки интеллектуальных указателей.

Что такое интеллектуальный указатель

Проще 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 распределенных данных и облегчает контроль жизненного цикла объекта.

Проблема с использованием обычных (простых) указателей

В отличие от других современных языков программирования, C++ предоставляет разработчику полную свободу в распределении, освобождении и управлении памятью. К сожалению, эта свобода — палка о двух концах. С одной стороны, она придает языку C++ его мощь, но с другой — позволяет программисту создавать такие связанные с памятью проблемы, как утечка памяти, когда динамически распределенные объекты не освобождаются правильно.

Например:

```
CData *pData = mObject.GetData ();
/*
    Вопрос: был ли объект, на который указывает указатель pData,
    динамически распределен 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, будет ли объект, на который указывает указатель `pData`,

- создан в распределяемой памяти, а поэтому в конечном счете должен быть освобожден;
- несет ли вызывающая сторона ответственность за его освобождение;
- будет 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 распределенными данными и указателями.

Чем помогут интеллектуальные указатели

С учетом описанной выше проблемы использования обычного указателя и обычных способов управления памятью следует заметить, что программист C++ не обязан использовать именно их, когда дело доходит до управления данными в распределяемой (динамической) памяти. Программист может выбрать более передовой способ резервирования и управления динамическими данными при помощи интеллектуальных указателей:

```
smart_pointer<CData> spData = mObject.GetData ();

// Использовать интеллектуальный указатель как обычный!
```

```
spData->Display ();
(*spData).Display ();

// Можно не заботиться об освобождении
// (деструктор интеллектуального указателя сделает это самостоятельно)
```

Таким образом, интеллектуальные указатели ведут себя как обычные указатели (давайте называть их теперь *простыми указателями* (raw pointer)), но предоставляют полезные средства при помощи своих *перегруженных операторов* (overloaded operator) и *деструкторов* (destructor), гарантирующих своевременное освобождение динамически распределенных данных.

Как реализованы интеллектуальные указатели

В настоящий момент этот вопрос может быть упрощен до "Почему интеллектуальный указатель `spData` способен функционировать как обычный указатель?" Ответ таков: чтобы позволить программисту использовать их как обычные указатели, классы интеллектуального указателя перегружают оператор обращения к значению (`*`) и оператор обращения к члену (`->`). Перегрузка операторов обсуждалась ранее на занятии 12, "Типы операторов и их перегрузка".

Кроме того, чтобы позволить управлять объектами в распределяемой памяти, тип которых вы выбираете сами, почти все хорошие классы интеллектуального указателя являются шаблонами класса и содержат обобщенную реализацию их функциональных возможностей. Будучи шаблонами, они обладают универсальностью и могут быть специализированы для управления объектами с типом по вашему выбору.

Листинг 26.1 содержит типичную реализацию 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* m_pRawPointer;
 5: public:
 6:     smart_pointer (T* pData) : m_pRawPointer (pData) {}
                                          // Конструктор
 7:     ~smart_pointer () {delete pData;}; // Деструктор
 8:
 9:     // конструктор копий
10:     smart_pointer (const smart_pointer & anotherSP);
11:     // Оператор присвоения копии
12:     smart_pointer& operator= (const smart_pointer& anotherSP);
13:
14:     T& operator* () const // оператор обращения к значению
15:     {
16:         return *(m_pRawPointer);
17:     }
```

```
18:
19:     T* operator-> () const // оператор обращения к члену
20:     {
21:         return m_pRawPointer;
22:     }
23: };
```

Анализ

Приведенный выше класс интеллектуального указателя демонстрирует реализацию этих двух 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 создали бы его экземпляр так:

```
smart_pointer <Tuna> pSmartTuna (new Tuna);
pSmartTuna->Swim();
// Альтернатива:
(*pSmartDog).Swim ();
```

Данный класс `smart_pointer` пока еще не имеет и не реализует функциональных возможностей, которые сделали бы его классом достаточно интеллектуального указателя и обеспечили бы его преимущество перед обычным указателем. Конструктор (строка 7) получает указатель, который сохраняется как внутренний объект указателя в классе интеллектуального указателя. Деструктор освобождает этот указатель, обеспечивая автоматическое освобождение памяти.

ПРИМЕЧАНИЕ

Реализация, которая делает интеллектуальный указатель действительно интеллектуальным, – это конструктор копий, оператор присвоения и деструктор. Именно они определяют поведение объекта интеллектуального указателя, когда он передается в функции, присваивается или выходит из области видимости (т.е. удаляется). Поэтому перед переходом к изучению полной реализации интеллектуального указателя следует рассмотреть возможные типы интеллектуальных указателей.

Типы интеллектуальных указателей

Управление ресурсом памяти (т.е. реализация модели принадлежности) и отличает классы интеллектуального указателя. Интеллектуальные указатели решают, что они делают с ресурсом при их копировании и присвоении. 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 интеллектуальный указатель, прежде чем он решит использовать его в своем приложении.

Классификация интеллектуальных указателей фактически основана на их стратегии управления ресурсами памяти.

- Глубокого копирования.
- Копирования при записи (COW).

- Подсчета ссылок.
- Списка ссылок.
- Деструктивного копирования.

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 (таким образом осуществляется глубокое копирование). Когда интеллектуальный указатель выходит из области видимости, он освобождает память, на которую указывает (при помощи деструктора).

Хотя интеллектуальный указатель глубокого копирования, казалось бы, не играет выдающейся роли при передаче объекта по значению, его преимущество становится очевидным при работе с полиморфными объектами, как демонстрирует следующий код, где он позволяет избежать *отсечения* (slicing):

```
// Пример отсечения 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); // Отсечение: функции MakeFishSwim()
                               // передается только часть Fish объекта Carp

Tuna marineFish;
MakeFishSwim(marineFish); // Снова отсечение
```

Проблема отсечения решается, когда разработчик задействует интеллектуальный указатель глубокого копирования, как представлено в листинге 26.2.

ЛИСТИНГ 26.2. Использование интеллектуального указателя глубокого копирования для передачи полиморфных объектов их базовым классам

```
0: template <typename T>
1: class deepcopy_smart_pointer
2: {
3: private:
4:     T* m_pObject;
5: public:
6:     // ... другие функции
7:
8:     // конструктор копий указателя глубокого копирования
```

```
 9:     deepcopy_smart_pointer (const deepcopy_smart_pointer& source)
10:     {
11:         // Clone() виртуальная: гарантирует глубокое копирование
            // объекта производного класса
12:         m_pObject = source->Clone ();
13:     }
14:
15:     // Оператор присвоения копии
16:     deepcopy_smart_pointer& operator= (const deepcopy_smart_pointer&
                                           source)
17:     {
18:         if (m_pObject)
19:             delete m_pObject;
20:
21:         m_pObject = source->Clone ();
22:     }
23:
24: };
```

Анализ

Как можно заметить, класс `deepcopy_smart_pointer` реализует конструктор копий в строках 9–13. Он обеспечивает глубокое копирование полиморфного объекта при помощи функции `Clone()`, которую должен реализовать объект. Точно так же он реализует оператор присвоения копии 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 эту функцию либо как параметр шаблона, либо как объект функции.

Таким образом, когда сам интеллектуальный указатель передается как указатель на базовый класс `Fish`, часть `Carp` не отсекается:

```
deepcopy_smart_ptr<Carp> freshWaterFish(new Carp);
MakeFishSwim (freshWaterFish); // Carp не будет 'отсечен'
```

Глубокое копирование, реализованное в конструкторе интеллектуального указателя, обеспечивает передачу объекта без отсечения, 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 во многих других случаях потеря производительности могла бы воспрепятствовать программисту задействовать интеллектуальный указатель в его приложении. Вместо этого он мог бы просто передать такой функции, как `MakeFishSwim()`, указатель базового класса (обычный указатель `Fish*`). Другие ссылочные типы пытаются решать эту проблему производительности иными способами.

Механизм копирования при записи

Копирование при записи (Copy on Write — COW) — одна из попыток оптимизировать производительность интеллектуальных указателей глубокого копирования за счет совместного использовании указателей до первой попытки записи объекта. При первой

попытке вызова не константной функции указатель COW, как правило, создает копию объекта, для которого вызвана непостоянная функция, тогда как другие экземпляры указателя продолжат совместно использовать исходный объект.

Указатели COW широко распространены и имеют множество приверженцев. Поклонники указателей COW полагают реализацию операторов (*) и (->) в их константных и не константных версиях ключевой функциональной возможностью указателя COW. Копию создает последняя.

Дело в том, что, выбирая реализацию указателя в соответствии с философией COW, следует убедиться в понимании подробностей его реализации, прежде чем перейти к его использованию. В противном случае вы можете оказаться в ситуации, где копий будет или слишком мало, или слишком много.

Интеллектуальные указатели подсчета ссылок

Подсчет ссылок (reference counting) — это механизм подсчета количества пользователей объекта. Когда их количество сокращается до нуля, объект освобождается. Подсчет ссылок — это очень хороший механизм для совместного использования объектов без необходимости их копирования. Если вы когда-нибудь работали с технологией Microsoft под названием COM, то концепция подсчета ссылок определенно вам знакома.

У таких интеллектуальных указателей должен быть счетчик ссылок, увеличивающийся при копировании объекта. 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 хранится в совместно используемом объекте.

Первый вариант, где счетчик ссылок содержится в объекте, называется *внедренным подсчетом ссылок* (intrusive reference counting), поскольку объект должен быть изменен. В данном случае объект содержит счетчик ссылок, осуществляет его инкремент и предоставляет любому классу интеллектуального указателя, контролирующему его. Кстати, этот подход выбран для технологии COM. Второй вариант, где счетчик ссылок содержится в совместно используемом объекте, является механизмом, где класс интеллектуального указателя способен хранить счетчик ссылок в динамической памяти (целое число в динамической памяти, например), а при копировании конструктор копий увеличивает это значение.

Таким образом, механизм подсчета ссылок удобен при работе интеллектуальных указателей только с объектами. Управление объектами при помощи интеллектуальных указателей при наличии простых указателей на них является плохой идеей, поскольку интеллектуальный указатель (интеллектуально) освобождает объект, когда содержащийся в нем счетчик доходит до нуля, но простой указатель продолжает указывать на ту область памяти, которая вашему приложению больше не принадлежит. Кроме того, подсчет ссылок может вызвать специфические проблемы в таких, например, ситуациях: два объекта, которые содержат указатели друг на друга, никогда не освободятся, поскольку *циклическая зависимость* (cyclic dependency) удерживает их счетчики ссылок в минимальном значении 1.

Интеллектуальный указатель списка ссылок

Интеллектуальный указатель *списка ссылок* (reference-linked) относится к тем, которые не считают количество ссылок, используя объект, а скорее желают знать, когда количество ссылок достигнет нуля, чтобы можно было освободить объект.

Они называются указателями списка 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 интеллектуальный указатель из этого списка. Такой указатель, подобно указателю подсчета ссылок, страдает от проблем, вызванных циклической зависимостью.

Деструктивное копирование

Деструктивное копирование (destructive copy) — это механизм, который при копировании интеллектуального указателя передает получателю собственность на обрабатываемый объект полностью, а сам освобождается:

```
destructive_copy_smartptr <SampleClass> pSmartPtr (new SampleClass ());

SomeFunc (pSmartPtr); // Собственность передается SomeFunc
// Не используйте больше pSmartPtr в вызывающей стороне!
```

Хотя принцип этого механизма не назовешь интуитивно понятным, преимущество использования интеллектуального указателя деструктивного копирования 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 деструктивный характер можно использовать в ваших интересах.

Реализация указателей деструктивного копирования 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::unique_ptr`, который не может быть передан по значению благодаря его закрытому конструктору копий и оператору присвоения копии. Он может быть передан как аргумент только по ссылке.

ЛИСТИНГ 26.3. Типичный интеллектуальный указатель деструктивного копирования

```
0: template <typename T>
1: class destructivecopy_pointer
2: {
3: private:
4:     T* pObject;
```

```
 5: public:
 6:     destructivecopy_pointer(T* pInput):pObject(pInput) {}
 7:     ~destructivecopy_pointer() { delete pObject; }
 8:
 9:     // конструктор копий
10:     destructivecopy_pointer(destructivecopy_pointer& source)
11:     {
12:         // Взять копию в собственность
13:         pObject = source.pObject;
14:
15:         // удалить первоисточник
16:         source.pObject = 0;
17:     }
18:
19:     // Оператор присвоения копии
20:     destructivecopy_pointer& operator= (destructivecopy_pointer& rhs)
21:     {
22:         if (pObject != source.pObject)
23:         {
24:             delete pObject;
25:             pObject = source.pObject;
26:             source.pObject = 0;
27:         }
28:     }
29: };
30:
31: int main()
32: {
33:     destructivecopy_pointer<int> pNumber (new int);
34:     destructivecopy_pointer<int> pCopy = pNumber;
35:
36:     // pNumber теперь недопустим
37:     return 0;
38: }
```

Анализ

В листинге 26.3 показана самая важная часть реализации интеллектуального указателя деструктивного копирования. Строки 10–17 и 20–28 содержат конструктор копий и оператор присвоения копии соответственно. Эти функции делают первоисточник недействительным при его копировании; т.е. после копирования конструктор копий устанавливает указатель, содержавшийся первоисточником, в NULL, оправдывая название *деструктивное копирование*. Оператор присвоения делает то же самое. Таким образом, указатель `pNumber` фактически объявляется недопустимым в строке 34, когда присваивается другому указателю. Это поведение противоестественно действию присвоения.

ВНИМАНИЕ!

Конструктор копий и оператор присвоения копии, которые критически важны для реализации интеллектуальных указателей деструктивного копирования, также продемонстрированные в листинге 26.3, вызывают максимум критики. В отличие от большинства классов C++, у этого не может быть конструктора копий и оператора присвоения получающих константные ссылки, поскольку они

должны объявлять первоисточник недопустимым после его копирования. Это не только отклонение от традиционной семантики конструктора копий и оператора присвоения, это делает использование класса интеллектуального указателя не интуитивно понятным. Немногие ожидают, что оригинал копии или первоисточник присвоения окажутся недопустимы после присвоения или копирования. Тот факт, что такие интеллектуальные указатели уничтожают первоисточник, также делает их неподходящими для использования в таких контейнерах библиотеки STL, как `std::vector`, или любой другой динамический класс коллекции, который вы могли бы использовать. Эти контейнеры должны копировать свое содержимое внутренне, а все заканчивается тем, что указатели оказываются недопустимыми.

По совокупности этих причин многие разработчики боятся интеллектуальных указателей деструктивного копирования как чумы.

СОВЕТ

До сих пор по стандарту C++ основным интеллектуальным указателем деструктивного копирования считался указатель `auto_ptr`. Теперь, в C++11, его применение не рекомендуется, и вместо него следует использовать класс `std::unique_ptr`.

C++11

Использование класса `std::unique_ptr`

Класс `std::unique_ptr` — нововведение стандарта C++11, он немного отличается от класса `auto_ptr` тем, что не позволяет копирование и присвоение.

СОВЕТ

Чтобы использовать класс `std::unique_ptr`, включите его заголовок:

```
#include <memory>
```

Класс `unique_ptr` — это простой интеллектуальный указатель, подобный представленному в листинге 26.1, но с закрытым конструктором копий и оператором присвоения, чтобы воспрепятствовать копированию при передаче в функции по значению или при присвоении. Его применение демонстрирует листинг 26.4.

ЛИСТИНГ 26.4. Использование класса `std::unique_ptr`

```
0: #include <iostream>
1: #include <memory>   // включить для использования std::unique_ptr
2: using namespace std;
3:
4: class Fish
5: {
6: public:
7:     Fish() {cout << "Fish: Constructed!" << endl;}
```

```
 8:     ~Fish() {cout << "Fish: Destructed!" << endl;}
 9:
10:     void Swim() const {cout << "Fish swims in water" << 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, поскольку MakeFishSwim
                                 // допускает ссылку
24:
25:     unique_ptr<Fish> copySmartFish;
26:     // copySmartFish = smartFish; // Ошибка: operator= закрытый
27:
28:     return 0;
29: }
```

Результат

```
Fish: Constructed!
Fish swims in water
Fish swims in water
Fish: Destructed!
```

Анализ

Как видно в выводе, конструкторы и деструкторы выполняются последовательно. Обратите внимание: даже при том, что объект, на который указывает указатель `smartFish`, был создан в функции `main()`, как и ожидалось, он был освобожден (автоматически) даже без явного вызова оператора `delete`. Таково поведение класса `unique_ptr`, где выходящий из области видимости указатель освобождает свой объект при помощи деструктора. Обратите внимание, как в строке 23 можно передать указатель `smartFish` в качестве аргумента функции `MakeFishSwim()`. Это не этап копирования, поскольку функция `MakeFishSwim()` получает параметр по ссылке, как показано в строке 13. Если вы удалите символ ссылки `&` из строки 13, то немедленно получите ошибку компиляции, поскольку конструктор копий закрытый. Аналогично присвоение одного объекта класса `unique_ptr` другому, как показано в строке 26, также не разрешается благодаря закрытому оператору присвоения копии.

Таким образом, указатель класса `unique_ptr` безопасней, чем указатель класса `auto_ptr` (который ныне не рекомендован), поскольку он не объявляет недопустимым исходный объект интеллектуального указателя во время копирования или присвоения. Но он все же обеспечивает простое управление памятью, освобождая объект во время деструкции.

Популярные библиотеки интеллектуальных указателей

Вполне очевидно, что версия интеллектуального указателя, предоставляемого стандартной библиотекой C++, не удовлетворяет требованиям каждого программиста. Вот почему существует 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. Подробную информацию о библиотеке интеллектуальных указателей Boost и ее загрузке можно найти по адресу `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`.

Точно так же при разработке приложений COM на платформах Windows для управления объектами COM имеет смысл использовать эффективные классы интеллектуального указателя среды выполнения ATL, такие как `CComPtr` и `CComQIPtr`, а не простые указатели.

Резюме

На сегодняшнем занятии рассматривалось, как использование подходящих интеллектуальных указателей способно сократить код, связанный с созданием экземпляров и вопросами собственности объекта. Вы также изучили различные типы классов интеллектуальных указателей и, что важней всего, их поведение в вашем приложении. Теперь вы знаете, что не должны использовать указатель `std::auto_ptr`, поскольку он объявляет недопустимым первоисточник при копировании и присвоении. Вы также узнали о новом классе интеллектуального указателя `std::unique_ptr`, совместимого со стандартом C++11.

Вопросы и ответы

- **Мне нужен вектор указателей. Могу ли я выбрать класс `auto_ptr` как тип объекта, который будет содержаться в векторе?**

  Как правило, вы вообще не должны использовать класс `std::auto_ptr`. Это не рекомендуется. Единственная операция копирования или присвоения может сделать исходный объект неприменимым.

- **Какие два оператора должен реализовать класс, чтобы называться классом интеллектуального указателя?**

  Операторы `operator*` и `operator->`. Они позволяют использовать объекты класса с семантикой обычного указателя.

- **У меня есть приложение, в котором классы `Class1` и `Class2` содержат атрибуты, которые указывают на объекты типа других. Должен ли я использовать в этом случае указатель подсчета ссылок?**

  Вероятно, нет, из-за циклической зависимости, которая не позволит обнулить счетчик ссылок, а следовательно, оставит объекты двух классов в распределяемой памяти, не освободив их.

- **Сколько интеллектуальных указателей существует?**
  Тысячи, а возможно и миллионы. Вы должны использовать только те интеллектуальные указатели, которые имеют хорошую документацию на функциональные возможности и предоставляются заслуживающим доверия производителем, таким как Boost.
- **Класс `string` также управляет символьным массивом, расположенным в динамической памяти. Так что, класс `string` тоже интеллектуальный указатель?**
  Нет. Эти классы обычно не реализуют операторы `operator*` и `operator->`, а поэтому они не относится к интеллектуальным указателям.

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 Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Где стоит осуществить поиск, прежде чем писать собственный класс интеллектуального указателя для своего приложения?
2. Значительно ли интеллектуальный указатель замедлил бы ваше приложение?
3. Где интеллектуальные указатели подсчета ссылок способны хранить данные счетчика ссылок?
4. Должен ли механизм указателя связанного списка использовать однонаправленный список ссылок или двунаправленный?

Упражнения

1. **Отладка:** Найдите ошибку в этом коде:

```
std::auto_ptr<SampleClass> pObject (new SampleClass ());
std::auto_ptr<SampleClass> pAnotherObject (pObject);
pObject->DoSomething ();
pAnotherObject->DoSomething();
```

   Используйте класс `unique_ptr` для создания экземпляра класса `Carp`, происходящего от класса `Fish`. Передайте объект как указатель на класс `Fish` и отметьте комментарием отсечение, если оно будет.

2. **Отладка:** Укажите ошибку в этом 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 занятия 1, "Первые шаги", где вы отображали на экране строку "Hello World", используя оператор `std::cout`. Пришло время обратить внимание на эту часть языка C++ и изучить потоки с практической точки зрения.

На сегодняшнем 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 по одинаковой схеме, независимо от устройства или области, с которой происходят обмен данными? 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 для чтения и записи данных. Эти схемы единообразны, 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 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;
```

Так и есть: `std::cout` — это потоковый объект класса `ostream` для вывода на консоль. Чтобы использовать класс `std::cout`, необходимо включить в код заголовок `<iostream>`, который предоставляет эту и другие функциональные возможности, такие как класс `std::cin`, позволяющий читать из потока.

Что я подразумеваю, когда говорю, что потоки обеспечивают единообразный и специфический для устройств доступ? Если бы необходимо было записать текст "Hello World" в текстовый файл, то можно было бы использовать такой синтаксис объекта файлового потока `fsHW`:

```
fsHW << "Hello World!" << endl; // "Hello World!" в файловый поток
```

Как можно заметить, после выбора подходящего потокового класса запись текста "Hello World" в файл не слишком 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 ввода с клавиатуры, из файла и т.д.

Стоит отметить, что на этом занятии потоки рассматриваются с практической точки зрения.

Важнейшие классы и объекты потоков C++

Язык C++ предоставляет набор стандартных классов и заголовков, позволяющих выполнять ряд наиболее важных и популярных операции ввода и вывода. Табл. 27.1 содержит список наиболее часто используемых классов.

ТАБЛИЦА 27.1. Наиболее популярные классы потоков C++ пространства имен `std`

Класс или объект	Задача
`cout`	Стандартный поток вывода, как правило, переадресовываемый на консоль
`cin`	Стандартный поток ввода, как правило, используемый для чтения данных в переменные
`cerr`	Стандартный поток вывода для ошибок
`fstream`	Класс потока ввода и вывода для файловых операций; происходит 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; происходит от классов `istringstream` и `ostringstream`; обычно используется для преобразования в строки (или из строк), а также другие типы

ПРИМЕЧАНИЕ

Объекты `cout`, `cin` и `cerr` являются глобальными объектами потоковых классов `ostream`, `istream` и `ostream` соответственно. Будучи глобальными объектами, они инициализируются перед запуском функции `main()`.

При использовании потокового класса есть возможность определения *манипуляторов* (manipulator), которые автоматически выполняют специфические действия самостоятельно. Одним из них является манипулятор `std::endl`, который вы использовали для вставки символа новой строки:

```
std::cout << "This lines ends here" << std::endl;
```

Несколько других подобных манипуляторов и флагов приведено в табл. 27.2.

ТАБЛИЦА 27.2. Наиболее используемые манипуляторы пространства имен `std` для работы с потоками

Манипуляторы вывода	Задача
`endl`	Вставляет символ новой строки
`ends`	Вставляет нулевой символ
Манипуляторы систем счисления	Задача
`dec`	Вынуждает поток интерпретировать ввод или отображать вывод как десятичное число
`Hex`	Вынуждает поток интерпретировать ввод или отображать вывод как шестнадцатеричное число
`oct`	Вынуждает поток интерпретировать ввод или отображать вывод как восьмеричное число
Манипуляторы представления значений с плавающей запятой	Задача

Окончание табл. 27.2

Манипуляторы вывода	Задача
`fixed`	Вынуждает поток отображать значения в форме с фиксированной точкой
`scientific <iomanip>`	Вынуждает поток отображать значения в экспоненциальном представлении
Манипуляторы	Задача
`setprecision`	Устанавливает точность десятичного представления как параметр
`setw`	Устанавливает ширину поля как параметр
`setfill`	Устанавливает символ заполнения как параметр
`setbase`	Устанавливает основание системы счисления, используя `dec`, `hex` или `oct` как параметр
`setiosflag`	Устанавливает флаги через маску входного параметра типа `std::ios_base::fmtflags`
`resetiosflag`	Восстанавливает исходные значения для специфического типа, определенного содержимым `std::ios_base::fmtflags`

Использование объекта `std::cout` для вывода отформатированных 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. Наконец пришло время вернуться к объекту `cout` и использовать некоторые из манипуляторов для изменения способа выравнивания и отображения данных.

Изменение формата представления чисел

Объект `std::cout` можно заставить отображать целые числа в шестнадцатеричном или восьмеричном формате. Листинг 27.1 демонстрирует использование объекта `cout` для отображения введенных чисел в различных форматах.

ЛИСТИНГ 27.1. Отображение целого числа в десятичном, восьмеричном и шестнадцатеричном форматах с использованием объекта `cout` и флагов `<iomanip>`

```
 0: #include <iostream>
 1: #include <iomanip>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:     cout << "Enter an integer: ";
 7:     int Input = 0;
 8:     cin >> Input;
 9:
10:     cout << "Integer in octal: " << oct << Input << endl;
```

```
11:     cout << "Integer in hexadecimal: " << hex << Input << endl;
12:
13:     cout << "Integer in hex using base notation: ";
14:     cout << setiosflags(ios_base::hex|ios_base::showbase|ios_base \
                            ::uppercase);
15:     cout << Input << endl;
16:
17:     cout << "Integer after resetting I/O flags: ";
18:     cout << resetiosflags(ios_base::hex|ios_base::showbase|ios_base \
                              ::uppercase);
19:     cout << Input << endl;
20:
21:     return 0;
22: }
```

Результат

```
Enter an integer: 253
Integer in octal: 375
Integer in hexadecimal: fd
Integer in hex using base notation: 0XFD
Integer after resetting I/O flags: 253
```

Анализ

Пример использует представленные в табл. 27.2 манипуляторы для изменения способа отображения объектом `cout` введенного пользователем целого числа. Обратите внимание на использование манипуляторов `oct` и `hex` в строках 10 и 11. В строке 14 функция `setiosflags()` используется для задания отображения числа прописными буквами в шестнадцатеричном формате. В результате объект `cout` отображает введенное целое число 253 как `0XFD`. В результате использования функции `resetioflags()` в строке 18 объект `cout` снова отображает целое число в десятичном формате. Вот другой способ смены отображения целых чисел в десятичном формате:

```
cout << dec << Input << endl; // отображать в десятичном формате
```

В формате отображения объектом `cout` таких чисел, как `Pi`, можно также задать точность, т.е. определить количество знаков десятичного числа после запятой, которое будет представлено, либо задать отображение числа в экспоненциальном представлении (листинг 27.2).

ЛИСТИНГ 27.2. Использование объекта `cout` для отображения числа `Pi` и площади круга в экспоненциальном представлении и с фиксированной точкой

```
0: #include <iostream>
1: #include <iomanip>
2: using namespace std;
3:
4: int main()
5: {
6:     const double Pi = (double)22.0 / 7;
7:     cout << "Pi = " << Pi << endl;
8:
9:     cout << endl << "Setting precision to 7: " << endl;
```

```
10:     cout << setprecision(7);
11:     cout << "Pi = " << Pi << endl;
12:     cout << fixed << "Fixed Pi = " << Pi << endl;
13:     cout << scientific << "Scientific Pi = " << Pi << endl;
14:
15:     cout << endl << "Setting precision to 10: " << endl;
16:     cout << setprecision(10);
17:     cout << "Pi = " << Pi << endl;
18:     cout << fixed << "Fixed Pi = " << Pi << endl;
19:     cout << scientific << "Scientific Pi = " << Pi << endl;
20:
21:     cout << endl << "Enter a radius: ";
22:     double Radius = 0.0;
23:     cin >> Radius;
24:     cout << "Area of circle: " << 2*Pi*Radius*Radius << endl;
25:
26:     return 0;
27: }
```

Результат

```
Pi = 3.14286

Setting precision to 7:
Pi = 3.142857

Fixed Pi = 3.1428571
Scientific Pi = 3.1428571e+000

Setting precision to 10:
Pi = 3.1428571429e+000
Fixed Pi = 3.1428571429
Scientific Pi = 3.1428571429e+000

Enter a radius: 9.99
Area of circle: 6.2731491429e+002
```

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 `6.2731491429e+002`.

Выравнивание текста и установка ширины поля с использованием объекта `std::cout`

Для установки ширины поля в символах можно использовать такие манипуляторы, 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 такой ситуации, которая показана в листинге 27.3.

ЛИСТИНГ 27.3. Установка ширины поля и символов заполнения с использованием манипуляторов `setw()` и `setfill()`

```
 0: #include <iostream>
 1: #include <iomanip>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:     cout << "Hey - default!" << endl;
 7:
 8:     cout << setw(35); // установка поля шириной 25 символов
 9:     cout << "Hey - right aligned!" << endl;
10:
11:     cout << setw(35) << setfill('*');
12:     cout << "Hey - right aligned!" << endl;
13:
14:     cout << "Hey - back to default!" << endl;
15:
16:     return 0;
17: }
```

Результат

```
Hey - default!
                Hey - right aligned!
***************Hey - right aligned!
Hey - back to default!
```

Анализ

Вывод демонстрирует результат применения манипулятора `setw(35)` в строке 8 и манипулятора `setfill('*')` в строке 11 для объекта `cout`. Как можно заметить, в предпоследней строке вывода свободное пространство, предшествующее тексту, заполнено звездочками, как определено манипулятором `setfill()`.

Использование объекта `std::cin` для ввода

Объект `std::cin` универсален, он позволяет читать данные простых типов, таких как `int`, `double` и строк `char*` в стиле C, а также читать с экрана строки и символы, используя такие методы, как `getline()`.

Использование объекта `std::cin` для ввода старых простых типов данных

При помощи объекта `cin` можно читать данные типа `int`, `double` и `char` непосредственно из стандартного устройства ввода. Листинг 27.4 демонстрирует применение объекта `cin` для чтения простых типов данных, введенных пользователем.

ЛИСТИНГ 27.4. Использование объекта `cin` для чтения в переменную типа `int`, в переменную типа `double` в экспоненциальной форме и трех символов типа `char`

```
 0: #include<iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Enter an integer: ";
 6:     int InputInt = 0;
 7:     cin >> InputInt;
 8:
 9:     cout << "Enter the value of Pi: ";
10:     double Pi = 0.0;
11:     cin >> Pi;
12:
13:     cout << "Enter three characters separated by space: " << endl;
14:     char Char1 = '\0', Char2 = '\0', Char3 = '\0';
15:     cin >> Char1 >> Char2 >> Char3;
16:
17:     cout << "The recorded variable values are: " << endl;
18:     cout << "InputInt: " << InputInt << endl;
19:     cout << "Pi: " << Pi << endl;
20:     cout << "The three characters: " << Char1 << Char2 << Char3
            << endl;
21:
22:     return 0;
23: }
```

Результат

```
Enter an integer: 32
Enter the value of Pi: 0.314159265e1
Enter three characters separated by space:
c + +
The recorded variable values are:
InputInt: 32
Pi: 3.14159
The three characters: 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 получилось заполнить три символьные переменные в пределах одной строки (строка 15).

Использование метода `std::cin::get()` для ввода в символьный буфер стиля C

Подобно тому, как объект `cin` позволяет записывать данные непосредственно в переменную типа `int`, то же самое можно сделать с символьным массивом стиля C:

```
cout << "Enter a line: " << endl;
char CStyleStr [10] = {0}; // может содержать максимум 10 символов
cin >> CStyleStr; // Опасно: пользователь может ввести больше 10 символов
```

При записи в буфер строки стиля С очень важно не превышать границ буфера, чтобы избежать аварийного отказа или бреши в системе безопасности. Поэтому лучше читать в строковый буфер стиля С так:

```
cout << "Enter a line: " << endl;
char CStyleStr[10] = {0};
cin.get(CStyleStr, 9);   // прекращение вставки на 9-ом символе
```

Этот более безопасный способ вставки текста в буфер стиля С приведен в листинге 27.5.

ЛИСТИНГ 27.5. Вставка в буфер стиля С без выхода за его границы

```
 0: #include<iostream>
 1: #include<string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:     cout << "Enter a line: " << endl;
 7:     char CStyleStr[10] = {0};
 8:     cin.get(CStyleStr, 9);
 9:     cout << "CStyleStr: " << CStyleStr << endl;
10:
11:     return 0;
12: }
```

Результат

```
Enter a line:
Testing if I can cross the bounds of the buffer
CStyleStr: Testing i
```

Анализ

Как демонстрирует вывод, благодаря использованию в строке 8 метода `cin::get()`, в буфер стиля С были записаны только первые девять символов. Это самый безопасный способ работы со строками стиля С.

СОВЕТ

По возможности не используйте строки стиля С и массивы типа `char`. Используйте вместо них тип `std::string`.

Использование объекта `std::cin` для ввода в переменную типа `std::string`

Объект `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 приведен ввод с использованием объекта `cin` в переменную типа `std::string`.

ЛИСТИНГ 27.6. Вставка текста в строку `std::string` с использованием объекта `cin`

```
0: #include<iostream>
1: #include<string>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Enter your name: ";
7:     string Name;
8:     cin >> Name;
9:     cout << "Hi " << Name << endl;
10:
11:    return 0;
12: }
```

Результат

```
Enter your name: Siddhartha Rao
Hi Siddhartha
```

Анализ

Вывод отобразил мое имя не полностью, поскольку так была реализована программа. Я ожидал, что переменная `Name`, заполняемая объектом `cin` в строке 8, будет содержать введенное мной имя и фамилию, а не только имя. Что же произошло? Объект `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 объектом `cin` показано в листинге 27.7.

ЛИСТИНГ 27.7. Чтение введенной пользователем строки полностью с использованием функции `getline()` и объекта `cin`

```
0: #include<iostream>
1: #include<string>
2: using namespace std;
3:
4: int main()
5: {
6:     cout << "Enter your name: ";
7:     string Name;
8:     getline(cin, Name);
9:     cout << "Hi " << Name << endl;
```

```
10:
11:     return 0;
12: }
```

Результат

```
Enter your name: Siddhartha Rao
Hi Siddhartha Rao
```

Анализ

Функция `getline()`, как показано в строке 8, решила проблему ввода символа пробела. Теперь вывод содержит введенную пользователем строку полностью.

Использование объекта `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::ofstream` для чтения из него.

Другими словами, класс `std::fstream` обеспечивает возможность как чтения, так и записи.

СОВЕТ

Чтобы использовать класс `std::fstream`, включите его заголовок:

```
#include <fstream>
```

Открытие и закрытие файла с использованием 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())
{
    // здесь 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, в котором открывается файл. Выбранный режим позволяет создать файл, 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 успеха выполнения метода `open()`.

ВНИМАНИЕ!

Закрытие файлового потока важно для сохранения его содержимого.

Есть альтернативный способ открытия файлового потока — через конструктор:

```
fstream myFile("HelloFile.txt",
               ios_base::in|ios_base::out|ios_base::trunc);
```

Если необходимо открыть файл только для записи, используйте следующий код:

```
ofstream myFile("HelloFile.txt", ios_base::out);
```

Если необходимо открыть файл только для чтения, используйте следующий код:

```
ifstream myFile("HelloFile.txt", ios_base::in);
```

СОВЕТ

Независимо от того, используете ли вы конструктор или метод `open()`, рекомендуется проверять успех открытия файла при помощи метода `is_open()`, прежде чем продолжать использовать соответствующий объект файлового потока.

Файловый поток может быть открыт в нескольких режимах.

- `ios_base::app`. Добавляет в конец существующего файла, не усекая его.
- `ios_base::ate`. Переводит в конец файла, но запись данных возможна в любое место.
- `ios_base::trunc`. Усекает существующий файл (принято по умолчанию).
- `ios_base::binary`. Создает двоичный файл (по умолчанию принят текстовый).
- `ios_base::in`. Открывает файл только для чтения.
- `ios_base::out`. Открывает файл только 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 с использованием объекта класса `ofstream`

```
 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 << "File open successful" << endl;
12:
```

```
13:         myFile << "My first text file!" << endl;
14:         myFile << "Hello file!";
15:
16:         cout << "Finished writing to file, will close now" << endl;
17:         myFile.close();
18:     }
19:
20:     return 0;
21: }
```

Результат

```
File open successful
Finished writing to file, will close now
```

Содержимое файла `HelloFile.txt`:

```
My first text file!
Hello file!
```

Анализ

Строка 7 открывает файл в режиме `ios_base::out` — т.е. исключительно для записи. В строке 9 проверяется успех выполнения метода `open()`, а затем осуществляется запись в файловый поток с использованием оператора вывода `operator<<`, как показано в строках 13 и 14. И наконец, файл закрывается в строке 17.

ПРИМЕЧАНИЕ

Листинг 27.8 демонстрирует, что писать в файловый поток используя объект `cout`, можно точно так же, как на стандартное устройство вывода (т.е. на консоль).

Это значит, что потоки C++ обеспечивают единообразный способ работы с различными устройствами, будь то запись текста на экран при помощи объекта `cout` или запись в файл при помощи объекта `ofstream`.

Чтение текстового файла с использованием метода open() и оператора >>

Для чтения файла можно воспользоваться объектом `fstream`, если открыть его с использованием флага `ios_base::in`, или использовать объект `ifstream`. Листинг 27.9 демонстрирует 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 << "File open successful. It contains: " << endl;
13:         string fileContents;
14:
15:         while (myFile.good())
16:         {
17:             getline (myFile, fileContents);
18:             cout << fileContents << endl;
19:         }
20:
21:         cout << "Finished reading file, will close now" << endl;
22:         myFile.close();
23:     }
24:     else
25:         cout << "open() failed: check if file is in right folder"
                 << endl;
26:
27:     return 0;
28: }
```

Результат

```
File open successful. It contains:
My first text file!
Hello file!
Finished reading file, will close now
```

ПРИМЕЧАНИЕ

Чтобы код листинга 27.9 прочитал текстовый файл `HelloFile.txt`, созданный в листинге 27.8, его следует либо переместить в рабочий каталог этого проекта, либо объединить этот код с предыдущим.

Анализ

Как обычно, вызов метода `is_open()` в строке 8 проверяет успех вызова метода `open()`. Обратите внимание на применение оператора извлечения >> при чтении содержимого файла непосредственно в строку, которая затем отображается при помощи объекта `cout` в строке 18. В этом 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 двоичного файла

Фактически процесс записи в двоичный файл не слишком отличается от процесса, который вам уже известен на настоящий момент. При открытии файла следует использовать флаг `ios_base::binary` как маску. Обычно используются методы `ofstream::write()` и `ifstream::read()`, как показано в листинге 27.10.

ЛИСТИНГ 27.10. Запись структуры в двоичный файл и ее восстановление из того же файла

```
 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* inName, int inAge, \
            const char* inDOB) : Age(inAge)
10:     {
11:         strcpy(Name, inName);
12:         strcpy(DOB, inDOB);
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, "May 1910");
23:
24:     ofstream fsOut ("MyBinary.bin", \
                        ios_base::out | ios_base::binary);
25:
26:     if (fsOut.is_open())
27:     {
28:         cout << "Writing one object of Human to a binary file"
                 << endl;
29:         fsOut.write(reinterpret_cast<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 << "Reading information from binary file: " << endl;
41:         cout << "Name = " << somePerson.Name << endl;
42:         cout << "Age = " << somePerson.Age << endl;
43:         cout << "Date of Birth = " << somePerson.DOB << endl;
44:     }
45:
46:     return 0;
47: }
```

Результат

```
Writing one object of Human to a binary file
Reading information from binary file:
Name = Siddhartha Rao
Age = 101
Date of Birth = May 1910
```

Анализ

В строках 22–31 создается экземпляр структуры Human, содержащей атрибуты Name, Age и DOB. Она сохраняется на диске в двоичном файле MyBinary.bin с использованием объекта класса ofstream. Затем, в строках 33–44, эта информация читается с использованием другого потокового объекта класса ifstream. Информация для вывода таких атрибутов, как Name и других, читается из двоичного файла. Этот пример демонстрирует также применение объектов ifstream и ofstream для чтения и записи файлов с использованием методов ifstream::read() и ofstream::write() соответственно. Обратите внимание на применение оператора reinterpret_cast в строке 29, фактически он вынуждает компилятор интерпретировать структуру как char*. В строке 38 применяется приведение в стиле C.

ПРИМЕЧАНИЕ

Если бы не цели демонстрации, я записал 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 данной версии придется добавить в нее новые атрибуты (например, NumChildren), то вам придется позаботиться о функциональных возможностях метода ifstream::read(), чтобы правильно читать двоичные данные прежних версий.

Использование объекта std::stringstream для преобразования строк

Предположим, есть строка, содержащая строковое значение "45". Как преобразовать это строковое значение в целое число со значением 45? И наоборот? Одной из весьма полезных утилит, предоставляемых языком C++, является класс stringstream, обеспечивающий выполнение множества преобразований.

СОВЕТ

Чтобы использовать класс std::stringstream, включите его заголовок:

```
#include <sstream>
```

Некоторые из основных операций класса stringstream демонстрирует листинг 27.11.

ЛИСТИНГ 27.11. Преобразование целочисленного значения в строковое, и наоборот, с использованием класса `std::stringstream`

```
 0: #include<fstream>
 1: #include<sstream>
 2: #include<iostream>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:     cout << "Enter an integer: ";
 8:     int Input = 0;
 9:     cin >> Input;
10:
11:     stringstream converterStream;
12:     converterStream << Input;
13:     string strInput;
14:     converterStream >> strInput;
15:
16:     cout << "Integer Input = " << Input << endl;
17:     cout << "String gained from integer, strInput = " << strInput
            << endl;
18:
19:     stringstream anotherStream;
20:     anotherStream << strInput;
21:     int Copy = 0;
22:     anotherStream >> Copy;
23:
24:     cout << "Integer gained from string, Copy = " << Copy << endl;
25:
26:     return 0;
27: }
```

Результат

```
Enter an integer: 45
Integer Input = 45
String gained from integer, strInput = 45
Integer gained from string, Copy = 45
```

Анализ

Пользователя просят ввести целочисленное значение. Сначала это целое число вставляется в объект класса `stringstream` (строка 12) при помощи оператора <<. Затем, в строке 14, оператор извлечения >> используется для преобразования этого целого числа в строку. Потом эта строка используется как отправная точка для получения целочисленного значения переменной `Copy`, представляющего числовое значение строки в переменной `strInput`.

РЕКОМЕНДУЕТСЯ	НЕ 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 при помощи метода `is_open()`. Используйте ее прежде, чем вставить или извлечь данные из файлового потока	**Не забывайте** закрывать файловый поток при помощи метода `close()` после его использования **Не забывайте**, что в результате чтения строки кодом `cin >> strData;` переменная `strData` содержит текст лишь до первого пробела в строке **Не забывайте**, что функция `getline(cin, strData);` извлекает всю строку из входного потока, включая пробелы

Резюме

На этом занятии рассматривались потоки C++ с практической точки зрения. Вы учились использовать такие потоки ввода и вывода, как `cout` и `cin`, с самого начала книги. Теперь вы знаете, как создавать простые текстовые файлы, а также читать или записывать в них. Вы узнали, чем класс `stringstream` может помочь в преобразовании простых типов, таких как `int`, в строки, и наоборот.

Вопросы и ответы

- **Если я могу использовать класс `fstream` и для записи, и для чтения из файла, то зачем мне использовать классы `ofstream` и `ifstream`?**

  Если ваш код или модуль должен только читать из файла, то следует использовать класс `ifstream`. Точно так же, если нужно только записать в файл, используйте класс `ofstream`. В обоих случаях прекрасно сработал бы и класс `fstream`, но для обеспечения целостности данных лучше иметь ограничительную политику, подобную использованию констант, которая также не обязательна.
- **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()` позволяет читать пользовательский ввод по одному символу за раз.
- **Когда мне использовать класс `stringstream`?**

  Класс `stringstream` обеспечивает удобный способ преобразования целых чисел и других простых типов в строку, и наоборот (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. Попытайтесь самостоятельно ответить на эти вопросы и выполнить задания, а потом

сверьте полученные результаты с ответами в приложении Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Необходима только запись в файл. Какой поток использовать?
2. Как использовать объект `cin` для получения из входного потока полной строки?
3. Необходима запись объекта класса `std::string` в файл. Выбрали бы вы режим `ios_base::binary`?
4. Вы открыли поток при помощи метода `open()`. Зачем беспокоиться об использовании метода `is_open()`?

Упражнения

1. **Отладка:** Найдите ошибку в следующем коде:

```
fstream myFile;
myFile.open("HelloFile.txt", ios_base::out);
myFile << "Hello file!";
myFile.close();
```

2. **Отладка:** Найдите ошибку в следующем коде:

```
ifstream MyFile("SomeFile.txt");
if(MyFile.is_open())
{
    MyFile << "This is some text" << 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 резервирование памяти всегда успешно, файлы всегда находятся и т.д. Однако действительность зачастую совершенно иная.

На сегодняшнем занятии.

- Что такое исключение.
- Как обрабатываются исключения.
- Как обработка исключений помогает создавать стабильные приложения C++.

Что такое исключение

Ваша программа резервирует 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, хоть и управляет гигабайтами! Вы отправляете 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 появляются первые жалобы. В некоторых из них будет упоминание о нарушении прав доступа, а в других — сообщение "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 той же аппаратной конфигурации. Причина этого 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 распределение памяти, которое отлично работало в ваших условиях, отказывает 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 сделать то, что оно намеревалось. Тем не менее ваше приложение способно обработать исключение и отобразить пользователю сообщение об ошибке, выполнить, по мере необходимости, операции по сохранению данных и завершить работу корректно.

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 ноль. Некоторые модули разрабатываются так, чтобы сообщить об ошибке, передавая исключения вызывающей стороне.

ПРИМЕЧАНИЕ

Чтобы защитить свой код от исключений, вы *обрабатываете* (handle) их, делая свой код устойчивым к исключениям (exception safe).

Реализация устойчивости к исключениям при помощи блоков `try` и `catch`

Когда дело доходит до реализации устойчивости к исключениям, ключевые слова C++ `try` и `catch` оказываются самыми важными. Чтобы сделать операторы устойчивыми к

исключениям, их следует поместить в блок try и обработать исключения, которые блок try передаст в блок catch:

```
void SomeFunc()
{
    try
    {
        int* pNumber = new int;
        *pNumber = 999;
        delete pNumber;
    }
    catch(...) // ... обрабатывает все исключения
    {
        cout << "Exception in SomeFunc(), quitting" << endl;
    }
}
```

Использование блока catch(...) для обработки всех исключений

Помните, на занятии 8, "Указатели и ссылки", я упоминал о том, что стандартная форма оператора new возвращает допустимый указатель на область в памяти, если все пройдет удачно, в противном случае передает исключение. В листинге 28.1 показано, как резервирование памяти с использованием оператора new можно сделать устойчивым к исключениям и находить выход из ситуации, когда компьютер не в состоянии зарезервировать запрошенную память.

ЛИСТИНГ 28.1. Использование блоков try и catch для обеспечения устойчивости к исключениям при резервировании памяти

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     cout << "Enter number of integers you wish to reserve: ";
 6:     try
 7:     {
 8:         int Input = 0;
 9:         cin >> Input;
10:
11:         // запрос области в памяти и ее последующее освобождение
12:         int* pReservedInts = new int [Input];
13:         delete[] pReservedInts;
14:     }
15:     catch (...)
16:     {
17:         cout << "Exception encountered. Got to end, sorry!" << endl;
18:     }
19:     return 0;
20: }
```

Результат

```
Enter number of integers you wish to reserve: -1
Exception encountered. Got to end, sorry!
```

Анализ

Для этого примера я указал количество резервируемых чисел `-1`. Это абсурд, но пользователи иногда делают абсурдные вещи. В отсутствие обработчика исключений работа программы закончилась бы некрасиво. Но благодаря обработчику исключения, как видите, вывод отображает осмысленное сообщение: `Got to end, sorry!` (Это конец, извините!).

ПРИМЕЧАНИЕ

Если вы запускаете программу в среде разработки Visual Studio, то можете увидеть сообщение режима отладки, представленное на рис. 28.1.

РИС. 28.1. Утверждение в связи с недопустимым размером резервируемой памяти

Чтобы позволить сработать обработчику исключения, щелкните на кнопке Ignore (Игнорировать). Это сообщение режима отладки, но обработка исключений позволяет вашей программе корректно закончить работу даже в режиме релиза.

Листинг 28.1 демонстрирует применение блоков `catch` и `try`. Блок `catch()` получает параметры, как обычная функция, а "`...`" означает, что блок `catch` принимает исключения всех типов. Но в данном случае мы могли бы захотеть принимать исключения только типа `std::bad_alloc`, поскольку именно они передаются при неудаче оператора `new`. Обработка исключений определенного типа позволяет рассматривать проблемы конкретного типа, в частности, например, отображать пользователю сообщение о том, что именно пошло не так.

Обработка исключения конкретного типа

Исключение в листинге 28.1 передавалось из стандартной библиотеки C++. Тип таких исключений известен, и их самостоятельная обработка проще, поскольку вы уже точно знаете причину исключения, можете лучше организовать очистку или, по крайней мере, отобразить пользователю конкретное сообщение, как в листинге 28.2.

ЛИСТИНГ 28.2. Обработка исключения типа `std::bad_alloc`

```
 0: #include <iostream>
 1: #include<exception> // включите для обработки исключения bad_alloc
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:     cout << "Enter number of integers you wish to reserve: ";
 7:     try
 8:     {
 9:         int Input = 0;
10:         cin >> Input;
11:
12:         // запрос области в памяти и ее последующее освобождение
13:         int* pReservedInts = new int [Input];
14:         delete[] pReservedInts;
15:     }
16:     catch (std::bad_alloc& exp)
17:     {
18:         cout << "Exception encountered: " << exp.what() << endl;
19:         cout << "Got to end, sorry!" << endl;
20:     }
21:     catch(...)
22:     {
23:         cout << "Exception encountered. Got to end, sorry!" << endl;
24:     }
25:     return 0;
26: }
```

Результат

```
Enter number of integers you wish to reserve: -1
Exception encountered: bad allocation
Got to end, sorry!
```

Анализ

Сравните вывод листинга 28.2 с выводом листинга 28.1. Как можно заметить, теперь вы в состоянии указать причину внезапного окончания работы приложения точней, а именно "bad allocation" (ошибка резервирования). Это связано с тем, что теперь есть дополнительный блок `catch` (да, два блока `catch`), который обрабатывает исключения конкретного типа `catch(bad_alloc&)`, как показано в строках 16–20.

СОВЕТ

Вы можете вставить столько блоков `catch()`, сколько вам нужно, располагая их один за другим в зависимости от типа ожидаемых и вероятных исключений.

Блок `catch(...)`, представленный в листинге 28.2, обрабатывает исключения всех типов, которые не были обработаны явно другими блоками `catch`.

Передача исключения конкретного типа с использованием оператора throw

Когда вы обрабатывали исключение std::bad_alloc в листинге 28.2, речь фактически шла об объекте класса std::bad_alloc, который передал оператор new. Но вы вполне можете самостоятельно передать исключение по собственному выбору. Для этого необходимо ключевое слово throw:

```
void DoSomething()
{
    if(something_unwanted)
        throw Value;
}
```

Давайте изучим применение оператора throw на примере собственного типа исключения, передаваемого при попытке деления на нуль, как представлено в листинге 28.3.

ЛИСТИНГ 28.3. Передача специального исключения при попытке деления на нуль

```
 0: #include<iostream>
 1: using namespace std;
 2:
 3: double Divide(double Dividend, double Divisor)
 4: {
 5:     if(Divisor == 0)
 6:         throw "Dividing by 0 is a crime";
 7:
 8:     return (Dividend / Divisor);
 9: }
10:
11: int main()
12: {
13:     cout << "Enter dividend: ";
14:     double Dividend = 0;
15:     cin >> Dividend;
16:     cout << "Enter divisor: ";
17:     double Divisor = 0;
18:     cin >> Divisor;
19:
20:     try
21:     {
22:         cout << "Result of division is: "
                 << Divide(Dividend, Divisor);
23:     }
24:     catch(char* exp)
25:     {
26:         cout << "Exception: " << exp << endl;
27:         cout << "Sorry, can't continue!" << endl;
28:     }
29:
30:     return 0;
31: }
```

Результат

```
Enter dividend: 2011
Enter divisor: 0
Exception: Dividing by 0 is a crime
Sorry, can't continue!
```

Анализ

Код не только демонстрирует возможность обработки исключения типа `char*`, как показано в строке 24, но также и то, что возможна обработка исключения, переданного в 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 передача исключения. Это хорошая общепринятая практика, поскольку обработка исключений также может уменьшать производительность вашего кода.

Как действует обработка исключений

В функции `Divide()` листинга 28.3 передавалось исключение типа `char*`, которое обрабатывалось обработчиком `catch (char*)` в вызывающей функции `main()`.

Когда исключение передается с использованием оператора `throw`, компилятор вставляет динамический поиск соответствующего блока `catch`, способного обработать это исключение. Логика обработки исключений подразумевает поиск передавшей исключение строки сначала в пределах блока `try`. Если это так, то начинается поиск библиотеки `catch`, который способен обработать исключение этого типа. Если оператор `throw` находится вне блока `try` или если нет блока `catch()`, соответствующего типу исключения, логика обработки ищет то же самое в вызывающей функции. Так, логика обработки исключений двигается по стеку вызовов вверх, перебирая одну вызывающую функцию за другой, отыскивая подходящий блок `catch`, способный обработать исключение данного типа. На каждом этапе прокрутки стека локальные переменные текущей функции уничтожаются в порядке, обратном их созданию (листинг 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 << "Constructed a struct A" << endl; }
 6:     ~StructA() {cout << "Destroyed a struct A" << endl; }
 7: };
 8:
 9: struct StructB
10: {
11:     StructB() {cout << "Constructed a struct B" << endl; }
12:     ~StructB() {cout << "Destroyed a struct B" << endl; }
13: };
14:
15: void FuncB() // передача
```

```
16: {
17:     cout << "In Func B" << endl;
18:     StructA objA;
19:     StructB objB;
20:     cout << "About to throw up!" << endl;
21:     throw "Throwing for the heck of it";
22: }
23:
24: void FuncA()
25: {
26:     try
27:     {
28:         cout << "In Func A" << endl;
29:         StructA objA;
30:         StructB objB;
31:         FuncB();
32:         cout << "FuncA: returning to caller" << endl;
33:     }
34:     catch(const char* exp)
35:     {
36:         cout << "FuncA: Caught exception, it says: " << exp << endl;
37:         cout << "FuncA: Handled it here, will not throw to caller"
                 << endl;
38:         // throw; // снимите комментирий для передачи в main()
39:     }
40: }
41:
42: int main()
43: {
44:     cout << "main(): Started execution" << endl;
45:     try
46:     {
47:         FuncA();
48:     }
49:     catch(const char* exp)
50:     {
51:         cout << "Exception: " << exp << endl;
52:     }
53:     cout << "main(): exiting gracefully" << endl;
54:     return 0;
55: }
```

Результат

```
main(): Started execution
In Func A
Constructed a struct A
Constructed a struct B
In Func B
Constructed a struct A
Constructed a struct B
About to throw up!
Destroyed a struct B
```

```
Destroyed a struct A
Destroyed a struct B
Destroyed a struct A
FuncA: Caught exception, it says: Throwing for the heck of it
FuncA: Handled it here, will not throw to caller
main(): exiting gracefully
```

Анализ

В листинге 28.4 функция `main()` вызывает функцию `FuncA()`, которая вызывает функцию `FuncB()`, передающую исключение в строке 21. Обе вызывающие функции, `FuncA()` и `main()`, являются устойчивыми к исключениям, поскольку у обеих реализован блок `catch(const char*)`. У функции `FuncB()`, передающей исключение, нет блоков `catch()`, а следовательно, первым обработчиком переданного ей исключения будет блок `catch` в пределах функции `FuncA()` (строки 34–39), поскольку функция `FuncA()` является вызывающей стороной для функции `FuncB()`. Обратите внимание: функция `FuncA()` решает, что это исключение не имеет серьезного характера, и не передает его дальше функции `main()`. Следовательно, функция `main()` продолжит свою работу, как будто никакой проблемы нет. Если снять комментарий со строки 38, исключение будет передаваться вызывающей стороне, т.е. функция `main()` также получит его.

Вывод демонстрирует также порядок создания объектов (тот же порядок, в котором они расположены в коде) и их уничтожения при передаче исключения (обратный порядку создания объектов). Это происходит не только в функции `FuncB()`, где было передано исключение, но также и в функции `FuncA()`, которая вызвала функцию `FuncB()` и обработала исключение.

ВНИМАНИЕ!

В листинге 28.4 показан порядок вызова деструкторов локальных объектов при передаче исключения.

Если деструктор объекта, вызванный в связи с исключением, также передал исключение, то это приведет к аварийному завершению вашего приложения.

Класс `std::exception`

При обработке исключения `std::bad_alloc` в листинге 28.2 вы фактически обрабатывали объект класса `std::bad_alloc`, переданный оператором `new`. Класс `std::bad_alloc` происходит от стандартного класса C++ `std::exception`, объявленного в заголовке `<exception>`.

Класс `std::exception` является базовым для следующих классов важных исключений.

- `bad_alloc`. Передается при неудаче резервирования памяти оператором `new`.
- `bad_cast`. Передается оператором `dynamic_cast` при попытке приведения неправильного типа (типа без отношений наследования).
- `ios_base::failure`. Передается функциями и методами библиотеки `iostream`.

Класс `std::exception` является базовым классом, предоставляющим очень полезный и важный виртуальный метод, `what()`, возвращающий более описательную информацию о причине и природе проблемы, вызвавшей исключение. Функция `exp.what()`

в строке 18 листинга 28.2 предоставляет информацию "bad allocation", сообщая о том, что резервирование памяти потерпело неудачу. Вы можете использовать класс std::exception, являющийся базовым классом для многих типов исключений, и создать еще один, catch(const exception&), способный обрабатывать все исключения, для которых класс std::exception является базовым:

```
void SomeFunc()
{
    try
    {
        // код, обеспечивающий устойчивость к исключениям
    }
    catch (const std::exception& exp) // обработать bad_alloc,
                                      // bad_cast и т.д.
    {
        cout << "Exception encountered: " << exp.what() << endl;
    }
}
```

Ваш собственный класс исключения, производный от класса std::exception

Вы можете передать исключение любого типа, какой пожелаете. Однако в наследовании от класса std::exception есть преимущество — все существующие обработчики для исключений catch(const std::exception&), для исключений bad_alloc, bad_cast и т.п. также работают и автоматически масштабируются до обработчика вашего нового класса исключения, поскольку они имеют 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, необходимый Reason
10:     CustomException(const char* why):Reason(why) {}
11:
12:     // переопределение виртуальной функции для возвращения 'Reason'
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: \
                                   Dividing by 0 is a crime");
23:
24:     return (Dividend / Divisor);
25: }
26:
27: int main()
28: {
29:     cout << "Enter dividend: ";
30:     double Dividend = 0;
31:     cin >> Dividend;
32:     cout << "Enter divisor: ";
33:     double Divisor = 0;
34:     cin >> Divisor;
35:     try
36:     {
37:         cout << "Result of division is: "
                 << Divide(Dividend, Divisor);
38:     }
39:     catch(exception& exp) // обрабатывает CustomException,
                              // bad_alloc и т.д.
40:     {
41:         cout << exp.what() << endl;
42:         cout << "Sorry, can't continue!" << endl;
43:     }
44:
45:     return 0;
46: }
```

Результат

```
Enter dividend: 2011
Enter divisor: 0
CustomException: Dividing by 0 is a crime
Sorry, can't continue!
```

Анализ

Это версия листинга 28.3, который передавал простое исключение типа `char*` при делении на нуль. Теперь, однако, создается 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, по существу, возвращая причину передачи исключения. Логика обработчика `catch(exception&)` в функции `main()` (строки 39–43) обрабатывает не только исключения класса `CustomException`, но и других типов, например `bad_alloc`, у которых тот же базовый 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 функцией `throw()`, означая, что данная функция сама, как ожидается, не передаст исключения — это очень важное и уместное ограничение для класса, объект которого используется как исключение. Если вы все же вставите оператор `throw` в эту функцию, то можете ожидать предупреждение от компилятора.

Если объявление функции заканчивается `throw(int)`, то это значит, что функция ожидает передачи исключения типа `int`.

РЕКОМЕНДУЕТСЯ	НЕ РЕКОМЕНДУЕТСЯ
Помните об обработке исключения типа `std::exception`	**Не передавайте** исключения из деструкторов
Помните о наследовании собственного специального класса исключения (и любого другого) от класса `std::exception`	**Не считайте** успех резервирования памяти само собой разумеющимся; код, применяющий оператор `new`, всегда следует заключать в блок `try` и создавать соответствующий обработчик
Передавайте исключения осмотрительно. Они не замена для возвращения таких значений, как `true` или `false`	**Не вставляйте** сложную логику или резервирование ресурсов в блок `catch()`. Нельзя передавать исключения при их обработке

Резюме

На этом занятии вы узнали очень важную часть практического программирования C++. Создание приложений, стабильных вне собственной среды разработки, важно для клиента, как и интуитивно понятные пользователям сообщения. Это именно то, что позволяют сделать исключения. Вы узнали, что код, который резервирует ресурсы или память, может подвести, а следовательно, он требует обработки исключения. Вы узнали, что язык C++ обладает классом исключения `std::exception`, а при необходимости создать собственный специальный класс исключения имеет смысл наследовать его.

Вопросы и ответы

- **Почему нужно передавать исключение, а не возвращать ошибку?**

  Не всегда есть право возвращать ошибку. При неудаче оператора `new` необходимо обработать переданное им исключение и воспрепятствовать отказу вашего приложения. Кроме того, если ошибка очень серьезная и делает невозможным последующее функционирование вашего приложения, следует обратить внимание на передачу исключений.

- **Почему мой класс исключения должен происходить от класса `std::exception`?**

  Это, конечно, не обязательно, но позволит вам многократно использовать все блоки `catch()`, которые уже обрабатывают исключения типа `std::exception`. Вы можете написать собственный класс исключения, который не

происходит ни от чего, но впоследствии вам придется добавить новые обработчики `catch(MyNewExceptionType&)` во всех соответствующих пунктах.

- **У меня есть функция, которая передает исключение. Должна ли она обрабатываться в той же функции?**

  Нет. Следует только удостовериться, что исключение передаваемого типа обработает одна из вызывающих функций выше в стеке вызовов.

- **Может ли передача исключения осуществляться из конструктора?**

  У конструкторов фактически нет выбора! У них нет возвращаемых значений, и передача исключения — наилучший способ оповещения о проблеме.

- **Может ли передача исключения осуществляться из деструктора?**

  Технически — да. Но это очень плохая идея, поскольку именно деструкторы и вызываются при прокрутке стека в связи с исключением. Если деструктор, вызванный в связи с исключением, сам передаст исключение, то может возникнуть весьма некрасивая ситуация, когда состояние приложения, пытающегося осуществить корректный выход, и так уже нестабильно.

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 Г, "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. Какое исключение передается при неудаче резервирования памяти оператором `new`?
3. Хороша ли идея зарезервировать в обработчике исключения (блок `catch`) место под миллион целых чисел для резервного копирования существующих данных, например?
4. Как вы обработали бы объект исключения класса `MyException`, происходящего от класса `std::exception`?

Упражнения

1. **Отладка:** Что не так со следующим кодом?

```
class SomeIntelligentStuff
{
    bool StuffGoneBad;
public:
    ~ SomeIntelligentStuff()
    {
```

```
        if(StuffGoneBad)
            throw "Big problem in this class, just FYI";
    }
};
```

2. **Отладка:** Что не так со следующим кодом?

```
int main()
{
    int* pMillionIntegers = new int [1000000];
    // сделать нечто с миллионом целых чисел

    delete []pMillionIntegers;
}
```

3. **Отладка:** Что не так со следующим кодом?

```
int main()
{
    try
    {
    int* pMillionIntegers = new int [1000000];
    // сделать нечто с миллионом целых чисел

    delete []pMillionIntegers;
    }
    catch(exception& exp)
    {
        int* pAnotherMillionIntegers = new int [1000000];
        // взять резервную копию pMillionIntegers и сохранить
        // ее на диске
    }
}
```

ЗАНЯТИЕ 29
Что дальше

Вы изучили основы программирования на языке C++. Фактически мы вышли за границы теоретических понятий, изучив стандартную библиотеку шаблонов (STL), шаблоны и то, как библиотека STL способна помочь вам писать эффективный и компактный код. Пришло время обратить внимание на производительность и получить несколько полезных советов по программированию.

На сегодняшнем занятии.

- Чем отличаются современные процессоры.
- Как приложение C++ может лучше использовать возможности процессора.
- Потоки и многопоточность.
- Полезные советы по программированию на C++.
- Как повышать свою квалификацию после прочтения этой книги.

ЗАНЯТИЕ 29
Что дальше

Вы изучили основы программирования на языке C++. Фактически мы вышли за границы теоретических понятий, изучив стандартную библиотеку шаблонов (STL), шаблоны и то, как библиотека STL способна помочь вам писать эффективный и компактный код. Пришло время обратить внимание на производительность и получить несколько полезных советов по программированию.

На сегодняшнем занятии.

- Чем отличаются современные процессоры.
- Как приложение C++ может лучше использовать возможности процессора.
- Потоки и многопоточность.
- Полезные советы по программированию на C++.
- Как повышать свою квалификацию после прочтения этой книги.

Чем отличаются современные процессоры

За последнее время процессоры компьютеров стали быстрее, их скорости обработки измеряются уже на в Герцах (Гц) и мегагерцах (МГц), а в гигагерцах (ГГц). Например, процессор Intel 8086 (рис. 29.1), выпущенный в 1978, был 16-разрядным и работал с тактовой частотой примерно 10 МГц.

РИС. 29.1. Микропроцессор Intel 8086

Процессоры в наши дни стали значительно быстрее, а что же приложения C++? Проще всего было бы положиться на постоянно улучшающиеся аппаратные средства и получить рост производительности программного обеспечения за счет повышения их быстродействия. Хотя современные процессоры становятся быстрее, истинное новаторство кроется в количестве ядер, которыми они обладают. На момент написания этой книги корпорация Intel уже продавала 64-битовый микропроцессор с шестью встроенными ядрами на 3,2 ГГц. Современный многоядерный процессор представлен на рис. 29.2. Фактически даже смартфоны уже обладают процессорами с несколькими ядрами.

РИС. 29.2. Многоядерный микропроцессор Intel

Многоядерный процессор можно считать одной микросхемой с несколькими процессорами, работающими параллельно. Каждый процессор имеет собственный кеш L1 и способен работать независимо от другого.

Чем быстрее процессор, тем выше скорость выполнения вашего приложения, что вполне логично. Но чем поможет несколько ядер процессора? Вполне очевидно, что каждое ядро способно выполнять приложения параллельно, но это не заставляет ваше приложение

работать немного быстрее. Однопотоковые приложения C++, которые вы видели до сих пор, возможно, и не извлекут выгоды от использования многоядерных систем. Такие приложения выполняются в одном потоке, а следовательно, только на одном ядре, как показано на рис. 29.3.

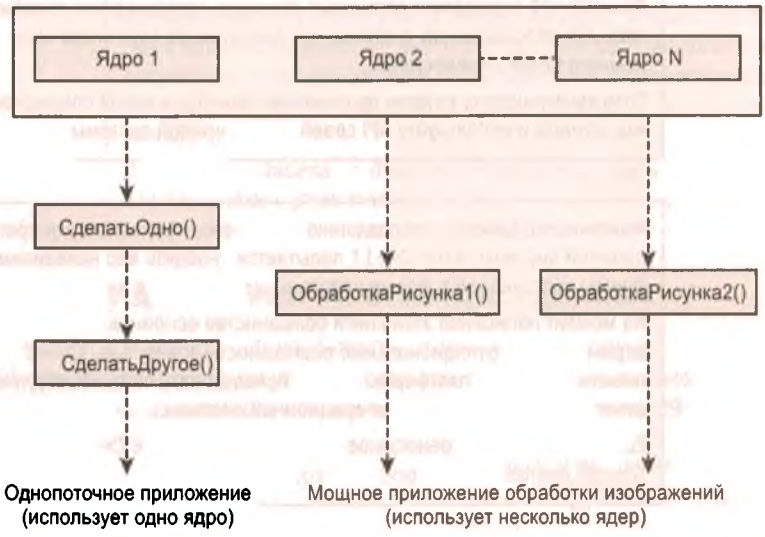


РИС. 29.3. Однопоточное приложение на многоядерном процессоре

Если ваше приложение выполнит все действия последовательно, то операционная система, возможно, предоставит ему в очереди столько же времени, сколько и другим приложениям, и оно займет на процессоре только одно ядро. Другими словами, ваше приложение выполняется на многоядерном процессоре таким же образом, как и многие годы назад.

Как лучше использовать несколько ядер

Выход — в создании многопоточных приложений, у которых каждый поток выполняется параллельно, позволяя операционной системе запускать потоки на нескольких ядрах. Хотя обсуждение потоков и многопоточности выходит за рамки этой книги, я вполне могу затронуть эту тему и дать вам начальное представление о высокопроизводительных приложениях.

Что такое поток

Код приложения всегда работает в потоке. *Поток* (thread) — это синхронная сущность выполнения, когда операторы выполняются один за другим. Код функции `main()`, как полагается, выполняет основной поток приложения. В этом основном потоке можно создать новые потоки, способные выполняться параллельно. Такие приложения, состоящие из одного или нескольких потоков, выполняющихся параллельно основному потоку, называются *многопоточными приложениями* (multithreaded application).

Операционная система предписывает, что потоки должны быть созданы, и вы можете создать их непосредственно, вызвав соответствующие функции API, предоставляемые операционной системой.

СОВЕТ

Язык C++11 определяет потоковые функции, которые сами позаботятся о вызове API операционной системы, что делает ваше многопоточное приложение немного более переносимым.

Если вы планируете создать приложение только для одной операционной системы, изучите и используйте API своей операционной системы.

ПРИМЕЧАНИЕ

Фактические действия по созданию потока специфичны для конкретной операционной системы. Язык C++11 попытается снабдить вас независимой от платформы абстракцией в форме класса `std::thread` в заголовке `<thread>`.

На момент написания этой книги большинство основных компиляторов не поддерживали эти функциональные возможности полностью. Кроме того, если вы пишете для одной платформы, лучше использовать потоковые функции, специфичные только для данной операционной системы.

Если необходимо переносимое поточное приложение C++, обратитесь к Boost Thread Libraries на `www.boost.org`.

Зачем создавать многопоточные приложения

Многопоточность используется в приложениях, которые должны осуществлять несколько определенных действий параллельно. Предположим, что от 1 до 10 000 пользователей пытаются осуществить покупку на Amazon.com. Конечно, веб-сервер Amazon.com не может заставить ждать 9 999 пользователей, пока один что-то покупает. Веб-сервер создают несколько потоков, обслуживая пользователей одновременно. Если веб-сервер выполняется на многоядерном процессоре (держу пари, что так и есть), потоки в состоянии извлечь преимущество из наличия доступных ядер и обеспечить оптимальную производительность каждому пользователю.

Еще одним общеизвестным примером многопоточности является прогресс-индикатор, который взаимодействует с пользователем, пока приложение осуществляет некую 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 на заднем плане осуществляет работу. К таким приложениям относятся инструменты дефрагментации диска. После запуска такого приложения создается рабочий поток, начинающий просмотр и дефрагментацию диска. Одновременно поток пользовательского интерфейса отображает прогресс процесса, предоставляя пользователю возможность отменить дефрагментацию. Чтобы поток пользовательского интерфейса мог отображать прогресс, осуществляющий дефрагментацию, рабочий поток должен регулярно сообщать ему об этом. Точно так же, чтобы рабочий поток узнал об отмене работы, 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 друг от друга, многопоточные приложения нуждаются в средстве "общения" потоков друг с другом.

Последовательность также важна. Вы ведь не хотите, чтобы поток пользовательского интерфейса закончил работу прежде, чем рабочий поток закончит дефрагментацию. Нередки ситуации, когда один поток должен ждать другой. Например, поток чтения из базы данных должен переждать поток, осуществляющий запись.

Действия по организации ожидания потоками друг друга 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. 29.4.

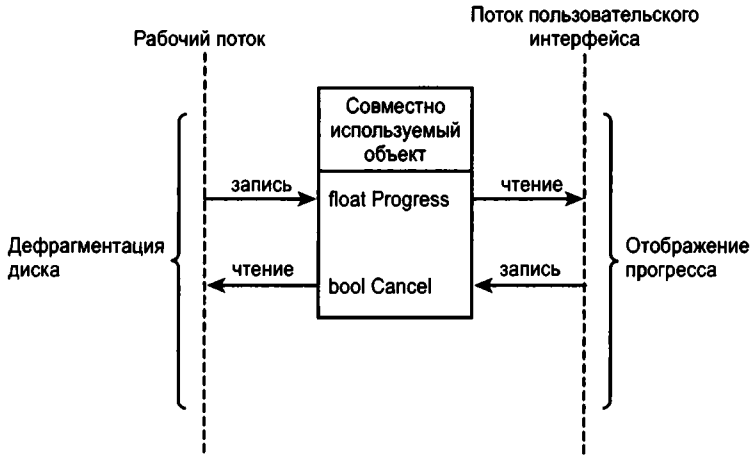


РИС. 29.4. Рабочий поток и поток пользовательского интерфейса совместно используют данные

Потоки могут общаться при записи и чтении данных, хранящихся в некой области памяти, к которой они способны обращаться, а следовательно, использовать совместно. В примере дефрагментации, где рабочему потоку известен прогресс, а потоку пользовательского интерфейса необходимо отображать его, рабочий поток может регулярно сохранить значение процента прогресса в целочисленной переменной, используемой потоком пользовательского интерфейса для отображения.

Это довольно простой случай: один поток создает информацию, а другой использует ее. Но что будет, если писать и читать из той же области будет несколько потоков? Некоторые потоки могли бы начать читать данные, когда другие еще не закончили писать их. Целостность данных оказалась бы 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 действий по синхронизации потоков *семафоры* (semaphore) и *мьютексы* (mutex).

Использование мьютекса обычно гарантирует доступ к части кода только одного потока за раз. Другими словами, мьютекс используется для организации раздела кода, перед выполнением которого поток должен подождать, пока другой поток не закончит его выполнение и не освободит мьютекс. Когда следующий поток приобретает мьютекс, он выполнит свою задачу и также освободит его.

Используя семафоры, можно контролировать количество потоков, которые выполняют раздел кода. Семафор, разрешающий доступ только одному потоку за раз, называется также *бинарным семафором* (binary semaphore).

ПРИМЕЧАНИЕ

В зависимости от используемой операционной системы вы могли бы иметь только эти или еще и другие объекты синхронизации. Операционная система Windows, например, предоставляет критические разделы, позволяющие выполнять участок кода только одному потоку за раз.

Проблемы, вызванные многопоточностью

Многопоточность, с ее потребностью в хорошей синхронизации потоков, способна стать причиной множества бессонных ночей, когда эффективность синхронизации оказывается недостаточной (читай: ошибочной). Вот две наиболее распространенные проблемы, с которыми сталкиваются многопоточные приложения.

- *Состояние гонки* (race condition). Два потока или более пытаются записать те же данные. Кто победит? Каково состояние этого объекта?
- *Взаимоблокировка* (deadlock). Два потока ожидают завершения друг друга, и оба находятся в состоянии ожидания. В результате приложение зависает.

При хорошей синхронизации вы можете избежать состояния гонки. Обычно, когда потокам позволено писать в совместно используемый объект, необходимо дополнительно позаботиться о следующем:

- одновременно может писать только один поток;
- никакому потоку не позволено читать, пока не закончится запись в объект.

Вы можете избежать взаимоблокировки, устранив ситуации, когда два потока вынуждены ожидать друг друга. У вас может быть главный поток, который синхронизирует рабочие потоки, или программа может действовать таким образом, что задачи распределяются между потоками, в результате чего достигается распределение рабочей нагрузки. Поток А может ждать поток В, но поток В никогда не должен ждать поток А.

Разработка многопоточных приложений сама по себе является специальностью. Следовательно, 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. Как только вы овладеете ими, сможете позиционировать свои приложения C++ как оптимальные для использования многоядерных процессоров, за которыми будущее.

Как писать отличный код C++

Язык C++ не только значительно развился со времени первого выпуска, усилия по стандартизации, предпринятые главными изготовителями компилятора, а также доступность утилит и функций помогут вам писать компактный и ясный код C++. Писать читаемые и надежные приложения C++ действительно просто.

Ниже приведены полезные советы, которые помогут создавать хорошие приложения C++.

- Присвойте переменным осмысленные имена (понятные не только вам).
- Всегда инициализируйте такие переменные, как `int`, `float`, и подобные им.
- Всегда инициализируйте указатели либо значением NULL, 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*` стиля C и такие функции, как `strlen()` и `strcpy()`. Тип `std::string` более безопасен и предоставляет много полезных вспомогательных методов, включая такие, которые позволяют находить длину, копировать и добавлять.
- Используйте статические массивы только тогда, когда абсолютно уверены в количестве элементов, которые они будут содержать. Если вы не уверены в этом, используйте динамический массив, такой как `std::vector`.
- При объявлении и определении функций, получающих параметры типов, отличных от POD (простые старые данные), постарайтесь избежать ненужного этапа копирования, передавая их при вызове функции по ссылке.
- Если ваш класс содержит член (или члены) в виде простого указателя, обдумайте принадлежность ресурса в памяти и управление им в случае копирования и присвоения. Таким образом, рассмотрите возможность создания конструктора копий и оператора присвоения копии.
- При написании вспомогательного класса, управляющего динамическим массивом или подобным, для повышения производительности не забудьте добавить конструктор перемещения и оператор присваивания при перемещении.
- Не забывайте об использовании констант. В идеале функция `get()` не должна изменять члены класса, а следовательно, должна быть константой. Точно так же параметры функций должны быть константными ссылками, если вы не хотите изменять значения, которые они содержат.
- Избегайте использования простых указателей. Используйте подходящие интеллектуальные указатели везде, где только можно.
- При создании вспомогательного класса не пожалейте усилий для поддержки всех тех операторов, которые сделают использование вашего класса проще.

- По возможности отдавайте предпочтение шаблону, а не макросу. Шаблоны безопасны и обобщены.
- При разработке класса, объекты которого будут храниться в контейнере, таком, как вектор или список, или использоваться как ключ карты, не забывайте предоставить оператор `operator<`, позволяющий определить заданный по умолчанию критерий сортировки.
- Если ваша лямбда-функция становится слишком большой, возможно, имеет смысл создать объект функции ее класса с оператором `operator()`, поскольку функтор обеспечивает повторное применение, а также единый пункт обслуживания.
- Никогда не будьте уверены в успехе оператора `new`. Код резервирования ресурсов всегда может передать исключение, поэтому заключайте его в блок `try` и предоставляйте соответствующий блок `catch()`.
- Никогда не используйте оператор `throw` в деструкторе класса.

Изучение C++ на этом не заканчивается

Поздравляю, вы достигли больших успехов в изучении языка C++. Наилучший способ постоянно повышать квалификацию — это практика, и еще раз практика!

Язык C++ довольно сложен. И чем больше вы программируете, тем выше ваш уровень понимания того, как он работает внутренне. Такие среды разработки, как Visual Studio, со средством intelli-sense помогут вам и удовлетворят ваше любопытство, например, показав члены строкового класса, которых вы еще не видели. Пришло время учиться на практике!

Сетевая документация

Если необходимо узнать больше о сигнатурах контейнеров STL, их методах или алгоритмах, а также иные подробности, обратитесь к библиотеке MSDN (`http://msdn.microsoft.com/`), где стандартная библиотека шаблонов описана очень хорошо.

СОВЕТ

> При чтении документации STL в библиотеке MSDN не забудьте выбирать подходящую версию Visual Studio, стандарт C++11 поддерживается лишь с версии Visual Studio 2010.

На момент написания этой книги основные компиляторы C++ не поддерживали полностью все средства C++11. Например, среда разработки Visual Studio 2010 не поддерживает шаблоны `variadic`. У компилятора GCC от GNU версии 4.6 реализация класса `std::thread` ошибочна. Имеет смысл ознакомиться с сетевой документацией вашего компилятора и выяснить, какие средства C++11 он уже поддерживает или планирует поддерживать. Группа Visual Studio поддерживает блог "C++11 Core Language Feature Support", который вы можете посетить по адресу `http://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx`, а у пользователей GCC есть страница поддержки по адресу `http://gcc.gnu.org/projects/cxx0x.html`.

ПРИМЕЧАНИЕ

На момент написания данной книги эти два компилятора уже поддерживали основной объем средств, рекомендованных стандартом C++11. Будьте уверены, что фрагменты кода книги были проверены с использованием их обоих.

Сетевые сообщества и помощь

У языка C++ богатые и яркие сетевые сообщества. Зарегистрируйтесь на таких сайтах, как CodeGuru (`www.CodeGuru.com`) и CodeProject (`www.CodeProject.com`), чтобы задавать технические вопросы и получать ответы от сообщества.

Когда почувствуете себя уверенно, не стесняйтесь способствовать этим сообществам. Вы ответите на спорные вопросы и научитесь многому в процессе общения.

Резюме

Это заключительное занятие — фактически начало самостоятельного изучения C++! Зайдя так далеко, вы изучили основные и дополнительные концепции языка. На сегодняшнем 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 знаете проблемы многопоточных приложений и способы их решения. И наконец, но не в последнюю очередь, вы изучили некоторые из основных полезных советов по программированию на языке C++. Вы знаете, что для написания хорошего кода C++ нужно не только использовать передовые концепции, но и присваивать переменным имена, которые понятны другим, что обрабатывая исключения, нужно позаботиться о непредвиденном, что можно использовать такие классы, как интеллектуальные указатели, вместо простых указателей. Теперь вы готовы перейти к профессиональному программированию C++.

Вопросы и ответы

- **Я вполне доволен производительностью моего приложения. Должен ли я все же реализовать многопоточность?**

  Нет. Не все приложения должны быть многопоточными. Она нужна только при необходимости выполнения нескольких задач одновременно или для обслуживания множества пользователей.

- **Основные компиляторы еще не поддерживают стандарт C++11 полностью. Так почему бы не использовать старый стиль программирования?**

  Два главных компилятора (Visual C++ от Microsoft и GCC от GNU) поддерживают большинство основных средств C++11, за исключением лишь некоторых. Кроме того, C++11 существенно упрощает программирование. Такие ключевые слова, как `auto`, избавляют вас от долгих и утомительных объявлений итераторов, а лямбда-функции делают ваши конструкции `for_each()` компактней и без объектов функций. Поэтому преимущества в программировании на C++11 уже существенны.

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 Г, "Ответы". Если остались неясными хотя бы некоторые из предложенных ниже вопросов, не приступайте к изучению материала следующего занятия.

Контрольные вопросы

1. Мое приложение обработки изображения перестает отвечать при исправлении контраста. Что мне делать?
2. Мое многопоточное приложение обеспечивает чрезвычайно быстрый доступ к базе данных. Но все же иногда я замечаю, что выбранные данные искажены. Что я делаю неправильно?

ЧАСТЬ VI
Приложения

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 индивидуальных 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 равны 1).

ПРИМЕЧАНИЕ

В десятичной системе счисления самыми важными являются степени числа 10. Цифры в числе умножаются на 10, 100, 1000 и т.д., 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 (двоичное) = $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5$ (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 т.д., 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 проведены степени числа два.

ТАБЛИЦА А.1. Степени числа 2

Степень	Значение	Двоичное представление
0	$2^0 = 1$	1
1	$2^1 = 2$	10
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, 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 напряжения) или как выключенное (при отсутствии разницы потенциалов или напряжения).

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 NOT, AND, OR и XOR, рассматривавшиеся на занятии 5, "Команды, выражения 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 кратно 2. Кроме того, восемь битов в байте позволяют передать до 2^8 (255) различных значений. Этих 255 индивидуальных значений достаточно для представления всех символов в наборе символов ASCII.

Сколько байт 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 быть 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:

$0x31F = 3 \times 16^2 + 1 \times 16^1 + F \times 16^0 = 3 \times 256 + 16 + 15$ (десятичное) = 799.

ПРИМЕЧАНИЕ По соглашению 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 числа от 0 до 7.

Преобразование в различные системы счисления

При работе с числами может возникнуть необходимость в просмотре 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 назначения. Полученный остаток заполняет места в цифровой системе назначения, начиная с самого низкого места. Следующее деление использует частное предыдущей операции деления с основанием в качестве делителя.

Так продолжается до тех пор, пока остаток в пределах цифровой системы назначения и частное не достигнет 0.

Этот процесс также вызывается *методом разбиения* (breakdown method).

Преобразование десятичного числа в двоичное

Чтобы преобразовать десятичные 33 в двоичное, вычитайте из него са высокую из возможных степеней числа 2 (32):
Знакоместо 1: 33 / 2 = частное 16, остаток 1
Знакоместо 2: 16 / 2 = частное 8, остаток 0
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): 100001
———————————————————————————
Аналогично 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 = частное0, остаток 1
———————————————————————————
Двоичный эквивалент числа 156: 10011100
———————————————————————————

Преобразование десятичного числа в шестнадцате

Процесс тот же, что и при преобразовании в двоичное число, но деление ется на основание 16, а не 2.

Преобразование десятичного числа 5211 в шестнадцатеричное:
Знакоместо 1: 5211 / 16 = частное 325, остаток B_{16} (11_{10} 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, н стую программу, подобную листингу 27.1 из занятия 27, "Примен для ввода и вывода". Она использует объект `std::cout` с манипу отображения целых чисел в шестнадцатеричной, десятичной и в формах записи.

Чтобы отобразить целое число в двоичном формате, испол `std::bitset`, который был описан на занятии 25, "Работа с би гами при использовании библиотеки STL". Черпайте вдохновени га 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.

```
False                    signed
Final                    sizeof
Float                    static
For                      static_assert
Friend                   static_cast
goto                     struct
if                       switch
inline                   template
int                      this
long                     throw
long long int            true
mutable                  try
namespace                typedef
new                      typeid
operator                 typename
override                 union
private                  unsigned
protected                using
public                   virtual
register                 void
reinterpret_cast         volatile
return                   wchar_t
short                    while
```

следующие ключевые слова также зарезервированы:

```
compl                    or_eq
not                      xor
not_eq                   xor_eq
or
```

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 табл. В.1, которого придерживается компилятор C++ во избежание двусмысленности.

ТАБЛИЦА В.1. Приоритет операторов

№	Название	Оператор
1	Область видимости	`::`
2	Прямое и косвенное обращение к члену класса, вызов функции, постфиксный инкремент и декремент	`. -> () ++ --`
3	Префиксный инкремент и декремент, инверсия и не унарные минус и плюс, получение адреса и ссылки, а также операторы `new`, `new[]`, `delete`, `delete[]`, `casting`, `sizeof()`	`++ -- ^ ! - + & * ()`
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 `AND`	`&`
11	Побитовое исключающее `OR`	`^`
12	Побитовое `OR`	`|`
13	Логическое `AND`	`&&`
14	Логическое `OR`	`||`
15	Троичный условный оператор	`?:`
16	Операторы присвоения	`= *= /= %= += -= <<= >>=`
17	Запятая	`,`

ПРИЛОЖЕНИЕ Г

Ответы

Ответы к занятию 1

Контрольные вопросы

1. Интерпретатор — это инструмент, который интерпретирует исходный код (или промежуточный бинарный код) и выполняет определенные действия. Компилятор получает на входе исходный код и создает объектный файл. В языке C++ после компиляции и компоновки получается исполняемый 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 исполняемого файла, который интегрирует все явные и неявные зависимости, является задачей компоновщика.
3. Код. Компиляция для создания объектного файла. Компоновка для создания исполняемого файла. Выполнение для проверки. Отладка. Устранение ошибок в коде и повторение предыдущих этапов. В большинстве случаев компиляция и компоновка — это один этап.
4. Стандарт C++11 поддерживает переносимую потоковую модель, позволяющую разработчику создавать многопоточные приложения, используя стандартные поточные функции C++. Таким образом, он позволяет многоядерному процессору работать оптимально при одновременном выполнении различных потоков в приложении на его нескольких ядрах.

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 начаться 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.

```
/* если вы пишете комментарий, используя такой синтаксис в стиле C,
   то можете расположить его в нескольких строках */
```

Упражнения

1. Причина неудачи в чувствительности к регистру у 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 подобна листингу 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 << "Performing subtraction 10 - 5 = " << 10 - 5 << endl;
    cout << "Performing multiplication 10 * 5 = " << 10 * 5 << endl;

    return 0;
}
```

Результат

```
Performing subtraction 10 - 5 = 5
Performing multiplication 10 * 5 = 50
```

Ответы к занятию 3

Контрольные вопросы

1. В знаковом целом числе самый старший разряд означает знак числа (плюс или минус). Знаковое целое число, напротив, используется только для положительных значений.
2. Директива препроцессора `#define` инструктирует компилятор осуществить глобальную текстовую замену указанного значения. Однако это не учитывает

безопасности типов и является примитивным способом определения констант. Поэтому его 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
// Или так..
enum YOURCARDS {ACE, JACK, QUEEN = 45, KING};
// ACE = 0, JACK = 1, QUEEN = 45, a KING = 46
```

2. Посмотрите код листинга 3.4 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 << "Enter circle's radius: ";
    double Radius = 0;
    cin >> Radius;

    cout << "Area = " << Pi * Radius * Radius << endl;
    cout << "Circumference = " << 2 * Pi * Radius << endl;

    return 0;
}
```

Результат

```
Enter circle's radius: 4
Area = 50.2656
Circumference = 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

```
Enter circle's radius: 4
Area = 50
Circumference = 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 отказу 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 на часть 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
    {
        NOTHING = 0,
        PAWN,
        ROOK,
        KNIGHT,
        BISHOP,
        KING,
        QUEEN
    };

    SQUARE ChessBoard[8][8];
    // Инициализировать клетки, содержащие ладьи
    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 присвоения значения. Вывод будет непредсказуем. Всегда инициализируйте переменные и массивы; в противном случае они будут содержать последнее значение, хранившееся в зарезервированной 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 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 у оператора сдвига, что приведет к сдвигу на 1 + 5 = 6 битов, а не на один бит.
6. Результатом операции XOR будет `false`, согласно табл. 5.5.

Упражнения

1. Вот правильное решение:

```
int Result = ((number << 1) + 5) << 1; // теперь вполне 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 << "Enter a boolean value true(1) or false(0): ";
    bool Value1 = false;
    cin >> Value1;

    cout << "Enter another boolean value true(1) or false(0): ";
    bool Value2 = false;
    cin >> Value2;

    cout << "Result of bitwise operators on these operands: " << endl;
    cout << "Bitwise AND: " << (Value1 & Value2) << endl;
    cout << "Bitwise OR: " << (Value1 | Value2) << endl;
```

```
    cout << "Bitwise XOR: " << (Value1 ^ Value2) << endl;

    return 0;
}
```

Результат

```
Enter a boolean value true(1) or false(0): 1
Enter another boolean value true(1) or false(0): 0
Result of bitwise operators on these operands:
Bitwise AND: 0
Bitwise OR: 1
Bitwise XOR: 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 решении упражнения 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 6.13, 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 MyInts1[ARRAY1_LEN] = {35, -3, 0};
    int MyInts2[ARRAY2_LEN] = {20, -1};

    cout << "Adding each int in MyInts1 by each in MyInts2:" << endl;

    for(int Array1Index=ARRAY1_LEN-1;Array1Index>=0;--Array1Index)
        for(int Array2Index=ARRAY2_LEN-1;Array2Index>=0;--Array2Index)
            cout<<MyInts1[Array1Index]<<" + "<<MyInts2[Array2Index] \
               << " = " << MyInts1[Array1Index] + MyInts2[Array2Index]
               << endl;

    return 0;
}
```

Результат

```
Adding each int in MyInts1 by each in 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 спрашивает у пользователя следующее:

```
cout << "How many Fibonacci numbers you wish to calculate: ";
int NumsToCal = 0;
cin >> NumsToCal;
```

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 << "Here are the available colors: " << 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 << "Choose one by entering code: ";
    int YourChoice = BLUE; // исходное
    cin >> YourChoice;

    switch (YourChoice)
    {
    case VIOLET:
    case INDIGO:
    case BLUE:
    case GREEN:
    case YELLOW:
    case ORANGE:
    case RED:
        cout << "Bingo, your choice is a Rainbow color!" << endl;
        break;

    default:
        cout << "The color you chose is not in the rainbow" << endl;
        break;
    }

    return 0;
}
```

Результат

```
Here are the available colors:
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
Choose one by entering code: 4
Bingo, your choice is a Rainbow color!
```

5. В выражении условия выхода из цикла `for` программист по невнимательности осуществил присвоение значения 10 счетчику, а не сравнение.
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 содержит не копию значения.
3. Рекурсивная функция.
4. Перегруженные функции.
5. На вершину! Стек похож на стопку тарелок; ту, что 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. Параметр со значением по умолчанию должен либо располагаться в конце, либо значения по умолчанию нужно определить для всех параметров.

5. Функция должна возвратить 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 изменять их. 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 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 резервирование 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 начаться 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 (помните, 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 Пи не предоставляется 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 << "Enter a radius: ";
    double Radius = 0;
    cin >> Radius;

    Circle MyCircle(Radius);

    cout << "Circumference = " << MyCircle.GetCircumference() << endl;
    cout << "Area = " << 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, принадлежащая производному классу, отсекается, и только часть, соответствующая базовому классу, передается по значению. Результат может быть непредсказуемым.
3. Композиция делает проект гибче.
4. Позволяет показать 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 `SubDerived`, а следовательно, они недоступны.

Упражнения

1. Конструкторы вызываются в порядке объявления: `Mammal - Bird - Reptile - Platypus`. Удаление осуществляется в обратном порядке. Программа показана ниже.

```
#include <iostream>
using namespace std;

class Mammal
{
public:
    void FeedBabyMilk()
    {
        cout << "Mammal: Baby says glug!" << endl;
```

```
    }

    Mammal()
    {
        cout << "Mammal: Contructor" << endl;
    }
    ~Mammal()
    {
        cout << "Mammal: Destructor" << endl;
    }
};

class Reptile
{
public:
    void SpitVenom()
    {
        cout << "Reptile: Shoo enemy! Spits venom!" << endl;
    }

    Reptile()
    {
        cout << "Reptile: Contructor" << endl;
    }
    ~Reptile()
    {
        cout << "Reptile: Destructor" << endl;
    }
};

class Bird
{
public:
    void LayEggs()
    {
        cout << "Bird: Laid my eggs, am lighter now!" << endl;
    }

    Bird()
    {
        cout << "Bird: Contructor" << endl;
    }
    ~Bird()
    {
        cout << "Bird: Destructor" << endl;
    }
};

class Platypus: public Mammal, public Bird, public Reptile
{
public:
    Platypus()
    {
        cout << "Platypus: Contructor" << endl;
    }
    ~Platypus()
    {
        cout << "Platypus: Destructor" << endl;
```

```
    }
};

int main()
{
    Platypus realFreak;
    //realFreak.LayEggs();
    //realFreak.FeedBabyMilk();
    //realFreak.SpitVenom();

    return 0;
}
```

2. 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 отсечению, результат которого непредсказуем:

```
Derived objectDerived;
SomeFunc(objectDerived); // будет отсечение
```

Ответы к занятию 11

Контрольные вопросы

1. Объявите абстрактный класс `Shape` с чистыми виртуальными функциями `Area()` и `Print()`, это заставит классы `Circle` и `Triangle` реализовать их.
2. Нет. Это создает таблицу виртуальной функции только для тех классов, которые содержат виртуальные функции, включая производные классы.
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 says hello!" << 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 says hello!" << endl;
    }
};

int main()
{
    Circle myRing(5);
    Triangle myWarningTriangle(6.6, 2);
```

```
    cout << "Area of circle: " << myRing.Area() << endl;
    cout << "Area of triangle: " << 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 виртуальный деструктор будет вызван только один `~Car()`.

Ответы к занятию 12

Контрольные вопросы

1. Нет, язык C++ не позволяет двум функциям с тем же именем иметь разные возвращаемые значения. Вы можете создать две реализации оператора `[]` с идентичными типами возвращаемого значения, но один оператор определен как константная функция, и другой нет. В данном случае компилятор C++ выбирает не константную версию для действий, связанных с присвоением, и константную версию в противном случае:

```
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.
3. Поскольку у него нет никаких динамически распределенных ресурсов, содержавшихся в пределах класса `Date`, способных вызвать ненужные циклы резервирования и освобождения памяти в пределах конструктора копий или оператора присвоения копии, этот класс не является хорошим кандидатом на наличие конструктора перемещения или оператор 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* pIntegers;

public:
    // Конструктор перемещения
    DynIntegers(DynIntegers&& MoveSource)
    {
        pIntegers = MoveSource.pIntegers; // взять собственность
        MoveSource.pIntegers = NULL; // освободить источник
                                     // от собственности
    }

    // Оператор присваивания при перемещении
    DynIntegers& operator= (DynIntegers&& MoveSource)
    {
        if(this != &MoveSource)
        {
            delete [] pIntegers; // освободить собственные ресурсы
            pIntegers = MoveSource.pIntegers;
            MoveSource.pIntegers = NULL;
        }
        return *this;
    }

    ~DynIntegers() {delete[] pIntegers;} // Деструктор

    // реализовать стандартный конструктор,
    // конструктор копий, оператор присвоения
};
```

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 допустимость:

```
void DoSomething(Base* pBase)
{
    Derived* pDerived = dynamic_cast <Derived*>(pBase);
```

```
    if(pDerived) // проверка на допустимость
        pDerived->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 сработает только при
    // использовании допустимого 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. Вот он:

```
#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. Шаблон функции `swap()` будет таким:

```
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];}
};
```

Ответы к занятию 15

Контрольные вопросы

1. Контейнер `deque`. Только он обеспечивает вставку в начало и в конец контейнера при постоянной продолжительности.
2. Контейнеры `std::set` или `std::map`, если у вас пары "ключ–значение". Если элементы могут дублироваться, выберите контейнеры `std::multiset` или `std::multimap`.
3. Да. Когда вы создаете 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 (быть настроенным на) каждый возможный 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, насколько это возможно.

Упражнения

1. Программа должна использовать функцию `std::reverse()`:

```
#include <string>
#include <iostream>
#include <algorithm>

int main ()
{
    using namespace std;

    cout << "Please enter a word for palindrome-check:" << endl;
    string strInput;
    cin >> strInput;

    string strCopy (strInput);
    reverse (strCopy.begin (), strCopy.end ());

    if (strCopy == strInput)
        cout << strInput << " is a palindrome!" << endl;
    else
        cout << strInput << " is not a palindrome." << endl;

    return 0;
}
```

2. Используйте функцию `std::find()`:

```
#include <string>
#include <iostream>

using namespace std;

// Найдите 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 << "Please enter a string:" << 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');

    // DIY: заглавные буквы обработать также..

    cout << "The number of vowels in that sentence is: " << nNumVowels;

    return 0;
}
```

3. Используйте функцию `toupper()`:

```
#include <string>
#include <iostream>
#include <algorithm>

int main ()
{
    using namespace std;

    cout << "Please enter a string for case-conversion:" << 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 << "The string converted to upper case is: " << 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 << "The sentence reads:" << endl;
    cout << strResult;

    return 0;
}
```

Ответы к занятию 17

Контрольные вопросы

1. Нет, не могут. За постоянное время элементы могут быть только добавлены в конец вектора.
2. Еще 10. При 11-й вставке произойдет повторное резервирование.
3. Извлекает последний элемент; т.е. удаляет элемент с конца.
4. Типа `CMammal`.
5. При помощи оператора индексирования (`[]`) или функции `at()`.
6. Итератор прямого доступа.

Упражнения

1. Одно из решений таково:

```
#include <vector>
#include <iostream>

using namespace std;

char DisplayOptions ()
{
    cout << "What would you like to do?" << endl;
    cout << "Select 1: To enter an integer" << endl;
    cout << "Select 2: Query a value given an index" << endl;
    cout << "Select 3: To display the vector" << endl << "> ";
    cout << "Select 4: To quit!" << endl << "> ";

    char ch;
    cin >> ch;

    return ch;
}
int main ()
{
    vector <int> vecData;
```

```
    char chUserChoice = '\0';
    while ((chUserChoice = DisplayOptions ()) != '4')
    {
        if (chUserChoice == '1')
        {
            cout << "Please enter an integer to be inserted: ";
            int nDataInput = 0;
            cin >> nDataInput;

            vecData.push_back (nDataInput);
        }
        else if (chUserChoice == '2')
        {
            cout << "Please enter an index between 0 and ";
            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 << "The contents of the vector are: ";
            for (size_t nIndex = 0; nIndex < vecData.size (); ++ nIndex)
                cout << vecData [nIndex] << ' ';
            cout << endl;
        }
    }
    return 0;
}
```

2. Используйте алгоритм `std::find()`:

```
vector <int>::iterator iElementFound = std::find (vecData.begin (),
                                    vecData.end (), nDataInput);
```

3. Усовершенствуйте код решения упражнения 1, разрешив пользователю ввод и вывод содержимого вектора на экран.

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 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 они указали 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;

    // Вставить содержимое вектора в начало списка
    listIntegers.insert (listIntegers.begin (),
                         vecData.begin (), vecData.end());

    cout << "The contents of the list are: ";

    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 << "The contents of the list are: ";

    list <string>::const_iterator iElement;
    for (iElement = listNames.begin(); iElement!=listNames.end();
                            ++iElement)
        cout << *iElement << " ";
    cout << endl;

    cout << "The contents after reversing are: ";
    listNames.reverse ();
    for (iElement = listNames.begin(); iElement!=listNames.end();
                            ++iElement)
        cout << *iElement << " ";
    cout << endl;

    cout << "The contents after sorting are: ";
    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 возвратит значение `true`, если первое число меньше второго.
2. Рядом, один за другим.
3. Для всех контейнеров библиотеки STL это функция `size()`.

Упражнения

1. Бинарный предикат может быть таким:

```
struct FindContactGivenNumber
{
    bool operator () (const CContactItem& lsh,
```

```
                        const CContactItem& rsh) const
    {
        return (lsh.strPhoneNumber < rsh.strPhoneNumber);
    }
};
```

2. Структура и определение мультимножества могли бы быть такими:

```
#include <set>
#include <iostream>
#include <string>

using namespace std;

struct PAIR_WORD_MEANING
{
    string strWord;
    string strMeaning;

    PAIR_WORD_MEANING (const string& sWord, const string& sMeaning)
                        : strWord (sWord), strMeaning (sMeaning) {}
    bool operator< (const PAIR_WORD_MEANING& pairAnotherWord) const
    {
        return (strWord < pairAnotherWord.strWord);
    }
};

int main ()
{
    multiset <PAIR_WORD_MEANING> msetDictionary;
    PAIR_WORD_MEANING word1 ("C++", "A programming language");
    PAIR_WORD_MEANING word2 ("Programmer", "A geek!");

    msetDictionary.insert (word1);
    msetDictionary.insert (word2);

    return 0;
}
```

3. Одно из решений приведено ниже.

```
#include <set>
#include <iostream>

using namespace std;

template <typename T>
void DisplayContent (const T& sequence)
{
    T::const_iterator iElement;

    for (iElement = sequence.begin(); iElement!=sequence.end();
                                    ++iElement)
        cout << *iElement << " ";
}

int main ()
```