А. Пол

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Книга Айры Пола, автора многих популярных изданий по C и C++, не является ни учебником по C++, ни курсом по объектно-ориентированному программированию (ООП) «вообще», хотя может быть полезна и в этих двух качествах. Задача книги (точно отраженная в названии) совершенно конкретна: научить читателя писать на C++ объектно-ориентированные программы. Для многих абстрактных идей и понятий ООП в C++ существуют конкретные воплощающие их конструкции. В каждой главе автор вводит и объясняет очередную «порцию» таких конструкций, демонстрирует технику их эффективного использования. В некоторых случаях возникающие перед программистом на C++ проблемы не могут быть непосредственно решены средствами языка. Здесь на помощь приходят разнообразные приемы, не владея которыми трудно программировать на C++ реальные задачи. Такие приемы поясняются автором на многочисленных примерах, причем наиболее показательные программы подвергаются тщательному разбору.

Много внимания на страницах книги уделено самым последним дополнениям C++: стандартной библиотеке шаблонов (STL), пространствам имен (namespaces), механизму идентификации типов во время выполнения (RTTI), явным приведениям типов (cast-операторам) и другим.

Книга в первую очередь рассчитана на программистов, желающих получить ясное представление о парадигме объектно-ориентированного программирования в C++. Издание также будет полезно преподавателям, студентам и всем, кто хочет освоить объектно-ориентированное программирование на C++.

Содержание

Предметный указатель

Символы

Предисловие

Эта книга задумана как введение в объектно-ориентированное программирование (ООП) на языке C++ стандарта ANSI[1]. Она предназначена для читателей или студентов, которые уже имеют опыт программирования. Книга объясняет основные черты C++ в контексте ООП.

В последнее время язык C++ пополнился рядом нововведений. Среди них — стандартная библиотека шаблонов (Standard Template Library — STL), пространства имен (namespaces), идентификация типов на этапе выполнения (Run-Time Type Identification — RTTI) и булевский тип. Все эти новшества без труда могут быть использованы опытными программистами на базовом C++. Надо, однако, заметить, что в большинстве книг эти темы не затрагивались. Данное издание может стать удобным руководством в освоении указанных нововведений.

C++ был разработан в середине 80-х Бьерном Страуструпом из компании Bell Labs. Этот язык стал современным и могучим наследником языка C. C++ дополняет стандартный C концепцией *классов* — механизмом для создания типов данных, определяемых пользователем. Такие типы данных называют также *абстрактными типами данных*. Абстрактные типы данных в сочетании с возможностями наследования и связывания на этапе выполнения поддерживают концепцию ООП в C++.

Примеры, приведенные в книге и доступные на Web-узле издательства Addison-Wesley, призваны продемонстрировать хороший стиль программирования. На Web-узле *www.awl.com* кроме кода, имеющегося в книге, можно найти полезные дополнительные примеры. Тексты программ, упакованные в формате ZIP, можно получить, обратившись по адресу:

```
www.awl.com/cseng/titles/0-201-89550-1
```

или

```
ftp.aw.com/cseng/authors/pohl/OOPUCPP2e/opus2e.zip
```

Все программы, представленные в тексте, действительно работают. Благодаря их тщательному подбору и разбору книга представляет собой простое и полное введение в программирование на C++. Разбор — это анализ и объяснение всех новых эле-

[1] Текст книги приблизительно соответствует окончательному варианту стандарта языка (ISO/IEC 14882), ратифицированному в августе 1998 года. — *Примеч. перев.*

ментов в программе, которые обучаемый видит в первый раз. Такой разбор поясняет ключевые моменты во многих примерах рабочего кода, используемого для обучения.

Эта книга предназначена для первого этапа обучения программированию на C++. Она может быть использована как дополнение к углубленному курсу по программированию, курсу по структурам данных, курсу по методологии программного обеспечения, сравнительному курсу по языкам программирования или в других курсах, где преподаватель выберет C++ в качестве основного языка. В каждой главе предлагается несколько программ, которые тщательно объясняются. Многие программы и функции детально разбираются.

Все наиболее важные части кода были проверены. С самого начала выбран четкий и правильный стиль программирования. Этот стиль является стандартом в сообществе профессиональных программистов на C++.

Так же как и в книге Эла Келли (Al Kelley) и Айры Пола (Ira Pohl) *A Book on C, Third Edittion* (Addison-Wesley Longman, 1995), в настоящей работе языки C и C++ и их использование трактуются комплексно, неотъемлемо друг от друга, что не встречается в другой литературе. Для начинающих можно посоветовать более простое введение в язык C: *C by Dessection: Essentials of C Programming, Third Edition*, Эла Келли (Al Kelley) и Айры Пола (Ira Pohl), Addison-Wesley Longman, 1995.

Каждая глава включает:

Объектно-ориентированную концепцию. Объясняется, как объектно-ориентированные концепции программирования поддерживаются свойствами языка.

Рабочий код. Небольшие фрагменты рабочего кода, иллюстрирующие рассматриваемые вопросы. Код демонстрирует особенности языка или концепции ООП.

Разбор. Рассматривается и разбирается программа, детально иллюстрирующая предмет главы. Цель разбора — объяснить читателю впервые встретившиеся программные элементы и идиомы.

Практические замечания. Подсказки, рекомендации по обходу ловушек, нюансы и советы по рассматриваемой теме.

Резюме. Полезный обзор главы в виде краткого перечня основных моментов.

Упражнения. Упражнения проверяют знание языка. Многие из них лучше выполнять в процессе чтения текста. Это поможет читателю последовательно освоить новые моменты. Часто упражнения расширяют познания читателя.

Книга сочетает в себе следующие идеи:

Объектно-ориентированное программирование. «Объектно-ориентированность» подчеркивается повсюду. Глава 1, «Зачем нужно объектно-ориентированное программирование на C++?», дает введение в использование C++ как объектно-ориентированного языка. В главе 2, «Собственные типы данных и инструкции», рассматриваются типы данных, выражения и простые инструкции. В главе 3, «Функции и указатели», обсуждаются сходства между функциями и сложными типами данных. Несколько следующих глав показывают, как работают классы. Классы служат основой для абстрактных типов данных и объектно-ориентированного программирования. Последние главы рассматривают детали использования наследования, шаблонов и исключений. В главе 12, «ООП на C++», обсуждаются ООП и *Платоническая* философия программирования. Эта книга способствует принятию программистом данной точки зрения. В любом месте текста читатель может остановиться для того, чтобы на практике применить новые знания.

Обучение на примере. Эта книга является пособием, в котором особое внимание уделено работающему коду. С самого начала представлены вполне функциональные программы. Предполагается, что читатель использует интерактивную среду программирования. Упражнения сочетаются с примерами, что поощряет к экспериментированию. В объяснении больших фрагментов кода мы избегаем чрезмерной детализации. В каждой главе содержится несколько важных программ-примеров. Главные элементы этих программ подвергаются разбору.

Структуры данных в C++. Рассмотрены основные типы данных, используемые в программировании. Стеки, статические и динамические (в том числе многомерные) массивы, списки, деревья и строки — все это обсуждается в тексте. Упражнения углубляют понимание того, как реализовать и использовать эти структуры. Реализация согласуется с концепцией абстрактного типа данных.

Стандартная библиотека шаблонов (STL). STL изучается и используется в главе 9, «Шаблоны, обобщенное программирование и STL». Объяснение и использование STL предваряется множеством примеров структур данных. Упор делается на изучение механизма шаблонов, необходимого для STL, и на идиому итератора, которая используется в STL.

Язык ANSI C++ и *iostream.h*. C++ развивается довольно быстро для давно существующего и широко используемого языка. Эта книга основана на самом последнем стандарте — документах Комитета ANSI по языку C++.[1] Краткий неофициальный справочник по языку представлен в приложении C, «Руководство по языку». Главные нововведения связаны с шаблонами и обработкой исключений. В примерах используется библиотека ввода-вывода *iostream.h*. Она заменила *stdio.h*, которая применялась в C. Использование библиотеки *iostream.h* описано в приложении D, «Ввод-вывод».

Справочные приложения. Приложение C, «Руководство по языку», представляет собой подручный неформальный справочник. Приложение содержит сжатое описание языка, правда, неофициальное. Есть также приложение по ключевым библиотекам ввода-вывода, *iostream.h* и *stream.h* — приложение D, «Ввод-вывод». Краткое руководство по библиотекам *string* и STL приведено в приложении E, «STL и строковые библиотеки».

Идиоматичность и основное направление языка. Автор пытается придерживаться основного направления при рассмотрении тех аспектов, которые наиболее важны и для студентов, и для профессионалов. Он избегает темных сторон языка, которые приводят к ошибкам и путанице. Подход к коду идиоматичен. Код можно легко скопировать и применить для решения других задач.

Проверенный в промышленном и учебном отношении материал. Книга является основой для многих учебных курсов, прочитанных автором. Автор использует содержание книги в обучении студентов и профессионалов, которое он проводит с 1986 года. Различные изменения в новом издании были проверены «в учении и в бою» и отражают значительный опыт преподавания и консалтинга, накопленный автором. Книга послужила основой для обширной серии учебных видеокассет и онлайновых курсов. Дополнительную информацию по этим курсам можно получить на Web-странице автора по адресу: *www.cse.ucsc.edu/~pohl*.

[1] Окончательный вариант стандарта (ISO/IEC 14882) был ратифицирован 8 августа 1998 года. — *Примеч. перев.*

Благодарности

Особая благодарность — моей жене Дебре Долсберри (Debra Dolsberry), которая вдохновляла меня на протяжении всего проекта. Она выступала в качестве дизайнера книги и технического редактора второго издания. Дебра разработала с помощью FrameMaker 4.0 подходящие форматы и стили оформления и руководила переносом текстов в формате troff из первого издания. Она также набрала и протестировала основную часть кода. Проведенное ею тщательное тестирование кода и упражнений существенно улучшило книгу. Стефен Клэмедж (Stephen Clamage) из корпорации TauMetric сделал существенные и толковые замечания по деталям языка. Уильям Энджелс (William Engles, University of Wisconsin) описал и улучшил процедуру перетасовки для примера «Покер». Данное издание также рецензировали: Джин Белл (Jean Bell, Colorado School of Mines); Артур Делчер (Arthur Delcher, Loyola University); Константин Лауфер (Konstantin Laufer, Loyola University); Джеймс Л. Мерфи (James L. Murphy, California State University); Кент Вулдбридж (Kent Wooldbridge, California State University); Ши-Хо Ванг (Shih-Ho Wang, University of California); Дэвид Б. Тиг (David B. Teague, Western California University); Лукаш Пруски (Lukasz Pruski, California State University) и Дэвид Грегори (David Gregory). Рэндел Бернс (Randal Burns) и Хироя Чиба (Hiroya Chiba), ассистенты и аспиранты по специальности «Компьютерные науки» Университета штата Калифорния в Санта-Крузе, также внесли свой вклад в рецензирование книги.

При работе над первым изданием автор получил помощь, вдохновение и поддержку от Питера Аперса (Peter Apers, University of Twente, The Netherlands); Генри Баля (Henri Bal, Vrije University, The Netherlands); Майкла Бисона (Michael Beeson, State University of California); Нана Борресона (Nan Borreson, Borland International); Дугласа Кэмпбела (Douglas Campbell, University of Connecticut); Кэти Коллинз (Cathy Collins, USC); Стива Демуржана (Steve Demurjian); Роберта Дорана (Robert Doran, University of Auckland, New Zealand); Роберта Дюрлинга (Robert Durling, UCSC); Дэниела Эдельсона (Daniel Edelson, USCS); Антона Элиенса (Anton Eliens, Vrije University, The Netherlands); Рэя Фуджиока (Ray Fujioka, USC); Томаса Джадсона (Thomas Judson. University of Portland); Эла Келли (Al Kelley, USCS); Джима Кемпфа (Jim Kempf, Sun Microsystems, Incorporated); Даррелла Лонга (Darrell Long, USCS); Чарли Макдаула (Charlie McDowell, USCS); Лауры Пол (Laura Pohl, Cottage Consultants); Рейнда ван де Рейта (Reind van de Reit, Vrije University, The Netherlands); Энтони Вассермана (Anthony Wasserman, IDE); и Сали Юртас (Salih Yurttas, Texas A&M University).

Второе издание было подготовлено при поддержке моего редактора Дж. Картера Шанклина (J. Carter Shanklin) и ассистента редакции Энджелы Бьюннинг (Angela Buenning). Наконец, я благодарю Бьерна Страуструпа (Bjarne Stroustrup) за изобретение такого мощного языка и за то, что он вдохновляет других на оказание помощи в изучении этого языка.

Айра Пол
Университет штата Калифорния, Санта-Круз

Глава 1

Зачем нужно объектно-ориентированное программирование на C++?

В этой главе дается обзор языка программирования C++. Она также служит введением в использование C++ в качестве объектно-ориентированного языка и представляет ряд программ, которые демонстрируют объектно-ориентированные возможности. Сложность программ постепенно увеличивается, последние разделы иллюстрируют некоторые концепции объектно-ориентированного программирования. Такой подход должен дать вам ощущение того, как работает язык. Будучи обзорной, эта глава содержит достаточно сложный материал, который может быть просмотрен бегло или пропущен теми читателями, которые хотят начать с простых основ программирования. Они могут сразу перейти к следующей главе.

Объектно-ориентированное программирование (ООП) — основная методология программирования 90-х годов. Она является результатом тридцатилетнего опыта и практики, которые берут начало в языке Simula 67 и продолжаются в языках Smalltalk, LISP, Clu и в более поздних — Actor, Eiffel, Objective C, Java и C++. ООП — это стиль программирования, который фиксирует поведение реального мира так, что детали разработки скрыты, а это позволяет тому, кто решает задачу, мыслить в терминах, присущих этой задаче, а не программированию.

C++ был создан в начале 80-х Бьерном Страуструпом. Страуструп имел перед собой две цели: (1) оставить C++ совместимым с обычным C и (2) расширить C конструкциями ООП, основанными на понятии класса в Simula 67. Язык C был разработан Деннисом Ричи в начале 70-х для создания UNIX и задумывался как язык системного программирования. Постепенно он приобрел популярность не только как системный язык, но и в качестве языка общего назначения.

Конечная цель создания C++ — предоставить профессиональному программисту язык, который можно использовать при создании объектно-ориентированного программного обеспечения, не жертвуя эффективностью или переносимостью C. Первые

шаги на этом пути были сделаны Деннисом Ричи, а продолжили его Бьерн Страуструпп и растущее сообщество современных практикующих программистов.

Подчеркнем два аспекта, связанных с C++. Во-первых, он превосходен как язык общего назначения, благодаря своим новыми свойствам. Во-вторых, он удачен и как объектно-ориентированный язык программирования.

1.1. Объектно-ориентированное программирование

Объектно-ориентированное программирование — это программирование, сфокусированное на данных, причем данные и поведение неразрывно связаны. Вместе данные и поведение представляют собой класс, а объекты являются экземплярами класса. Например, многочлен имеет область значений, и она может изменяться такими операциями, как сложение и умножение многочленов.

ООП рассматривает вычисления как моделирование поведения. То, что моделируется, является объектами, представленными вычислительной абстракцией. Допустим, мы хотим улучшить наши навыки игры в покер; для этого мы должны научиться вычислять вероятность выпадения различных карточных комбинаций. Нам надо смоделировать перетасовку колоды, кроме того следует найти подходящий способ для оперирования мастями и достоинствами карт. Мы можем открыто[1] использовать названия мастей: пики, черви, бубны и трефы, но технически масти представлены целыми числами. Это внутреннее представление скрыто и, поэтому, не может повлиять на наши расчеты. Также как материальные колоды карт могут иметь разные физические свойства, но при этом ведут себя одинаковым ожидаемым от них образом, так и различные моделируемые колоды должны вести себя одинаково.

Мы будем применять термин *абстрактный тип данных, АТД* (abstract data type, ADT) для обозначения определяемого пользователем расширения исходных типов языка. АТД состоит из набора значений и операций, которые могут влиять на эти значения. Например, в C нет типа данных для комплексных чисел, а C++ позволяет добавить такой тип и интегрировать его с существующими.

Объекты (objects) являются переменными класса. Объектно-ориентированное программирование позволяет легко создавать и использовать АТД. Объектно-ориентированное программирование использует механизм *наследования* (inheritance). Наследование выгодно тем, что позволяет получать производные типы из уже определенных пользователем типов данных. Этот механизм сродни биологической таксономии. И грызуны, и кошки — млекопитающие. Если категория «млекопитающие» несет в себе информацию о свойствах и поведении, истинную для каждого из объектов соответствующего биологического класса, то создание категорий «кошачьи» и «грызуны» из категории «млекопитающие» чрезвычайно экономично.

В ООП объекты отвечают за свое поведение. Например, многочлены, комплексные числа, целые числа, числа с плавающей точкой — все это объекты, «понимающие» сложение. Каждый из этих типов включает в себя код для выполнения сложения. Компилятор предоставляет надлежащий код для целых и чисел с плавающей точкой. АТД «многочлен» содержит функцию, определяющую сложение, в соответствии с особенностями своей реализации. Поставщик АТД должен включать в него код для описания любого поведения, которое обычно можно ожидать от соответству-

[1] Под открытостью здесь понимается открытый (public) доступ к данным. — *Примеч. перев.*

ющих объектов. То, что объект сам отвечает за свое поведение, значительно упрощает задачу программирования для пользователя этого объекта.

Представим себе класс объектов под названием «фигуры». Если мы хотим нарисовать на экране какую-нибудь фигуру, нам надо знать, где будет находиться ее центр и как ее рисовать. Некоторые фигуры, например многоугольники, относительно легко нарисовать. В общем случае, процедура рисования фигуры может быть очень трудоемкой, так как, возможно, потребуется хранить большое число отдельных граничных точек. Напротив, вариант с многоугольником несомненно удобен. Если отдельная фигура прекрасно понимает, как себя нарисовать, программист при использовании такой фигуры должен лишь передать объекту сообщение «нарисовать(ся)».

В C++ новое понятие классов предоставляет механизм *инкапсуляции* (encapsulation) для реализации АТД. Инкапсуляция сочетает в себе, с одной стороны, внутренние детали реализации конкретного типа и, с другой, доступные извне операции и функции, которые могут действовать на объекты этого типа. Детали реализации могут быть недоступны для программы, которая использует данный тип. Например, стек может быть реализован как массив фиксированной длины, а доступные всем операции должны включать в себя функции `push` (поместить в стек) и `pop` (извлечь из стека). Изменение внутренней реализации на связный список не должно повлиять на то, как `push` и `pop` используются снаружи класса. Код, который использует АТД, называется *клиентом АТД*. Реализация стека скрыта от его клиентов.

К понятию ООП имеет отношение целый набор концепций, включая следующие:

Концепции ООП

- Моделирование действий из реального мира
- Наличие типов данных, определяемых пользователем
- Сокрытие деталей реализации
- Возможность многократного использование кода благодаря наследованию
- Интерпретация вызовов функций на этапе выполнения

Некоторые из этих понятий довольно расплывчаты, некоторые — абстрактны, другие носят общий характер. Чтобы смягчить возможные разочарования и смятение, мы постараемся проиллюстрировать ООП примерами, которые демонстрируют конкретные преимущества для программиста.

1.2. Пример программы на C++

C++ — это тесный союз программирования на низком и высоком уровнях. С был разработан как системный язык, близкий к машинному. C++ дополнен объектно-ориентированными свойствами, которые позволяют программисту создавать или импортировать библиотеки, присущие конкретной задаче. Пользователь может написать код на проблемном уровне, в то же время поддерживая контакт с машинным уровнем реализации деталей.

В файле hello.cpp

```
//Здороваемся с миром на C++

#include <iostream.h>      //библиотека ввода-вывода
#include <string>          //строковый тип
```

```
using namespace std; //пространство имен стандартных библиотек
inline void pr_message(string s = "Hello world!")
                  //Hello world! – Здравствуй, мир!
  { cout << s << endl; }
int main()
{
  pr_message();
}
```

При выполнении эта программа напечатает сообщение:

```
Hello world!
```

Программа на C++ — это набор объявлений и функций; выполнение начинается с функции `main()`. Когда программа компилируется, сначала выполняются все директивы препроцессора, такие, как директива `#include`. Она импортирует необходимый файл, обычно определения библиотеки. В нашем случае, библиотека ввода-вывода находится в файле *iostream* или *iostream.h*.

Библиотека *string* является частью стандартной библиотеки C++ и должна быть включена для использования объявления `string`. Пространство имен `std` зарезервировано для использования со стандартными библиотеками. Пространства имен были введены в ANSI C++ для предоставления области видимости, которая позволяет различным поставщикам кода избежать конфликта глобальных имен. Объявление `using` позволяет использовать идентификаторы из стандартной библиотеки без уточнения полного имени. При отсутствии такого объявления программа должна была бы использовать запись `std::string`.

Символ // (двойная косая черта) используется для добавления комментария в конце строки. Текст программы может располагаться в любом месте страницы, пустое пространство между строками игнорируется. Вообще, пробелы, комментарии, абзацные отступы в тексте программы, — все это используется лишь для создания хорошо документированной программы и не влияет на ее семантику.

Модификатор `inline` функции `pr_message()` сообщает компилятору, что ее надо компилировать, по возможности отказавшись от генерации инструкций вызова и возврата. Это позволяет обеспечить высокую эффективность. Как видно из текста, функция `pr_message()` имеет строковый параметр `s` со значением по умолчанию `"Hello world!"`. Это значит, что когда передается пустой список параметров, выполняется `pr_message("Hello world!")`.

Идентификатор `cout` определен в *iostream* как стандартный выходной поток, в большинстве систем направляющий вывод на экран. Идентификатор `endl` является стандартным *манипулятором* (manipulator), который очищает выходной буфер, так что печатается все содержимое буфера и осуществляется переход на новую строку. Оператор << направляет в `cout` (в данном случае) все, что за ним следует.

В C++ функция может возвращать неопределенный тип `void`. Это значит, что функция не возвращает никакого значения, как в случае с `pr_message()`. Специальная функция `main()` возвращает выполняющей системе целое значение; в большинстве случаев неявно возвращается ноль, что означает нормальное завершение. Другие значения `main()` должны возвращаться явно инструкцией `return`.

Вот еще один вариант `main()`:

В файле dinner.cpp

```
int main()
{
   pr_message();
   pr_message("Лаура Пол");
   pr_message("Пора обедать!");
}
```

При выполнении программа напечатает:

```
Hello world!
Лаура Пол
Пора обедать!
```

1.3. Инкапсуляция и расширяемость типов

ООП — это сбалансированный подход к написанию программного обеспечения. Данные и поведение упакованы вместе. Такая инкапсуляция создает определяемые пользователем типы данных, расширяющие собственные типы языка и взаимодействующие с ними. *Расширяемость типов* — это возможность добавлять к языку определяемые пользователем типы данных, которые так же легко использовать, как и собственные типы.

Абстрактный тип данных, например строка, является описанием идеального, всем известного поведения типа. Пользователь строки знает, что операции, такие как конкатенация или печать, имеют определенное поведение. Операции конкатенации и печати называются *методами*. Конкретная реализация АТД может иметь ограничения; например, строки могут быть ограничены по длине. Эти ограничения влияют на открытое всем поведение. В то же время, внутренние или закрытые детали реализации не влияют прямо на то, как пользователь видит объект. Например строка часто реализуется как массив; при этом внутренний базовый адрес элементов этого массива и его имя не существенны для пользователя.

На терминологию ООП сильно повлиял язык Smalltalk. Создатели Smalltalk хотели, чтобы программисты порвали со своими старыми привычками и приняли новую методологию программирования. Они изобрели термины, такие как *сообщение* и *метод*, взамен традиционных понятий *вызов функции* и *функция-член*.

Инкапсуляция — это способность скрывать внутренние детали при предоставлении открытого интерфейса к определяемому пользователем типу. В C++ для обеспечения инкапсуляции используются объявления класса и структуры (`class` и `struct`) в сочетании с ключевыми словами доступа `private` (закрытый), `protected` (защищенный) и `public` (открытый).

Ключевое слово `public` показывает, что доступ к членам, которые стоят за ним, является открытым безо всяких ограничений. Без этого ключевого слова члены класса по умолчанию закрыты. Закрытые члены доступны только функциям-членам своего класса. Открытые члены доступны любой функции внутри области видимости объявления класса. Закрытость позволяет спрятать часть реализации класса, предотвращая тем самым непредвиденные изменения структуры данных. Ограничение доступа или сокрытие данных является особенностью объектно-ориентированного про-

граммирования. Давайте напишем класс под названием my_string, в котором реализована ограниченная форма строки.

В файле string1.cpp

```
//Простейшая реализация типа my_string

const int max_len = 255;

class my_string{
public:          //всеобщий доступ к интерфейсу
   void assign(const char* st);
   int length() const { return len; }
   void print() const
      { cout << s << "\nДлина: " << len << endl;}
private:         //ограниченный доступ к реализации
   char s[max_len];
   int len;
};
```

Скрытое представление — это массив из max_len символов и переменная len, в которой хранится длина строки. Объявление функций-членов позволяет АТД иметь отдельные функции, влияющие на его закрытое представление. Например, функция-член length() возвращает длину строки. Функция-член print() выводит строку и ее длину. Функция-член assign() помещает символьную строку в скрытую переменную s, затем вычисляет ее длину и сохраняет ее в скрытой переменной len. Функции-члены, которые не изменяют значения переменных, объявлены как const.

Теперь мы можем использовать тип данных my_string, как если бы это был основной тип языка. Новый тип удовлетворяет стандартным правилам С. Код, который будет использовать этот тип, называется *клиентом* типа; он может обращаться только к открытым членам для того, чтобы воздействовать на переменные my_string.

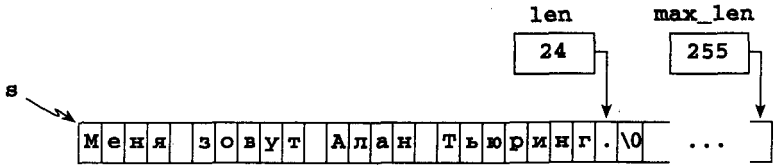
В файле string1.cpp

```
//Проверка класса my_string

int main()
{
   my_string one, two;
   char three[40] = {"Меня зовут Чарльз Бэббидж."};

   one.assign("Меня зовут Алан Тьюринг.");
   two.assign(three);
   cout << three;
   cout << "\nДлина: " << strlen(three) << endl;
        //Печать наиболее короткой из one и two
   if (one.length() <= two.length())
     one.print();
   else
     two.print();
}
```

Переменные one и two — типа my_string (см. рисунок). Переменная three имеет тип указателя на символ и не совместима с my_string.



Функции-члены вызываются с использованием *оператора «точка»* (*оператора доступа к члену структуры*). Как видно из их определений, эти функции-члены действуют на скрытые члены данных соответствующих переменных. Вот результат работы программы:

```
Меня зовут Чарльз Бэббидж.
Длина: 26
Меня зовут Алан Тьюринг.
Длина: 24
```

1.4. Конструкторы и деструкторы

В терминологии ООП переменная называется *объектом*. Функция-член, единственная работа которой заключается в инициализации объекта класса, называется *конструктором* (constructor). Во многих случаях инициализация предполагает динамическое распределение памяти. Конструкторы вызываются всякий раз, когда создается объект данного класса. Конструктор с одним аргументом может производить преобразование типов, если только при объявлении такого конструктора не используется ключевое слово explicit (явный). *Деструктор* (destructor) — это функция-член, задача которой состоит в том, чтобы завершить существование переменной класса. Этот процесс часто предполагает динамическое освобождение памяти. Деструктор вызывается неявно, когда автоматический объект выходит за пределы своей области видимости.

Давайте изменим наш пример my_string. Теперь память для каждой переменной класса будет выделяться динамически. Мы заменим закрытый член данных (массив) на указатель. Переделанный класс будет использовать конструктор для динамического выделения нужного объема памяти. Для этого применим оператор new.

В файле string2.cpp

```
//Реализация динамической my_string

class my_string {
public:         //конструктор
  explicit my_string(int n) { s = new char[n + 1]; len = n; }
  void assign(const char* st);
  int length() const { return len; }
  void print() const
    { cout << s << "\nДлина: " << len << endl; }
```

```
private:
   char*  s;
   int    len;
};
```

Для такого представления нам потребуется вариант функции-члена `assign()`, динамически выделяющий память:

```
void my_string::assign(const char* str)
{
   delete [] s;
   len = strlen(str);
   s = new char[len + 1];
   strcpy(s, str);
}
```

Имя конструктора совпадает с именем класса. При выделении памяти для инициализации объектов конструктор часто использует оператор `new`. Это унарный оператор, который получает в качестве аргумента тип данных, в частности, размер массива определенного типа. Оператор `new` выделяет необходимый объем памяти для хранения данного типа и возвращает указатель на адрес выделенной памяти. В предыдущем примере из свободной памяти должно быть выделено `n + 1` байт. Так, при объявлении:

```
my_string a(40), b(100);
```

для переменной `a` потребуется 41 байт , на которые укажет `a.s`, и 101 байт будет выделен для переменной `b`, а укажет на эти байты `b.s`. Мы добавили один байт для символа конца строки `\0`. Оператор `new` выделяет память на постоянной основе, и она не освобождается автоматически при выходе из блока. Если требуется освободить память, в класс должна быть включена функция-деструктор. Деструктор записывается, как обычная функция-член, имя которой совпадает с именем класса, но начинается с символа ~ (тильда). Для уничтожения объекта, распределенного оператором `new`, деструктор использует унарный оператор `delete`, — еще одно дополнение к языку, — чтобы автоматически освободить память, на которую направлено соответствующее выражение-указатель.

```
//Добавлено как функция-член к классу my_string
~my_string() { delete []s; }     //деструктор
```

Конструкторы часто перегружают, задавая несколько функций-конструкторов, чтобы предоставить несколько способов инициализации объекта. Рассмотрим, например, инициализацию `my_srting` указателем на символьное значение. Такой конструктор будет выглядеть так:

```
my_string(const char* p)
{
   len = strlen(p);
   s = new char[len + 1];
   strcpy(s, p);
}
```

Типичное объявление, вызывающее эту версию конструктора:

```
char* str = "Я пришел пешком.";
my_string a("Я приехал на автобусе."), b(str);
```

Желательно также иметь конструктор без аргументов:

```
my_string() { len = 0; s = new char[1]; }
```

Здесь объявление производится без аргументов, и по умолчанию будет выделен один байт памяти. А следующее объявление вызовет все три конструктора:

```
my_string a, b(20), c("Я приехал верхом.");
```

Перегруженный конструктор выбирается в зависимости от формы каждого из объявлений. Переменная a не имеет параметров, поэтому под нее будет выделен один байт. У переменной b есть целый параметр, и для нее отведен 21 байт. Переменная c получает в качестве параметра указатель на символьную строку "Я приехал верхом.", и для нее выделено 18 байт, причем заданная строка копируется в закрытый член s.

1.5. Перегрузка

Перегрузкой (overloading) называется практика придания нескольких значений оператору или функции. Выбор конкретного значения зависит от типов аргументов, полученных оператором или функцией. Давайте перегрузим функцию print() из предыдущего примера. Это будет второе определение функции print().

В файле string3.cpp

```
class my_string {

public:     //общий доступ
   .....
   void  print() const
      { cout << s << "\nДлина: " << len << endl; }
   void  print(int n) const
      { for (int i = 0; i < n; ++i)
         cout << s << endl; }
   .....
}
```

Новая версия функции print() принимает целый аргумент. Она напечатает строку n раз.

```
three.print(2);     //печать строки three дважды
three.print(-1);    //строка three не печатается
```

Большинство операторов C++ может быть перегружено. Мы, например, перегрузим оператор +, чтобы представить конкатенацию двух строк. Для этого нам понадобятся два новых ключевых слова: friend и operator. Ключевое слово operator располагается перед изображающим оператор значком и вместе с этим значком заменяет то, что было бы именем функции в объявлении обычной функции. Ключевое слово friend предоставляет функции доступ к закрытым членам переменной класса. Дру-

жественная (friend) функция не является членом класса, но обладает привилегиями функции-члена того класса, другом которого она объявлена.

В файле string4.cpp

```
//Перегруженный оператор +

class my_string {
public:
   my_string() { len = 0; s = new char[1]; }
   explicit my_string(int n){ s = new char[n + 1]; len = n; }
   void  assign(const char* st);
   int  length() const { return len; }
   void  print() const
      { cout << s << "\nДлина: " << len << endl; }
   my_string& operator=(const my_string& a);
   friend my_string& operator+
      (const my_string& a, const my_string& b);
private:
   char* s;
   int  len;
};

//Перегрузка +
my_string& operator+(const my_string& a, const my_string& b)
{
   my_string* temp = new my_string(a.len + b.len);

   strcpy(temp->s, a.s);
   strcat(temp->s, b.s);
   return *temp;
}

void print(const char* c) //print с областью видимости файла
                          //(не путать с print из my_string!)
{
   cout << c << "\nДлина: " << strlen(c) << endl;
}

int main()
{
   my_string one, two, both;
   char three[40] = {"Меня зовут Чарльз Бэббидж."};

   one.assign("Меня зовут Алан Тьюринг.");
   two.assign(three);
   print(three); //Вызов print с областью видимости файла
                 //Печать наболее короткой строки из one или two
   if (one.length() <= two.length())
      one.print();  //вызов функции-члена print
```

```
    else
      two.print();
    both = one + two;  //плюс перегружен как конкатенация
    both.print();
}
```

Разбор функции operator+()

- `my_string& operator+(const my_string& a, const my_string& b)`

Плюс перегружен. У него два аргумента, оба — `my_string`. Аргументы передаются по ссылке. Запись вида *тип& идентификатор* объявляет идентификатор как переменную-ссылку. Ключевое слово `const` показывает, что аргументы не могут быть изменены.

- `my_string* temp = new my_string(a.len + b.len);`

Функция должна возвратить значение типа `my_string`. Этот локальный указатель будет использован чтобы вернуть значение `my_string` после выполнения конкатенации. Для выделения достаточного объема памяти используется оператор `new`.

- `return *temp;`

разыменованный указатель `temp` ссылается на объединенную my_string.

1.6. Шаблоны и обобщенное программирование

Ключевое слово `template` используется в C++ для обеспечения *параметрического полиморфизма*. Параметрический полиморфизм позволяет использовать один и тот же код применительно к разным типам, причем тип является параметром кода. Код пишется обобщенно. Особенно важно применение такой техники при написании общих *контейнерных классов*. Контейнерный класс используется для хранения данных определенного типа. Стеки, векторы, деревья, списки — все это примеры стандартных контейнерных классов. Вот пример контейнерного класса `stack`, определенного как параметризованный тип:

В файле tstack.cpp

```
//Реализация шаблона stack

template <class TYPE>
class stack {
public:
  explicit stack(int size = 1000) : max_len(size)
    { s = new TYPE[size]; top = EMPTY; }
  ~stack() { delete []s; }
  void  reset() { top = EMPTY; }          //очистить стек
  void  push(TYPE c) { s[++top] = c; }    //поместить в стек
  TYPE  pop() { return s[top--]; }        //извлечь из стека
```

```
    TYPE  top_of() { return s[top]; }     //считать верхний
                                          //элемент
    bool  empty() { return (top == EMPTY); }  //стек пуст?
    bool  full() {return (top == max_len-1);} //стек заполнен?
  private:
    enum {EMPTY = -1};
    TYPE*  s;
    int   max_len;      //максимальная длина
    int   top;          //вершина
  };
```

Объявление класса выглядит так:

```
template <class идентификатор>
```

Идентификатор является аргументом шаблона, который на самом деле будет подставлен на место произвольного типа. Везде в тексте определения класса аргумент шаблона может использоваться в качестве имени типа. Сам аргумент подставляется при фактическом объявлении. Вот пример объявления `stack`, использующий вышесказанное:

```
stack<char>     stk_ch;          //стек из 1000 символов
stack<char*>    stk_str(200);    //стек из 200 указателей
stack<complex> stk_cmplx(100); //стек из 100 комплексных чисел
```

Этот механизм позволяет нам не переписывать каждый раз объявления классов, которые отличаются лишь типами. При работе с таким типом в качестве части объявления всегда должны использоваться угловые скобки < >. Вот две функции, использующие шаблон `stack`:

```
//Обращение последовательности указателей char*,
//представляющих строку

void reverse(char* str[], int n)
{
  stack<char*> stk(n); //в стеке хранятся char*

  for (int i = 0; i < n; ++i)
    stk.push(str[i]);
  for (int i = 0; i < n; ++i)
    str[i] = stk.pop();
}
```

В функции `reverse()` используется стек `stack<char*>`. Он принимает `n` строк, а затем строки извлекаются в обратном порядке.

```
//Инициализация стека комплексными числами из массива

void init(complex c[], stack<complex>& stk, n)
{
  for (int i = 0; i < n; ++i)
    stk.push(c[i]);
}
```

В функции `init()` переменная типа `stack<complex>` передается по ссылке; в стек помещаются n комплексных чисел.

1.7. Стандартная библиотека шаблонов (STL)

Стандартная библиотека шаблонов (Standard Template Library — STL) является стандартной библиотекой C++, которая позволяет использовать обобщенное программирование для множества распространенных структур данных и алгоритмов. Эта библиотека предлагает: контейнерные классы, такие как векторы, очереди, отображения; средства перебора элементов контейнера с помощью классов итераторов; а также алгоритмы для различного использования контейнеров — функции сортировки и поиска данных и т. п. Мы предлагаем краткое описание STL с акцентом на эти три основные составляющие, — контейнеры, итераторы и алгоритмы, — которые создают основу для обобщенного программирования.

Библиотека построена с использованием шаблонов, ее дизайн вполне ортогонален. Компоненты можно комбинировать друг с другом, используя в качестве параметров как «родные» типы C++, так и типы, определяемые пользователем. Необходимо только следить за правильностью инстанцирования элементов библиотеки STL.

В файле stl_list.cpp

```
//Использование контейнера list (список)

#include <iostream>
#include <list>       //контейнер списков
#include <numeric>    //нужно для функции accumulate()
using namespace std;

void print(const list<double> &lst)
{  //используем итератор для путешествия по lst
   list<double>::const_iterator where;

   for (where = lst.begin(); where !=lst.end(); ++where)
     cout << *where << '\t';
   cout << endl;
}

int main()
{
   double w[4] = {0.9, 0.8, 88, -99.99};
   list<double> z;

   for( int i = 0; i < 4; ++i)
     z.push_front(w[i]);
   print(z);
   z.sort();
   print(z);
   cout << "Сумма равна "<< accumulate(z.begin(), z.end(), 0.0)
        << endl;
}
```

В этом примере списочный контейнер должен хранить переменные с двойной точностью. Массив из таких переменных помещается в список. Функция `print()` использует итератор для печати по очереди всех элементов списка. Итераторы имеют стандартный интерфейс, частью которого являются функции-члены `begin()` и `end()` для определения начала и конца контейнера. Кроме того, интерфейс включает в себя алгоритм сортировки — функцию-член `sort()`. Функция `accumulate()` использует 0.0 в качестве начального значения и вычисляет сумму элементов списочного контейнера путем прохода от начальной позиции `z.begin()` до конечной `z.end()`.

1.8. Наследование

Особенностью ООП является поощрение повторного использования кода при помощи механизма наследования. Новый класс *производится* от существующего, называемого *базовым* классом. Производный класс использует члены базового класса, но может также изменять и дополнять их.

Многие типы представляют собой вариации на темы существующих. Часто бывает утомительно разрабатывать новый код для каждого из них. Кроме того, новый код — новые ошибки. Производный класс наследует описание базового класса, делая ненужными повторную разработку и тестирование кода. Отношения наследования иерархичны. Иерархия — это метод, позволяющий копировать элементы во всем их многообразии и сложности. Она вводит классификацию объектов. Например, в периодической системе элементов есть газы. Они обладают свойствами, присущими всем элементам системы. Инертные газы — следующий важный подкласс. Иерархия заключается в том, что инертный газ, например, аргон — это газ, а газ, в свою очередь является элементом системы. Такая иерархия позволяет легко толковать поведение инертных газов. Мы знаем, что их атомы содержат протоны и электроны, что верно и для прочих элементов. Мы знаем, что они пребывают в газообразном состоянии при комнатной температуре, как все газы. Мы знаем, что ни один газ из подкласса инертных газов не вступает в обычную химическую реакцию ни с одним из элементов, и это свойство всех инертных газов.

Методология объектно-ориентированного проектирования

1. Выбери надлежащую совокупность типов.
2. Спроектируй взаимосвязи типов в коде, используя наследование.

Предположим, нам надо разработать базу данных для университета. Студенческий отдел должен вести учет студентов разных категорий. Базовый класс, который нам необходимо разработать, будет фиксировать описание студента. Две основные категории студентов — это аспирант и просто студент (не аспирант).

Вот пример наследования:

В файле student1.cpp

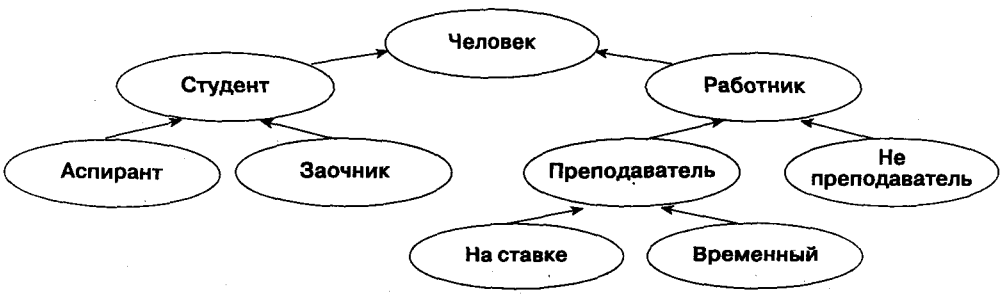
```
            //тип финансовой поддержки
enum support { ta, ra, fellowship, other };
            //курс (год обучения)
enum year { fresh, soph, junior, senior, grad };
```

```
class student {          //класс студентов
public:                  //конструктору передаются имя,
                         //номер, средний балл, курс
  student(char* nm, int id, double g, year x);
  void print() const;
private:
  int student_id;        //номер
  double gpa;            //средний балл
  year y;                //курс
  char name[30];         //имя
};

class grad_student : public student { //класс аспирантов
public:                  //конструктору дополнительно передаются:
                         //тип финансовой поддержки,
                         //название кафедры, тема диссертации
  grad_student(char* nm, int id, double g,
               year x, support t, char* d, char* th);
  void print() const;
private:
  support  s;            //финансовая поддержка
  char   dept[10];       //кафедра
  char   thesis[80];     //диссертация
};
```

В этом примере `grad_student` — производный класс, а `student` — базовый. Использование в заголовке производного класса ключевого слова `public`, следующего за двоеточием, означает, что открытые члены класса `student` должны наследоваться как открытые члены `grad_student`. Закрытые члены базового класса недоступны производному классу. Открытое наследование означает также, что производный класс `grad_student` является подтипом `student`.

Структуры наследования образуют каркас для построения достаточно общих систем. Например, база данных, содержащая информацию обо всех людях в университете, может быть унаследована от базового класса `person` (человек). Базовый класс `student` можно использовать для создания производного класса студентов-юристов, как следующей значимой категории объектов. Аналогично, `person` может служить базовым классом для различных категорий работников. Иерархическая структура наследования показана на рисунке.



1.9. Полиморфизм

Полиморфная функция или оператор имеет несколько вариантов. Например, в C++ оператор деления полиморфен. Если аргументы оператора деления целые, выполняется целочисленное деление. Если один или оба аргумента — с плавающей точкой, используется деление для чисел с плавающей точкой.

В C++ имя функции или оператора может быть перегружено. То, какая именно из одноименных функций будет вызвана, определяется *сигнатурой*, которая представляет собой перечисление типов аргументов в списке параметров функции.

Например, в выражении деления

```
a/b     //тип определяется естественными правилами приведения
```

результат зависит от аргументов, которые автоматически приводятся к более широкому типу. Так, если оба аргумента целые, результатом будет целое частное. Но если хотя бы один из аргументов — с плавающей точкой, результат тоже будет с плавающей точкой.

Другой пример — оператор вывода

```
cout << a;  //полиморфизм через перегруженную функцию
```

в котором оператор сдвига << вызывает функцию, которая умеет выводить объекты типа a. Таким образом, если a — целое, то и вывод будет целым. Если оно — с плавающей точкой, то и результат — с плавающей точкой.

Полиморфизм локализует ответственность за поведение. Клиентский код часто не требует пересмотра, когда к системе добавляется функциональность с помощью улучшений кода АТД.

Реализация набора процедур для задания типа геометрической фигуры может основываться на исчерпывающем описании произвольной фигуры. Например, структура

```
srtuct shape {   //фигура
                 //окружность, прямоугольник
   enum { CIRCLE, RECTANGLE, ..... } e_val;
   double center, radius; //центр, радиус
   .....
};
```

должна содержать все члены, необходимые для произвольной фигуры, которую на данном этапе может нарисовать система. Она должна включать переменную перечисляемого типа, с тем, чтобы фигуру можно было идентифицировать. Тогда процедура вычисления площади должна быть записана так:

```
duoble area(shape* s)
{
   switch(s -> e_val) {
   case CIRCLE: return(PI * s -> radius * s -> radius);
   case RECTANGLE: return(s -> heght * s -> width);
   .....
   }
}
```

Что означает пересмотр этого кода с целью включить новую фигуру? Понадобится дополнительная строка case в теле кода и дополнительные члены структуры. К сожалению, это повлечет за собой изменения во всем теле кода, поскольку каждая процедура построена так, что придется добавить дополнительный case, даже когда такой case выступает в роли еще одной метки в уже существующем case. Таким образом, локальное улучшение требует глобальных перемен.

Объектно-ориентированная техника программирования на C++ использует иерархию фигур для решения той же проблемы. Иерархия очевидна, когда круг и прямоугольник наследуются из фигуры. В процессе пересмотра кода возможные улучшения делаются в новом производном классе, так что дополнительные описания локализованы. Программист замещает смысл любой измененной процедуры. В нашем случае, необходимо задать новую формулу для вычисления площади. Клиентский код, не использующий новый тип, остается без изменений; код, который должен быть улучшен добавлением нового типа, обычно изменяется минимально.

Программа на C++, следующая изложенной схеме, использует shape как *абстрактный базовый класс*. Это класс, содержащий хотя бы одну чисто виртуальную функцию, как показано в следующем коде:

В файле shape1.cpp

```
//shape – абстрактный базовый класс (фигура)

class shape {
public:
  virtual double area() = 0;      //чисто виртуальная функция
                                  //(площадь)
};

class rectangle : public shape { //прямоугольник
public:
  rectangle(double h, double w) :
          height(h), width(w){}
  double area() {return (height * width);}
private:
  double height, width;           //высота, ширина
};

class circle : public shape {     //окружность
public:
  circle(double r) : radius(r) { }
  double area() {return(3.14159 * radius * radius);}
private:
  double radius;                  //радиус
};
```

Клиентский код для вычисления произвольной площади полиморфен. Надлежащая функция area() выбирается на этапе выполнения:

```
shape* ptr_shape;
  .....
```

```
    cout << "Площадь = " << ptr_shape -> area();
    .....
```

Теперь представим, что мы хотим дополнить нашу иерархию типов и разработать класс `square` (квадрат):

```
class square : public rectangle {
public:
   square(double h) : rectangle(h,h) { }
   double area() { return (rectangle::area()); }
};
```

Клиентский код остается без изменений, в отличие от не-ООП-кода, рассмотренного выше.

Иерархическая схема должна минимизировать интерфейс передачи параметров. Каждый уровень стремится сокрыть в себе (в пределах своей реализации) детали строения, на которые влияет вызов функции. В обычной схеме (не ООП) это иногда может сопровождаться внесением изменений в глобально определяемые данные. Такая практика почти всегда осуждается, потому что она ведет к неясному стилю программирования с *побочными эффектами*, а это затрудняет отладку, пересмотр и поддержку кода.

1.10. Исключения в C++

C++ вводит механизм обработки исключений, чувствительный к контексту. Контекстом для возбуждения исключения является блок `try`. Обработчики, объявляемые ключевым словом `catch`, находятся ниже блока `try`.

Исключение возбуждается с помощью ключевого слова `throw`. Исключение будет обработано вызовом соответствующего обработчика, выбранного из списка, который идет сразу за блоком `try`. Ниже приведен простой пример:

В файле stackex.cpp

```
//Конструктор стека с исключениями

stack::stack(int n)
{
   if (n < 1)
      throw (n);     //хотим положительное значение
   p = new char[n]; //создается символьный стек
   if (p == 0)      //new возвращает 0 при неудачном завершении
      throw ("СВОБОДНАЯ ПАМЯТЬ ИСЧЕРПАНА");
   top = EMPTY;
   max_len = n;
}

void g()
{
   try {
      stack a(n), b(n);
      .....
   }
```

```
    catch (int n) {.....}          //некорректный размер
    catch (char* error) {.....}    //свободная память исчерпана
}
```

Первый `throw()` имеет целый аргумент и соответствует сигнатуре `catch(int n)`. Этот обработчик должен выполнить соответствующие действия, когда конструктору в качестве аргумента передается некорректный размер массива. Например, вывести сообщение об ошибке и прервать программу. Второй `throw()` получает в качестве аргумента указатель на символ и соответствует сигнатуре `catch(char* error)`.

1.11. Преимущества объектно-ориентированного программирования

Центральным элементом ООП является инкапсуляция совокупности данных и соответствующих им операций. Понятие класса с его функциями-членами и членами данных предоставляет программисту подходящий для реализации инкапсуляции инструмент. Переменные класса являются объектами, которыми можно управлять.

Кроме того, классы обеспечивают сокрытие данных. Права доступа могут устанавливаться или ограничиваться для любой группы функций, которым необходим доступ к деталям реализации. Тем самым обеспечивается модульность и надежность.

Еще одной важной концепцией ООП является поощрение повторного использования кода с помощью механизма наследования. Суть этого механизма — получение нового производного класса из существующего, называемого базовым. При создания производного класса базовый класс может быть дополнен или изменен. Таким путем могут создаваться иерархии родственных типов данных, которые используют общий код.

Объектно-ориентированное программирование зачастую более сложно, чем обычное процедурное программирование, как оно выглядит на C. Существует по крайней мере один дополнительный шаг на этапе проектирования, перед алгоритмизацией и кодированием. Он состоит в разработке таких типов данных, которые соответствовали бы поставленной проблеме. Зачастую проблема решается «обобщеннее», чем это действительно необходимо.

Есть уверенность, что использование ООП принесет дивиденды в нескольких отношениях. Решение будет более модульным, следовательно, более понятным и простым для модификации и обслуживания. Кроме того, такое решение будет более пригодно для повторного использования. Например, если в программе нужен стек, то он легко заимствуется из существующего кода. В обычном процедурном языке программирования такие структуры данных часто «вмонтированы» в алгоритм и не могут экспортироваться.

ООП много значит для многих людей. Все попытки дать ему определение напоминают старания слепых мудрецов описать слона. Я бы предложил еще одно утверждение:

ООП = расширяемость типов + полиморфизм

Глава 2

Собственные типы данных и инструкции

В этой главе, также как в главе 3, «Функции и указатели» будет дано введение в программирование на C++ с использованием *собственных типов* (native types) и свойств языка, не относящихся к ООП. Собственные типы данных предоставляются самим языком. В C++ это простые типы, такие как символьный (character), целый (integer), с плавающей точкой (floating-point) и булевский (boolean). К собственным типам относятся также и производные от простых типы: массивы (array), указатели (pointer) и структуры (structure). В этой главе мы сосредоточимся на изучении простых собственных типов данных и инструкций.

Цель этой главы, а также главы 3, «Функции и указатели» — дать читателю навыки программирования на подмножестве языка C++, которое приближается к традиционным императивным языкам, таким как Pascal и C. Это подмножество мы называем *ядром языка*. Об этом же пойдет речь и в некоторых разделах главы 4, «Реализация АТД в базовом языке».

Важной чертой ООП является *расширяемость типов*. Эта возможность языка позволяет разрабатывать новые типы в соответствии с проблемной областью. Для того, чтобы функционировать должным образом, новый тип должен работать также, как собственные типы языка. В объектно-ориентированном проектировании создание определяемых пользователем типов должно имитировать вид и свойства собственных типов.

2.1. Элементы программы

Программа состоит из элементов, называемых *лексемами* (*token*). Лексемы — это наборы символов; они формируют базовый словарь, распознаваемый компилятором. Набор символов, используемых в языке, включает:

- Прописные и строчные буквы: `a b ····· z A B ····· Z`[1]
- Цифры: `0 1 ····· 9`

[1] Большинство компиляторов допускает также использование символов кириллицы, по крайней мере в строковых константах. — *Примеч. перев.*

- Операторы: `+ * = ..... <`
- Знаки пунктуации: `; , ' ..... "`

Между лексемами можно оставлять пустое пространство и вставлять комментарии для удобочитаемости и документированности программ. Существует пять типов лексем: ключевые слова (keywords), идентификаторы (identifiers), литералы (literals), операторы (operators) и знаки пунктуации (punctuators) (см. Приложение С.2, «Лексические элементы» на стр. 353).

2.1.1. Комментарии

В C++ многострочный комментарий выглядит так: */* можно несколько строк */*. Кроме того, C++ допускает однострочный комментарий: *//до конца строки.*

```
/* Многострочные комментарии часто используются как вводные
   Программист:  Лаура Пол
   Дата:         1 января 1989
   Версия:   DJD v4.2
*/

#include <iostream.h>//заголовочный файл ввода-вывода
```

2.1.2. Ключевые слова

Ключевые слова (keywords) — четко определенные зарезервированные слова, имеющие конкретное значение в C++. К ним относятся слова, используемые для объявления типов: `int`, `char`, `float`; слова, используемые в синтаксисе составных инструкций, например: `do`, `for`, `if` ; слова для управления доступом: `public`, `protected`, `private`. В следующей таблице приведены ключевые слова, употребляемые в большинстве современных систем C++.

Ключевые слова

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

2.1.3. Идентификаторы

Идентификатор (identifier) — это последовательность букв, цифр и символов подчеркивания (_). Идентификатор не может начинаться с цифры. Прописные и строчные буквы различаются. Хотя, в принципе, идентификаторы могут быть произвольной длины, многие системы распознают только первые 31 символ. Вот некоторые примеры:

```
n          //обычно целая переменная
count      //имя переменной выбрано по смыслу
           //(count — счетчик, номер)
buff_size  //стиль C++, разделитель слов — знак подчеркивания
buffSize   //стиль Pascal, разделитель слов — прописная буква
q2345      //невразумительно
cout       //стандартный выходной поток
_foo       //избегайте знака подчеркивания
           //в качестве первого символа
```

Следующие примеры не являются идентификаторами:

```
for        //ключевое слово
3q         //идентификатор не может начинаться с цифры
-count     //не путайте - и _
too__bad   //двойное подчеркивание
           //для системного использования
_Sysfoo    //подчеркивание и прописная
           //для системного использования
```

2.1.4. Литералы

Литералы (literals) — это постоянные значения, такие как 1 или 3.14159. Для каждого собственного типа C++ существуют литералы, включая символьный и булевский типы, целые, числа с плавающей точкой и указатели. Возможны строковые литералы. Вот некоторые примеры литералов:

```
5          //целая константа
5u         //u или U означает unsigned (беззнаковая)
5L         //l или L означает long (длинная)
05         //целая константа в восьмеричном виде
0x5        //целая константа в шестнадцатиричном виде
true       //булевская константа
5.0        //константа с плавающей точкой, трактуемая как double
5.0F       //f или F — с плавающей точкой,
           //обычно одинарной точности
5.0L       //l или L означает длинную константу
           //с двойной точностью
'5'        //символьная константа, символ с номером ASCII 53
'\n'       //этот символ начинает новую строку
L'XYZ'     //wchar_t символ XYZ
"5"        //строка, состоящая из символа '5'
```

```
"строка и символ перехода на новую строку\n"
5555555555555555   //целое, слишком велико
                   //для большинства машин
```

Символьные литералы обычно даются в виде *'символ'*. Например:

```
'A'       //прописная буква A, номер ASCII 65
'a'       //строчная буква a, номер ASCII 97
'\0'      //нулевой символ – символ конца строки
'+'       //символ оператора сложения (+)
```

Для некоторых непечатаемых и специальных символов требуются escape-последовательность.

Символьные константы

`'\a'`	звуковой сигнал (alert)
`'\\'`	обратная косая черта (backslash)
`'\b'`	возврат на шаг (backspace)
`'\r'`	возврат каретки (carriage return)
`'\"'`	двойные кавычки (double quote)
`'\f'`	прогон листа (formfeed)
`'\t'`	табуляция (tab)
`'\n'`	перевод строки (newline)
`'\0'`	нулевой символ (null character)
`'\''`	апостроф (single quote)
`'\v'`	вертикальная табуляция (vertical tab)
`'\101'`	восьмеричный ASCII-код `'A'`
`'\x041'`	шестнадцатиричный ASCII-код `'A'`
`L'oop'`	`wchar_t` константа

Строковый литерал хранится в памяти как последовательность символов, заканчивающаяся символом со значением 0. Строковые литералы – это `static char[]` константы. Специальные символы внутри строки должны предваряться символом обратной косой черты \.

```
"a"       //два байта для хранения 'a' и '\0'
"a\tb\n"  //пять байт 'a' '\t' 'b' '\n' '\0'
"1 \\"    //четыре байта '1' ' ' '\\' '\0'
"\""      //два байта '"' '\0'
""        //один байт для хранения '\0', пустая строка
```

При выводе этих строк управляющие символы ведут себя соответствующим образом. Так, вторая строка из приведенного примера напечатает символ a, затем – несколько пробелов, что определяется знаком табуляции, потом символ b и переход на новую строку.

Строковые литералы, разделенные только пустым местом, понимаются как одна строка.

```
"Это одна строка,"
"Так как она разделена только "
"пробелами и переводами строки."
```

Литералы с плавающей точкой могут записываться с экспоненциальной частью (включающей, если нужно, знак) или без таковой:

```
3.14f  1.234F    //константы с плавающей точкой
                 //и одинарной точностью
0.1234567        //с двойной точностью
0.123456789L     //длинная с двойной точностью
3. 3.0 0.3E1     //все это 3.0 с двойной точностью
300e-2           //тоже 3.0
```

2.1.5. Операторы и знаки пунктуации

В C++ многие символы и последовательности символов имеют специальное значение. Примеры:

```
+ - * / %       //арифметические операторы
-> ->*          //указатель и указатель на член (инструкции)
&&  ||          //логические операторы
= += *=         //операторы присваивания
```

Операторы (operators)[1] используются в выражениях и имеют тот или иной смысл, когда сопровождаются соответствующими аргументами. В C++ существует множество операторов (см. приложение B, «Приоритет и порядок выполнения операторов»). Они имеют строгий приоритет и порядок выполнения. Конкретный символ может по-разному трактоваться в тех или иных операторах, в зависимости от контекста. Например, знак «минус» (–) может быть унарным минусом (изменение знака), или знаком вычитания.

Знаки пунктуации (punctuators) включают: круглые скобки, фигурные скобки, двоеточия и т. п. Они структурируют элементы программы:

```
foo(a, 7, b + 8)   //в круглых скобках – разделенный
                   //запятыми список аргументов
{ a = b; c = d; }  //в фигурных скобках – список
                   //инструкций (блок)
```

Операторы, знаки пунктуации и пробелы служат для разделения элементов языка.

2.2. Ввод-вывод

Ввод-вывод не является непосредственно частью языка. Он добавлен в виде набора типов и процедур, находящихся в стандартной библиотеке. Стандартным заголовочным файлом ввода-вывода в C++ служит *iostream* или *iostream.h*. Имя файла без расширения *.h* является официальным стандартом ANSI и используется с пространством имен `namespace std`. Кроме того, широко используются заголовочные файлы стандартной библиотеки ANSI C *stdio.h* и *cstdio*. ANSI-стандартные заголовочные файлы библиотек C в соответствии с официальными правилами начинаются с буквы *c* и заканчиваются старым названием без расширения *.h*. Мы будем использо-

[1] Здесь и далее мы переводим слово *statement* как *инструкция*, а слово *operator* — как *оператор*. — *Примеч. перев.*

вать *iostream.h*, поскольку рассматриваем текущую практику программирования. В приложении D, «Ввод-вывод» приведено более полное описание этих и других аспектов ввода-вывода. Данный раздел — вступительный, он должен дать читателю лишь необходимую для начала работы информацию.

Библиотека *iostream* перегружает два оператора побитового сдвига:

```
<<      //поместить в выходной поток
>>      //считать с входного потока
```

Кроме того, в ней объявляются три стандартных потока:

```
cout    //стандартный поток вывода
cin     //стандартный поток ввода
cerr    //стандартный поток диагностики
```

Использование потоков со значениями и переменными аналогично инструкции присваивания:

В файле io.cpp

```
int i;
double x;

   cout << "\nВведите число с двойной точностью: ";
   cin >> x;
   cout << "\nВведите положительное целое: ";
   cin >> i;
   if (i < 1)
      cerr << "Ошибка, i = " << i << endl;
   cout << "i * x = " << i * x;
```

Первая инструкция (инструкция вывода) помещает строку на экран. Вторая ожидает ввода с клавиатуры, чтобы присвоить переменной с двойной точностью `x` преобразованное значение введенной строки. Введенная строка представляет собой выражение, которое либо имеет двойную точность, либо может конвертироваться при присвоении в число с двойной точностью. Ввод чего-либо другого вызовет ошибку. Обратите внимание, как две последние инструкции производят множественные перенаправления в выходные потоки. Инструкции выполняются слева направо. Например, если `i` получит значение –1, на экране появится сообщение об ошибке:

```
Ошибка, i = -1
```

Идентификатор `endl` называется *манипулятором*. Он очищает поток `cerr` и добавляет новую строку. Последняя инструкция печатает строку `i * x =`, а затем — значение выражения `i * x`.

2.3. Структура программы

Программа на C++ — это набор функций и объявлений. Язык имеет блочную структуру; память для переменных, объявленных внутри блоков, выделяется автоматически при входе в блок. Параметры передаются по значению (call-by-value), если не ука-

зано иное. Следующая программа вычисляет наибольший общий делитель для двух целых.

В файле gcd.cpp

```
//Программа для вычисления наибольшего общего делителя
//(GCD – Greatest Common Divisor)

#include  <iostream.h>
#include  <assert.h>

int gcd(int m, int n)   //определение функции
{                       //блок
  int  r;               //объявление остатка

  while (n != 0) {      //не равно
    r = m % n;          //оператор деления с остатком
    m = n;              //присваивание
    n = r;
  }                     //конец цикла while
  return m;             //выход из функции gcd со значением m
}

int main()
{
  int  x, y, g;

  cout << "\nПРОГРАММА GCD НА C++";
  do {
    cout << "\nВведите два целых числа: ";
    cin >> x >> y;
    assert(x * y != 0);              //предусловие для gcd
    cout << "\nGCD(" << x << ", " << y << ") = "
         << (g = gcd(x, y)) << endl;
    assert(x % g == 0 && y % g ==0); //постусловие
  } while (x != y);
}
```

Организация программы на C++

- Ввод-вывод в C++ обеспечивается внешней стандартной библиотекой. Информация, необходимая программе для использования этой библиотеки, находится в файле *iostream.h*.
- Проверка утверждений (assertions) в C++ обеспечивается внешней стандартной библиотекой. Информация, необходимая программе для использования этой библиотеки, находится в файле *assert.h*.
- Для преобразования программы из предварительного формата в чистый синтаксис C++ используется препроцессор, обрабатывающий набор директив, таких как директива `include`. Эти директивы предваряются символом #.
- Программа на C++ состоит из объявлений, которые могут находиться в разных файлах. Каждая функция располагается на внешнем (глобальном)

уровне; объявления функций не могут быть вложенными. Файлы программы выступают в качестве модулей и могут компилироваться отдельно друг от друга.

- Функция `main()` используется как начальная точка входа для выполнения программы. Она подчиняется правилам C++ для объявления функций. Обычно функция `main()` неявно возвращает целое значение 0, что означает нормальное завершение программы. Другие возвращаемые значения нужно задавать явно (с помощью ключевого слова `return`); они означают ошибку.
- Макро assert проверяет выполнение условий и прерывает программу, если «тест не сдан».

Компиляторы C++ могут обрабатывать программы из многих файлов. Большие программы готовятся в виде нескольких отдельных файлов. Каждый файл — это концептуальный программный модуль со своими объявлениями и определениями. Во многих системах C++ исходные файлы имеют расширение *.c*. Компилятор обычно вызывается командой *CC*. Например:

```
CC module1.c module2.c my_main.c
```

Это — команда компилятора C++ в системе UNIX, обрабатывающая три файла: *module1.c*, *module2.c* и *my_main.c*. Если компиляция не вызовет ошибок, будет создан исполняемый файл *a.out*.

2.4. Простые типы

Простыми собственными типами в C++ являются `bool`, `double`, `int` и `char`. Эти типы имеют набор значений и представление, привязанные к низкоуровневой архитектуре машины, на которой работает компилятор. В старых системах C++ нет собственного булевского типа. В них используется ноль для обозначения *false* и ненулевые значения — для *true*.

Простые типы C++ могут модифицироваться ключевыми словами `short`, `long`, `signed` и `unsigned` для получения дополнительных простых типов. В следующей таблице приведены эти типы в последовательности от коротких к более длинным. Здесь длина означает количество байт, отводимых для хранения типа.

Фундаментальные типы данных

`bool`		
`char`	`signed char`	`unsigned char`
`wchar_t`		
`short`	`int`	`long`
`unsigned short`	`unsigned`	`unsigned long`
`float`	`double`	`long double`

Этот список начинается с самого короткого типа `bool` и заканчивается самым длинным `long double`. Существует требование, согласно которому каждый более длинный тип должен занимать при хранении не меньше места, чем предыдущий тип. На большинстве машин `bool` и `char` хранятся в одном байте. На многих персональных компьютерах `int` занимает два байта, в то время как `long`, `float` и `double` требуют четыре байта для хранения.

В C++ существует оператор `sizeof`. Он используется для определения количества байт, необходимого для хранения конкретного объекта или типа.

```
//Проверим-ка!
cout << sizeof(int) << " <= " << sizeof(long) << endl;
```

Диапазон целых значений для данной системы определен в стандартном заголовочном файле *limits.h*. Вот некоторые примеры:

```
#define CHAR_BIT 8              //бит на символ
#define SCHAR_MIN (-128)        //signed char min
#define SCHAR_MAX 127           //signed char max
#define UCHAR_MAX 255           //unsigned char max
#define INT_MAX 2147483647      //int max
#define INT_MIN (-2147483648)   //int min
#define UINT_MAX 429496729U     //unsigned int max
```

Диапазон значений с плавающей точкой для данной системы определен в стандартном заголовочном файле *float.h*. Вот некоторые примеры:

```
#define FLT_EPSILON ((float)1.19209290e-07)  //single
#define FLT_MIN ((float)1.17549435e-38)      //float min
#define FLT_MAX ((float)3.40282347e+38)      //float max
#define DBL_EPSILON 2.2204460492503131e-16   //double
#define DBL_MIN 2.2250738585072014e-308      //double min
#define DBL_MAX 1.7976931348623157e+308      //double max
```

В новых системах (см. упражнение 15 на стр. 66) файл *limits* содержит шаблон `numeric_limits`, который позволяет, например, следующую запись:

```
numeric_limits<type>::max()  //максимальное значения для <типа>
```

2.4.1. Инициализация

Объявление переменной связывает имя переменной с типом. Оно не обязательно влечет выделение памяти для хранения переменной. Например:

```
//Вычисление площади круга
extern const double PI;      //объявлено, но не определено
int main()
{
  double  radius = 5.5;      //объявлено, определено,
                             //инициализовано
  double  area;              //объявлено, определено,
                             //не инициализовано

  cout << "Площадь круга: "
       << (PI * radius * radius);
}
```

Переменные `radius` и `area` объявляются и определяются внутри функции `main()`. Обычно память для их хранения выделяется из системного стека. Объявление пере-

менной является и определением, если обеспечивается выделение памяти под переменную. Неформально говоря, мы рассматриваем определение как создание объекта.

Определение может инициализовать переменную. Синтаксически инициализация представляет собой имя идентификатора с последующим *инициализатором*. Для простых переменных это обычно выглядит так:

```
тип идентификатор = выражение
```

Вот некоторые примеры:

```
{
   int  i = 5;                  //i инициализуется значением 5
   char c1, c2 = 'B';           //c1 не инициализована
   double x = 0.777, y = x + i;

   cout << x << '\t' << y;      //вывод 0.777 5.777
   cout << c2;                  //вывод 'B'
   cout << c1;                  //зависит от системы
   .....
}
```

Инициализовать можно при помощи произвольного выражения при условии, что все переменные и функции, используемые в выражении, определены. В предыдущем примере y инициализуется на основе только что определенного x. От неинициализованной переменной c1 нельзя получить какое-либо конкретное значение. Недопустимо использовать в вычислениях неинициализованную переменную до того, как ей присвоено подходящее значение. Хорошее правило: если возможно, лучше сразу инициализовать переменную, а не определять ее сначала как неинициализованную, а потом присваивать значение. Своевременная инициализация делает код более читабельным, менее подверженным ошибкам и более эффективным.

Заметьте, что объявления сами являются инструкциями и могут встречаться в любом месте блока. В приведенном фрагменте мы могли бы поместить объявление char после первой инструкции cout без всяких последствий для вывода:

```
   .....
   cout << x << '\t' << y;      //вывод 0.777 5.777
   char c1, c2 = 'B';           //инструкция объявления
   cout << c2;                  //вывод 'B'
   .....
```

2.5. Традиционные преобразования

Любое арифметическое выражение вроде x + y имеет тип и значение. Например, если обе переменные x и y одного типа (скажем int), то x + y — тоже целое. Если x и y различных типов, то x + y называется *смешанным выражением*. Предположим, x типа short, а y типа int. Тогда значение x будет преобразовано к целому, и выражение x + y получит тип int. Заметьте, что значение x в памяти остается неизменным. Преобразуется только временная копия x при вычислении значения выражения. Теперь предположим, что обе переменные x и y имеют тип short. Хотя x + y

и не является смешанным выражением, все равно произойдет автоматическое преобразование; x и y будут повышены до int и выражение будет целым.

Общие правила просты:

Автоматическое преобразование выражения «x операция y»

1. Все bool, char, short или enum повышаются до int. Целые типы, которые не могут быть представлены как int, повышаются до unsigned.
2. Если после первого шага выражение остается смешанным, то в соответствии с иерархией типов,

```
int < unsigned < long < unsigned long < float <
    < double < long double
```

операнд более низкого типа повышается до более высокого типа, и значение выражения получает этот тип. Заметьте, что unsigned повышается до unsigned long, если long не может содержать все значения unsigned.

Чтобы проиллюстрировать неявные преобразования, мы сделаем следующие объявления и перечислим различные смешанные выражения вместе с соответствующими типами:

Объявления

```
char c;          long lg;          double d;
short s;         float f;          unsigned u;      int i;
```

Выражение	Тип	Выражение	Тип
c - s / i	int	u * 3 - i	unsigned
u * 3.0 - i	double	f * 3 - i	float
c + 1	int	3 * s * lg	long
c + 1.0	double	d + s	double

Автоматическое преобразование может происходить при присваивании. Например,

```
d = i
```

переведет целое значение i в double и затем присвоит его d; тип всего выражения будет double. *Повышение* (или *расширение*) типа, как в выражении d = i, обычно будет выполнено правильно, а вот *понижение* (или *сужение*) типа, как в выражении i = d, может привести к потере информации. Здесь дробная часть d будет отброшена. Что именно произойдет, в каждом случае зависит от машины.

В дополнение к неявным преобразованиям, которые могут происходить при присваиваниях и в смешанных выражениях, существует явное преобразование, называемое *приведением* (cast)[1]. Пусть i — целое, тогда

```
static_cast<double>(i)
```

выполнит приведение значения i так, что выражение будет иметь тип double. Сама переменная i остается неизменной. Конструкция static_cast позволяет организовать преобразование типов корректно, переносимо и обратимо. Примеры:

[1] Здесь и далее когда речь идет о типах мы переводим слово *conversion* как *преобразование*, а слово *cast* — как *приведение*. — *Примеч. перев.*

```
static_cast<char>('A' + 1.0)
x = static_cast<double>(static_cast<int>(y) + 1)
```

Приведения типов, зависящие от представления или системы, используют `reinterpret_cast`:

```
i = reinterpret_cast<int>(&x)  //системно-зависимое
```

Подобные приведения типов нежелательны, и их лучше избегать.

В C++ существуют еще два специальных приведения типов: `const_cast` и `dynamic_cast`. Обсуждение `dynamic_cast` требует понимания наследования (см. раздел 10.9, «Идентификация типа на этапе выполнения» на стр. 304). Использование модификатора `const` приводит к тому, что значение переменной нельзя изменить. Изредка такое ограничение надо обойти. Это называется отменой постоянства (cast away constness). В подобных случаях используется `const_cast`, как здесь:

```
foo(const_cast<int>(c_var));//используется для вызова foo
```

Старые системы C++ предоставляют нестрогую форму приведения типов:

(тип) выражение или *тип(выражение)*

Вот некоторые примеры:

```
y = i/double(7);          //разделит с двойной точностью
ptr = (char*)(i + 88);    //целое к значению указателя
```

Такие устаревшие конструкции считаются вышедшими из употребления и далее в книге не встречаются, но они используются во многих старых компиляторах и исходных текстах. Старые системы не отличают сравнительно безопасные приведения типов, такие как `static_cast`, от менее надежных системно-зависимых, вроде `reinterpret_cast`. Кроме того, имена новых приведений типов говорят сами за себя. Например, назначение `const_cast` очевидно из названия.

Следующая программа переводит мили в километры. Значение в милях будет храниться как целое, а километры должны вычисляться с плавающей точкой.

В файле m_to_k.cpp

```
//Перевод миль в километры

#include  <iostream.h>
const double m_to_k = 1.609;//коэффициент преобразования

inline double  mi_to_km(int miles)
{
   return (miles * m_to_k);
}

int main()
{
  int miles;
  double kilometers;

  do {
```

```
        cout << "\nВведите расстояние в милях: ";
        cin >> miles;
        kilometers = mi_to_km(miles);
        cout << "\nЭто примерно "
             <<  static_cast<int>(kilometers)
             << " км"<< endl;
    } while (miles > 0);
}
```

Эта программа содержит две функции. Каждая из них имеет собственную локальную область видимости, в которой объявлены ее переменные. Каждая переменная имеет тип. Переменная `m_to_k` инициализована — ей присвоено значение 1.609, а модификатор `const` обеспечивает неизменность этого значения. Давать переменным мнемонические имена — это хорошая программистская практика, делающая код самодокументированным. Обратите внимание, что переменная, объявленная как `const`, должна быть инициализована в момент объявления. Ключевое слово `inline` модифицирует определение функции; оно указывает компилятору, что в точке обращения к функции определяющий ее код не вызывается, а встраивается.

Тип выражения `miles * m_to_k` повышается до `double`. Понятно, что тип `int` переменной `miles` *уже*, чем `double`. Инструкция ввода `cin >> miles` принимает ввод с клавиатуры в форме строки, преобразуемой в целое. Например, введенное число 5.45 будет преобразовано и присвоено переменной `miles` как целое значение 5.

Надежное преобразование `static_cast<int>(kilometers)` обрезает значение с двойной точностью до целого. Без этого явного преобразования переменная `kilometers` напечаталась бы как `double`.

2.6. Перечислимые типы

Ключевое слово `enum` используется для объявления особого целого типа с набором именованных целых констант, называемых *константами перечислимого типа* или перечислимыми константами (enumerators). Рассмотрим объявление:

```
enum suit { clubs, diamonds, hearts, spades };
```

Здесь создается целый тип с четырьмя названиями мастей карт как целыми именованными константами. Перечислимые константы имеют идентификаторы `clubs`, `diamonds`, `hearts` и `spades`, а их значения равны 0, 1, 2 и 3 соответственно. Эти значения присваиваются по умолчанию, первой перечислимой константе дается целое значение 0. Каждый следующий элемент списка больше на единицу, чем его сосед слева. Теперь идентификатор `suit` — это уникальный целый тип, отличающийся от других целых типов. Такой идентификатор называется *теговым именем* (tag name).

Перечислимые константы могут быть инициализованы произвольными целыми константами:

```
enum ages { laura = 7, ira, harold = 59, philip = harold + 7 };
```

Перечислимые константы могут быть инициализованы целыми выражениями. Заметьте, что когда нет явного инициализатора, выполняется правило по умолчанию, то есть значение `ira` равно 8.

Теговые имена и имена перечислимых констант должны быть уникальны в пределах области видимости. Значения перечислимых констант не обязательно различаются. Перечислимые константы могут быть неявно преобразованы к обычному целому типу, но не наоборот.

В файле enum_tst.cpp

```
enum signal { off, on } a = on;  //a присвоено значение on
enum answer { no, yes, maybe = -1 } b;
enum negative { no, false } c;   //недопустимо, no и
                                 //false переобъявлены
int  i, j = on;                  //допустимо, on
                                 //преобразована в 1
a = off;                         //допустимо
i = a;                           //допустимо, i становится 0
b = a;                           //недопустимо,
                                 //два разных типа
b = static_cast<answer>(a);      //допустимо, явное приведение
b = (a ? no : yes);              //допустимо,перечислимые
                                 //константы типа answer
```

Перечислимые константы могут объявляться *анонимно*, без тегового имени. Например:

```
enum { LB = 0, UB = 99 };
enum { lazy, hazy, crazy } why;
```

Первое объявление иллюстрирует обычный способ задания мнемонических целых констант. Во второй строке объявляется переменная `why` (причина) перечислимого типа с допустимыми значениями `lazy` (ленивый), `hazy` (поддатый) и `crazy` (чокнутый).

2.7. Выражения

В C++ имеется множество форм операторов и выражений. Например, присваивание является выражением. Следующее допустимо в C++:

```
a = b + (c = d + 3);
```

Использование арифметических выражений в C++ согласуется с обычной практикой. Одно важное отличие состоит в том, что результат оператора деления (/) зависит от типа аргументов.

```
a = 3 / 2;
a = 3 / 2.0;
```

Второе важное отличие заключается в том, что в C++ принята терпимая позиция по отношению к смешению типов и автоматическому преобразованию. C++ допускает расширяющие преобразования типов, так что `int` может быть расширен до `double` при присваивании. С другой стороны, в C++ разрешаются присваивания с преобразованием к более узкому типу; так, `double` может быть присвоен типу `int` и даже `char`.

```
int  i;
char ch;
double b = 1.9;

i = b / 2.0;        //i присвоено значение 0
ch = 'A' + 1.0;     //ch равно 'B'
```

Такое либеральное отношение C++ к преобразованиям потворствует плохому стилю программирования и не должно поощряться. Сужающие тип преобразования и смешанные типы могут влиять на правильность программы и должны применяться с осторожностью.

В современных системах C++ для контроля за потоком управления в инструкциях различных типов используются булевские значения `true` и `false`. В следующей таблице представлены инструкции C++, наиболее часто используемые для управления логикой выполнения.

Операторы сравнения, равенства и логические

Операторы *сравнения*	меньше	<
	больше	>
	меньше или равно	<=
	больше или равно	>=
Операторы *равенства*	равно	==
	не равно	!=
Логические операторы	(унарное) отрицание	!
	логическое и	&&
	логическое или	||

Для операторов сравнения, равенства и логических, также как и для всех остальных, существуют правила приоритета и порядка выполнения, точно определяющие, как должны вычисляться выражения, содержащие эти операторы (см. приложение B, «Приоритет и порядок выполнения операторов»). Оператор отрицания `!` является унарным. Все остальные операторы сравнения, равенства и логические операторы — бинарные. Они действуют на выражения и вырабатывают булевские значения `false` или `true`. Это заменяет более раннее соглашение C++, когда типа `bool` в языке не существовало, и за `false` принимался ноль, а за `true` — не ноль. Однако там, где ожидается логическое значение, арифметическое выражение автоматически преобразуется в соответствии с этим соглашением, так что ноль понимается как `false`, а не ноль — как `true`. То есть старый стиль программирования по-прежнему работает корректно.

Одна из ловушек C++ заключается в том, что операторы равенства и присваивания визуально похожи. Выражение `a == b` является проверкой на равенство, тогда как `a = b` — это выражение присваивания. Одна из наиболее распространенных ошибок программирования на C++ может выглядеть как-нибудь так:

```
if (i = 1)
  //делать что-нибудь
```

Конечно, подразумевалось

```
if (i == 1)
  //делать что-нибудь
```

В первой инструкции `if` переменной `i` присваивается значение 1, и результат выражения присваивания также равен 1, поэтому условие в скобках будет *всегда* выполнено (не ноль, то есть `true`). Подобную ошибку бывает очень трудно найти.

Логические операторы `!`, `&&` и `||` вырабатывают булевское значение `true` или `false`. Логическое отрицание может быть применено к любому выражению. Если выражение имело значение `false`, то его отрицание выработает `true`.

Приоритет `&&` выше, чем `||`, но оба оператора имеют более низкий приоритет, чем все унарные, арифметические операторы и операторы сравнения. Порядок их выполнения — слева направо.

При вычислении выражений, являющихся операндами `&&` и `||`, процесс вычисления прекращается, как только известен результат, `true` или `false`. Это называется *вычислением по короткой схеме* (short-circuit evaluation). Пусть *expr1* и *expr2* — выражения. Если *expr1* имеет значение `false`, то в

```
expr1 && expr2
```

expr2 не будет вычисляться, поскольку значение логического выражения уже определено как `false`. Аналогично, если *expr1* есть `true`, то в

```
expr1 || expr2
```

expr2 не будет вычисляться, поскольку значение логического выражения уже определено как `true`.

Из всех операторов C++ у оператора *запятая* самый низкий приоритет. Это бинарный оператор с выражениями в качестве операндов. В выражении с запятой вида

```
expr1 , expr2
```

первым вычисляется *expr1*, затем *expr2*. Все выражение с запятой в целом имеет значение и тип своего правого операнда. Например,

```
sum = 0, i = 1
```

Если `i` была объявлена как целое, то данное выражение с запятой имеет значение 1 и тип `int`. Оператор запятая обычно применяется в контрольном выражении итерационного процесса, когда одного действия недостаточно. Порядок выполнения операторов запятая — слева направо.

Условный оператор `?:` необычен тем, что это тройной оператор. Он принимает в качестве операндов три выражения. В конструкции

```
expr1 ? expr2 : expr3
```

первым вычисляется выражение *expr1*. Если оно истинно, то вычисляется *expr2*, и оно становится значением условного выражения в целом. Если *expr1* ложно, то вычисляется *expr3*, и уже оно становится значением условного выражения в целом. В следующем примере условный оператор используется для присваивания переменной x наименьшего из двух значений:

```
x = (y < z) ? y : z;
```

Скобки здесь не обязательны, так как оператор сравнения имеет более высокий приоритет, чем оператор присваивания. Однако, использование скобок — это хороший стиль, поскольку с ними ясно, что с чем сравнивается.

Тип условного выражения

```
expr1 ? expr2 : expr3
```

определяется операндами *expr2* и *expr3*. Если они разных типов, то применяются обычные правила преобразования. Тип условного выражения не может зависеть от того, какое из двух выражений *expr2* или *expr3* будет вычислено. Порядок выполнения условных операторов ? : — справа налево.

C++ предоставляет битовые операторы. Они действуют на машинно-зависимое битовое представление целых операндов.

Битовый оператор	Значение
~	побитовое отрицание
<<	побитовый сдвиг влево
>>	побитовый сдвиг вправо
&	побитовое И
^	побитовое исключающее ИЛИ
|	побитовое ИЛИ

Обычно операторы сдвига перегружаются для осуществления ввода-вывода.

В C++ *вызов функции* `()` и *индексация массива* `[]` рассматриваются как операторы. Существуют также оператор *определения адреса* & и оператор *обращения по адресу* или *разыменования* *. Оператор определения адреса является унарным; он возвращает адрес памяти, по которому хранится объект. Оператор обращения по адресу также является унарным и применяется к указателям. С его помощью можно получить значение по адресу, на который ссылается указатель. Это называется разыменованием указателя (см. раздел 3.10.1, «Определение адреса и разыменование» на стр. 84).

В C++ также имеется оператор `sizeof`, который используется для определения того, сколько байт потребуется для хранения конкретного объекта. Для динамически распределяемых объектов очень важно выделять достаточный объем памяти.

2.8. Инструкции

В C++ много различных типов *инструкций* (statements)[1]. Для обозначения конца инструкции используется точка с запятой. Имеется также инструкция-выражение. Например, инструкция присваивания в C++ синтаксически является выражением присваивания с последующей точкой с запятой.

2.8.1. Присваивание и выражения

В C++ присваивание выполняется как часть выражения присваивания, которое встречается в нескольких формах.

```
a = b + 1;      //(b + 1) присваивается a
```

В этом выражении вычисляется правая часть присваивания, затем она преобразуется к значению, совместимому с переменной в левой части. Полученное значение при-

[1] См. сноску на стр. 41.

сваивается левой части. Левая часть выражения должна быть *именующим выражением* (*lvalue*). Lvalue — это место в памяти, где значение может храниться и откуда оно извлекается. Простые переменные — это lvalue.

Компиляторы C++ свободно обращаются с последовательностью вычисления выражений. На практике нужная последовательность вычисления выражений может быть гарантирована использованием оператора , (запятая) или ; (завершение инструкции). Обе эти формы требуют, чтобы выражения вычислялись последовательно.

```
a = b() + e() * f();                    //порядок может быть
                                        //изменен
a = b(), t=e(), t=t * f(), a=a + t;     //фиксированный порядок
a = b(); t=e(); t=t * f(); a=a + t;     //фиксированный порядок
```

C++ допускает несколько присваиваний в одной инструкции.

```
a = b + (c = 3);
```

что равнозначно

```
c = 3; a = b + c;
```

C++ предлагает операторы присваивания, комбинирующие присваивание с каким-нибудь другим оператором:

```
a += b;      равнозначно  a = a + b;
a *= a + b; равнозначно  a = a * (a + b);
```

В C++ имеются также оператор увеличения на единицу (инкремента) (++) и уменьшения на единицу (декремента) (--) в префиксной и постфиксной форме. В префиксной форме оператор инкремента прибавляет 1 к значению, хранящемуся в lvalue-выражении, на которое он действует. Аналогично, оператор декремента вычитает 1 из значения, хранящегося в lvalue-выражении, на которое он направлен.

```
++i;       равнозначно  i = i + 1;
--x;       равнозначно  x = x - 1;
```

Постфиксная форма ведет себя по-другому, нежели префиксная. В постфиксной форме lvalue изменяется после того, как вычислена остальная часть выражения.

```
j = ++i;       равнозначно i = i + 1; j = i;
j = i++;       равнозначно j = i; i = i + 1;
i = ++i + i++; //опасная практика, результат зависит от системы
```

Замечание: На самом деле приведенные выше слева и справа варианты не совсем равнозначны. В комбинированных операторах присваивания выражение в левой части вычисляется один раз. Поэтому для сложных выражений с побочными эффектами результаты двух обсуждаемых форм могут отличаться.

Пустая инструкция записывается как отдельная точка с запятой. Она не вызывает никаких действий. Пустая инструкция обычно используется там, где какая-нибудь инструкция необходима синтаксически, но никаких действий не требуется. Такая ситуация иногда встречается в инструкциях, управляющих последовательностью выполнения.

2.8.2. Составная инструкция

Составная инструкция в C++ — это последовательность инструкций, заключенных в фигурные скобки { и }. Составные инструкции в основном применяются для группировки простых инструкций в функциональные звенья программы. Тело функции в C++ — всегда составная инструкция. В C такая инструкция называется *блоком*. В блоке объявления должны находиться в начале. Это правило ослаблено в C++, где объявления могут встречаться в любом месте списка инструкций. Везде, где можно вставить инструкцию, допускается и составная инструкция.

2.8.3. Инструкции if и if-else

В общем виде инструкция `if` выглядит так:

```
if (условие)
    инструкция
```

Если *условие* истинно, то выполняется *инструкция*; в противном случае *инструкция* пропускается. После того, как инструкция `if` выполнена, управление передается следующей инструкции. *условие* — это выражение или объявление с инициализацией, с помощью которого выбирается логика выполнения. В следующем примере фраза `Выше точки замерзания!` будет напечатана только когда `temperature` больше или равна 0. Вторая инструкция выполняется всегда. Обычно выражение в инструкции `if` является выражением сравнения, равенства или логическим выражением.

В файле if_test.cpp

```
if (temperature >= 0)
    cout << "Выше точки замерзания!\n";
cout << "Температура " << temperature << endl;
```

С инструкцией `if` тесно связана инструкция `if-else`. В общем виде она имеет форму:

```
if (условие)
    инструкция1
else
    инструкция2
```

Если *условие* истинно, то выполняется *инструкция1*, а *инструкция2* пропускается; если *условие* ложно, то пропускается *инструкция1*, а *инструкция2* выполняется. После того, как инструкция `if-else` выполнена, управление передается следующей инструкции. Рассмотрим такой код:

В файле if_test.cpp

```
if (x < y)
  min = x;
else
  min = y;
cout << "min = " << min;
```

Если условие `x < y` истинно, то переменной `min` будет присвоено значение `x`; если оно ложно, то `min` будет присвоено значение `y`. После выполнения инструкции `if-else` будет напечатано значение `min`.

2.8.4. Инструкция while

В общем виде инструкция `while` выглядит так:

```
while (условие)
   инструкция
```

Сначала вычисляется *условие*. Если оно истинно, то выполняется *инструкция*, и управление передается обратно в начало цикла `while`. В результате тело цикла `while`, а именно *инструкция*, выполняется несколько раз, до тех пор, пока условие не станет ложно. С этого момента управление передается следующей за циклом инструкции. То есть *инструкция* может быть выполнена ноль или более раз.

Вот пример инструкции `while`:

В файле while_t.cpp

```
while (i <= 10) {
   sum += i;
   ++i;
}
```

Допустим, что начальное значение `i` было 1, а `sum` — 0. В цикле `while` значение `sum` увеличивается на текущее значение `i`, а затем `i` увеличивается на 1. После того как тело цикла выполнится 10 раз, значение `i` станет равным 11, и значение выражения `i <= 10` будет ложно. Тело цикла не будет выполняться и управление перейдет к следующей инструкции. При выходе из цикла значение `sum` будет равно 55.

2.8.5. Инструкция for

Вот общий вид инструкции `for`:

```
for (начальная_инструкция условие; выражение)
   инструкция
следующая_инструкция
```

Семантика инструкции `for` такова, что сначала выполняется *начальная_инструкция*; она инициализует переменную, используемую в цикле. Затем проверяется *условие*. Если оно истинно, то выполняется *инструкция*, вычисляется *выражение*, и управление передается обратно в начало цикла `for` с той разницей, что *начальная_инструкция* уже не выполняется. Это продолжается до тех пор, пока *условие* не станет ложно, после чего управление передается *следующей_инструкции*.

Начальной_инструкцией может быть инструкция-выражение или просто объявление. Будучи объявленной, переменная имеет область видимости инструкции `for`. Инструкция `for` является итерационной инструкцией, используемой обычно с увеличивающейся или уменьшающейся переменной. Например, в следующем коде инструкция `for` используется для сложения целых чисел от 1 до 10.

В файле for_test.cpp

```
sum = 0;
for (i = 1; i <= 10; ++i)
  sum += i;
```

Другой пример показывает, как оператор *запятая* может быть использован для инициализации более одной переменной.

В файле for_test.cpp

```
for (factorial = n, i = n - 1; i >= 1; --i)
  factorial *= i;
```

Любое или все выражения в инструкции `for` могут отсутствовать, но две точки с запятой должны быть обязательно. Если пропущена *начальная_инструкция*, то никакая инициализая в цикле `for` не выполняется. Если пропущено *выражение*, то не производится приращение, а в случае отсутствия *условия* не производится проверка. Есть специальное правило для тех случаев, когда пропущено *условие*; в такой ситуации компилятор будет считать условие выполненным всегда. Так, цикл `for` в следующем фрагменте бесконечен.

В файле for_test.cpp

```
for (i = 1, sum = 0 ; ; sum += i++ )
  cout << sum << endl;
```

В инструкции `for` может присутствовать локальное объявление переменной управления циклом, как в следующем примере.

В файле for_test.cpp

```
for (int i = 0; i < N; ++i)
  sum += a[i];      //сумма элементов массива
                    //a[0] + ... + a[N - 1]
```

Целая переменная `i` является локальной в предложенном цикле. В ранних системах C++ она считалась объявленной внутри окружающего блока. Это может вносить путаницу, поэтому лучше объявлять все автоматические программные переменные в заголовке блока.

2.8.6. Инструкция do

Инструкция `do` представляет собой вариант инструкции `while`. Но вместо проверки условия в начале цикла, в инструкции `do` она производится в конце. Это значит, что инструкция, контролируемая условием `do`, выполняется по крайней мере один раз, тогда как `while` может вообще не передать управление своему телу, если условие изначально не выполняется. Вот общий вид цикла `do`:

```
do
  инструкция
```

```
while (условие);
следующая_инструкция
```

Сначала выполняется *инструкция*, затем вычисляется *условие*. Если оно истинно, то управление передается обратно на начало инструкции do и процесс повторяется. Когда значение *условия* становится ложно, управление передается *следующей_инструкции*. Допустим, мы хотим ввести положительное целое и убедиться, что оно в самом деле положительно.

В файле do_test.cpp

```
do {
   cout << "\nВведите положительное целое: ";
   cin >> n;
} while (n <= 0);
```

Пользователю предлагается ввести положительное целое. Отрицательное или нулевое значение вызовут повторение цикла с запросом нового значения. Управление выйдет из цикла только после того, как будет введено положительное целое.

2.8.7. Инструкции break и continue

Чтобы прервать нормальное выполнение цикла, программист может использовать две специальные инструкции:

```
break;     и     continue;
```

Инструкция `break` кроме использования в циклах может применяться в инструкции `switch`. Она вызывает выход из тела цикла или инструкции `switch`.

Следующий пример иллюстрирует использование инструкции `break`. Производится проверка на отрицательное значение, и если условие выполняется (значение отрицательно), инструкция `break` вызывает выход из цикла `for`. Управление программой перескакивает к инструкции, следующей сразу за циклом.

В файле for_test.cpp

```
for (i = 0; i < 10; ++i) {
   cin >> x;
   if (x < 0.0) {
      cout << "Все!" << endl;
      break;    //выход из цикла,
                //если значение отрицательно
   }
   cout << sqrt(x) << endl;
}

//break прыгает сюда
.....
```

Это типичное применение инструкции `break`. Когда выполняется специальное условие, производится соответствующее действие и выход из цикла.

Инструкция continue вызывает остановку текущей итерации цикла и немедленный переход к началу очередной итерации. Следующий фрагмент обрабатывает все символы, кроме цифр.

В файле for_test.cpp

```
for (i = 0; i < MAX; ++i) {
   cin.get(c);
   if (isdigit(c))
      continue;
.....        //обработка других символов

//continue прыгает сюда
}
```

Когда выполняется инструкция continue, управление перепрыгивает в точку перед закрывающей фигурной скобкой, что приводит к выполнению цикла с головы. Обратите внимание, что continue завершает текущую итерацию, в то время как инструкция break заканчивает цикл.

Инструкция break может встречаться только внутри тела инструкций for, while, do или switch. Инструкция continue может использоваться только внутри тела инструкций for, while или do.

2.8.8. Инструкция switch

Инструкция switch — разветвляющаяся условная инструкция, похожая на if-else. В общем виде она выглядит так:

```
switch (условие)
   инструкция
```

где *инструкция* — обычно составная инструкция, содержащая метки case и необязательную метку default. Как правило, switch включает несколько case. *Условие*, стоящее в круглых скобках за ключевым словом switch, определяет, какой из case будет выполняться, если это вообще произойдет.

В следующем примере инструкция switch подсчитывает количество оценок по категориям.

В файле switch_t.cpp

```
switch (score){
case 9: case 10:
   ++a_grades; break;
case 8:
   ++b_grades; break;
case 7:
   ++c_grades; break;
default:
   ++fails;
}
```

В инструкции switch каждая метка case должна быть уникальна. Форма метки case такова:

```
case целое_постоянное_выражение:
```

Обычно действие, выполняемое после каждой метки case, заканчивается инструкцией break. Если нет инструкции break, выполнение «спускается» к следующей инструкции, идущей за очередным case или default.

Если не была выбрана ни одна метка case, управление передается к метке default, если она есть. Метка default не обязательна, но ее использование весьма полезно. Если не выбрана ни одна метка case и нет метки default, произойдет выход из инструкции switch. Чтобы выявить ошибки, программисты часто включают метку default, даже когда все варианты case предсказуемы.

Ключевые слова case и default не могут использоваться вне инструкции switch.

Как работает инструкция switch

1. Вычисляется выражение в круглых скобках, стоящих за switch.
2. Выполняется метка case, совпадающая с тем значением, которое было найдено на этапе 1; если ни одна из case не соответствует этому значению, выполняется метка default; если метки default нет, switch прерывается.
3. Выполнение switch прерывается, когда встречается инструкция break или когда достигается конец switch.

2.8.9. Инструкция goto

Инструкция goto — наиболее примитивный способ изменения нормального хода выполнения программы. Это безусловный переход к произвольной помеченной инструкции в теле функции. В большинстве работ по современной методологии программирования инструкция goto рассматривается как вредная конструкция, поскольку она может разрушить все полезные структуры, предоставляемые другими механизмами управления логикой выполнения (for, while, do, if и switch).

Метка — это идентификатор. При выполнении инструкции goto, имеющей вид

```
goto метка;
```

управление безусловно переходит к помеченной инструкции. Вот пример:

В файле goto_tst.cpp

```
if (d == 0.0)
  goto error;
else
  ratio = n / d;
.....
error:  cerr << "ОШИБКА: Деление на ноль!\n";
```

И инструкция goto, и соответствующая ей метка должны находиться в теле одной функции. Вообще, goto лучше не использовать.

2.9. Практические замечания

Использование простой логики выполнения программы существенно для написания свободного от ошибок кода. В C++ существует множество способов повлиять на процесс выполнения, поэтому написать разветвленный код легко.

Избегайте использования инструкции `goto`. Она не так уж необходима. Вместо нее для передачи управления применяйте `break`, `continue` или `assert`, в зависимости от контекста. Фрагмент

```
//плохо!
while (условие1) {
    if (условие2)
       goto exit;
.....
}
exit:
```

равнозначен

```
//лучше
while (условие1) {
   if (условие2)
      break;
.....
}
```

Применяйте инструкцию `while` вместо инструкции `for` в тех случаях, когда при итерациях переменная цикла не увеличивается:

```
while (условие) {
.....
}
```

предпочтительнее, чем

```
for ( ; условие; ) {
.....
}
```

Советую не использовать современную возможность объявления локальной переменной в инструкции `for`, так как это может привести к ошибке на старых системах.

Избегайте приведения типов, а там, где это необходимо, применяйте новые надежные преобразования, такие как `static_cast<>`.

В программах на C директивы `define` используются для создания макро, позволяющих избежать накладных расходов на вызов функции. В C++ использовать `const` и `inline` намного предпочтительнее, чем «эквивалентные» макро `define`.

Резюме

1. Программа на C++ включает объявления, которые могут находиться в разных файлах. Каждая функция находится на внешнем или глобальном уровне;

объявление функций не может быть вложенным. Файлы исходного текста выступают в качестве модулей и могут компилироваться отдельно.

2. Функция `main()` служит в качестве точки входа в программу. Она подчиняется правилам C++ для объявления функций.

3. Для преобразования программы из предварительного формата в чистый синтаксис C++ используется препроцессор, обрабатывающий набор директив, таких как директива `include`. Подобные директивы предваряются символом #.

4. Ввод-вывод в C++ обеспечивается внешней стандартной библиотекой. Информация, необходимая программе для использования этой библиотеки, находится в файле *iostream.h*.

5. Основными простыми типами C++ являются `bool`, `double`, `int`, `char` и `wchar_t`.

6. В C++ много различных типов инструкций. Точка с запятой используется для обозначения конца инструкции. Существует также инструкция-выражение. Например, инструкция присваивания в C++ синтаксически является выражением присваивания с последующей точкой с запятой.

7. В общем виде инструкция `if` выглядит так:

```
if (условие)
  инструкция
```

Если *условие* истинно, то выполняется *инструкция*, в противном случае *инструкция* пропускается. После того как инструкция `if` выполнена, управление передается следующей инструкции.

8. Общий вид инструкции `while`:

```
while (условие)
  инструкция
```

Сначала вычисляется *условие*. Если оно истинно, то выполняется *инструкция*, и управление передается обратно в начало цикла `while`. В результате тело цикла `while`, а именно *инструкция*, выполняется несколько раз, до тех пор пока условие не станет ложно. С этого момента управление передается следующей инструкции.

9. Чтобы прервать нормальное выполнение цикла, программист может использовать две специальные инструкции

```
break;    и    continue;
```

10. Инструкция `break` кроме использования в циклах может применяться в инструкции `switch`. Она вызывает выход из тела цикла или инструкции `switch`.

11. Инструкция `goto` — наиболее примитивный способ изменения нормального хода выполнения программы. Это безусловный переход к произвольной помеченной инструкции в теле функции. Инструкция `goto` рассматривается как вредная конструкция, ее лучше избегать.

Упражнения

1. Следующий фрагмент кода — правильный и рабочий, но написан плохо. Перечислите все недостатки.

```
//Плохо написанная функция определения
//наибольшего общего делителя
int foo(int q1, int q2){int q3;
while(q2){q3=q1%q2;q1=q2;q2=q3;}return
q1;}
```

2. Перепишите функцию `gcd()` из раздела 2.3, «Структура программы», на стр. 42, заменив цикл `for` на цикл `while`.

3. Напишите функцию `main()`, вызывающую функцию `gcd()`. Предложите пользователю ввести два целых числа, вычислите и напечатайте результат. Выход из программы должен осуществляться после вычисления пяти наибольших общих делителей.

4. Перепишите программу *gcd* так, чтобы считывать значение переменной `how_many` (сколько), в которой хранится количество подлежащих вычислению наибольших общих делителей. Переменная `how_many` будет использоваться для выхода из цикла `for`.

5. В большинстве систем стандартный поток ввода может быть переключен на файл. Допустим, программа *gcd* была скомпилирована в исполняемый файл с именем *gcd*. Команда

```
gcd < gcd.dat
```

принимает ввод из файла gcd.dat и записывает ответы на экран. Проверьте это, используя файл, содержащий:

```
4   4 6   6 21   8 20   15 20
```

В большинстве систем вывод также может быть перенаправлен в файл. Команда

```
gcd > gcd.ans
```

поместит результат в файл, принимая ввод с клавиатуры. Введите те же данные и проверьте файл *gcd.ans* чтобы убедиться, что в нем содержится четыре правильных ответа. Можно скомбинировать два перенаправления:

```
gcd < gcd.dat > gcd.ans
```

Здесь ввод принимается из файла *gcd.dat*, а вывод направляется в файл *gcd.ans*. Проверьте это на своей системе.

6. На машине, поддерживающей кодовую таблицу ASCII, напишите программу с использованием цикла `for`, которая печатала бы по очереди все прописные и строчные буквы.

7. Вычисление по короткой схеме — важное свойство C++. Следующий фрагмент иллюстрирует это на типичном примере:

```
//Нахождение корней уравнения: a*x*x+b*x+c

cin >> a >> b >> c;
discr = b*b -4*a*c;
if ((discr > 0) && (sq_disc = sqrt(discr))) {
  root1 = (-b + sq_disc) / (2 * a);
  root2 = (-b - sq_disc) / (2 * a);
}
else if (discr < 0) {//комплексные корни
.....
}
else
  root1 = root2 = -b / (2 * a);
```

Функция `sqrt()` вызовет ошибку при отрицательных значениях, но вычисление по короткой схеме защищает программу от этой ошибки. Закончите эту программу, вычислив и напечатав корни уравнения при следующих исходных значениях:

```
a =1.0, b = 4.0, c = 3.0
a =1.0, b = 2.0, c = 1.0
a =1.0, b = 1.0, c = 1.0
```

8. *Инструкция C++* `switch` *допускает выполнение двух и более* `case` *для одного значения, путем «спуска» по телу* `switch`.

```
switch (i) {                              //"спускаемся"; оценки:
case 0: case 1: ++hopeless;               //безнадежно,
case 2: case 3: ++weak;                   // слабо,
case 4: case 5: ++fails; break;           //неудовлетворительно,
case 6: case 7: ++c_grades; break;        //удовлетворительно,
case 8: ++b_grades; break;                //хорошо,
case 9: ++a_grades; break;                //отлично
default: cout << "Неверная оценка " << i << endl;
}
```

Воспроизведите вручную работу этой инструкции для `i`, равного 1.

9. Некоторые теоретики программирования не признают семантику «спуска», считая, что она приводит к ошибкам в управлении логикой программы. Перепишите `switch` из предыдущего упражнения так, чтобы избежать «спуска».

10. *Следующая программа на C подсчитывает различные символы в ASCII-файле.* Запишите эту программу в файл *count.c*. Откомпилируйте ее и проверьте, используя переключение ввода на файл *count.c*.

```
/* Подсчет пробелов, цифр, букв,
   переводов строки и остальных символов
*/
#include  <stdio.h>
```

```
int main()
{
   int  blank_cnt = 0, c, digit_cnt = 0, //пробелы, цифры
   letter_cnt = 0, nl_cnt = 0, other_cnt = 0;
      //буквы, переводы строки, все остальные символы
   while ((c = getchar()) != EOF)
      if (c == ' ')
          ++blank_cnt;
      else if (c >= '0' && c <= '9')
          ++digit_cnt;
      else if (c >= 'a' && c <= 'z' || c >= 'A' &&
          c <= 'Z')
          ++letter_cnt;
      else if (c == '\n')
          ++nl_cnt;
      else
          ++other_cnt;

   printf("%10s%10s%10s%10s%10s%10s\n\n",
      "blanks", "digits", "letters", "lines",
      "others", "total");
   printf("%10d%10d%10d%10d%10d%10d\n\n",
      blank_cnt, digit_cnt, letter_cnt, nl_cnt,
      other_cnt, blank_cnt + digit_cnt + letter_cnt +
      nl_cnt + other_cnt);
}
```

Функция `getchar()` считывает один символ со стандартного ввода. В большинстве систем константа `EOF` (End Of File) имеет значение –1. Это системное значение считывается, когда достигается конец файла. В упражнении 22 на стр. 67 мы переделаем эту программу, чтобы она была независима от ASCII-кода.

11. Измените предыдущий пример так, чтобы в нем использовалась библиотека *iostream.h*. Многие по-прежнему предпочитают пользоваться *stdio.h*, поскольку она очень гибкая и проверена годами, однако она ненадежна с точки зрения типов. В объявлении

```
int printf(const char* cntrl_str, ...)
```

используется эллипсический[1] аргумент, означающий, что функция может быть вызвана с произвольным списком фактических параметров неопределенных типов. Такой изменяемый список аргументов легко может привести к ошибкам на этапе выполнения.

12. Измените код из упражнения 10 на стр. 64 так, чтобы в нем использовалась инструкция `switch`.

[1] Эллипсис (*лингвист.*) — пропущенная, но подразумеваемая конструкция. — *Примеч. перев.*

13. Переделайте упражнение 10 на стр. 64 так, чтобы с помощью `assert` проверялась правильность подсчетов. То есть общее число символов должно равняться сумме сосчитанных символов всех видов.

14. Используйте оператор `sizeof` для определения количества байт, требуемого в вашей системе для хранения: `bool`, `char`, `short`, `int`, `long`, `float`, `double`, `long double`. То же самое проделайте для перечислимых типов:

```
enum bounds { lb = -1, ub = 511 };
enum suit { clubs, diamonds, hearts, spades };
```

15. С помощью следующей программы выясните предельные значения в вашей системе:

```
#include <iostream.h>
#include <limits>
using namespace std;

int main()
{
  numeric_limits<double> dbl_typ;
  cout << "Предельные значения double: "
       <<  dbl_typ.min() << " , "
       << dbl_typ.max() << endl;
}
```

Вместо `double` могут быть записаны другие типы. Это часть новой стандартной библиотеки шаблонов.

16. Измените предыдущую программу так, чтобы она печатала диапазоны всех основных числовых типов.

17. Напишите программу для перевода температуры из шкалы Цельсия в шкалу Фаренгейта. В программе должны использоваться целые значения, и выводимые результаты также должны округляться до целых. Напомним, что 0 по Цельсию составляет 32 по Фаренгейту. А в одном градусе по Цельсию — 1.8 градуса по Фаренгейту.

18. Напишите программу, понимающую на входе значения температуры и по Цельсию, и по Фаренгейту, и выводящую значения в другой шкале. Например, на входе 0C — на выходе 32F; на входе 212F — на выходе 100C.

19. Добавьте проверку утверждений (assert) к программе преобразования Цельсий-Фаренгейт, для того, чтобы проверять корректность входных (предусловие) и выходных (постусловие) значений. Постусловие может проверяться с помощью обратного преобразования, чтобы убедиться в том, что вы получили изначально введенное значение. Не забудьте принять во внимание округления и машинные ошибки точности.

20. Вспомните, что выражение с запятой — это последовательность (слева направо) разделенных запятой подвыражений, вычисляемая в строгом порядке, и упростите следующий код:

```
for (sum = i = 0, j = 2, k = i + j; i < 10 || k < 15;
     ++i, ++j, ++k)
  sum += (i < j)? k : i;
```

21. (*Сложное*) В мире C более гибкий файловый ввод-вывод возможен с применением объявления `FILE` и файловых операций, доступных из *stdio.h*. В C++ используется *fstream.h*, как описано в приложении D, «Ввод-вывод». Ознакомьтесь с этой библиотекой и переделайте программу из упражнения 5 на стр. 63 так, чтобы в ней использовалась *fstream.h*. Программа должна получать аргументы из командной строки:

```
gcd  gcd.dat  gcd.ans
```

22. Многие наши упражнения и примеры подразумевают использование кодировки ASCII. Существуют другие широко используемые кодировки, среди них IBM EBCDIC. Используя стандартную библиотеку *ctype.h*, можно сделать обработку символов переносимой и не зависящей от кодировки. Переделайте упражнения 10 на стр. 64 и 11 на стр. 65, применяя *ctype.h* (см. раздел D.7, «Функции и макро в ctype.h», на стр. 416). Например, проверка того, не является ли символ десятичной цифрой, будет выглядеть так:

```
  .....
else if (isdigit(c))
  ++digit_cnt;
  .....
```

23. Проверьте следующие преобразования и попытайтесь определить, что происходит в каждом случае:

```
int i = 3, *j = &i;
bool flag = true;
double x = 1.5;
```

Используйте старые приведения и запишите каждое из значений как `(int)` и как `(double)`. Посмотрите, меняется ли что-нибудь, если использовать `static_cast<>`.

24. Создайте перечислимый тип для шахматных фигур. Напишите программу, которая будет считывать в шахматную переменную тип фигуры и красиво распечатывать названия фигур.

```
cin >> piece      //p n b r q k (первые буквы
                  //английских названий фигур)
```

Глава 3

Функции и указатели

Эта глава продолжает обсуждение базового языка C++. Основное внимание в ней уделяется функциям, указателям и массивам. Функция в C++ является основным элементом в структуре программы. Составные типы данных представлены в C++ массивами и структурами. Указатели используются в обоих случаях как механизм для доступа к этим данным.

3.1. Функции

Всякая задача может быть разбита на подзадачи, каждую из которых можно либо непосредственно представить в виде кода, либо разбить на еще более мелкие подзадачи. Данный метод носит название *пошагового уточнения* (stepwise refinement). *Функции* в C++ служат для записи программного кода этих непосредственно решаемых подзадач. Такие функции используются другими функциями, и в конечном — функцией `main()` для решения исходной задачи. Функции в C++ решают самые разнообразные задачи программирования. Некоторые функции, например `strcpy()` и `rand()`, поставляются в библиотеках. Другие могут быть написаны программистом.

3.1.1. Вызов функции

Программа на C++ составляется из одной или более функций, одна из которых — `main()`. Выполнение программы всегда начинается с функции `main()`. Когда при выполнении программы встречается имя функции, происходит обращение к этой функции (она вызывается[1]). То есть управление программой передается функции. После того как функция выполнила свою работу, управление возвращается в то место, откуда функция была вызвана — в вызывающее окружение (calling environment). В качестве простого примера рассмотрим программу *bell*, которая издает звуковой сигнал:

[1] В английском языке наряду с термином *обратиться к функции (to call function)* употребляется также термин *вызвать функцию (to invoke function)*. Английское слово *invoke* помимо прочего означает *заклинать, вызывать духов*. Пожалуй, подобные ассоциации способны облегчить понимание философии программирования на C++. — *Примеч. перев.*

В файле bell.cpp

```
//Позвоним, используя символ '\a' (звуковой сигнал)

#include  <iostream.h>
const char  BELL = '\a';

void ring()
{
   cout << BELL;
}

int main()
{
   ring();
}
```

3.2. Определение функции

В C++ код, описывающий, что делает функция, называется *определением функции* (function definition). Формально это выглядит так:

```
заголовок_функции
{
   инструкции
}
```

Все, что стоит перед первой фигурной скобкой, составляет *заголовок* определения функции, а то, что находится между фигурными скобками, является *телом* определения функции.

Заголовок функции — это:

```
тип имя(список_объявлений_параметров)
```

Спецификация *типа*, стоящая перед именем функции, является *возвращаемым типом*. Он определяет тип значения, возвращаемого функцией (если оно вообще возвращается). Механизм возврата значений объясняется в разделе 3.3, «Инструкция return», на стр. 71.

В определении функции `ring()` из предыдущего примера список параметров пуст, то есть объявлений параметров нет. Тело функции состоит из единственной инструкции. Поскольку функция не возвращает значения, возвращаемый тип этой функции — `void`. Синтаксически параметры — это идентификаторы, они могут использоваться внутри тела функции. Иногда параметры в определении функции называют *формальными параметрами*. Тем самым подчеркивается их сущность: формальные параметры — это то, вместо чего будут подставлены фактические значения, передаваемые функции в момент ее вызова. После вызова функции значение аргумента, соответствующее формальному параметру, используется в теле выполняемой функции. В C++ такие параметры являются *вызываемыми по значению* (call-by-value). Когда применяется вызов по значению, переменные передаются функции как аргументы, их значения копируются в соответствующие параметры функции, а сами переменные не изменяются в вызывающем окружении. В сущности, вызываемые по значению пара-

метры являются локальными в своей процедуре. Им могут передаваться выражения, значения которых присваиваются этим локальным переменным (параметрам).

Чтобы проиллюстрировать изложенные идеи, давайте перепишем предыдущую программу так, чтобы у функции `ring()` появился формальный параметр, задающий количество звонков.

В файле bellmult.cpp

```
//Несколько звоноков

#include  <iostream.h>
const char  BELL = '\a';

void ring(int k)
{
   int  i;

   for (i = 0; i < k; ++i)
      cout << BELL;
}

int main()
{
   int  n;

   cout << "\nВведите небольшое положительное целое:  ";
   cin >> n;
   ring(n);
}
```

3.3. Инструкция return

Инструкция `return` используется для двух целей. Когда она выполняется, управление программой немедленно передается обратно в вызывающее окружение. Кроме того, если за ключевым словом `return` следует какое-либо выражение, то его значение также передается в вызывающее окружение. Это значение должно допускать неявное преобразование к типу, указанному в заголовке определения функции.

Инструкция `return` имеет одну из двух форм:

```
return;
return выражение;
```

Вот некоторые примеры:

```
return;
return 3;
return (a + b);
```

Скобки в выражении `return` не обязательны; некоторые программисты ставят их для удобочитаемости.

В качестве примера напишем программу, которая вычисляет наименьшее из двух целых.

В файле min_int.cpp

```
//Найти наименьшее из двух целых

#include  <iostream.h>

int min(int x, int y)
{
   if (x < y)
     return x;
   else
     return y;
}

int main()
{
   int  j, k, m;

   cout << "Введите два целых числа:  ";
   cin >> j >> k;
   m = min(j, k);
   cout << '\n' << m << " наименьшее из " << j
        << " и " << k << endl;
}
```

Функция `min()` создана для работы с целыми значениями. Допустим, что теперь мы хотим использовать значения типа `double`. Перепишем `min()`:

В файле min_dbl.cpp

```
double min(double x, double y)
{
   if (x < y)
     return x;
   else
     return y;
}
```

3.4. Прототипы функций

Синтаксис функций в C++ инспирировал синтаксис *прототипов функций*, используемый компиляторами со стандартного (ANSI) C. По сути, возвращаемый тип, идентификатор функций и типы параметров составляют прототип функции. *Прототип функции* (function prototype) — это объявление функции, но не ее определение. Благодаря явному объявлению возвращаемого типа и списка типов аргументов, в C++ при обращении к функции возможны строгая проверка типов и неявные преобразования типов.

Итак, функция может быть объявлена до того, как она определена. Определение функции может идти позже в этом же файле, браться из библиотеки или из указанного пользователем файла. Прототип функции имеет следующую форму:

```
тип имя(список_объявлений_аргументов);
```

Список_объявлений_аргументов может быть пустым, содержать единственное объявление или несколько объявлений, разделенных запятыми. Если функция не имеет параметров, допускается использование ключевого слова `void`. В C++ функция с пустым списком параметров выглядит как *имя_функции()*. Такая информация позволяет компилятору отслеживать совместимость типов. Аргументы преобразуются к указанным в объявлении типам так же, как в случае присваивания.

В программе *min_int* из раздела 3.3, «Инструкция return», на стр. 71 мы использовали функцию `min()`. Ее прототип будет:

```
int min(int, int);
```

И возвращаемый тип функции, и типы аргументов в списке указываются явно. Определение функции `min()` должно соответствовать этому объявлению. Прототип функции может также содержать имена аргументов. В случае `min()` это могло бы выглядеть так:

```
int min(int x, int y);
```

В C++ используется эллипсический символ (ellipsis)[1] (`...`) для обозначения списка аргументов, который не указан, но подразумевается. Функция `printf()` из стандартной библиотеки *stdio.h* имеет следующий прототип:

```
int printf(const char* cntrl_str, ...);
```

Такая функция может быть вызвана с произвольным списком фактических параметров, но подобной практики лучше избегать, так как она не обеспечивает безопасность типов.

Вот еще одна программа, иллюстрирующая применение прототипов функций:

В файле add3.cpp

```
//Сложение трех целых – иллюстрация прототипов функций

#include  <iostream.h>

int  add3(int, int, int);
double  average(int);

int main()
{
   int  score_1, score_2, score_3, sum;

   cout << "\nВведите три оценки: ";
   cin >> score_1 >> score_2 >> score_3;
   sum = add3(score_1, score_2, score_3);
   cout << "\nИх сумма равна " << sum;
   cout << "\nИх среднее равно " << average(sum);
   sum = add3(1.5 * score_1, score_2, 0.5 * score_3);
   cout << "\nИх взвешенная сумма равна " << sum << ".";
   cout << "\nИх взвешенное среднее равно "
        << average(sum) << "." << endl;
}
```

[1] См. сноску к упражнению 11 в главе 2 на стр. 65.

```
int add3(int a, int b, int c)
{
   return (a + b + c);
}

double average(int s)
{
   return (s / 3.0);
}
```

Разбор программы add3

- `int add3(int, int, int,);`
  `double average(int);`

Эти объявления являются прототипами функций. Они информируют компилятор о типе и количестве аргументов, предполагаемых для каждой объявленной таким образом и определенной в другом месте функции.

- `sum = add3(1.5 * score_1, score_2, 0.5 * score_3);`

В C++ это выражение преобразуется к целому значению согласно описанию прототипа функции.

- ```
  int add3(int a, int b, int c)
  {
     return (a + b + c);
  }
  ```

Это собственно определение функции. Оно может быть импортировано и из другого файла. Оно соответствует объявлению прототипа функции перед `main()`.

Так как список аргументов в прототипе функции может включать имена переменных,

```
int add3(int a, int b, int c);
```

тоже допустимо.

3.5. Аргументы по умолчанию

Формальному параметру может быть задан *аргумент по умолчанию* (default argument). Обычно это константа, которая часто встречается при вызове функции. Использование аргумента по умолчанию позволяет не писать его значение при каждом вызове. Следующая функция демонстрирует сказанное:

В файле def_args.cpp

```
int sqr_or_power(int n, int k = 2)    //k = 2 по умолчанию
{
```
```

```
    if (k == 2)
      return (n * n);
    else
      return (sqr_or_power(n, k - 1) * n);
}
```

Здесь предполагается, что чаще всего эта функция применяется для вычисления значения n в квадрате.

```
sqr_or_power(i + 5)   //вычислит (i + 5) * (i + 5)
sqr_or_power(i + 5, 3)  //вычислит (i + 5) в кубе
```

Только несколько *последних* параметров функции могут иметь значения по умолчанию:

```
void foo(int i, int j = 7);                    //допустимо
void foo(int i = 3, int j);                    //недопустимо
void foo(int i, int j = 3, int k = 7);         //допустимо
void foo(int i = 1, int j = 2, int k = 3);     //допустимо
void foo(int i, int j = 2, int k);             //недопустимо
```

3.6. Перегрузка функций

Обычно выбор имени функции основан на стремлении отразить в названии ее основное назначение. «Читабельные» программы, как правило, содержат разнообразные и грамотно подобранные идентификаторы. Иногда различные функции используются для одних и тех же целей. Рассмотрим, например, функцию, которая вычисляет среднее значение массива чисел с двойной точностью и такую же функцию, но оперирующую массивом целых чисел. И ту, и другую удобно назвать avg_arr, как в следующем примере.

Перегрузка (overloading) использует одно и то же имя для нескольких вариантов оператора или функции. Выбор конкретного варианта зависит от типов аргументов, используемых оператором (функцией). Здесь мы ограничимся обсуждением перегрузки функций. Рассмотрение перегрузки операторов оставим до следующих глав, так как она в основном используется в контексте классов (см. главу 7, «Ad hoc полиморфизм»). В следующем фрагменте перегружается функция avg_arr().

В файле avg_arr.cpp

```
//Нахождение среднего значения элементов массива

double avg_arr(const int a[], int size)
{
  int  sum = 0;

  for (int i = 0; i < size; ++i)
    sum += a[i];  //выполняется сложение для int
  return (static_cast<double>(sum) / size);
}

double avg_arr(const double a[], int size)
{
  double  sum = 0.0;
```

```
    for (int i = 0; i < size; ++i)
      sum += a[i];  //выполняется сложение для double
    return (sum / size);
}
```

Следующий код демонстрирует обращение к `avg_arr()`:

```
int main()
{
   int w[5] = { 1, 2, 3, 4, 5 };
   double x[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

   cout << avg_arr(w, 5) << " среднее для int" << endl;
   cout << avg_arr(x, 5) << " среднее для double" << endl;
}
```

Компилятор выбирает функцию в соответствии с типами аргументов и их количеством. Правило, по которому осуществляется этот выбор, называется *алгоритмом соответствия сигнатуре* (signature matching algorithm). (см. раздел 7.2, «Перегрузка и выбор функций», на стр. 193). Под *сигнатурой* (signature) мы понимаем список типов, который используется в объявлении функции.

3.7. Встраивание

C++ предоставляет ключевое слово `inline`. Оно располагается перед объявлением функции, когда программист желает, чтобы код, заменяющий обращение к функции, встраивался по месту вызова функции.

В файле inline_t.cpp

```
inline double cube(double x)
{
   return (x * x * x);
}
```

Компилятор анализирует семантику этой функции точно так же, как и ее невстраиваемой версии. Ограничения компилятора не позволяют встраивать сложные функции.

Обычно для размещения в программе встроенного кода используют вызов функции, однако существует и другая схема — *раскрытие макро*. Директива препроцессора `#define` поддерживает макроподстановку, как показано в следующем примере:

В файле inline_t.cpp

```
#define  SQR(X) ((X) * (X))
#define  CUBE(X) (SQR(X)*(X))
#define  ABS(X) (((X) < 0)? -(X) : X)
   .....
   y = SQR(t + 8) - CUBE(t - 8);
   cout << sqrt(ABS(y));
```

Препроцессор раскрывает макро и передает получившийся текст компилятору. Так, приведенное выше равносильно:

В файле inline_t.cpp

```
    y = ((t+8) * (t+8)) - ((((t-8)) * (t-8)) * (t-8));
    cout << sqrt((((y) < 0)? -(y) : y));
}
```

Основная причина использования скобок — стремление избежать ошибок в последовательности вычислений, как это происходит в следующем фрагменте:

```
#define  SQR(X) X * X
   .....
   y = SQR(t + 8);  //раскроет макро в t + 8 * t + 8
```

Раскрытие макро не обеспечивает надежности с точки зрения типов, в отличие от механизма передачи параметров. Еще одна проблема возникает, когда макро приводит к вычислению параметра более одного раза, в то время как подразумевалось, что аргумент должен вычисляться только один раз, как при вызове функции.

3.8. Область видимости и класс памяти

В базовом языке существует два основных вида области видимости (scope): локальная область видимости и область видимости файла. Локальная область видимости относится к блоку. Тело функции — это пример блока; оно содержит набор объявлений, включая параметры функции. В область видимости файла входят имена, которые являются внешними (глобальными). Правила для области видимости класса мы обсудим позднее.

Основное правило области видимости состоит в том, что идентификаторы доступны только внутри блока, в котором они объявлены. Они не известны за границами блока. Вот простой пример:

В файле scope_t.cpp

```
{
   int  a = 2;              //a вне блока
   cout << a << endl;       //напечатается 2
   {                        //вход во внутренний блок
     int  a = 7;            //a внутри блока
     cout << a << endl;     //напечатается 7
   }                        //выход из внутреннего блока
   cout << ++a << endl;     //напечатается 3
}
```

Каждый блок вводит свою собственную номенклатуру. Внешнее по отношению к блоку имя действительно до тех пор, пока оно не переопределено внутри блока. Если оно переопределено, то внешнее имя скрывается (или маскируется) от внутреннего блока. Вложенность блоков может быть произвольной глубины, которая определяется ограничениями системы.

В C++ объявления могут встречаться в любом месте блока. В стандартном C все объявления для области видимости блока находятся в начале блока. Например:

В файле scope_t.cpp

```
//В C++, но не в C

int max(int c[], int size)
{
   cout << "размер массива: " << size << endl;
   int  comp = c[0];                  //объявление comp
   for (int i = 1; i < size; ++i)     //объявление i
      if (c[i] > comp)
         comp = c[i];
   return comp;
}
```

В C++ область видимости идентификатора начинается сразу после его объявления и продолжается до конца самого внутреннего блока, в котором он объявлен.

Несмотря на то, что в C++ не обязательно помещать объявления в начале блока, иногда бывает полезно делать именно так. Поскольку блоки часто невелики, такой подход помогает обеспечить хороший стиль документирования.

Помещение объявлений внутри блока позволяет инициализовать переменную введенным или вычисленным значением; лучше помещать объявления как можно ближе к тому месту, где они используются, особенно в больших блоках.

3.8.1. Класс памяти auto

Каждая переменная и функция в базовом языке C++ имеет два атрибута: *тип* и *класс памяти*. Классов памяти четыре: автоматический, внешний, регистровый и статический; им соответствуют ключевые слова:

```
auto    extern    register    static
```

Переменные, объявленные внутри тела функции, по умолчанию являются автоматическими, поэтому автоматический класс памяти наиболее часто встречается. Если составная инструкция содержит объявления переменных, к этим переменным можно обращаться внутри области видимости составной инструкции. Составная инструкция с объявлениями является *блоком*.

Объявления переменных внутри блоков неявно получают автоматический класс памяти. Для явного указания класса памяти можно использовать ключевое слово `auto`. Например:

```
auto int   a, b, c;
auto float f = 7.78;
```

Поскольку по умолчанию класс памяти автоматический, ключевое слово `auto` используется редко.

Система выделяет память для автоматических переменных при входе в блок. Переменные с инициализаторами инициализуются. В большинстве систем для таких

переменных используется та или иная форма размещения в стеке. Переменные, определенные в конкретном блоке, рассматриваются *локально* по отношению к этому блоку. При выходе из блока система больше не резервирует память, отведенную под автоматические переменные. Поэтому их значения теряются. Такой механизм легко реализовать с помощью стека, где занимаемое переменными пространство будет освобождено посредством переустановки вершины стека. Если вход в блок происходит повторно, система еще раз выделяет память, при этом предыдущие значения неизвестны. Если определение функции содержит блок, то каждый вызов этой функции устанавливает новое окружение.

3.8.2. Класс памяти register

Класс памяти `register` сообщает компилятору, что соответствующие переменные должны храниться в быстрых регистрах памяти, если это физически и семантически возможно. Поскольку ограниченность ресурсов и семантика не всегда позволяют разместить переменную в быстрой памяти, класс памяти `register` по умолчанию становится автоматическим, если компилятор не может выделить подходящий физический регистр. Обычно компилятору доступно лишь несколько таких регистров. Многие из них необходимы системе и не могут распределяться иначе.

Когда важна скорость вычислений, программист может выбрать несколько переменных, обращение к которым происходит наиболее часто, и объявить их с классом памяти `register`. Обычно кандидатами на это бывают переменные циклов и параметры функций.

```
{
   for (register i = 0; i < LIMIT; ++i) {
      .....
   }
}
```

Объявление

```
register i;     равнозначно  register int i;
```

Если в объявлении указан класс памяти, а тип переменной отсутствует, по умолчанию принимается тип `int`.

Класс памяти `register` не слишком полезен. Он понимается только как *совет* компилятору. Кроме того, современные оптимизирующие компиляторы часто более проницательны, чем программист.

3.8.3. Класс памяти extern

Один из способов передачи информации между блоками и функциями заключается в использовании внешних переменных. Когда переменная объявлена вне функции, память для нее выделяется на постоянной основе, и ее класс памяти — `extern`. Объявление внешней переменной может выглядеть так же, как объявление переменной, которая встречается внутри функции или блока. Внешняя переменная рассматривается как *глобальная* для всех функций, объявленных после нее, и при выходе из блока или функции такая переменная продолжает существовать. Подобные

переменные не могут иметь автоматический или регистровый класс памяти. Можно использовать ключевое слово `static` (см. раздел 3.8.4, «Класс памяти static», на стр. 81).

Ключевое слово `extern` используется для того, чтобы приказать компилятору «где-то поискать переменную», в этом файле или в каком-то другом. Таким образом, два файла можно компилировать отдельно. Использование ключевого слова `extern` в одном из них говорит компилятору, что переменная будет *определена* где-то — возможно в этом же файле, а может быть и в другом. Возможность компилировать файлы отдельно важна при написании больших программ.

Внешние переменные никогда не исчезают. Так как они существуют на протяжении всей жизни программы, их можно использовать для передачи значений между функциями. Они могут, однако, быть скрыты, если идентификатор переопределен. Внешние переменные можно рассматривать как переменные, объявленные в блоке, который включает в себя весь файл.

Информацию можно передать функции двумя способами: внешними переменными и с помощью механизма параметров. Хотя существуют исключения, передача параметров предпочтительнее. Такой подход ведет к улучшению модульности кода и уменьшает вероятность нежелательных побочных эффектов.

Вот простой пример использования внешних объявлений. Программа здесь размещается в двух отдельных файлах:

В файле circle3.cpp

```
const double  pi = 3.14159; //локально по отношению
                            //к circle3.cpp

double circle(double radius)
{
   return (pi * radius * radius);
}
```

В файле cir_main.cpp

```
#include  <iostream.h>

double circle(double);       //внешняя функция (автоматически)

int main()
{
   double  x;

   .....
   cout << circle(x) << " площадь круга радиуса "
        << x << endl;
}
```

На нашей UNIX-системе компиляция запускается командой *CC circle3.cpp cir_main.cpp*.

Модификатор `const` задает для переменной `pi` область видимости файла; она не может прямо импортироваться в другой файл. Если подобное определение требуется где-либо еще, оно должно быть явно модифицировано ключевым словом `extern`.

3.8.4. Класс памяти static

Статические объявления имеют два различных важных применения. Простейшее из них заключается в том, что локальная переменная сохраняет предыдущее значение при повторном входе в блок, — в отличие от обычных автоматических переменных, которые теряют свои значения при выходе из блока и в дальнейшем инициализуются заново.

В качестве примера сохранения значения с помощью `static` напишем функцию, которая подсчитывает, сколько раз она была вызвана.

В файле stat_tst.cpp

```
int f()
{
   static int called = 0;
   ++called;
   .....
   return called;
}
```

В первый раз, когда вызывается функция, переменная `called` инициализуется нулем. При выходе из функции значение `called` сохраняется в памяти. Когда функция вызывается снова, `called` *не* переинициализуется; она хранит свое значение с последнего вызова функции.

Второе и более тонкое применение `static` связано с внешними объявлениями, с которыми этот класс памяти обеспечивает механизм *закрытости* (privacy). Он очень важен для модульности программы. Под закрытостью мы понимаем ограничение *обозримости* или *области видимости* переменных или функций, доступных при прочих равных условиях.

Такое применение `static` ограничивает область видимости функции. Статические функции видны только внутри файла, в котором они определены. В отличие от обычных функций, к которым возможен доступ из других файлов, статическая функция доступна повсюду в своем файле, но ни в каком другом. Еще раз отметим: эта возможность полезна при разработке закрытых модулей определения функций. Заметьте, что в системах C++, поддерживающих пространства имен (namespaces), этот механизм может быть заменен безымянным пространством имен.

```
static int goo(int a)
{
   .....
}
int foo(int a)
{
   .....
   b = goo(a);
        //goo() доступна здесь, но не в других файлах
   .....
}
```

В C++ и внешние, и статические переменные, которые не были инициализованы программистом, инициализуются нулем самой системой. Это относится к массивам, строкам, указателям, структурам и объединениям. Для массивов и строк это означает, что каждый элемент инициализуется нулем, для структур и объединений — что каждый член будет инициализиван нулем. Напротив, автоматические и регистровые переменные обычно не инициализуются системой. Это значит, что они могут содержать мусор.

3.8.5. Таинства компоновки

Многофайловые программы требуют должной компоновки (linkage). C++ требует соблюдения определенных правил во избежание скрытых противоречий. Как уже отмечалось, имя, явно объявленное в области видимости файла как `static`, является локальным и скрыто от других файлов. Так же ведут себя функции, объявленные `inline` и переменные с модификатором `const`. Переменной `const`, находящейся в области видимости файла, но не являющейся статической, может быть задан внешний тип компоновки с помощью объявления ее `extern`. Наконец, совместная компоновка с кодом на C возможна с использованием формы:

```
extern "C" { код или включаемый файл }
```

Возможность компоновки совместно с программами на других языках зависит от системы. Некоторые системы, например, допускают `"Pascal"`.

3.9. Пространства имен

C++ унаследовал от C единое глобальное пространство имен. При объединении программ, написанных разными коллективами, могут возникать конфликты имен. В то же время C++ поощряет использование библиотек от различных поставщиков. Это привело к появлению в ANSI C++, области видимости пространства имен (namespace scope):

```
namespace LMPinc {
   class puzzles { ····· };
   class toys { ····· };
   ·····
}
```

Идентификатор пространства имен может быть использован как часть идентификатора, разрешающего область видимости:

```
идентификатор_пространства_имен::идентификатор
```

Существует также объявление `using`, которое позволяет клиенту получить доступ ко всем именам из пространства имен. Например:

```
using namespace LMPinc;
toys top;      //LMPinc::toys
```

Пространства имен могут быть вложенными:

В файле namespac.cpp

```
namespace LMPout {
   int  n;
   namespace LMPin {
      int  sq(){ return n * n; }      //LMPout::n
      void  pr_my_logo();
   }
   void  LMPin::pr_my_logo()
      { cout << "LMPinc" << endl; }
}
```

Как упоминалось в разделе 3.8.4, «Класс памяти static», на стр. 81, пространства имен могут использоваться вместо глобальных статических объявлений для обеспечения уникальной области видимости. Это делается с помощью объявления безымянного пространства имен (anonymous namespace):

```
namespace { int count = 0; }          //здесь count уникальна
//count доступна в остальной части файла
void chg_cnt(int i) { count = i; }
```

В соответствии с новыми стандартами ANSI заголовочные файлы библиотек больше не будут использовать расширение *.h*. Такие файлы как *iostream* или *complex* будут объявляться с `namespace std`. Несомненно, производители по-прежнему будут продолжать поставку заголовков и в старом стиле, таких как *iostream.h* и *complex.h*, так что старый код можно выполнять без изменений.

3.10. Типы указателей

Указатели в C++ используются для связи переменных с машинными адресами. Понятие указателя тесно сплетено с обработкой массивов и строк. Массивы в C++ могут рассматриваться как специальная форма указателя, связанная с непрерывной областью памяти для хранения индексируемой последовательности значений.

Указатели используются в программах для доступа к памяти и манипуляции с адресами. Если `v` — переменная, то `&v` — это адрес, или место в памяти, где хранится ее значение. Оператор определения адреса `&` (address operator) является унарным и имеет такой же приоритет и порядок выполнения справа налево, как и остальные унарные операторы. В программе можно объявить переменные-указатели и затем использовать их для получения адресов в качестве значений. Объявление

```
int* p;
```

говорит, что переменная `p` будет иметь тип «указатель на целое». Допустимая область значений для любого указателя всегда включает специальный адрес 0, а также набор положительных целых, которые интерпретируются как машинные адреса в зависимости от конкретной системы.

Вот некоторые примеры присваивания значений указателю `p`:

```
p = &i;                          //адрес объекта i
p = 0;                           //специальное значение
p = static_cast<int*>(1507);     //абсолютный адрес
```

В первом примере мы подразумеваем, что p «ссылается на i», или «указывает на i», или «содержит адрес i». Компилятор решает, какой адрес присвоить переменной i. Этот адрес может отличаться от машины к машине и даже может быть разным при разных запусках программы на одной и той же машине.

Второй пример — присваивание указателю p специального значения 0. Это значение обычно используется для индикации возникновения особых условий. Например, указатель со значением ноль возвращается оператором new, если исчерпана свободная память. Значение 0 также используется для обозначения того, что достигнут конец динамической структуры данных, такой как дерево или список.

В третьем примере приведение необходимо, чтобы избежать ошибки типов, и используется фактический адрес памяти.

3.10.1. Определение адреса и разыменование

Оператор *разыменования* (dereferencing) или *непрямого обращения* * является унарным и имеет такой же приоритет и порядок выполнения справа налево, как и другие унарные операторы. Если p — указатель, то *p — значение переменной, на которую указывает p. Прямое значение p — это адрес памяти, тогда как *p — это непрямое значение p, то есть значение, находящееся по адресу, который храниться в указателе p. В некотором смысле, * — это оператор, обратный &. Вот фрагмент, показывающий их взаимосвязь:

```
int  i = 5, j;
int* p = &i      //указатель инициализован
                 //адресом переменной i
cout << *p << " = i, хранится по адресу " << p << endl;
j = p;           //недопустимо, указатель
                 //не преобразуется в целое
j = *p + 1;      //допустимо
p = &j;          //p указывает на j
```

3.10.2. Вызов по ссылке на основе указателей

В этом разделе мы опишем, как адреса переменных можно использовать в качестве аргументов функций, реализуя таким образом *вызов по ссылке* (call-by-reference). В результате значения переменных могут изменяться в вызывающем окружении. Указатели используются в списке параметров для определения адресов переменных, значения которых могут изменяться. При обращении к функции адреса переменных должны передаваться как аргументы.

В файле call_ref.cpp

```
//Вызов по ссылке на основе указателей
void  order(int*, int*);
int main()
```

```
{
   int  i = 7, j = 3;

   cout << i << '\t' << j << endl;  //напечатается 7  3
   order(&i, &j);
   cout << i << '\t' << j << endl;  //напечатается 3  7
}

void order(int* p, int* q)
{
   int  temp;

   if (*p > *q) {
      temp = *p;
      *p = *q;
      *q = temp;
   }
}
```

Большая часть работы этой программы выполняется функцией `order()` Обратите внимание, что в качестве аргументов передаются адреса `i` и `j`. Эти адреса передаются по значению, и следовательно, не могут быть изменены в вызывающем окружении. Тем не менее, значения `i` и `j` (а не их адреса) могут изменяться в вызывающем окружении.

Разбор функции order().

- `void order(int* p, int* q)`

Оба параметра `p` и `q` — указатели на целое. Приведенная конструкция реализует вызов по ссылке на основе указателей. Параметрам должны передаваться адреса. В `main()` функция была вызвана как `order(&i, &j);`

- ```
  if (*p > *q) {
     temp = *p;
     *p = *q;
     *q = temp;
  }
  ```

Здесь сравниваются и, в случае неправильного порядка, меняются местами в вызывающем окружении произвольные значения, хранящиеся по адресам, на которые указывают `p` и `q`.

Следующий список резюмирует правила использования аргументов-указателей для выполнения обращения по ссылке.

Вызов по ссылке с использованием указателей

1. Объявить параметры-указатели в заголовке функции.
2. Использовать разыменованный указатель в теле функции.
3. Передать адреса в качестве аргумента при вызове функции.
  ```

3.11. Применение void

Ключевое слово `void` используется как возвращаемый тип функции, не возвращающей значения. Более важное применение `void` заключается в объявлении типа *обобщенного указателя* (generic pointer) — указателя на `void`, например `void* gp`. Такому указателю можно присвоить адрес переменной любого типа, но он не может быть разыменован. Разыменование — это операция `*`, действующая на значение указателя (адрес) для получения того, на что он указывает. Она не будет иметь смысла при разыменовании указателя на значение `void`, потому что неизвестно, какого типа информация хранится по адресу указателя. Вот несколько простых примеров использования `void*`:

```
void* gp;                          //обобщенный указатель
int* ip;                           //указатель на int
char* cp;                          //указатель на char

gp = ip;                           //допустимое преобразование
cp = ip;                           //недопустимое преобразование
*ip = 15;                          //допустимое разыменование
                                   //указателя на целое
*ip = *gp                          //недопустимое разыменование
                                   //обобщенного указателя
ip = static_cast<int*>(gp);        //допустимое преобразование
```

В основном данный тип применяется в качестве формального параметра. Например, библиотечная функция `memcpy` объявляется в *cstring* (в более ранних системах C++ или в C используйте *cstring.h*) как

```
void* memcpy(void* s1, void* s2, size_t n);
```

Эта функция копирует `n` символов из объекта, размещенного в `s2`, в объект, размещенный в `s1`. Она работает с любыми типами указателей в качестве фактических аргументов. Тип `size_t` определен в *stddef.h* и часто является синонимом `unsigned int`.

Еще одно применение `void` — вместо списка параметров в объявлении функции. Это означает, что функция не принимает аргументов.

```
int foo();      в C++ равнозначно    int foo(void);
```

Любой тип указателя может быть преобразован к обобщенному указателю типа `void*`. Вот еще преобразования указателей: имя массива является указателем на его базовый элемент; нулевое значение указателя может быть преобразовано к любому типу; тип «функция, возвращающая `T`» может быть преобразован в указатель на функцию, возвращающую `T`.

3.12. Массивы и указатели

Массив (array) — это тип данных, который используется для представления последовательности однородных значений. Доступ к элементам массива производится с помощью индексов. В C++ возможны массивы любых типов, включая массивы массивов. Объявление типичного массива выделяет память начиная с базового адреса. На

самом деле имя массива является постоянным указателем, инициализованным этим базовым адресом.

Чтобы проиллюстрировать эти идеи, давайте напишем небольшую программу, которая заполняет массив, выводит значения и суммирует элементы:

В файле sum_arr1.cpp

```
//Простая обработка массива

#include  <iostream.h>
const int  SIZE = 5;

int main()
{
   int  a[SIZE]; //резервируем место
                 //для a[0],·····,a[4]
   int  i, sum = 0;

   for (i = 0; i < SIZE; ++i) {
     a[i] = i * i;
     cout << "a[" << i << "] = " << a[i] << "     ";
     sum += a[i];
   }
   cout << "\nсумма = " << sum << endl;
}
```

Вывод этой программы выглядит так:

```
a[0] = 0     a[1] = 1     a[2] = 4     a[3] = 9     a[4] = 16
sum = 30
```

Вышеприведенному массиву требуется память для хранения пяти целых значений. Так если `a[0]` хранится по адресу 1000, то в системе, отводящей для хранения целого 4 байта, остальные элементы массива последовательно размещаются по адресам 1004, 1008, 1012 и 1016. Считается хорошей программистской привычкой определять размер массива как символическую константу. Поскольку большая часть кода может зависеть от этого значения, удобно иметь возможность изменять единственную строчку `#define` для обработки массивов различного размера. Старайтесь обращать внимание на ясность и лаконичность различных частей инструкции `for` при вычислениях над массивами.

3.12.1. Индексирование

Предположим, было сделано объявление вида:

```
int i, a[размер];
```

Тогда для доступа к элементу массива мы можем написать: `a[i]`. В более общем случае эта запись выглядит так: `a[`*выражение*`]`, где *выражение* — это целое выражение. Указанное *выражение* мы называем *индексом* (subscript или index) элемента массива `a`. В C++ значение индекса должно лежать в диапазоне от 0 до *размер*–1. Значение

индекса массива вне этого диапазона часто вызывает ошибку выполнения. Такая ситуация называется превышением границ массива или выходом индекса за границы. Это распространенная ошибка программирования. Эффект от подобной ошибки в программе на C++ зависит от системы и может быть довольно неожиданным. Одним из частых результатов является то, что будет возвращено или изменено значение некой не связанной с массивом переменной. Таким образом, программист всегда должен быть уверен, что все индексы находятся в пределах границ.

3.12.2. Инициализация

Массив может быть инициализован с помощью заключенного в фигурные скобки списка выражений, разделенных запятыми:

```
int a[4] = {9, 8, 7};//a[0]=9, a[1]=8, a[2]=7
```

Когда список инициализаторов короче размера массива, остальные элементы инициализуются нулем. Неинициализованные внешние и статические массивы автоматически инициализуются нулем. Однако автоматические массивы открываются с неопределенными значениями.

Массиву, объявленному с явным списком инициализаторов, но без задающего его размер выражения, дается размер, соответствующий количеству инициализаторов:

```
char laura[] = {'l', 'm', 'p' };
```

равнозначно

```
char laura[3] = {'l', 'm', 'p' };
```

3.13. Связь между массивами и указателями

Имя массива само по себе является *адресом* или *значением указателя*. Массивы и указатели почти идентичны в смысле их использования для доступа к памяти. Однако существуют едва уловимые, но важные различия. Указатель — это переменная, принимающая в качестве значения адрес. А имя массива является конкретным фиксированным адресом, который может пониматься как постоянный указатель. Когда объявляется массив, компилятор должен выделить базовый адрес и достаточный объем памяти для размещения всех элементов массива. Базовый адрес массива является начальным положением в памяти, где хранится массив; это адрес первого (с индексом 0) элемента массива. Допустим, что мы написали следующее объявление:

```
const int N = 100;
int a[N], *p;
```

и что система назначила байты памяти 300, 304, 308, ... , 696 в качестве адресов для хранения `a[0]`, `a[1]`, `a[2]`, ... , `a[99]` соответственно; при этом адрес 300 стал базовым адресом массива `a`. Мы полагаем, что каждый байт памяти адресуем, а для хранения целого используется четыре байта. Две инструкции

```
p = a;        и        p = &a[0];
```

являются равнозначными и присваивают p значение 300. Арифметика указателей предлагает альтернативу индексированию массивов. Две инструкции

```
p = a + 1;        и        p = &a[1];
```

равнозначны и присваивают p значение 304. Предположив, что элементам a были присвоены значения, мы можем использовать следующий код для суммирования массива:

В файле sum_arr2.cpp

```
sum = 0;
for (p = a; p < &a[N]; ++p)
   sum += *p;
```

это равносильно

```
sum = 0;
for (i = 0; i < N; ++i)
   sum += a[i];
```

В этом цикле переменная указателя p инициализуется базовым адресом массива a. Тогда последовательные значения p эквивалентны `&a[0]`, `&a[1]`, ... , `&a[N-1]`. Вообще, если `i` — целая переменная, то `p + i` — это *i*-ое смещение от адреса p. Аналогично, `a + i` — это *i*-ое смещение от базового адреса массива a. Вот еще один способ суммирования массива:

```
sum = 0;
for (i = 0; i < N; ++i)
   sum += *(a + i);
```

Также как выражение `*(a + i)` равносильно `a[i]`, выражение `*(p + i)` равносильно `p[i]`.

Во многих случаях массивы и указатели могут рассматриваться как одно и то же, но имеется существенная разница. Поскольку массив a является постоянным указателем, а не переменной, такие выражения как:

```
a = p        ++a        a += 2
```

недопустимы. Мы не можем изменить адрес a.

Отметьте, что `sizeof(a)` и `sizeof(p)` различны. Первое выражение дает размер всего массива, тогда как второе — размер выражения-указателя.

3.14. Передача массивов функциям

В определении функции формальный параметр, который объявлен как массив, по сути является указателем. Когда функции передается массив, на самом деле передается по значению его базовый адрес. Сами элементы массива не копируются. Для удобства записи компилятор допускает *запись с квадратными скобками* `[]` (как у массивов) при объявлении указателей в качестве параметров. Такая запись напоминает программисту и тем, кто будет читать код, что функция должна вызываться с массивом. Чтобы проиллюстрировать это, мы напишем функцию, которая суммирует элементы массива типа `int`.

В файле sum_arr3.cpp

```
int sum(const int a[], int n)  //n — это размер массива a[]
{
  int  i, s = 0;

  for (i = 0; i < n; ++i)
    s += a[i];
  return s;
}
```

Как часть заголовка определения функции объявление

```
int a[];        равнозначно        int *a;
```

В других контекстах они не равнозначны.

Предположим, что v была объявлена как массив со 100 элементами типа int. После того, как элементам присвоены значения, мы можем использовать приведенную выше функцию sum() для сложения различных элементов v. Следующая таблица содержит некоторые примеры.

Суммирование элементов массива

Вызов	Что будет вычислено и возвращено
sum(v, 100)	v[0] + v[1] + ... + v[99]
sum(v, 88)	v[0] + v[1] + ... + v[87]
sum(v + 7, k)	v[7] + v[8] + ... + v[k+6]

Последний вызов функции еще раз иллюстрирует использование арифметики указателей. Базовый адрес v смещен на 7, и функция sum() инициализует локальную переменную-указатель a этим адресом. В результате вызванная функция будет производить все вычисления адресов с учетом заданного смещения.

В C++ функция с формальным параметром-массивом может вызываться с фактическим аргументом-массивом любого размера при условии, что массив имеет правильный базовый тип.

3.15. Объявления ссылок и вызов по ссылке

Объявление ссылки вводит идентификатор, который будет альтернативным именем, или *псевдонимом* (alias), для объекта, указанного при инициализации ссылки, и допускает более простую форму параметров при вызове по ссылке. Например:

```
int     n;
int&    nn = n;       //nn — альтернативное имя для n
double  a[10];
double& last = a[9]; //last — псевдоним для a[9]
```

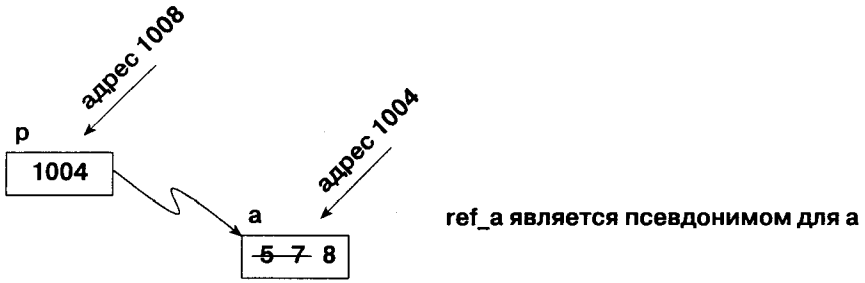
Объявление ссылки, которое является ее определением, должно содержать инициализатор. Обычно ссылки инициализуются простыми переменными. Инициализатор является именующим выражением (lvalue), которое дает адрес переменной. В примерах выше имена n и nn — псевдонимы друг для друга, то есть они ссылаются на один

и тот же объект. Изменение nn равнозначно изменению n и наоборот. Имя last является альтернативным именем для конкретного элемента массива a[9]. Будучи однажды инициализованы, эти имена не могут быть изменены.

Когда объявляется переменная i, она получает связанный с ней адрес памяти. Когда объявляется переменная-ссылка r и инициализуется значением i, она идентична i. Она ничем не отличается от остальных имен этого же объекта.

Следующие определения используются для демонстрации взаимосвязей между указателями, разыменованием и псевдонимами. Предположим, что память по адресу 1004 используется для хранения целой переменной a, а память по адресу 1008 — для переменной-указателя p.

```
int   a = 5;        //объявление a
int*  p = &a;       //p указывает на a
int&  ref_a = a;    //псевдоним для a
*p = 7;             //выражению lvalue присваивается 7
a = *p + 1;         //к rvalue 7 добавляется 1 и a
                    //присваивается 8
```



Обратите внимание, что любое изменение значения a равносильно изменению ref_a. Такое изменение влияет на разыменованное значение p. Указателю p может быть присвоен другой адрес, и он потеряет свою связь с a. С другой стороны, a и ref_a являются псевдонимами, и в пределах области видимости они должны ссылаться на один и тот же объект.

Обсуждаемые объявления могут использоваться с вызываемыми по ссылке аргументами, что в C++ позволяет применять такие аргументы напрямую (без разыменования). Такая возможность отсутствовала в C.

Код функции order(), использующий этот механизм, выглядит так:

В файле order.cpp

```
void order(int& p, int& q)
{
   int temp;

   if (p > q) {
      temp = p;
      p = q;
      q = temp;
   }
}
```

Прототип и вызов из `main()` функции `order()` может выглядеть так:

```
void order (int&, int&);

int main()
{
   int i, j;
   .....
   order(i, j);
   .....
}
```

Давайте используем этот механизм для создания функции `greater()`, которая меняет местами два значения, если первое больше второго.

В файле greater.cpp

```
int greater(int& a, int& b)
{
   if (a > b) {      //перестановка
      int  temp = a;

      a = b;
      b = temp;
      return 1;
   }
   else
      return 0;
}
```

Если `i` и `j` являются целыми переменными, то

```
greater(i, j)
```

будет использовать ссылку на `i` и ссылку на `j` для перестановки (если это необходимо) их значений. В традиционном C эта операция должна выполняться с помощью указателей и разыменования.

3.16. Проверка утверждений и корректность программы

Проверка утверждения (assertion) — это контрольный элемент программы, который (в случае нарушения проверяемого утверждения) осуществляет аварийный выход. На наш взгляд, утверждение является гарантийным договором между поставщиком кода (производителем) и клиентом кода (пользователем). В этой модели клиент гарантирует, что условия для применения кода существуют, а производитель гарантирует, что код будет работать правильно при этих условиях.

Корректность программы частично может рассматриваться как гарантия того, что вычисление завершится корректным выводом, зависящим от корректного ввода. Пользователь вычислений несет ответственность за корректность ввода. Это *предусловие*. Вычисление, если оно удачно, удовлетворяет *постусловию*. Такие проверки утверждений могут отслеживаться в процессе выполнения, что обеспечивает весьма

полезную диагностику. В самом деле, дисциплина, заставляющая продумывать надлежащие утверждения, часто помогает программисту избежать ошибок и ловушек.

Сообщество программистов на C++ придает все большее значение использованию утверждений. Стандартная библиотека *assert.h* предоставляет макро `assert`, оно вызывается так, как если бы существовала функция с сигнатурой:

```
void assert(выражение);
```

Если результат *выражения* — ложь, то выполнение прерывается с выводом диагностического сообщения. Проверка утверждений не выполняется, если определено макро NDEBUG.

Следующую программу мы снабдили проверками утверждений, чтобы продемонстрировать обсуждаемую технику. Программа просматривает отрезок массива в поисках наименьшего элемента и помещает этот элемент в первую просмотренную позицию массива.

В файле order.cpp

```
//Нахождение наименьшего элемента в отрезке массива

#include <iostream.h>
#include <assert.h>

void order(int& p, int& q)
{
   int temp = p;

   if (p > q) {
      p = q;
      q = temp;
   }
}

int place_min(int a[], int size, int lb = 0)
{
   int i, min;
   assert(size >= 0);    //предусловие

   for(i = lb; i < lb + size; ++i)
      order(a[lb], a[i + 1]);
   return a[lb];
}

int main()
{
   int a[9] = { 6, -9, 99, 3, -14, 9, -33, 8, 11};

   cout << "Minimum = " << place_min(a, 3, 2) << endl;
   assert(a[2]<=a[3] && a[2]<=a[4]);   //постусловие
}
```

Проверка утверждения предусловия в функции `place_min()` гарантирует, что будет просмотрено неотрицательное число элементов. Постусловие в `main()` удостоверяет, что наименьший элемент был найден и помещен в нужное место.

3.17. Строки: соглашение о char*

Сообщества C и C++ «договорились» считать тип `char*` формой строкового типа. Соглашение заключается в том, что строки заканчиваются символьным значением ноль, и что с этими абстракциями вызываются функции из *cstring* (*string.h* в более ранних системах). В ANSI C++ библиотека *string* в качестве шаблонного класса предоставляет стандартизованный строковый тип, использовать который предпочтительнее, чем `char*`.

Язык частично поддерживает абстракцию `char*`, устанавливая, что строковые литералы заканчиваются нулевым символом. Строка `char*` или `char[]` может быть инициализована строкой символов. Заметьте, что завершающий ноль является частью списка инициализаторов:

```
char* s = "c++"; //s[0]='c', s[1]='+', s[2]='+', s[3]=0
```

Пакет *cstring* содержит более 20 функций, включая следующие:

Некоторые функции из библиотеки cstring

- `size_t strlen(const char* s);`
  Вычисляет длину строки
- `char* strcpy(char* s1, const char* s2);`
  Копирует строку `s2` в строку `s1`. Возвращается значение `s1`.
- `int strcmp(const char* s1, const char* s2);`
  Возвращает целое значение, отражающее лексикографическое сравнение `s1` и `s2`. Если строки одинаковы — возвращается ноль. Если `s1` меньше, чем `s2` — возвращается отрицательное целое. Если `s2` меньше, чем `s1` — возвращается положительное целое.

Придерживаясь вышеизложенных соглашений, программист получает возможность многократно использовать код со строками. Библиотечные процедуры позволяют использовать переносимый, легко понимаемый код.

В файле str_func.cpp

```
//Реализация строковых функций

size_t strlen(const char* s)
{
   int i;
   for (i = 0; s[i]; ++i)
      ;
   return i;
}

int strcmp(const char* s1, const char* s2)
{
   int i;
   for(i=0; s1[i] && s2[i] && (s1[i]==s2[i]); ++i)
      ;
  return (s1[i] - s2[i]);
}
```

```
char* strcpy(char* s1, const char* s2)
{
   int i;
   for (i = 0; s1[i] = s2[i]; ++i)
      ;
   return s1;
}
```

Заметьте, как эти функции используют для выхода из своих основных циклов соглашение о завершении строки нулем. Функция `strcpy()` прекращает цикл `for`, когда `s2[i] == 0`.

Кроме того, обратите внимание на хорошую привычку вставлять ключевое слово `const` перед теми строковыми переменными, содержимое которых не будет изменяться.

3.18. Многомерные массивы

Язык C++ допускает массивы любого типа, включая массивы массивов. С двумя парами квадратных скобок мы имеем двухмерный массив. Продолжая в том же духе, *можно получить массивы более высокой размерности. С каждой новой парой* квадратных скобок мы наращиваем размерность массива.

Объявления массивов

`int a[100];`	одномерный массив
`int b[3][5];`	двухмерный массив
`int c[7][9][2];`	трехмерный массив

Любой k-мерный массив имеет размер по каждому из своих k измерений. Пусть s_i представляет величину i-того измерения, тогда объявление массива выделит память для $s_1 \times s_2 \times ... \times s_k$ элементов. В приведенной таблице `b` содержит 3×5 элементов, а `c` — 7 x 9 x 2 элементов. Начиная с базового адреса массива все элементы хранятся в памяти последовательно.

Многомерный массив может быть инициализован заключенным в фигурные скобки списком инициализаторов, причем каждый ряд инициализуется своим списком в фигурных скобках:

```
int a[2][3] = { {1, 2, 3}, {4, 5, 6} };
                //то же, что {1, 2, 3, 4, 5, 6}
char name[3][9] = { "laura", "michele", "pohl" };
                  //завершенные '\0'
```

3.19. Операторы свободной памяти new и delete

Унарные операторы `new` и `delete` служат для управления *свободной памятью* (free store). Свободная память — это предоставляемая системой область памяти для объектов, время жизни которых напрямую управляется программистом. Программист создает объект с помощью ключевого слова `new`, а уничтожает его, используя `delete`. Это важно при работе с динамическими структурами данных, такими как списки и деревья.

В C++ оператор new принимает следующие формы:

```
new имя_типа
new имя_типа инициализатор
new имя_типа[выражение]
```

В каждом случае имеется по крайней мере два эффекта. Во-первых, выделяется надлежащий объем свободной памяти для хранения указанного типа. Во-вторых, возвращается базовый адрес объекта (в качестве значения оператора new). Когда память недоступна, оператор new либо возвращает значение 0 либо возбуждает соответствующее исключение (см. раздел 11.10, «Стандартные исключения и их использование», на стр. 324).

В следующем примере используется оператор new:

```
int *p, *q;
p = new int(5);     //выделение памяти
                    //и инициализация
q = new int[10];    //получаем массив от q[0] до q[9]
                    //причем q = &q[0]
```

В этом фрагменте указателю на целое p присваивается адрес ячейки памяти, полученный при размещении целого объекта. Место в памяти, на которое указывает p, инициализуется значением 5. Такой способ обычно не используется для простых типов вроде int, так как гораздо удобнее и естественнее разместить целую переменную в стеке или глобально. Пример с указателем q на массив встречается чаще.

Оператор delete уничтожает объект, созданный с помощью new, отдавая тем самым распределенную память для повторного использования. Оператор delete может принимать следующие формы:

```
delete выражение
delete[] выражение
```

Первая форма используется, если соответствующее выражение new размещало не массив. Во второй форме присутствуют пустые квадратные скобки, показывающие, что изначально размещался массив объектов. Оператор delete не возвращает значения, поэтому можно сказать, что возвращаемый им тип — void.

В следующем примере эти конструкции используются для динамического распределения памяти.

В файле dynarray.cpp

```
double avg_arr(const int a[], int size)
{
   int  sum = 0;
   for (int i = 0; i < size; ++i)
      sum += a[i];
   return static_cast<double>(sum)/size;
}

int main()
```

```
{
   int* data;
   int size;

   for (int n_loop = 0; n_loop < 5; ++n_loop) {
      cout << "\nВведите размер массива: ";
      cin >> size;
      assert(size > 0);
      data = new int[size];//размещение массива
      assert(data != 0);
      for (int j = 0; j < size; ++j)
         cin >> data[j];
      cout << " среднее равно "
           << avg_arr(data, size) << endl;
      delete[] data;//освобождение памяти,
                    //занимаемой массивом
   }
}
```

Разбор программы dynarray

- ```
  cout << "\nВведите размер массива: ";
  cin >> size;
  assert(size > 0);
  data = new int[size];
  assert(data != 0);
  ```

Переменная-указатель `data` используется в качестве базового адреса динамически размещаемого массива с количеством элементов, задаваемым значением `size`. Оператор `new` выделяет из свободной памяти место, достаточное для хранения объекта типа `int[size]`. В системах, где целое занимает два байта, данная операция выделит 2 × `size` байт. С этого момента переменной `data` присвоен базовый адрес созданного объекта. Вторая проверка утверждения гарантирует, что распределение памяти выполнено. В новых C++ системах оператор `new` в случае неудачи приводит к исключению, которое автоматически прерывает программу.

- ```
  delete[] data;   //освобождение памяти
  ```

Оператор `delete` отдает свободной памяти область, связанную с переменной-указателем `data`. Такое можно проделывать только с объектами, размещенными `new`. Используется форма с квадратными скобками, так как соответствующее распределение было выполнено для массива.

Наше вводное обсуждение операторов свободной памяти относится к простым случаям. Более подробно операторы свободной памяти изучаются в разделе 7.12, «Перегрузка операторов new и delete», на стр. 214.

3.20. Практические замечания

Когда аргументы функции предполагаются неизменными, их можно эффективно и корректно вызывать по ссылке как const. Именно так поступают, например, со структурами:

```
struct large_size {
   int mem[N];
   .....     //остальное
};

void print(const large_size& s)
{
     //поскольку s не будет изменяться,
     //избегайте копирования с помощью вызова по значению
     .....
}
```

Функция print() не должна создавать локальную копию переменной s. Если бы s передавалась по значению, при каждом вызове print() копировались бы N целых. При использовании вызова по ссылке передается адрес фактического объекта и локального копирования не происходит. При этом экономятся как память, так и время (и то, и другое расточительно расходуется при копировании). Для переменных, которые не будут изменяться, но при этом не требуют много памяти, обычно применяется вызов по значению.

3.20.1. void* и reinterpret_cast

ANSI C допускает сужающие преобразования от void* к произвольному типу указателя, в то время как C++ — нет. Глядя на присвоения указателей в следующем примере,

```
int* ip;
void* gp;

.....
ip = gp;                    //недопустимо в C++, но допустимо в C
ip = reinterpret_cast<int*>(gp);    //допустимо в C++
```

можно заметить, что в C++ применяется более строгая и безопасная с точки зрения типов практика преобразований. Вот один пример. C++ рассматривает обобщенный указатель как более широкий тип, чем указатель на любой конкретный тип. Расширяющее преобразование указателя любого конкретного типа к обобщенному указателю безопасно и допустимо как неявное. Сужающее преобразование обобщенного указателя к указателю на конкретный тип потенциально небезопасно и зависит от системы. Поэтому следует использовать reinterpret_cast<> — оператор приведения, который требуется для (возможно) непереносимого преобразования значения.

3.20.2. Замена объявлений static extern

Для обеспечения уникальной области видимости вместо глобальных статических объявлений следует использовать безымянные пространства имен (см. раздел 3.9,

«Пространства имен», на стр. 82). Это делается объявлением пространства имен без указания имени, например:

```
namespace { int seed = 0; }       //создает уникальное имя
void srand(int i) { seed = i; }   //уникальное::seed
```

Объявления, заданные внутри фигурных скобок, могут использоваться в оставшейся части файла, они скрыты внутри этого файла и неэкспортируемы. Сокрытие данных — это принцип, важный и для объектно-ориентированного, и для модульного программирования.

Резюме

1. Код, описывающий, что делает функция, называется определением функции. Формально это выглядит так:

   ```
   заголовок_функции
   {
     инструкции
   }
   ```

   Каждый раз, когда переменные передаются функции как аргументы, их значения копируются в соответствующие параметры функции, а сами переменные в вызывающем окружении не изменяются. Это называется вызовом по значению. В C++ для осуществления вызова по ссылке применяются объявления ссылок.

2. В C++ использование функции недопустимо до ее объявления. Она может быть определена в том же файле позднее, а может браться из библиотеки или из указанного пользователем файла. Прототип функции явно задает тип и количество параметров. Он имеет следующую обобщенную форму:

   ```
   тип имя(список_объявлений_аргументов);
   ```

   Обычно *список_объявлений_аргументов* — это разделенные запятыми имена типов; за каждым из имен типов может следовать идентификатор соответствующего аргумента. Эта информация позволяет компилятору обеспечить совместимость типов.

3. Для выхода из функции применяется инструкция `return`. Если используется форма `return` *выражение*, то *выражение* должно допускать неявное преобразование к возвращаемому функцией типу.

4. Массив — это тип данных, который используется для представления последовательности однородных значений. Массив размещается с нулевого элемента. Доступ к элементам массива производится с помощью индексов. Таким образом, массив из количества *размер* элементов индексируется от 0 до *размер*-1. Имя массива, в сущности, является постоянным указателем, инициализованным базовым адресом массива.

5. Каждая переменная и функция в базовом языке C++ имеет класс памяти. Классов памяти четыре: автоматический, внешний, регистровый и статический; им соответствуют ключевые слова:

```
auto    extern    register    static
```

Переменные, объявленные внутри тела функции, по умолчанию являются автоматическими.

6. Указатели C++ используется для установления соответствия между переменными и машинными адресами. Массивы в C++ могут рассматриваться как специальная форма указателя, связанного с непрерывной областью памяти для хранения последовательности значений, которую можно проиндексировать. Указатели используются в программах для доступа к памяти и манипуляции с адресами. Например, если `v` — переменная, то `&v` — это адрес (место в памяти), где хранится ее значение. Объявление

```
int* p;
```

говорит, что переменная `p` имеет тип «указатель на целое». Допустимая область значений для любого указателя всегда включает специальный адрес 0.

7. Объявления ссылок позволяют присвоить объекту псевдоним (альтернативное имя). Эти объявления можно использовать для аргументов, вызываемых по ссылке. Например, функция `order()`, использующая этот механизм, объявлена как:

```
void order(int &p, int &q);
```

8. Объявление `void*` является типом обобщенного указателя. Указателю, объявленному как указатель на `void`, например, `void* gp`, можно присвоить значение указателя любого основного базового типа, но он не может быть разыменован.

9. В определении функции формальный параметр, объявленный как массив, на самом деле является указателем. Когда функции передается массив, она, на самом деле, получает (по значению) его базовый адрес. Сами элементы массива не копируются. Для удобства компилятор позволяет запись с квадратными скобками (как у массивов) при объявлении указателей в качестве параметров. Такая запись напоминает программисту и тем, кто будет читать код, что функция должна вызываться с массивом.

10. Унарные операторы `new` и `delete` служат для управления свободной памятью. Свободная память — это предоставляемая системой область памяти для объектов, время жизни которых напрямую управляется программистом. Программист создает объект с помощью ключевого слова `new`, а уничтожает его, используя `delete`. Это важно при работе с динамическими структурами данных, такими как списки и деревья.

Упражнения

1. Указатель на символьную строку по соглашению завершается значением 0. Следующая функция реализует проверку равенства строк. Заметьте, что в ней используется арифметика указателей. Конструкция `*s1++` означает: разыменовать указатель `s1`, использовать полученное значение в выражении и, наконец, увеличить `s1` на единицу.

```
bool streq(const char* s1, const char* s2)
```

```
{
   while ( *s1 != 0 && *s2 != 0)
      if ( *s1++ != *s2++)
         return false;
   return (*s1 == *s2);
}
```

Напишите и проверьте функцию

```
bool streq(const char* s1, const char* s2, int n);
```

которая возвращает значение `true`, если первые `n` символов в двух строках совпадают, и `false` — если они отличаются.

2. Переработайте приведенную выше функцию с использованием нотации массива.

```
bool streq(char s1[], char s2[])
{
   int  i = 0;
   while ( s1[i] != 0 && s2[i] != 0)
      if ( s1[i] != s2[i])
         return false;
      ++i;
         .....
}
```

3. Стандартный заголовочный файл *cstring* содержит прототипы для некоторых полезных строковых функций, входящих в стандартную библиотеку. Среди них:

```
size_t strlen(const char* s);
```

Эта функция возвращает длину строки. В тексте главы было дано ее краткое описание; вот другой способ кодирования `strlen()`:

```
//итерационное вычисление длины строки
size_t strlen(const char *s)
{
   size_t  len = 0;
      while (*s != '\0') {  //символ конца строки
         ++len;             //приращение длины
         ++s;               //продвижение указателя
      }
      return len;
}
```

Этот алгоритм последовательно перемещает по строке указатель `s` в поисках символа конца строки. Будучи внешним по отношению к функции, значение указателя не изменяется, так как передается по значению. Напишите рекурсивный вариант функции.

4. Нахождение наибольшего общего делителя, НОД (Greatest Common Divisor, GCD), для двух целых рекурсивно определено в следующем псевдокоде:

```
НОД (m,n) есть:
   если m делить по модулю на n равно 0, то n;
   иначе НОД(n, m делить по модулю на n);
```

Напомним, что в C++ оператор деления по модулю выглядит как `%`. Закодируйте эту процедуру на C++.

5. Мы хотим посчитать количество рекурсивных вызовов функции `gcd()`. Обычно использование глобальных переменных внутри функции — плохая практика. В C++ можно использовать локальную статическую переменную вместо глобальной.

```
int gcd(int m, int n)
{
   static int fcn_calls = 0;
   int r;       //остаток

   ++fcn_calls;
    .....
}
```

Завершите и проверьте эту версию.

6. Что напечатает следующая программа?

```
#include  <iostream.h>

int foo(int n)
{
   static int  count = 0;

   ++count;
   if ( n <= 1) {
      cout << " счетчик: " << count << endl;
      return n;
   }
   else
      foo(n / 3);
}

int main()
{
   foo(21);
   foo(27);
   foo(243);
}
```

7. Класс памяти `static` полезен при многофайловой компиляции. Предскажите, что напечатает следующая программа:

```
//Файл A.cpp

static int  foo(int i)
{
   return (i * 3);
}
```

```
int  goo(int i)
{
  return (i * foo(i));
}

//Файл B.cpp

#include  <iostream.h>

int foo(int i)
{
  return (i * 5);
}

int  goo(int i);  //импортируется из файла A.cpp

int main()
{
  cout << "foo(5) = " << foo(5) << endl;
  cout << "goo(5) = " << goo(5) << endl;
}
```

Программа компилируется как *CC A.cpp B.cpp*. Функции с областью видимости файла по умолчанию внешние. Функция `foo()` из файла *A.cpp* видна только в этом файле, она закрыта для остальных файлов, в отличие от `goo()`. Поэтому переопределение `foo()` в файле *B.cpp* не вызовет ошибки. Еще раз попробуйте откомпилировать программу, убрав ключевое слово `static`, чтобы увидеть сообщение об ошибке. Затем проведите третью проверку, задав функцию `goo()` как `inline` в *A.cpp*, и вы убедитесь, что компилятор снова выдаст сообщение об ошибке.

8. Переделайте предыдущую программу, заменив внешние объявления `static` на безымянное пространство имен.

9. C++ позволяет передавать функции `main()` аргументы командной строки. Следующий код выводит свои аргументы командной строки:

```
//Вывод аргументов командной строки начиная с самого правого
#include  <iostream.h>

int main(int argc, char **argv)
{
  for (--argc; argc >= 0; --argc)
    cout << argv[argc] << endl;
}
```

Аргументу `argc` передается количество аргументов командной строки. Каждый аргумент является строкой, помещенной в двухмерный массив `argv`. Откомпилируйте программу в исполняемый файл с именем *echo_arg*. Запустите его со следующими аргументами:

```
echo_arg a man a plan a canal panama
```

[1] Можно и с такими аргументами: *а роза упала на лапу азора* или *аргентина манит негра*. —*Примеч. перев.*

10. Модифицируйте предыдущую программу так, чтобы она выводила аргументы командной строки слева направо и нумеровала их.

11. В ANSI C выражение `i[a]` допустимо и эквивалентно `a[i]`, хотя это невразумительная и плохая практика. Проверьте, так ли это в C++.

```
/* Работает в ANSI C */
  int  a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int i;

  for (i = 0; i < 10; ++i)
    if (a[i] != i[a])
      printf("Не одно и то же при индексе %d : %d != %d\n",
         i, a[i], i[a]);
```

12. Одно из преимуществ C++ по сравнению с традиционными языками заключается в расширяемости типов. Вы можете импортировать тип комплексных чисел с помощью `#include <complex.h>`, который можно использовать совместно с собственными арифметическими типами. Перегрузите и проверьте

```
complex avg_arr(const complex a[], int size)
```

13. Используйте проверку утверждений в программе *add3* из раздела 3.4, «Прототипы функций», на стр. 72, для гарантии того, что введенные значения лежат между 0 и 100. Используйте постусловие для дополнительной гарантии корректности вывода.

14. Проблема применения обобщенного указателя `void*` заключается в том, что его нельзя разыменовать. Поэтому для выполнения полезных действий над обобщенным указателем его необходимо привести к стандартному работающему типу, такому как `char*`, например. Запишите и проверьте:

```
void* memcpy(void* s1, const void* s2, unsigned n)
{
  char *from = static_cast<char*>(s2),
       *to = static_cast<char*>(s1);
  .....
}
```

15. Напишите программу, обращающую строку. Результат `s1` должен содержать обращение `s2` (предполагается, что длина `s1` достаточна для этого).

```
char* strrev(char* s1, const char* s2);
```

16. Напишите обращающую строку программу, используя память, распределенную с помощью `new`. В результате строка `s1` должна содержать обращение строки `s2`. Используйте `new` для размещения строки `s1` длиной в `strlen(s2) + 1` (что достаточно).

```
char* strrev(char*& s1, const char* s2);
```

17. Напишите программу для обращения строк, используя тип `string` ANSI C++ из стандартной библиотеки. В строку `s1` следует поместить обращение строки `s2`.

В нашей системе тип `string` находится в *bstring.h*. В системе, поддерживающей полный стандарт, этот тип будет находиться в файле *string*, а строковые функции в стиле `char*` — в *cstring*.

18. Напишите программу, которая размещает в свободной памяти одномерный массив. Его нижняя и верхняя границы задаются пользователем. Программа должна убедиться, что верхняя граница больше нижней. Если это не так, осуществляется аварийный выход. Используйте пакет *assert.h*:

```
#include <assert.h>
  .....     //ввод нижней и верхней границ
assert(ub - lb > 0);
  .....
```

Размер массива будет (*верхняя граница – нижняя граница + 1*) элементов. Дан стандартный массив C++ именно такой длины. Напишите функцию, которая использует данный стандартный массив для инициализации динамического массива. Проверьте ее, выведя в красиво отформатированном виде оба массива до и после инициализации.

19. (*Проект*) Используйте *cstirng* или *string.h* для построения лексикографически упорядоченного списка из всех идентификаторов, содержащихся во входном файле — программе на C++. Каждый идентификатор будет считываться в `char buffer[MAXLEN]`. После того как длина идентификатора выяснена, поместите его в массив, размещенный в свободной памяти, на который указывает `char* id_list[i]`. Добивайтесь, чтобы каждая функция делала одно дело. Кодирование больших кусков программы путем сведения ее к набору относительно легко кодируемых небольших функций называется пошаговым уточнением. Напишите функцию `get_id()`, которая получает из файла очередной идентификатор. Можете использовать перенаправление, так что функция `get_id()` окажется способной читать из `cin`. Используйте стандартные библиотечные функции где только возможно. Применяйте *ctype.h* для того, чтобы избежать зависимости от различных кодировок символов. Лексемы, не являющиеся идентификаторами, в том числе ключевые слова, должны игнорироваться. Рассмотрите возможность исключения повторов. В этой связи вы должны добавить счетчик встречаемости для каждого идентификатора. Постарайтесь применять быструю процедуру сортировки. Наконец, вы, возможно, захотите написать пакет для использования с аргументами командной строки. В таком случае вам понадобится *fstream.h* для обработки файлов (см. раздел D.5, «Файлы», на стр. 413). Изощренный пакет может принимать набор входных файлов и распечатывать список с указанием имени файла и номера строки, в которой встречается каждый из идентификаторов.

Глава 4

Реализация АТД в базовом языке

Эта глава знакомит читателя с понятием структуры и применением структур в базовом языке. Такие определяемые пользователем типы данных, как стек, комплексные числа и колоды карт являются примерами реализации АТД. В данной главе каждый из этих типов запрограммирован на C++ и представлен в примерах.

Значительная часть процесса объектно-ориентированного проектирования заключается в придумывании АТД, соответствующих задаче. Хороший АТД не только моделирует основные черты данной задачи, но часто также может использоваться в других программах.

4.1. Агрегатный тип struct

Тип структуры позволяет программисту объединить компоненты в переменную с одним именем. Структура содержит индивидуально именованные компоненты, называемые *членами*[1] (structure members). Поскольку члены структуры бывают различных типов, программист может создавать агрегаты (данных), позволяющие описывать сложные данные.

Давайте в качестве простого примера определим структуру, описывающую игральную карту. Метки на карте, представляющие ее достоинство, называются *очками* (pips). Например, тройка пик имеет достоинство (значение очков) — 3, и значение масти — пики. Можно объявить тип структуры, как показано в следующем фрагменте.

В файле poker1.cpp

```
enum suit {clubs, diamonds, hearts, spades}; //масти
typedef int pips;     //pips является синонимом int

struct card {         //карта
  suit s;
  pips p;   //от 1 до 13 соответствует Т, 2, ·····, В, Д, К
};
```

[1] В некоторых изданиях термин *member* применительно к данным переводится как *элемент* (элемент структуры, элемент данных). — *Примеч. перев.*

В C++ имя структуры или *теговое имя* (tag name) является типом. В данном объявлении `struct` — это ключевое слово, `card` — теговое имя структуры, а переменные `p` и `s` — это члены структуры. Переменная `p` будет принимать значения от 1 до 13, что соответствует картам от туза до короля. Она является переменной типа `pips`, объявленного с помощью ключевого слова `typedef` и созданного как синоним `int`. Объявление с таким именем придает программе документированность и сводит сложные объявления к ясным идентификаторам. Вот еще несколько примеров:

```
typedef int Miles;              //Miles – это int
typedef char* Cstring;          //указатель на char (C-строка)
typedef void* Gen_ptr;          //обобщенный указатель
typedef double (*F)(double);    //указатель на функцию
```

Объявление `struct card` можно рассматривать как «чертеж»: оно создает тип `card`, но не фактические экземпляры. Объявление

```
card c1, c2;
```

выделяет память для идентификаторов `c1` и `c2` типа `card`. Для доступа к членам структур `c1` и `c2` используется оператор выбора члена структуры, представляющий собой точку «`.`». Допустим, мы хотим присвоить `c1` значение, соответствующее пятерке бубен, а `c2` — дамы пик. Можно написать:

```
c1.p = 5;
c1.s = diamonds;
c2.p = 12;
c2.s = spades;
```

Конструкция вида

переменная_структуры.имя_члена

используется как переменная — точно таким же образом, как простая переменная или элемент массива. Имя члена должно быть уникальным внутри данной структуры. Поскольку член структуры всегда должен предваряться идентификатором переменной (теговым именем) структуры, никакой путаницы между членами, имеющими одинаковые имена в разных структурах, не возникает. Вот пример:

```
struct fruit {
   char name[15];
   int  calories;
};

struct vegetable {
   char name[15];
   int  calories;
};

fruit     a;
vegetable b;
```

Сделав такие объявления, можно обращаться к `a.calories` и `b.calories` безо всякой двусмысленности.

Вообще, структура объявляется ключевым словом `struct`, за ним следует идентификатор (теговое имя), затем в фигурных скобках идет список объявлений членов. Теговое имя необязательно, но если оно есть, имя должно отражать суть моделируемого АТД. Когда теговое имя отсутствует, объявление структуры является безымянным и может использоваться только для немедленного (следующего сразу за объявлением безымянной структуры) объявления переменных этого типа, например:

```
struct {
   int a, b, c;
} triples [2] = { {3, 3, 6}, {4, 5, 5} };
```

4.2. Оператор указателя структуры

Мы уже видели применение оператора выбора члена структуры «.» для доступа к ее членам. Этот раздел знакомит с оператором указателя структуры ->.

C++ предоставляет оператор указателя структуры -> для доступа к членам структуры посредством указателя. С клавиатуры этот оператор вводится как знак «минус», за которым следует знак «больше». Если переменной-указателю присвоен адрес структуры, то доступ к члену структуры может быть осуществлен с помощью конструкции вида:

```
указатель_на_структуру -> имя_члена
```

То же самое можно задать и так:

```
(*указатель_на_структуру).имя_члена
```

Операторы «->» и «.», также как () и [], имеют самый высокий приоритет и выполняются слева направо. В сложных ситуациях оба способа доступа могут комбинироваться. Следующая таблица иллюстрирует их использование на простых примерах.

Объявления и присваивания

```
card cd, *pc = &cd;
card deck[52];
cd.p = 5;
cd.s = spades;
deck[0] = cd;
```

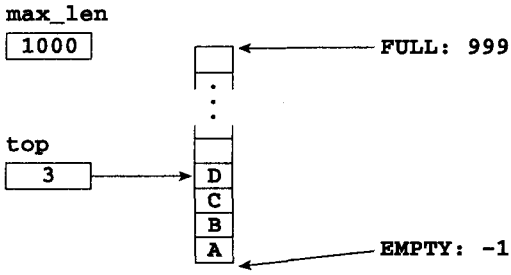
Выражение	Эквивалентное выражение	Значение
`cd.p`	`pc -> p`	5
`cd.suit`	`pc -> suit`	spades
`deck[0].p`	`deck -> p`	5
`(*pc).suit`	`pc -> suit`	spades

4.3. Пример: стек

Стек — одна из наиболее полезных структур данных. Стек позволяет вставлять и удалять данные, причем выполнение этих операций осуществляется исключительно над вершиной стека (то есть над самым верхним его элементом). Такой порядок называется «первым вошел, последним вышел» (last-in-first-out, LIFO). По идее, стек

ведет себя как стопка подносов: поднос оказывается наверху, если предыдущий забрали, или «опускается» в стопке вниз, если сверху положили еще один. Структура данных, основное назначение которой — хранение информации, называется *контейнером* (container).

Стек допускает следующие операции: *push*, *pop*, *top*, *empty* и *full* (поместить, извлечь, вершина, пусто, заполнен). Операция *push* помещает значение в стек, *pop* извлекает значение из стека и удаляет его, *top* возвращает верхнее значение в стеке, *empty* проверяет, не пуст ли стек, а *full* проверяет, не полон ли он. Стек является типичным АТД.



Реализуем стек как тип данных C++ для хранения символьных значений, используя структуру так, как она понимается в базовом языке. Реализация будет использовать символьный массив фиксированной длины для хранения содержимого стека. Вершиной стека будет член с именем `top`, принимающий целые значения. Различные операции со стеком будут реализованы в виде функций, список аргументов каждой из которых включает в качестве параметра указатель на `ch_stack`. Это позволит изменять стек и избежать его копирования при вызове по значению.

В файле ch_stac1.h

```
//Простая реализация типа ch_stack

const int  max_len = 1000;
enum { EMPTY = -1, FULL = max_len - 1 };

struct ch_stack {
   char  s[max_len];
   int  top;
};

//Стандартный набор операций с ch_stack

void reset(ch_stack* stk)        //обнуление стека
{
   stk -> top = EMPTY;
}

void push(ch_stack* stk, char c)
{
   stk -> s[++stk -> top] = c;    //приоритет: ++(stk -> top)
}
```

```
char pop(ch_stack* stk)
{
  return (stk -> s[stk -> top--]);
}

char top(ch_stack* stk)
{
  return (stk -> s[stk -> top]);
}

bool empty(const ch_stack* stk)
{
  return (stk -> top == EMPTY);
}

bool full(const ch_stack* stk)
{
  return (stk -> top == FULL);
}
```

Разбор функций ch_stack

- ```
  const int  max_len = 1000;
  enum { EMPTY = -1, FULL = max_len - 1 };

  struct ch_stack {
    char  s[max_len];
    int  top;
  };
  ```

Объявление `struct` создает новый тип — `ch_stack`. В нем два члена: член-массив `s` и целый член `top`.

- ```
  void reset(ch_stack* stk)
  {
    stk -> top = EMPTY;
  }
  ```

Эта функция используется для инициализации. Члену `top` присваивается значение `EMPTY`. Конкретный `ch_stack`, с которым работает эта функция, является аргументом, передаваемым как адрес.

- ```
  void push(ch_stack* stk, char c)
  {
    stk -> s[++stk -> top] = c;
  }

  char pop(ch_stack* stk)
  {
    return (stk -> s[stk -> top--]);
  }
  ```
  ```

Операция помещения в стек (push) реализована как функция двух аргументов. Член top увеличивается на единицу. Значение c помещается в вершину ch_stack. Эта функция полагает, что ch_stack не заполнен. Операция извлечения из стека (pop) реализована таким же образом; в ней предполагается, что ch_stack не пуст. Возвращается значение вершины стека, и член top уменьшается на единицу.

- ```
  bool empty(const ch_stack* stk)
  {
    return (stk -> top == EMPTY);
  }

  bool full(const ch_stack* stk)
  {
    return (stk -> top == FULL);
  }
  ```

Эти функции возвращают значение типа bool. Каждая из них проверяет член top на соответствующее условие. Например, перед вызовом push() программист может проверить, что ch_stack не полон, для того чтобы функция push() работала корректно. Эти функции не изменяют ch_stack, на который они направлены. Поэтому мы можем объявить аргументы-указатели как const. Во всех функциях аргумент ch_stack передается как адрес, а для доступа к членам используется оператор указателя структуры ->.

При условии, что описанные выше объявления находятся в файле *ch_stac1.h*, можно проверить операции со стеком с помощью следующей программы, которая помещает символы строки в стек и извлекает их, печатая символы в обратном порядке:

В файле ch_stac1.cpp

```
//Проверим ch_stack с помощью обращения строки

#include  "ch_stac1.h"
#include  <iostream.h>

int main()
{
  ch_stack  s;
  char str[40] = {"Книгу перевел Денис Ковальчук!"};
  int   i = 0;

  cout << str << endl;          //печать строки
  reset(&s);
  while (str[i] && !full(&s)) //помещение в стек
    push(&s, str[i++]);
  while (!empty(&s))            //печать обращенной строки
    cout << pop(&s);
  cout << endl;
}
```
```

Вывод этой тестовой программы выглядит так:

```
Книгу перевел Денис Ковальчук!
!кучьлавоК синеД леверeп угинК
```

Отметьте, что одним из фактических аргументов в каждой из функций является `&s`, то есть адрес переменной типа `ch_stack`, объявленной в `main()`. Мы задаем этот аргумент, поскольку функции ожидают адрес переменной типа `ch_stack`.

4.4. Объединения

Объединение (union) — это производный тип, синтаксис которого такой же, как и у структур, за исключением того, что ключевое слово `union` заменяет `struct`. Члены объединения совместно используют память, и их значения перекрываются. То есть объявление объединения позволяет интерпретировать его значение как набор типов входящих в него членов. Объединение инициализуется значением в фигурных скобках, причем оно присваивается первому члену объединения. Рассмотрим следующее объявление:

В файле union.cpp

```
union int_dbl {
   int i;
   double x;
} n = {0};          //член i инициализован нулем
```

Переменная `n` может использоваться или как целая, или как переменная с двойной точностью.

```
n.i  = 7;           //в n хранится целое значение 7
cout << n.i << " целое.";
cout << n.x << " двойной точности - зависит от машины.";
n.x  = 7.0;         //значение с двойной точностью
                    //в n хранится 7.0
```

Этот пример также иллюстрирует, почему объединения могут быть опасны и часто машинно-зависимы. На некоторых системах не все битовые комбинации являются допустимыми значениями перекрывающихся типов. В этом случае верное значение одного типа, при обращении к нему как к другому типу, может возбудить исключение.

Объединение может быть безымянным, как в следующем фрагменте:

В файле weekend.cpp

```
enum week { sun, mon, tues, weds, thurs, fri, sat };
            //дни недели
union {
   int i;
   week  w;
};
i = 5;
```

```
if (w == sat || w == sun)      //суббота или воскресенье
   cout << " Выходной! ";
```

Объявление безымянного объединения позволяет использовать идентификаторы отдельных членов как переменные. Имена членов должны быть уникальными в пределах области видимости; кроме того, никакие другие переменные безымянного типа объявлять нельзя. Отметьте, что безымянное объединение, объявленное в области видимости файла, должно быть статическим (`static`).

Усовершенствуем стек, переделав его в виде объединения различных типов. Суть в том, что стек является структурой данных, полезной для хранения значений любых типов. Такой стек можно использовать для обработки разнородных значений, применяя метод LIFO:

```
//Стек, усовершенствованный для хранения многих типов

enum type { int_type, dbl_type, chr_type, str_type };

union rainbow {
   int    i;
   double x;
   char   c;
   char*  p;
};

struct rbdata {
   type    t;
   rainbow d;
};

struct u_stack {
   rbdata s[max_len];
   int top;
};
```

Структура данных `u_stack` более гибкая, за что приходится расплачиваться дополнительной памятью, которая требуется для хранения переменной `t` типа `type` для каждого из элементов. Операции стека кодируются так же, как в `ch_stack`. Например, `push()`:

```
void push(u_stack* stk, str[i++])
{
   stk -> s[++stk -> top = c;
}
```

При использовании `u_stack` член `t` отслеживает, значение какого типа содержится в каждом элементе стека. В случае с разнородными данными, вероятно, подошли бы операторы `switch`, как в следующем примере:

```
rdata x;
u_stack * pa;
.....
x = pop(pa);
```

```
switch (x.t) {   //извлечение значения правильного типа
case int_type:
  v = x.d.i;     //v — целое
  break;
case dbl_type:
  y = x.d.i;     //y — двойной точности
  break;
.....
}
```

Однако такой подход неуклюж и ведет к ошибкам. Неточность в задании t приведет к трудно определяемым ошибкам на этапе выполнения. ООП-средства C++ предоставляют ряд возможностей, которые позволяют делать те же вещи более удобным и безопасным способом.

4.5. Комплексные числа

Множество научных вычислений нуждается в комплексных числах. Напишем АТД для комплексных чисел.

В файле complex1.cpp

```
struct complex {
  double  real, imag;
};

void assign(complex* pc, double r, double i = 0.0)
{
  pc -> real = r;
  pc -> imag = i;
}

complex add(complex a, complex b)
{
  complex temp;

  temp.real = a.real + b.real;
  temp.imag = a.imag + b.imag;
  return temp;
}
```

Обратите внимание, что если задан аргумент по умолчанию, то можно присвоить значение с двойной точностью комплексной переменной. В этом случае мнимая часть принимается равной нулю.

В следующем фрагменте складываются действительное и комплексное числа:

```
double f = 2.5;
complex w, x, z;

assign(&x, 5.5, -3.2);     //x = 5.5 - 3.2i
assign(&w, f);             //w = 2.5 - 0i
z = add(w, x);             //z = 8.0 - 3.2i
```

В упражнениях с 7 по 9 на стр. 126 вы дополните пакет для работы с комплексными числами другими элементами. Традиционный подход, приведенный выше, неудовлетворителен тем, что не позволяет использовать обычную запись выражения для задания вычисления, такую как

```
z = f + w;          //ООП допускает перегрузку +
```

Полный набор возможностей ООП допускает естественную запись кода.

4.6. Пример: флеш

Мы хотим определить вероятность раздачи флеша (flush)[1]. Флешем называется комбинация из по крайней мере пяти карт одинаковой масти. Мы смоделируем перетасовку колоды карт с помощью генератора случайных чисел. Программа написана с применением структур для представления необходимых типов данных и требуемой функциональности.

В файле poker.cpp

```
//Вычисление флеша в покере

enum suit {clubs, diamonds, hearts, spades};
typedef int pips;

struct card {
   suit s;
   pips p;
};

struct deck {
   card d[52];
};
```

Решение задать `deck` как отдельный тип, описывающий колоду, принято при проектировании. Мы можем разработать программу как набор процедур, выполняющих действия исключительно с массивом карт. По сути, колода представляет собой больше, чем просто набор карт. Ведь это 52 совершенно различные карты: по 13 штук треф, бубен, червей и пик. Конечно, объединяя 52 карты в колоду, мы просто моделируем нашу задачу.

```
pips assign_pips(int n)   //n - номер карты (от 0 до 51)
{
   return (n % 13 + 1);
}

suit assign_suit(int n)
{
   return (static_cast<suit>(n / 13));
}
```

Эти две процедуры обеспечивают преобразование из области целых в значения типа `pips` и `suit`. Функции преобразования важны для того, чтобы позволить новым за-

[1] Карточный термин, используемый, например, при игре в покер. — *Примеч. перев.*

данным пользователем типам взаимодействовать с собственными и заданными ранее типами. Позднее мы увидим, как встроить такие преобразования в поведение нового типа (см. раздел 7.1, «Преобразования АТД», на стр. 192).

```
void assign(int n, card& c)
{
  c.s = assign_suit(n);
  c.p = assign_pips(n);
}

pips get_pip(card c)
{
  return c.p;
}

suit get_suit(card c)
{
  return c.s;
}
```

Функция `assign()` устанавливает соответствие между целым со значением от 0 до 51 и уникальной парой значений `suit` и `pips`. Значение 0 становится тузом треф, а значение 51 — королем пик. Следующие две функции обеспечивают читабельный вывод при распечатке колоды.

```
void print_card(card c)
{
  cout << c.p;
  switch(c.s) {
   case clubs:
     cout << "Т ";   //трефы
     break;
   case diamonds:
     cout << "Б ";   //бубны
     break;
   case hearts:
     cout << "Ч ";   //черви
     break;
   case spades:
     cout << "П ";   //пики
  }
}

void print_deck(deck& dk)
{
  for (int i = 0; i < 52; ++i)
    print_card(dk.d[i]);
}
```

Остальные функции используются для перетасовки и раздачи карт.

```
void  shuffle(deck& dk)      //перетасовка колоды
{
   card t;

   for (int i = 0; i < 52; ++i) {
      int k = ((rand() % (52- i)));
      t = dk.d[i];
      dk.d[i] = dk.d[k];
      dk.d[k] = t;            //меняем местами две карты
   }
}

void deal(int n, int pos, card* hand, deck& dk)
{           //раздача
   for (int i = pos; i < pos + n; ++i)
      hand[i - pos] = dk.d[i];
}

void  init_deck(deck& dk)    //инициализация колоды
{
   for (int i = 0; i < 52; ++i)
      assign(i, dk.d[i]);
}
```

Функция `init_deck()` вызывает `assign()` для привязки целых значений к значениям карт. Функция `shuffle()` использует библиотечный генератор псевдослучайных чисел `rand()` для того чтобы попарно менять местами карты в колоде. Опыт показывает, что эта функция дает приемлемое соответствие хорошей перетасовке. Функция `deal()` по очереди вынимает карты из колоды и раздает их по рукам.

Теперь мы можем использовать эти функции для оценки вероятности того, что при раздаче выпадет флеш. Пользователь может выбрать, по сколько карт раздавать (от пяти до девяти).

```
int main()
{
   card one_hand[9];          //максимум 9 карт при раздаче
   deck dk;
   int  i, j, k, fcnt = 0, sval[4];
   int  ndeal, nc, nhand;

   do {
      cout << "\nПо сколько карт раздаем? (от 5 до 9):";
      cin  >> nc;
   } while (nc < 5 || nc > 9);
   nhand = 52 / nc;

   cout << "\nСколько раз играем? ";
   cin  >> ndeal;

   srand(time(NULL));         //time()для "затравки" rand()
   init_deck(dk);
```

```
    print_deck(dk);

    for (k = 0; k < ndeal; k += nhand) {
      if ((nhand + k) > ndeal)
        nhand = ndeal - k;
      shuffle(dk);
      for (i = 0; i < nc * nhand; i += nc) {
        for (j = 0; j < 4; ++j)    //обнуление счетчика масти
          sval[j] = 0;
      deal(nc, i, one_hand, dk);   //очередная сдача

      for (j = 0; j < nc; ++j)
        sval[one_hand[j].s]++;     //увеличение счетчика масти
      for (j = 0; j < 4; ++j)
        if (sval[j] >= 5)          //5 или больше — это флеш
          fcnt++;
      }
    }

    cout << "\n\nИз " << ndeal << " ";
    cout << nc << "-карточных раздач выпало ";
    cout << fcnt << " флешей\n";
}
```

Разбор программы poker

- ```
  card one_hand[9];  //максимум 9 карт при раздаче
  deck dk;
  int  i, j, k, fcnt = 0, sval[4];
  int  ndeal, nc, nhand;
  ```

Это переменные, размещаемые в памяти при входе в блок при выполнении `main`. Переменная `one_hand` — это массив из девяти элементов (наибольшего числа карт в одной руке). В нем содержатся розданные из колоды карты. Переменная `dk` представляет колоду и размещается автоматически. Число карт в одной раздаче хранится в переменной `nc`, а количество раздач содержится в переменной `ndeal`. Переменная `fcnt` служит счетчиком флешей. Массив `sval` содержит число карт одной масти у игрока.

- ```
  do {
    cout << "\nПо сколько карт раздаем? (от 5 до 9):";
    cin  >> nc;
  } while (nc < 5 || nc > 9);
  nhand = 52 / nc;

  cout << "\nСколько раз играем? ";
  cin  >> ndeal;
  ```

Программа запрашивает количество карт в одной раздаче. Для продолжения пользователь должен ввести число от 5 до 9. Число раздач, допускаемое размером колоды

(52 карты) и количеством карт в одной раздаче (nc), вычисляется и помещается в переменную nhand. Затем программа запрашивает, сколько раз вы хотите «сыграть» в покер, то есть сколько всего раздач необходимо осуществить (ndeal).

```
• srand(time(NULL));              //time()для "затравки" rand()
  init_deck(dk);
  print_deck(dk);

  for (k = 0; k < ndeal; k += nhand) {
    if ((nhand + k) > ndeal)
      nhand = ndeal - k;
    shuffle(dk);
```

Функция time() используется генератором случайных чисел для затравки. Переменная dk инициализуется и колода перетасовывается каждый раз при проходе основного цикла.

```
• for (i = 0; i < nc * nhand; i += nc) {
    for (j = 0; j < 4; ++j)       //обнуление счетчика масти
      sval[j] = 0;
  deal(nc, i, one_hand, dk);      //очередная сдача

  for (j = 0; j < nc; ++j)
    sval[one_hand[j].s]++;        //увеличение счетчика масти
  for (j = 0; j < 4; ++j)
    if (sval[j] >= 5)             //5 или больше - это флеш
      fcnt++;
  }
```

Массив sval содержит количество карт каждой масти и инициализуется нулем при каждой сдаче. Функция deal() сдает карты в массив one_hand. Выражение one_hand[j].s является значением масти конкретной карты — например 0, если карта трефовая. Затем это выражение служит индексом массива sval, в котором подсчитываются масти. С помощью переменной fcnt считаются флеши, выпавшие во всех этих испытаниях. Так как число испытаний равно ndeal, вероятность флеша будет fcnt/ndeal.

Программа *poker* расширена в упражнениях с 13 на стр. 127 по 16 на стр. 128.

4.7. Битовые поля

Член целого типа может состоять из определенного количества бит. Такой член называется *битовым полем* (bit field), а число соответствующих бит называется его *шириной*. Ширина определяется неотрицательным постоянным целым выражением, стоящим после двоеточия. Например:

```
struct pcard {//сжатое представление карты
  unsigned s : 2;
```

```
    unsigned p : 4;
};
```

Компилятор попытается упаковать битовые поля в памяти последовательно. При этом он может перейти к следующему байту или слову в целях выравнивания. Массивы битовых полей не допускаются. Кроме того, оператор определения адреса & не может быть применен к битовым полям.

Битовые поля используются для удобной адресации информации в упакованном виде. На многих машинах слово состоит из 32 бит, и битовая операция может выполняться параллельно (то есть одновременно со всеми битами слова). В этом случае битовые манипуляции можно использовать как технику реализации множеств, состоящих не более чем из 32 элементов, как показано ниже.

В файле set.cpp

```
struct word {
unsigned w0:1,w1:1,w2:1, w3:1, w4:1, w5:1, w6:1, w7:1,
   w8:1, w9:1,w10:1,w11:1, w12:1, w13:1, w14:1, w15:1,
   w16:1,w17:1,w18:1,w19:1, w20:1, w21:1, w22:1, w23:1,
   w24:1,w25:1,w26:1,w27:1, w28:1, w29:1, w30:1, w31:1;
};
```

Можно перекрыть `word` и `unsigned` в объединении для создания структуры данных, позволяющей манипулировать битами:

```
union set {
   word     m;
   unsigned u;
};

int main()
{
   set  x, y;

   x.u = 0x0f100f10;
   y.u = 0x01a1a0a1;
   x.u = x.u | y.u; //объединение множеств
   cout << "элемент 9 "
        << ((x.m.w9)? "присутствует" : "отсутствует") << endl;
}
```

В большинстве систем такая операция объединения множеств выполняется как параллельная операция над словами.

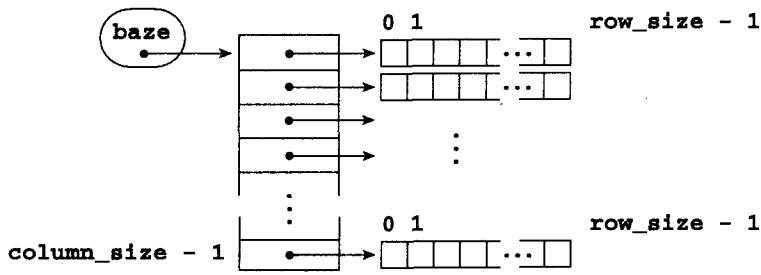
4.8. Пример: двумерные динамические массивы

В базовом языке многомерные динамические массивы (dynamic array) недопустимы. В научных, инженерных и других расчетах интенсивно используются двумерные массивы, называемые *матрицами* (matrix). Было бы неудобно писать специальные процедуры для каждого из возможных размеров матрицы. Соответствующая абстракция может быть реализована следующим образом.

В файле array_2d.cpp

```
//Двумерный динамический массив
struct twod {
   double**  base;              //базовый адрес
   int  row_size, column_size; //размер строки и колонки
};
```

Лежащая в основе структура данных очень проста. Указатель `base` является указателем на указатель на `double`. Указатель `base` содержит начальный адрес массива указателей, каждый из которых является начальным адресом ряда переменных с двойной точностью.



В следующем фрагменте обратите внимание, что функция `allocate()` сначала использует `new` для размещения вектора указателей на колонку, а затем — для размещения строки значений с двойной точностью. Освобождение памяти происходит в обратном порядке.

```
void allocate(int r, int s, twod& m)
{
   m.base = new double*[s];
   for (int i = 0; i < s; ++i)
      m.base[i] = new double[r];
   m.row_size = r;
   m.column_size  = s;
}

void deallocate(twod& m)
{
   for (int i = 0; i < m.column_size; ++i)
      delete [] m.base[i];
   delete [] m.base;
   m.row_size = 0;
   m.column_size  = 0;
}
```

Теперь напишем функцию `find_max()` — каноническую процедуру для обработки подобных двумерных динамических массивов.

```
double find_max(const twod& m)
```

```
{
   int i, j;
   double max = m.base[0][0];

   for (i = 0; i < m.column_size; ++i)
      for (j = 0; j < m.row_size; ++j)
       if (m.base[i][j] > max)
          max = m.base[i][j];
   return (max);
}
```

Обратите внимание как в представленной ниже `main()` функция `find_max()` работает с динамически распределяемым структурами различного размера:

```
#include <iostream.h>

int main()
{
   twod a, b;
   int  i, j;

   allocate(2, 3, a);
   allocate(4, 6, b);
   for (i = 0; i < a.column_size; ++i)
      for (j = 0; j < a.row_size; ++j)
         a.base[i][j] = i * j ;
   for (i = 0; i < b.column_size; ++i)
      for (j = 0; j < b.row_size; ++j)
         b.base[i][j] = i * j ;

   cout << find_max(a) << " наибольшее в массиве 2 * 3\n";
   cout << find_max(b) << " наибольшее в массиве 4 * 6\n";
}
```

Разбор программы array_2d

- ```
  struct twod {
     double**  base;
     int  row_size, column_size;
  };
  ```

Объявление структуры создает новый тип `twod`. Базовый адрес двумерного массива хранится в члене `base`. В переменных `row_size` и `column_size` будут храниться размеры строки и колонки.

- ```
  void allocate(int r, int s, twod& m)
  {
     m.base = new double*[s];
     for (int i = 0; i < s; ++i)
        m.base[i] = new double[r];
  ```

```
    m.row_size = r;
    m.column_size  = s;
}
```

Эта функция используется для размещения массива на этапе выполнения. Сначала размещается колонка указателей на `double`. Каждый из этих указателей содержит базовый адрес для строки из `double`. Эта память выделяется из кучи итеративно с использованием цикла `for`.

- ```
  void deallocate(twod& m)
  {
    for (int i = 0; i < m.column_size; ++i)
      delete [] m.base[i];
    delete [] m.base;
  ```

Функция `deallocate()` работает в обратном порядке по сравнению с `allocate()`. Каждая строка очищается с помощью `delete[]`. Было бы ошибкой сначала очистить `m.base`, причем результат в случае подобной ошибки зависит от системы. Только после того, как память, занимаемая каждой строкой из `double`, освобождена, можно безопасно удалить колонку указателей, представленную `m.base`.

- ```
  double find_max(const twod& m)
  ```

Эта функция работает с произвольными двумерными массивами, она находит наибольший элемент в заданном двумерном массиве.

Функция `find_max()` не привязана к конкретному линейному распределению памяти, что имело бы место в случае с двумерными массивами, размещаемыми в стеке.[1]

4.9. Практические замечания

Применение структур в базовом языке во многих случаях вытесняется созданием АТД — классов с закрытым доступом (для сокрытия реализации) и функциями-членами (для задания поведения АТД). Материал этой главы является промежуточным и помогает программисту понять более примитивную технику инкапсуляции, которая все еще широко применяется. Стеки, комплексные числа и динамические массивы, обсуждаемые выше, контрастируют с более изощренными реализациями в последующих примерах.

Резюме

1. Структура объявляется с помощью ключевого слова `struct`. Она служит механизмом для реализации АТД, таких как комплексные числа и стеки.

[1] Например, чтобы разместить в стеке массив из 50 строк и 50 столбцов понадобился бы непрерывный отрезок свободной памяти на 2500 элементов массива. При динамическом размещении матрицы в куче эти 2500 элементов могут размещаться там, где возможно (не обязательно подряд). — *Примеч. перев.*

2. Тип структуры позволяет программисту объединить компоненты в переменную с одним именем. Структура содержит индивидуально именованные компоненты, называемые *членами*. Решающим моментом при обработке структур является доступ к их членам. Доступ осуществляется либо с помощью оператора выбора члена структуры «.», либо посредством оператора указателя структуры ->. Эти операторы, наряду с () и [], имеют самый высокий приоритет.

3. Объединение — это производный тип, синтаксис которого такой же, как и у структур, за исключением того, что ключевое слово `union` заменяет `struct`. Члены объединения совместно используют память, и их значения перекрываются. Поэтому объединение позволяет интерпретировать свое значение как набор типов, соответствующих объявленным членам. Объединение инициализуется значением в фигурных скобках, причем оно присваивается первому члену объединения.

4. Член целого типа может состоять из определенного количества бит. Такой член называется битовым полем (bit field), а число соответствующих бит называется его шириной. Ширина определяется неотрицательным постоянным целым выражением, стоящим после двоеточия.

5. Оператор `new` можно использовать для создания двумерных массивов и массивов большей размерности, размер которых задается во время выполнения. Функции, использующие `new`, могут работать независимо от размера строк и колонок.

Упражнения

1. Создайте структуру C++ для молочных продуктов (dairy), включающую название, вес порции, энергетическую ценность (в калориях), содержание белков, жиров и углеводов. Двадцать пять грамм Американского сыра содержат 375 калорий, 5 грамм белков, 8 грамм жира и 0 грамм углеводов. Покажите, как присвоить эти значения переменным-членам вашей структуры. Напишите функцию, которая по заданным переменной типа `struct dairy` и весе в граммах (размер порции), возвращала бы число калорий в этой порции.

2. Используя структуру `card`, описанную в разделе 4.1, «Агрегатный тип struct», на стр. 107, напишите процедуру сортировки розданных игроку карт. В карточных играх большинство игроков держит свои карты, отсортировав их по достоинству. Эта процедура должна расположить вначале тузы, затем короли, и так далее до двоек. У игрока пять карт.

3. Следующие объявления не компилируются (содержат ошибки). Объясните, что здесь не так.

```
struct brother {
   char          name[20];
   int           age;
   struct sister sib;
} a;

struct sister {
   char           name[20];
```

```
    int             age;
    struct brother sib;
} a;
```

4. В этом упражнении используйте структуру `ch_stack`, описанную в разделе 4.3, «Пример: стек», на стр. 109. Напишите функцию:

```
void reverse(char s1[], char s2[]);
```

Строки `s1` и `s2` должны быть одинакового размера. Строка `s2` должна стать обращенной копией строки `s1`. Внутри `reverse` используйте `ch_stack` для выполнения обращения.

5. Перепишите функции `push()` и `pop()`, обсуждавшиеся в разделе 4.3, «Пример: стек», на стр. 109 так чтобы проверить, что `push()` не оперирует заполненным `ch_stack`, а `pop()` — пустым. При обнаружении одного из этих условий выведите сообщение об ошибке, используя `cerr`, и выполните `exit(1)` (из *stdlib.h*) для прерывания программы. Сопоставьте это с подходом, в котором применяется проверка утверждений (asserts).

6. Напишите для типа `ch_stack`:

```
//поместить n символов из s1[] в ch_stack stk
void pushm(int n, const char s1[], ch_stack stk);

//извлечь n символов из stk в s1[]
void popm(int n, char s1[],const ch_stack stk);
```

7. Добавьте в пакет комплексных чисел (см. раздел «Комплексные числа», на стр. 115) процедуру вычитания `complex sub(complex& a, complex& b)` и проверьте ее.

8. Добавьте туда же процедуры умножения и деления комплексных чисел.

9. Добавьте процедуру, возвращающую `complex`, когда ей передается `double`. Используйте ее для программирования `complex add(complex, double)` и `complex add(double, complex)`. Заметьте, как здесь перегружается функция `add()`, что обеспечивает полезную интеграцию при смешивании двух типов `double` и `complex`. Подобные перегрузки желательны для обеспечения коммутативности сложения.

10. Мы хотим определить структуру `ch_deque`[1] для реализации двусторонней очереди. Двусторонняя очередь допускает помещение в оба конца (push) и извлечение с обоих концов (pop). Это обычная форма контейнера.

```
struct ch_deque {
  char s[max_len];
  int bottom, top;
};

void reset(ch_deque* deq)
{
```

[1] Double Ended Queue, deque — двусторонняя очередь. — *Примеч. перев.*

```
    deq -> top = deq -> bottom = max_len / 2;
    deq -> top--;
}
```

Объявите и реализуйте `push_t()`, `pop_t()`, `push_b()`, `pop_b()`, `out_stack()`, `top_of()`, `bottom_of()`, `empty()` и `full()`. Функция `push_t()` выполняет *помещение наверх* (push on top). Функция `push_b()` отвечает за *помещение вниз* (push on bottom). Функция `out_stack()` должна выводить всю очередь сверху донизу. Функции `pop_t()` и `pop_b()` соответствуют *извлечению сверху* (pop from top) и *извлечению снизу* (pop from bottom). Если вершина ниже основания, то это означает пустую (empty) очередь. Проверьте каждую функцию.

11. Расширьте тип данных `ch_deque` добавлением функции `relocate()`. Если `ch_deque` заполнена, то вызывается `relocate()`, и содержимое `ch_deque` перемещается в пустую область памяти, выравненную относительно центра `max_len/2` массива `s`. Вот заголовок объявления этой функции:

```
//Возвращает true, если завершается успешно, false – если нет
bool relocate(ch_deque* deq)
```

12. Напишите функцию, которая меняет местами содержимое двух строк. Если вы поместите символьную строку в стек, а затем извлечете ее, она станет обращенной. Но при обмене строками нам нужен неизменный порядок символов. Используйте двустороннюю очередь `deque` для выполнения такого обмена. Строки будут храниться в двух символьных массивах одинакового размера, но сами строки могут быть разной длины. Вот прототип этой функции:

```
void swap(char s1[], char s2[]);
```

13. Напишите функцию `pr_hand()`, которая распечатывает карты игроков. Добавьте ее в программу *poker* из раздела 4.6, «Пример: флеш», на стр. 116 и используйте для распечатки каждого флеша.

14. В разделе 4.6, «Пример: флеш», на стр. 116 функция `main()` определяет наличие флеша. Напишите функцию

```
bool isflush(const card hand[], int nc);
```

которая возвращает значение `true`, если у игрока есть флеш.

15. Напишите функцию

```
bool isstraight(const card hand[], int nc);
```

которая возвращает значение `true`, если игроку выпал стрит. Стрит (straight) — это пять карт последовательного достоинства. Самый младший стрит составляют туз, двойка, тройка, четверка, пятерка. Самый старший — десятка, валет, дама, король, туз. Проведите эксперименты, чтобы определить вероятность того, что розданные карты составят стрит, и сравните результаты при раздаче пяти и семи карт. *Совет*: Вы можете задать массив из 15 целых счетчиков всех возможных значений достоинств карт. Не забудьте, что тузы должны рассматриваться

как младшая карта (значение 1) или как старшая карта (со значением 14), в зависимости от стрита.

16. Используйте предыдущее упражнение для определения вероятности того, что игроку в покер достанется флеш-стрит, то есть флеш, являющийся одновременно и стритом. Это редчайший покерный расклад, и он имеет наивысшее старшинство. Заметьте, что при сдаче более пяти карт недостаточно просто проверки наличия и стрита, и флеша для определения того, что игроку достался флеш-стрит.

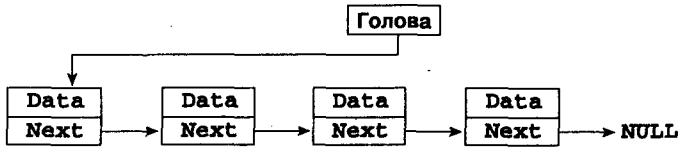
17. *(Сложное).* Используя битовые поля, напишите процедуру перетасовки колоды, которая на вашей машине хранит колоду в наименьшем числе машинных слов. Постарайтесь сделать перетасовку колоды случайной и эффективной. Если вы используете шесть бит для хранения карты (два бита на масть, четыре на достоинство), то некоторые сочетания бит никогда не будут использоваться. Убедитесь, что ваша процедура генерирует 52 различные карты.

18. *(Проект).* Этот проект можно реализовывать по-разному и развивать в различных направлениях. Напишите покерную программу, которая позволяет играть против машины. Если можно, используйте графику для обеспечения приемлемого отображения выпавших карт. Нужно реализовать стандартную форму покера (с раздачей пяти карт). Машина должна уметь оценивать свои карты и делать подходящие ставки. Машина также должна обладать способностью блефовать, то есть делать ставки, не соответствующие силе ее карт.

19. Давайте определим структуру с членом данных, который указывает на тот же тип структуры, так чтобы она позволяла оперировать произвольным числом таких структур, связанных вместе.

```
struct slist {
    char   data;
    slist* next;
};
```

Каждая переменная типа `slist`[1] имеет два члена: `data` (данные) и `next` (следующая). Переменная-указатель `next` является *связью*. Каждая структура связана с последующей при помощи члена `next`. Переменная-указатель `next` содержит или адрес расположения в памяти следующего элемента `slist`, или специальное значение NULL, которое определено как символическая константа со значением 0.



Напишите функцию, создающую список из строки. Функция будет возвращать указатель на начало итогового списка. Тело функции создает элемент списка, выделяя память и присваивая членам значения.

[1] Singly Linked List, slist — односвязный список. — *Примеч. перев.*

```
//Создание списка с помощью рекурсии
slist* string_to_list(char s[])
{
  slist* head;
  if (s[0] == '\0')  //базовый случай
       //что должно быть здесь?
  else {
    head = new slist;
       //а здесь что делать?
    head -> next = string_to_list(s + 1);
    return head;
  }
}
```

20. Напишите списочную операцию для подсчета элементов списка. Подразумевается рекурсия до конца списка с выходом после обнаружения указателя NULL. Если список оказывается пустым, возвращается значение 0, в противном случае возвращается число элементов в списке.

```
int count(slist* head);
```

21. Напишите функцию `lookup()`, которая ищет в списке конкретный элемент `c`. Если элемент найден, возвращается указатель на этот элемент, в противном случае возвращается указатель на NULL.

```
slist* lookup(char c, slist* head);
```

Глава 5

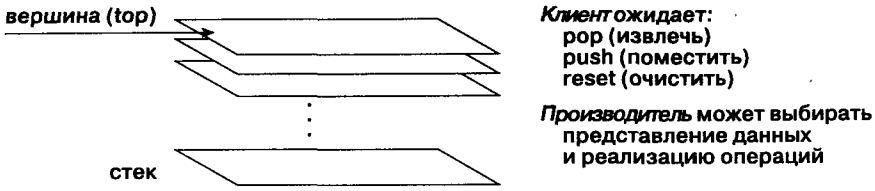
Сокрытие данных и функции-члены класса

Первоначально Страуструп назвал свой язык «С с классами». Класс является расширением понятия структуры, используемого в С. Класс объединяет тип данных вместе со связанными с ним функциями и операторами. В C++ структуры могут включать в себя функции-члены; кроме того, некоторая часть описаний структур может быть сокрыта. И то, и другое расширение языка С будут описаны в этой главе.

Мы будем использовать термин *клиент* для обозначения пользователя[1] абстрактного типа данных (АТД), а термин *производитель* — для обозначения поставщика АТД. Компания Honda производит автомобиль под названием Accord. Хоть я и владею Honda Accord, но меня мало интересует внутреннее устройство ее рулевого механизма. Для меня автомобиль — это некий черный ящик с определенными возможностями. Как клиент я хочу, чтобы моя машина работала эффективно и обеспечивала мне тот уровень сервиса, который можно ожидать от автомобиля. Например, позволяла бы мне проехать достаточное расстояние на одном баке бензина и могла бы перевезти некоторое количество груза в багажнике. Honda Accord приближается к абстракции «автомобильность», которая удовлетворяет моим потребностям.

Клиент АТД также предполагает примерное соответствие некой абстракции. Для того, чтобы можно было использовать стек, он должен быть приемлемого размера. Комплексное число должно быть представлено с разумной точностью. Колода должна тасоваться так, чтобы при раздаче карты распределялись случайным образом. Внутренние особенности вычисления поведения всех этих объектов не являются непосредственной заботой клиента. Чтобы быть конкурентоспособным, производитель, при разумном сочетании цены и эффективности, должен состязаться в реализации некой формы АТД в своих продуктах.

[1] Пользователи АТД бывают разные, как то: люди или организации, купившие библиотеку классов; другие АТД, опирающиеся на данный (более примитивный) АТД и т. п. — *Примеч. перев.*



Производитель заинтересован в сокрытии деталей разработки. Такое сокрытие значительно упрощает все то, что производитель должен объяснять клиенту. Это дает ему свободу для дальнейших усовершенствований продукта, которые не влияют на особенности использования продукта клиентом. Оно защищает клиента от опасного, пусть даже непреднамеренного искажения продукта.

Структуры и обычные функции позволяют строить полезные АТД, но они не поддерживают разграничения между клиентом и производителем. Клиент имеет доступ к внутренним деталям и может модифицировать их нежелательным образом. Например, клиент `ch_stack` из раздела 4.3, «Пример: стек», на стр. 109 может обратиться к внутреннему массиву, используемому для представления `ch_stack`. Это нарушает абстракцию LIFO (Last-In-First-Out, первым вошел — последним вышел), которую `ch_stack` реализует.

Чтобы исключить подобное вмешательство, ООП требуется механизм сокрытия данных. В C++ сокрытие данных осуществляется с помощью ключевого слова `class` и ключевых слов доступа `public` (открытый), `private` (закрытый) и `protected` (защищенный). Закрытые члены класса (как понятно из названия) скрыты от клиентской программы, а открытые доступны ей. Можно изменить представление скрытых данных, но не следует изменять доступ к открытым членам и их функциональность. Если сокрытие данных произведено правильно, клиентская программа не нуждается в изменениях, когда модифицируется представление скрытых данных.

5.1. Функции-члены класса

Концепция `struct` расширена в C++ так, что функции могут быть членами структур. Объявление функции включается в объявление структуры и эта функция вызывается с использованием методов доступа, которые определены для членов структуры. Идея заключается в том, что функциональность, необходимая для работы с типом данных `struct`, должна быть прямо включена в объявление структуры. Такая конструкция улучшает инкапсуляцию абстрактного типа данных `ch_stack`, поскольку упаковывает прямо в эту структуру надлежащее представление данных.

Давайте перепишем наш пример `ch_stack` и объявим как функции-члены различные функции, связанные с `ch_stack`.

В файле ch_stac2.h

```
struct ch_stack {
//представление данных
   int top;
   enum{max_len = 100, EMPTY = -1, FULL = max_len-1};
   char s[max__len];
```

```
//операции, представленные как функции-члены
  void reset() { top = EMPTY; }
  void push (char c) { top++; s[top] = c; }
  char pop() { return s[top--]; }
  char top_of() { return s[top]; }
  bool empty() { return (top == EMPTY); }
  bool full() { return (top == FULL); }
};
```

Функции-члены (member function) записаны подобно обычным функциям. Единственное отличие состоит в том, что они могут использовать имена членов класса «как есть». Так, функции-члены ch_stack используют переменные top и s без явного указания имени структуры. При вызове функций-членов применительно к конкретному объекту типа ch_stack они действуют на указанные члены данных именно этого объекта.

Следующий пример иллюстрирует изложенные идеи. Пусть описаны две переменных типа ch_stack:

```
ch_stack data, operands;
```

тогда

```
data.reset ();
operands.reset();
```

вызывают функцию-член reset, которая устанавливает data.top и operands.top в значение EMPTY. Если объявлен указатель на ch_stack

```
ch_stack* ptr_operands = &operands;
```

то

```
ptr_operands -> push('A');
```

вызывает функцию-член push, которая приводит к приращению operands.top и устанавливает operands.s[top] в 'A'. И последнее наблюдение: имя функции-члена top_of() изменено по сравнению с предыдущей реализацией, чтобы избежать конфликта имен.

Функции-члены, которые определены внутри struct, являются неявно встраиваемыми. Как правило, только короткие, интенсивно используемые функции-члены должны определяться внутри struct, как в только что приведенном примере. Чтобы определить функцию-член вне struct, используется оператор разрешения области видимости (scope resolution). Давайте проиллюстрируем это, заменив определение функции push в структуре ch_stack на соответствующий прототип. Перепишем наш пример, используя оператор разрешения области видимости. В этом случае функция уже не будет неявно встраиваемой.

```
struct ch_stack {
//представление данных
  int top;
  enum {max_len = 100, EMPTY = -1, FULL = max_len-1};
  char s[max_len];
```

```
//операции, представленные как функции-члены
   void reset() { top = EMTY; }   //неявно встроенная функция
   void push (char c);            //прототип функции
   .....
};

void ch_stack::push (char c) //определение невстроенной функции
{
   top++;
   s[top] = c;
}
```

Оператор разрешения области видимости позволяет функциям-членам из разных типов `struct` иметь одинаковые имена. То, какая именно функция-член вызывается, зависит от типа объекта, на который она направлена. Функции-члены внутри одной структуры могут быть перегружены. Добавим к типу данных `ch_stack` еще одну операцию извлечения элементов из стека (pop), которая имеет целый параметр — количество элементов, подлежащих извлечению. Можно поместить в структуру следующий прототип функции:

```
struct ch_stack {
   .....
   char pop(int n);   //в пределах ch_stack
   .....
};
char ch_stack::pop(int n)
{
   while (n-- > 1)
      top--;
   return s[top--];
}
```

Что конкретно будет вызвано, зависит от текущих аргументов функции `pop`:

```
data.pop();      //вызывает стандартный pop
data.pop(5);     //вызывает многократный pop
```

Объявление

```
ch_stack s, t, u;
```

создает три отдельных объекта `ch_stack` размером в `sizeof(ch_stack)` байт. Каждая из этих переменных имеет свои собственные члены данных:

```
int top;
char s[max_len];
```

Функция-член по определению является частью типа. Не существует отдельной функции-члена для каждой из этих трех переменных `ch_stack`.

Спецификация `inline` может использоваться явно с функциями-членами, определенными в пределах области видимости файла. Это позволяет избежать включения тела встраиваемой функции в определение класса.

```
struct ch_stack {
   .....
   void reset();
   void push(char c);
   .....
};
inline void ch_stack::reset()
{
   top =EMPTY;
}
inline void ch_stack::push(char c)
{
   s[++top] = c;
}
```

Объединение операций с данными подчеркивает их «объектность». Объекты имеют описание и поведение. Представим себе объект как существительное, а его поведение — как глаголы, которые чаще всего употребляются с этим существительным. Объектно-ориентированный подход к разработке фокусируется на концепциях и данных.

5.2. Доступ: закрытый и открытый

Понятие структуры расширено в C++, так что появилась возможность вводить закрытые и открытые члены данных и функции-члены. Внутри структуры использование ключевого слова `private` с последующим двоеточием ограничивает доступ к членам, которые идут за этой конструкцией. Закрытые члены могут использоваться лишь некоторыми категориями функций, чьи полномочия включают право доступа к этим членам. Таковыми являются функции-члены структур. Другие категории функций, имеющие подобное право, будут рассмотрены позднее.

Изменим наш пример `ch_stack`, чтобы скрыть представление данных:

В файле ch_stac3.h

```
struct ch_stack {
public:
   void reset() {top = EMPTY; }
   void push(char c) { top++; s[top] =c; }
   char pop() { return s[top--]; }
   char top_of() { return s[top]; }
   bool empty() { return (top == EMPTY); }
   bool full() { return (top == FULL); }
private:
   enum {max_len = 100, EMPTY = -1, FULL = max_len-1};
   char s[max_len];
   int top;
};
```

Теперь перепишем функцию `main()` из раздела 4.3, «Пример: стек» на стр. 112, чтобы проверить наши операции:

В файле ch_stac3.cpp

```
//Обращение строки с использованием ch_stack

int main()
{
   ch_stack s;
   char str[40] = {"Меня зовут Дон Кнут!"};
   int i = 0;

   cout << str <<endl;
   s.reset();                  //s.top = EMPTY было бы ошибкой
   while (str[i] && !s.full())
      s.push(str[i++]);
   while (!s.empty())          //печать обращенной строки
      cout << s.pop();
   cout << endl;
}
```

Результат работы этой версии тестовой программы будет выглядеть так:

```
Меня зовут Дон Кнут!
!тунК ноД тувоз яневМ
```

Как указано в комментарии, доступ к скрытой переменной `top` ограничен. Она может быть изменена с помощью функции-члена `reset()`, но к `top` нельзя обратиться напрямую. Отметьте также, как переменная `s` передается каждой функции-члену с использованием оператора выбора члена структуры.

Структура `ch_stack` имеет закрытую часть, содержащую описания данных, и открытую часть, содержащую функции-члены, необходимые для осуществления операций в `ch_stack`. Полезно рассматривать закрытую часть как код, доступный только разработчику, а открытую часть — как описание интерфейса, который используется клиентами. Разработчик может изменить закрытую часть, и это не повлияет на правильность использования типа `ch_stack` клиентом.

Методика сокрытия данных — важный компонент ООП. Ее использование ведет к упрощению отладки и сопровождения кода, поскольку ошибки и изменения локализованы. Клиентские же программы должны лишь иметь представление о спецификации интерфейса типов.

5.3. Классы

В C++ классы вводятся ключевым словом `class`. Они являются формой структуры, права доступа к членам которой по умолчанию определены как закрытые (private). Таким образом, `srtuct` и `class` взаимозаменяемы, но необходимо определять надлежащие права доступа.

Давайте перепишем пример с комплексными числами:

```
struct complex {
public:
   void assign (double r, double i);
   void print()
```

```
    { cout << real << " + " << imag << "i ";}
private:
  double real, imag;
};

inline void complex::assign(double r, double i = 0.0)
{
  real = r;
  imag = i;
}
```

А вот эквивалентное представление класса:

В файле complex2.cpp

```
class complex {
public:                                    //стиль «должен знать»,
  void assign (double r, double i);        //см. ниже
  void print()
    { cout << real << " + " << imag << "i ";}
private:
  double real, imag;
};

inline void complex::assign(double r, double i = 0.0)
{
  real = r;
  imag = i;
}
```

Также возможно было бы:

```
class complex {            //старый стиль, подразумевающий
  double real, imag;       //закрытый доступ
public:
  void assign (double r, double i);
  void print()
    { cout << real << " + " << imag << "i ";}
};
```

Здесь используется тот факт, что по умолчанию доступ к членам класса является закрытым. В дальнейшем мы будем предпочитать класс структуре, если не все члены являются открытыми. Мы также будем явно использовать ключевые слова доступа и в тексте программы сперва располагать открытые члены, а затем — закрытые. Мы называем такой стиль «должен знать» («need-to-know» style), потому что все должны знать об открытом интерфейсе, но только поставщику класса необходимы сведения о закрытых деталях его реализации.

В чем преимущество такого подхода по сравнению с описанием типа данных `complex` в разделе 4.5, «Комплексные числа», на странице 115? Мы приобрели мо-

дульность. Наличие функций-членов внутри класса показывает четкую связь между типом данных `complex` и связанными с ним операциями `assign()` и `print()`. Кроме того, уменьшается вероятность неверного использования представления, так как подробности реализации данных `real` и `imag` являются закрытыми. Попытка прямо изменить эти члены приведет к синтаксической ошибке нарушения прав доступа, поэтому клиент этой версии `complex` обязан использовать функции-члены, а они правильно взаимодействуют с комплексными переменными.

5.4. Область видимости класса

Классы привносят новые правила определения области видимости (scope) в дополнение к существующим в ядре языка C. (См. раздел 3.8, «Область видимости и класс памяти», на стр. 77). Одна из целей, преследуемых при использовании классов — обеспечить применение техники инкапсуляции. Это означает, что все имена, объявленные внутри класса, должны трактоваться внутри их собственного пространства имен, в отличие от внешних имен, имен функций и имен других классов. Появляется потребность в ведении оператора разрешения[1] области видимости (scope resolution).

5.4.1. Оператор разрешения области видимости ::

Оператор разрешения области видимости является оператором самого высокого приоритета в языке. Он может принимать две формы:

```
::i             //унарный оператор – указывает
                //на внешнюю область видимости
foo_bar::i      //бинарный оператор – указывает
                //на область видимости класса
```

Унарная форма используется для того, чтобы раскрыть имя (получить к нему доступ), которое имеет внешнюю область видимости и было скрыто объявлением в локальной области видимости или в области видимости класса:

В файле how_many.cpp

```
int count = 0;   //внешняя переменная

void how_many(double w[], double x, int& count)
{
   for (int i = 0; i < N; ++i)
      count += (w[i] == x);
   ++ ::count;
}
```

Бинарная форма используется для того, чтобы устранить неоднозначность в именах, которые повторно используются в пределах класса, как показано ниже. Позже мы будем применять эту форму с наследованием и пространствами имен.

```
class widgets { public: void f(); };
class gizmos  { public: void f(); };
```

[1] Здесь слово «разрешение» употребляется в значении «определение», «уточнение», а не в смысле «позволение». — *Примеч. перев.*

```
void f() {/*что-нибудь*/} //просто внешняя функция f
void widgets::f() {/*что-нибудь*/}    //f с областью видимости
                                      //widgets
void gizmos::f() {/*что-нибудь*/}     //f с областью видимости
                                      //gizmos
```

Можно представить себе оператор разрешения области видимости, как то, что указывает путь к конкретному идентификатору. Отсутствие модификатора области видимости означает, что применяются обычные правила видимости имен.

Продолжая предыдущий пример:

```
widgets w;
gizmos g;
g.f();
w.f();
g.gizmos::f();      //верно и избыточно
g.widgets::f();     //неверно, widgets::f() не может
                    //действовать на gizmos
```

5.4.2. Вложенные классы

Классы, также как блоки и пространства имен, могут быть вложены. Области видимости и вложенность позволяют локально скрывать имена и выделять ресурсы. Часто случается, что класс нужен как часть реализации некой большей конструкции (см. раздел 6.7, «Пример: односвязный список», на стр. 169, где `slistelem` вложен в `slist`). Правила для вложенных классов изменились по сравнению со стандартным C. Следующие вложенные классы иллюстрируют действующие правила C++:

В файле nested.cpp

```
char c;          //внешняя область видимости  ::c

class X {        //объявление объемлющего класса X::
  char c;        //X::c
public:
  class Y {      //объявление вложенного класса X::Y::
  public:
    void foo(char e) {X t; ::c = t.X::c = c = e;}
  private:
    char c;      //X::Y::c
  };
};
```

В классе `Y` функция-член `foo()`, используя `::c`, ссылается на глобальную переменную `c`; используя `X::c`, ссылается на переменную объемлющего класса; используя `c`, ссылается на переменную `X::Y::c` вложенного класса. Все три переменные с именем `c` доступны при использовании оператора разрешения области видимости.

Более того, классы с чисто локальной областью видимости могут создаваться внутри блоков. Их определения недоступны вне контекста соответствующих блоков.

```
void foo()
{
   class local { ..... } x;
                //что-нибудь
}
local y;        //неверно: local виден
                //только внутри foo()
```

Отметим, что C++ позволяет вкладывать определения функций, используя вложение классов. Это ограниченная форма вложения функций. Функции-члены должны быть определены внутри локального класса и на них нельзя ссылаться вне его области видимости. Как и в C, простое вложение функций недопустимо.

5.5. Пример: пересмотрим флеш

Программа poker из раздела 4.6, «Пример: флеш», на стр. 116 будет записана с использованием классов, чтобы представить нужные типы данных и функциональность.

В файле poker2.cpp

```
//Вычисление флеша в покере

#include <iostream.h>
#include <stdlib.h>  //для генерации случайного числа
#include <time.h>    //для затравки случайного числа

enum suit { clubs, diamonds, hearts, spades };
                     //масть (трефы, бубны, червы, пики)

class pips {         //достоинство карты
public:
   void assign(int n) { p = n % 13 + 1; }
   int getpip() { return p; }
   void print() {cout << p; }
private:
   int p;
};

class card {         //карта
public:
   suit s;
   pips p;
   void assign(int n)
      { cd = n; s = static_cast<suit>(n/13); p.assign(n); }
   void pr_card();
private:
   int cd;           //cd - от 0 до 51
};
```

```
class deck {          //колода
public:
   void init_deck();  //инициализация колоды
   void shuffle();    //перетасовка колоды
   void deal(int, int, card*); //сдача
   void pr_deck();
private:
   card d[52];
};
```

В этом примере за счет объединения функций-членов с членами данных достигается лучшая модульность по сравнению с аналогичными конструкциями ядра языка в разделе 4.6, «Пример: флеш», на стр. 116. Поведение и описание логически сгруппированы вместе. Каждый уровень описания скрывает сложность предыдущего уровня.

```
void deck::init_deck()
{
   for (int i = 0; i < 52; ++i)
      d[i].assign(i);
}

void deck::shuffle()
{
   for (int i = 0; i < 52; ++i) {
      int k = i + ((rand() % (52-i)));
      card t = d[i];//меняем местами две карты
      d[i] = d[k];
      d[k] = t;
   }
}

void deck::deal(int n, int pos, card* hand)
{
   for (int i = pos; i < pos + n; ++i)
      hand[i - pos] = d[i];
}
```

Параллель с разделом 4.6. «Пример: флеш», на стр. 116 очевидна. Функция `init_deck()` вызывает `card::assign()`, чтобы преобразовать целые в достоинство карт. Функция `shuffle()` использует библиотечный генератор псевдослучайных чисел `rand()` чтобы попарно менять местами карты в колоде. Функция `deal()` по очереди вынимает карты из колоды и раздает их по рукам.

```
int main()
{
   card one_hand[9];  //максимум по 9 карт в одни руки
   deck dk;
   .....
```

```
    dk.init_deck();
    dk.print();
    .....
      dk.shuffle();
        .....
        for (j = 0; j < nc; ++j)
          sval[one_hand[j].s]++;
    .....
}
```

5.6. Статические члены данных

Члены данных могут быть объявлены с использованием модификатора класса памяти `static`. Член данных, который объявлен как `static`, разделяется всеми переменными своего класса и хранится в одном месте. Нестатические члены данных создаются для каждого экземпляра класса. Если бы не наличие статических членов данных, сведения, необходимые всем экземплярам класса, должны были бы объявляться глобальными. Это разорвало бы отношения между данными и их классом. Статический член позволяет данным класса, которые не специфичны для отдельного экземпляра, существовать в области видимости класса.

Так как статический член данных не зависит от конкретного экземпляра, к нему можно обратиться следующим образом:

имя-класса : : *идентификатор*

Здесь используется оператор разрешения области видимости. Статический член глобального класса должен быть явно объявлен и определен в области видимости файла. Например:

```
class str {
public:
   static int how_many;     //объявление
   void print();
void assign(const char*);
   .....
private:                     //реализовано в виде символьного
   char s[100];              //массива фиксированной длины
};

int str::how_many = 0        //определение и инициализация
```

В нашем примере функция how_many может отслеживать, сколько памяти используется для хранение переменной `str`. Итак,

```
str s1, s2, s3, *p;

str::how_many = 3;           //предпочтительнее использовать ::
   .....
str t;
t.how_many++;                //оператор доступа «точка»
   .....
```

```
p = new str;
p -> how_many++;    //оператор доступа через указатель
   .....
delete p;
str::how_many--;
```

Предпочтительным стилем программирования для доступа к статическим членам является использование разрешения области видимости. Доступ с использованием указателя и оператора «точка» может ввести в заблуждение; не видно, что член является статическим. Новым является разрешение инициализации статической константы в пределах объявления класса:

```
class ch_stack {
.....
private:
   static const int max_len = 10000;   //инициализатор
.....
};

const stack::int max_len;              //необходимо объявление
```

5.7. Указатель this

Ключевое слово `this` обозначает объявленный неявно указатель на себя. Он может использоваться только в нестатических функциях-членах. В статических функциях-членах неявные аргументы недопустимы. Простая иллюстрация использования указателя `this` приведена ниже.

В файле c_pair.cpp

```
//Указатель this

class c_pair {
public:
   void init(char b) { c2 = 1 + (c1 = b); }
   c_pair increment() { c1++; c2++; return (*this); }
   c_pair* where_am_I() { return this; }
   void print() { cout << c1 << c2 << '\t'; }
private:
   char c1, c2;
};

int main()
{
   c_pair a, b;

   a.init('A');
   a.print();
   cout << " is at " << a.where_am_I() << endl;
   b.init('B');
   b.print();
```

```
    cout << " is at " << b.where_am_I() << endl;
    b.increment().print();
}
```

Функция-член `increment` использует неявный указатель `this`, чтобы возвратить приращенные значения `c1` и `c2`. Функция-член `where_am_I` возвращает адрес заданного объекта. Ключевое слово `this` предоставляет встроенный не требующий объявления указатель. Это то же самое, как если бы в `c_pair` неявно объявлялся закрытый член `c_pair* const this`. Указатель `this` нельзя изменить.

5.8. Функции-члены типа static и const

C++ позволяет использовать функции-члены типа `static` и `const`. Синтаксически статическая функция-член содержит модификатор `static`, предшествующий возвращаемому типу функции внутри объявления класса. Определение вне класса не должно включать этот модификатор:

```
class foo {
.....
   static int foo_fcn();   //сначала – ключевое слово static
.....
};

int foo::foo_fcn()         //здесь не должно быть слова static
{ /* определение функции */ }
```

Синтаксически, функция-член типа `const` вводится модификатором `const`, следующим за списком аргументов внутри объявления класса. Определение вне класса также должно включать этот модификатор:

```
class foo {
.....
   int foo_fcn() const;
.....
};

int foo::foo_fcn() const     //необходимо ключевое слово const
{ /* определение функции */ }
```

Особенности применения функций-членов `const` и `static` можно усвоить, используя указатель `this`. Обычная функция-член, вызываемая как

```
x.mem(i, j, k);
```

имеет явный список аргументов и неявный список аргументов, а именно, список всех членов данных объекта x. Неявные аргументы могут пониматься как данные, доступные через указатель `this`. В противоположность этому, статическая функция-член не получает неявных аргументов. Постоянная функция-член не может модифицировать свои неявные аргументы. Объявление постоянных функций-членов и постоянных параметров называется *контролем постоянства* (const-correctness). Контроль постоянства — важное подспорье при написании кода. Благодаря такому контролю можно

быть уверенным в том, что компилятор убедится в неизменности значений объектов. Кроме того, контроль постоянства позволяет компилятору производить специальную оптимизацию, например, расположить объект const в памяти только для чтения.

Следующий пример иллюстрирует вышесказанное.

В файле salary.cpp

```
//Вычисление оклада с использованием статических членов

class salary {                          //оклад
public:
   void init(int b) {b_sal = b; your_bonus = 0;}
   void calc_bonus(double perc)   //вычисление личной премии
     { your_bonus = b_sal * perc; }
   //премия для всех сотрудников
   static void  reset_all(int p) { all_bonus = p; }
   int  comp_tot() const          //суммарный оклад
     { return (b_sal + your_bonus + all_bonus); }
private:
   int         b_sal;
   int         your_bonus;
   static int all_bonus;          //объявление
};

//объявление и определение
int salary::all_bonus = 100;

int main()
{
   salary w1, w2;

   w1.init(1000);
   w2.init(2000);
   w1.calc_bonus(0.2);
   w2.calc_bonus(0.15);
   salary::reset_all(400);
   cout << " w1 " << w1.comp_tot() << " w2 "
        << w2.comp_tot() << endl;
}
```

Разбор программы salary

- ```
  class salary {
     . . .
  private:
     int         b_sal;
     int         your_bonus;
     static int all_bonus;//объявление
  };
  ```
  ```

В этом классе три закрытых члена данных. Статический член `all_bonus` нуждается в объявлении, область видимости которого — весь файл. Он существует независимо от всех переменных типа `salary`, которые были объявлены.

- `void init(int b) {b_sal = b; your_bonus = 0;}`

Здесь значение `b` присваивается члену `b_sal`. Данная функция-член задает начальное значение основного оклада. Переменная `your_bonus` инициализуется нулем. Хотя в нашем маленьком примере это и необязательно, инициализовать все переменные — хорошая привычка. В следующей главе рассматриваются специальные функции, называемые конструкторами. Они используются, когда необходимы инициализация и создание объектов.

- ```
  static void  reset_all(int p)
        { all_bonus = p; }
  ```

Модификатор `static` должен находиться перед типом, возвращаемым функцией.

- ```
  int  comp_tot() const
        { return (b_sal + your_bonus + all_bonus); }
  ```

Модификатор `const` стоит между списком аргументов и телом определения функции. Он показывает, что никакие члены данных не изменят своего значения, что придает коду большую ясность. Можно сказать, что указатель на себя передается как `const salary* const this`.

- `salary::reset_all(400);`

Статическая функция-член может быть вызвана с использованием оператора разрешения области видимости. Она может быть также вызвана как:

```
w1.reset_all(400);
```

но это вводит в заблуждение, поскольку, на самом деле, в переменной `w1` нет ничего такого, что связывало бы именно ее (а не какую-то другую переменную класса) со статической функцией-членом `reset_all()`.

Обратите особое внимание, что ключевое слово `static` используется только в определении класса и должно опускаться, когда данные или функция-член определяются вне класса.

5.8.1. Изменчивость (mutable)

Ключевое слово `mutable` позволяет членам класса, переменные которого были объявлены как константы, быть, тем не менее, изменяемыми. Таким образом отпадает необходимость отказываться от постоянства, используя конструкцию `const_cast<>`. Это относительно новая возможность, поддерживаемая не всеми компиляторами C++. Вот как она применяется:

В файле mutable.cpp

```
//Класс с членами mutable
class person {                    //человек
```

```
public:
   person(const char*, int, unsigned long);
   void bday() { ++age; }          //в день рождения
                                   //увеличиваем возраст
   .....
private:
   const char* name;
   mutable int age;                //возраст всегда изменяется
   unsigned long soc_sec;
};
.....
{
   const person ira("ira pohl", 38, 1110111);
.....
   ira.bday();                     //правильно, ira.age - mutable
}
```

5.9. Контейнеры и доступ к их содержимому

Контейнерные классы, такие как стеки и двумерные динамические массивы,— очень полезные типы данных. Код twod из раздела 4.8, «Пример: двумерный динамический массив», на стр. 121 может быть переписан в виде класса. При этом можно добавить возможность доступа к отдельным членам объектов и их модификации.

В файле twod.cpp

```
//Двумерный динамический массив

class twod {
public:
   bool allocate(int r, int s);   //псевдоконструктор
   void deallocate();             //псевдодеструктор
   double& element_lval(int i, int j) const
      { return base[i][j]; }
   double element_rval(int i, int j) const
      { return base[i][j]; }
   int  r_size() const { return row_size; }
   int  c_size() const { return column_size; }
private:
   double**  base;
   int       row_size, column_size;
};

bool twod::allocate(int r, int s)
{
   base = new double*[s];
   if (base == 0)//выделение памяти не выполнено
      return false;
```

```
    for (int i = 0; i < s; ++i) {
      base[i] = new double[r];
      if (base[i] == 0)          //выделение памяти не выполнено
          return false;
    }
    row_size = r;
    column_size = s;
    return true;
}

void twod::deallocate()
{
    for (int i = 0; i < column_size; ++i)
      delete [] base[i];
    delete [] base;
    row_size = 0;
    column_size = 0;
}
```

Обратите внимание, как в этом варианте `twod` скрыто внутреннее представление. Функциям-членам, действующим на объект неявно, требуется на один аргумент меньше, чем в соответствующей реализации с использованием обычных функций. Конечно, аргумент все равно существует, но он передается посредством указателя `this`. Функция-член `allocate()` используется явно для создания объекта `twod`. Работа по конструированию настолько важна при производстве объектов, что для этой цели существует специальная категория функций-членов, называемых *конструкторами (*constructors*)*. Конструкторы могут использоваться неявно (см. главу 6, «Создание и уничтожение объектов»).

Выше определены две почти идентичные функции-члена — `element_lval()` и `element_rval()`. Они служат для иллюстрации разницы между понятиями *lvalue* (левостороннее значение или именующее выражение) и *rvalue* (правостороннее значение). Отличие кроется в возвращаемом типе. Возвращаемый тип `double&` предполагает, что возвращается ссылка на объект. Так, `element_lval()` определяет lvalue для манипулирования отдельными индексированными элементами. Функция `double element_rval()` требует, чтобы было возвращено rvalue индексированных элементов. Изложенное можно проиллюстрировать так:

```
twod m;
m.allocate(5, 10);

m.element_lval(1, 1) = 5;          //поместим 5 в m.base[1][1]
m.element_rval(2, 2) = 6.5;        //недопустимое присваивание
                                   //rvalue
cout << (m.element_rval(1, 1) ==
         m.element_lval(1, 1));    //правильно, 5 == 5
```

Мы можем написать функцию `find_max()`, которая находит наибольшее значение элемента двумерного объекта:

```
double find_max(const twod& m)
{
   int    i, j;
   double max = m.element_rval(0,0);
   for (i = 0; i < m.c_size(); ++i)
      for (j = 0; j < m.r_size(); ++j)
         if (m.element_rval(i, j) > max)
            max = m.element_rval(i, j);
   return max;
}
```

Заметьте, что `element_lval()` тоже можно было бы использовать. Теперь давайте напишем алгоритм, транспонирующий двумерную матрицу.

```
void transpose(twod& m)
{
   int    i, j;
   double temp;

   for (i = 0; i < m.c_size() - 1 ; ++i)
      for (j = i + 1; j < m.r_size(); ++j) {
         temp = m.element_lval(i, j);
         m.element_lval(i, j) = m.element_lval(j, i);
         m.element_lval(j, i) = temp;
      }
}
```

Поскольку этот алгоритм манипулирует значениями, которые хранятся в элементах контейнерного объекта m, он предполагает использование lvalue.

5.10. Практические замечания

Традиционно при определении класса сперва писались члены с закрытыми правами доступа:

```
class ch_stack {
private:
   int top;
   enum {max_len = 100, EMPTY = -1, FULL = max_len-1};
   char s[max_len];
public:
   void reset() {top = EMPTY; }
   void push(char c) { top++; s[top] =c; }
   char pop() { return s[top--]; }
   char top_of() const { return s[top]; }
   bool empty() const { return (top == EMPTY); }
   bool full() const { return (top == FULL); }
};
```

Так сложилось, потому что в оригинальном C++ не было ключевых слов доступа `private` и `protected`. По умолчанию, доступ к членам конструкции `class` был закрыт, следовательно, члены `private` должны были вводиться в начале.

Наш стиль, когда сперва пишутся открытые члены, становится нормой. Он следует правилу, согласно которому доступ к открытым членам необходим более широкой аудитории. Специализированная информация располагается позднее в объявлении класса.

В общем случае члены данных должны быть закрытыми. Это важная заповедь программирования. Обычно данные отражают конкретный выбор способа реализации класса. Доступ к ним должен осуществляться через открытые функции-члены. Если такие функции-члены не изменяют данные, они называются *функциями доступа* (accessor functions). Функции доступа необязательно неэффективны — простые функции доступа могут быть встроенными. В классе `stack` функции-члены `top_of()`, `empty()` и `full()` — встроенные функции доступа. Функции доступа должны объявляться как `const`. Функция-член `reset()` является *модификатором* (mutator). Она допускает ограниченный круг действий со скрытой переменной `top`. Обратите внимание, насколько безопаснее такая схема. Если бы `top` была доступна напрямую, легко было бы изменить ее неподходящим образом.

В объектно-ориентированном программировании открытые члены обычно являются функциями и понимаются как *интерфейс* типа. Это — ожидаемые от объекта действия или поведение. Если мы представим себе тип объекта как существительное, его поведение будет выражаться глаголами. При реализации члены данных обычно имеют закрытый доступ. Это ключевой принцип сокрытия данных, а именно: реализация заключена внутрь «черного ящика», который не может прямо эксплуатироваться пользователем объекта.

Резюме

1. Первоначально Страуструп назвал свой язык «C с классами». Классы —это способ реализации типа данных и связанных с ним функций и операторов. В C++ существует особый механизм для реализации АТД, таких как комплексные числа и стеки.

2. Структуры позволяют программисту собрать различные компоненты в переменную с единым именем. Структура включает в себя члены, имеющие собственные имена. Чрезвычайно важным при работе со структурами является доступ к их членам. Он осуществляется либо с помощью оператора доступа к члену «`.`» (точка), либо — оператором доступа к члену через указатель `->`. Эти операторы, наравне с `()` и `[]`, имеют второй приоритет. Самым высоким приоритетом обладает оператор разрешения области видимости `::`.

3. Концепция `struct` расширена в C++ так, что функции могут быть членами структур. Объявление функций включается в объявление структуры; функции вызываются с использованием тех же методов доступа, что и члены структуры. Идея заключается в том, что функциональность, необходимая для работы с типом данных `struct`, должна быть прямо включена в объявление структуры.

4. Функции-члены, которые определяются внутри структуры, являются неявно встраиваемыми. Как правило, только короткие, интенсивно используемые функции-члены должны определяться внутри структуры. Для определения функции-члена вне структуры используется оператор разрешения области видимости.

5. Оператор разрешения области видимости позволяет функциям-членам из разных типов структур иметь одинаковые имена. То, какая именно функция-член вызывается, зависит от типа объекта, на который она направлена. Функции-члены внутри одной структуры могут быть перегружены.

6. В C++ структуры позволяют осуществлять сокрытие данных. Внутри структуры ключевое слово `private` с последующим двоеточием ограничивает доступ к членам, которые идут за этой конструкцией. Закрытые члены используются только функциями-членами структуры.

7. Классы в C++ — это разновидность структуры, у которой по умолчанию установлен закрытый тип доступа. Структуры и классы взаимозаменяемы, но необходимо определять надлежащие права доступа.

8. Члены данных могут объявляться с использованием модификатора класса памяти `static`. Член данных, объявленный как `static`, разделяется всеми переменными данного класса и хранится в одном месте.

9. Классы могут быть вложены. Вложенный класс находится внутри области видимости объемлющего класса. Это не согласуется с семантикой стандартного языка C.

Упражнения

1. Разработайте класс `person`, содержащий члены для хранения имени, возраста, пола, телефона. Напишите функции-члены, которые могут изменять эти члены данных по отдельности. Напишите функцию-член `person::print()`, которая печатала бы красиво отформатированные данные о человеке.

2. В этом упражнении используйте структуру `ch_stack`, описанную в разделе 5.2, «Доступ: закрытый и открытый», на стр. 135. Напишите функцию

```
void reverse(const char s1[], char s2[]);
```

Строки `s1` и `s2` должны быть одинакового размера. Строка `s2` должна превращаться в обращенную копию строки `s1`. Внутри `reverse` используйте `ch_stack` для выполнения обращения.

3. Для типа `ch_stack` из раздела 5.2, «Доступ: закрытый и открытый», на стр. 135 запишите как функции-члены:

```
//Помещение n символов из s1 в ch_stack
void push(int n, const char s1[]);

//Извлечение n символов из ch_stack в символьную строку
void pop(int n, char s1[]);
```

Подсказка: Не забудьте вставить символ конца строки перед тем, как ее выводить.

4. Объясните разницу между структурой:

```
struct a {
  int i, j, k;
};
```

и классом:

```
class a {
  int i, j, k;
};
```

Объясните, почему такое объявление класса не имеет смысла. Как можно использовать ключевое слово `public`, чтобы превратить объявление класса в объявление, эквивалентное структуре `a`?

5. Представьте как класс код `deque` из главы 4, «Реализация АТД в базовом языке», упражнение 10, на стр. 126; `deque` — двусторонняя очередь, допускающая вход и выход с обоих концов.

```
class deque {
public:
  void reset();
.....
private:
  char c[max_len];
  int bottom, top;
};
```

Объявите и реализуйте функции `push_t`, `pop_t`, `push_b`, `pop_b`, `out_stack`, `top_of`, `bottom_of`, `empty` и `full`. Функция `push_t` служит для помещения элемента в начало очереди; функция `push_b` — в конец. Функция `out_stack` должна выводить всю очередь от начала до конца. Функции `pop_t` и `pop_b` отвечают за извлечение элементов из начала и конца очереди соответственно. Ситуация, когда начало очереди опускается ниже конца, означает пустую очередь. Протестируйте каждую функцию.

6. Расширьте тип данных `deque`, добавив функцию-член `relocate`. Если очередь заполнена целиком, вызывается функция `relocate`, и содержимое `deque` перемещается в пустую область памяти, выровненную относительно центра `max_len/2` массива `s`. Прототип этой функции должен выглядеть так:

```
//Возвращает true, если завершается успешно,
//false - если нет

bool deque::relocate()
```

7. Напишите функцию, которая меняет содержимое двух строк (иными словами, переделайте упражнение 12 из главы 4, «Реализация АТД в базовом языке», на стр. 127). Если вы поместите символьную строку в `ch_stack`, а затем извлечете ее, она станет обращенной. Но при обмене строками нам нужен неизменный поря-

док символов. Используйте deque для выполнения такого обмена. Строки будут храниться в символьных массивах одинакового размера, но сами строки могут быть разной длины. Вот прототип этой функции:

```
void swap(char s1[], char s2[]);
```

8. Допишите функции-члены:

```
complex complex::plus(complex a, complex b);
//выполняет бинарное сложение c = plus(a, b)

  complex complex::mpy(complex a, complex b);
//выполняет бинарное умножение c = mpy(a, b)
```

9. Напишите процедуру, которая вычисляет корни квадратного уравнения. Процедура должна иметь следующий прототип:

```
void solve_quadratic(
  double a, //квадратичный член a*x*x
  double b, //линейный член b*x
  double c, //свободный член c
  complex& root1,  //корни могут быть комплексными
  complex& root2
);
```

10. Перепишите класс twod из раздела 5.9, «Контейнеры и доступ к их содержимому», на стр. 147 так, чтобы он мог содержать комплексные значения. Проверьте его, инициализовав этот контейнер набором комплексных чисел, и произведите контрольный вывод. Затем выполните транспонирование и выведите результат.

11. Напишите класс chess_piece (шахматная фигура), который описывал бы положение фигуры на шахматной доске. Шахматная доска имеет вертикали от *a* до *h* и горизонтали от *1* до *8*. Вначале белый король стоит на *e1*, а черный — на *e8*. Проверьте ваш класс, введя начальные позиции для всех 32 шахматных фигур и напечатав содержимое доски.

12. Перепишите вложенный цикл функции transpose() из раздела 5.9, «Контейнеры и доступ к их содержимому», на стр. 149, заменив где это возможно вызовы функции twod::element_lval() на вызов функции twod::element_rval(). Затем перепишите этот цикл, заменив три оператора присваивания на обращение к

```
inline void swap(double&, double&);
```

Проверьте все три версии.

13. Напишите процедуру для умножения матриц, используя переменные twod.

```
//ab = a*b   Обратите внимание: ab должна быть
//правильно размещена в памяти
void mmpy(const twod& a, const twod& b, twod& ab);
```

14. Напишите функцию-член для умножения матриц, используя переменные `twod`.

```
twod twod::mmpy(const twod& a, const twod& b);
```

Проверьте эффективность этого алгоритма по сравнению с предыдущим примером. Отличаются ли они по производительности; если да, то почему? Для сравнения используйте системный профайлер. Если у вас нет профайлера, проведите тесты на время. В любом случае, массивы `twod` должны содержать по крайней мере 100 элементов.

15. *(Проект)* Включите в класс `chess_piece` генератор правильных ходов, который мог бы находить все правильные ходы для заданной фигуры.

Глава 6

Создание и уничтожение объектов

Любому *объекту* требуется память и некоторое начальное значение. В C++ это обеспечивается с помощью объявлений, являющихся одновременно определениями. В большинстве случаев при обсуждении объявлений мы подразумеваем именно такие объявления. Например, в

```
void foo()
{
    int     n = 5;
    double z[10] = {0.0};
    struct gizmo { int i, j; } w = {3, 4};
    .....
}
```

все объекты создаются при входе в блок, когда вызывается `foo()`. В типичных реализациях используется динамический системный стек. Так, объект `n` типа `int` в системе, где целые хранятся в четырех байтах, получит их из стека инициализованными значением 5. Объект `w` типа `gizmo` требует восьми байт для представления своих двух целых членов. Массиву из объектов с двойной точностью `z` необходимо 10 раз по `sizeof(double)` для хранения его элементов. В каждом случае система заботится о создании и инициализации этих объектов. При выходе из `foo()` память освобождается автоматически.

При создании сложных агрегатов пользователь вправе ожидать аналогичного подхода к управлению объектами, определенными классом. Чтобы клиент мог использовать объекты как собственные типы, класс нуждается в механизме создания и уничтожения объектов.

Конструктор (constructor) — это функция-член, имя которой совпадает с именем класса. Он создает значения типа своего класса. Этот процесс включает в себя инициализацию членов данных и (зачастую) распределение свободной памяти с помощью `new`. *Деструктор* (destructor) — это функция-член, имя которой представляет собой имя класса, предваряемое символом ~ (тильда). Обычно деструктор предназначен для уничтожения значений типа класса, как правило, с помощью `delete`.

Из этих двух специальных функций-членов более сложны конструкторы. Они могут быть перегружены, могут принимать аргументы; ни то, ни другое невозможно для деструкторов[1]. Конструктор вызывается: когда связанный с ним тип используется в определении; когда для передачи значения функции применяется вызов по значению; или когда результат, возвращаемый функцией, предполагает создание значения соответствующего типа. Деструкторы вызываются неявно, когда объект выходит за пределы области видимости. Конструкторы и деструкторы не имеют возвращаемого типа и не могут использовать инструкцию `return` *выражение*.

6.1. Классы с конструкторами

Простейшее применение конструктора — использование его при инициализации. В этом и следующих разделах мы создадим несколько примеров, в которых конструкторы используются для инициализации членов данных класса.

Наш первый пример — реализация типа данных `mod_int` для хранения чисел, над которыми производятся вычисления по модулю.

В файле modulo.cpp

```
//Числа для вычислений по модулю и инициализация конструктора
class  mod_int {
public:
   mod_int(int i); //объявление контруктора
   void  assign(int i) { v = i % modulus; }
   void  print() const { cout << v << '\t'; }
   const static int modulus = 60;
private:
   int  v;
};
        //определение конструктора
mod_int::mod_int(int i) { v = i % modulus; }
const int mod_int::modulus;
```

Целая переменная `v` ограничена значениями 0, 1, 2, ..., `modulus` – 1. Ответственность за выполнение этого ограничения лежит на программисте; все функции-члены должны вести себя, соблюдая это правило.

Функция-член `mod_int::mod_int(int)` является конструктором. Она не имеет возвращаемого типа. Она вызывается, когда объявляются объекты типа `mod_int`. Это функция одного аргумента. При вызове конструктор ожидает выражение, преобразуемое по умолчанию к его целому параметру. Затем он создает и инициализует объявленную переменную.

Вот некоторые примеры объявлений с использованием данного типа:

```
mod_int a(0);    //a.v = 0
mod_int b(61);   //a.v = 1
```

[1] К чести деструкторов надо сказать, что они (в отличие от конструкторов) могут быть виртуальными. — *Примеч. перев.*

но не так:

```
mod_int a;        //недопустимо, т.к. нет списка параметров
```

Поскольку у этого класса лишь один конструктор со списком аргументов int, объявление mod_int должно получать целое выражение, передаваемое в качестве инициализующего значения. Заметьте, что не допуская объявления переменной типа mod_int без выражения-инициализатора, мы тем самым предотвращаем ошибки на этапе выполнения из-за неинициализованных переменных.

6.1.1. Конструктор по умолчанию

Конструктор, не требующий аргументов, называется *конструктором по умолчанию*. Это может быть конструктор с пустым списком аргументов, или конструктор, у которого все аргументы имеют значения по умолчанию. Конструктор по умолчанию имеет специальное назначение при инициализации массивов объектов своего класса.

Зачастую удобно перегружать конструктор с помощью нескольких объявлений функций. В нашем примере было бы желательно, чтобы v имела значение по умолчанию 0. Если мы добавим конструктор по умолчанию

```
mod_int() { v = 0; }
```

как функцию-член mod_int, то можно будет сделать следующие объявления:

```
mod_int s1, s2; //оба инициализуют закрытый член v нулем
mod_int d[5];   //массивы правильно инициализованы
```

В обоих объявлениях будет вызван конструктор с пустым списком параметров.

Если у класса нет конструктора, система предоставляет конструктор по умолчанию. Если конструктор у класса есть, но нет конструктора по умолчанию, размещение массива вызовет синтаксическую ошибку.

Обратите внимание, что в нашем примере с классом mod_int один и тот же конструктор может служить и для обычной инициализации, и в качестве конструктора по умолчанию:

```
inline mod_int::mod_int(int i = 0)
  {v = i % modulus; }
```

6.1.2. Инициализатор конструктора

Существует специальный синтаксис для инициализации отдельных частей объекта с помощью конструктора. *Инициализаторы конструктора* (constructor initializer) для членов структуры и класса могут быть заданы через запятую в списке, который идет после двоеточия за списком параметров конструктора и предшествует телу. Давайте перепишем предыдущий пример:

```
//Конструктор по умолчанию для mod_int
mod_int::mod_int(int i = 0) : v(i % modulus) {}
```

Обратите внимание, как инициализация заменила присваивание. Отдельные члены должны быть инициализованы как:

```
имя_члена (список выражений)
```

Не всегда членам можно присвоить значения в теле конструктора. Список инициализаторов необходим, когда нестатический член либо постоянен, либо является ссылкой.

6.1.3. Конструкторы как преобразования

Конструктор с единственным параметром автоматически является функцией преобразования, если только он не объявлен с ключевым словом `explicit`. Рассмотрим следующий класс, назначение которого — напечатать невидимые символы с их представлением ASCII; например, код с восьмеричным номером 07 — это звонок (звуковой сигнал).

В файле printabl.cpp

```
//Печатаемые символы ASCII

class  pr_char {
public:
   pr_char(int i = 0) : c(i % 128) {}
   void print() const { cout << rep[c]; }
private:
   int    c;
   static const char*  rep[128];
};

const char*  pr_char::rep[128] = { "nul", "soh", "stx",
   .....
 "w", "x", "y", "z", "{", "|", "}", "~", "del" };

int main()
{
   pr_char  c;

   for (int i = 0; i < 128; ++i) {
      c = i;    //или: c=static_cast<pr_char>(i)
      c.print();
      cout << endl;
   }
}
```

Конструктор выполняет автоматическое преобразование типа `int` в тип `pr_char`. Отметьте, что инструкция в цикле

```
c = i;
```

как раз и заключает в себе это преобразование. Возможно и явное использование приведений. Более подробно преобразования раскрываются в главе 7, «Ad hoc полиморфизм». В ООП для абстрактных типов данных используются неявные преобразования, поскольку желательно, чтобы АТД выглядели и вели себя так же, как и собственные типы.

6.2. Создание динамического стека

Конструктор можно также использовать для выделения пространства из свободной памяти. Изменим тип ch_stack из раздела 5.2, «Доступ: закрытый и открытый», на стр. 135 так, чтобы его максимальная длина инициализовалась конструктором.

По замыслу объект ch_stack содержит скрытые детали реализации. Члены данных помещены в закрытую часть класса ch_stack. Открытый интерфейс предоставляет клиентам ожидаемую абстракцию — стек. Это открытые функции-члены, такие как push() и pop(). Некоторые из этих функций являются *функциями доступа* (accessor function), которые не изменяют объект стека, такие как top_of() и empty(). Обычно эти функции-члены объявляются как const. Некоторые из обсуждаемых функций являются *модификаторами* (mutator function) — они изменяют объект ch_stack (например, push() и pop()). Функции-члены-конструкторы выполняют работу по созданию и инициализации объектов ch_stack.

В файле ch_stac4.h

```
//Реализация ch_stack с конструктором

class ch_stack {
public:
//открытый интерфейс для АТД ch_stack
   explicit ch_stack(int size) : max_len(size), top(EMPTY)
   { assert(size > 0); s = new char[size]; assert(s != 0);}
   void  reset() { top = EMPTY; }
   void  push(char c) { s[++top] = c; }
   char  pop() { return s[top--]; }
   char  top_of() const { return s[top]; }
   bool  empty() const { return (top == EMPTY); }
   bool  full() const { return (top == max_len - 1); }
private:
   enum  { EMPTY = -1 };
   char*  s;   //было s[max_len]
   int   max_len;
   int   top;
};
```

Теперь клиент, использующий ch_stack, может выбрать требуемый размер. Пример объявления ch_stack с вызовом конструктора выглядит так:

```
ch_stack data(1000);          //размещение 1000 элементов
ch_stack more_data(2 * n);    //размещение 2*n элементов
```

Создадим два дополнительных конструктора. Один — с пустым списком параметров, он будет размещать ch_stack определенной длины. Другой — с двумя параметрами, его второй параметр типа char* будет использоваться для инициализации ch_stack. Их можно записать так:

```
//конструктор по умолчанию для ch_stack
ch_stack::ch_stack() : max_len(100),top(EMPTY)
{
  s = new char[100];
  assert(s != 0);
}

//инициализация стека строкой
ch_stack::ch_stack(int size, const char str[]) : max_len(size)
{
  int i;
  assert(size > 0);
  s = new char[size];
  assert(s != 0);
  for (i = 0; i < max_len && str[i] != 0; ++i)
    s[i] = str[i];
  top = --i;
}
```

Соответствующие прототипы функций должны быть включены как члены в класс `ch_stack`. Используем эти конструкторы следующим образом:

```
ch_stack data;              //создает s[100]
ch_stack d[N];              //N стеков по 100 элементов
ch_stack w(4, "ABCD");      //w.s[0]='A' ... w.s[3]='D'
```

6.2.1. Копирующий конструктор

Рассмотрим наш стек и подсчитаем, сколько раз встречается заданный символ. Можно многократно извлекать элементы из стека, проверяя каждый из них по очереди, до тех пор пока стек не будет пуст. Но что если мы хотим сохранить нетронутым содержимое стека? Вызываемые по значению параметры именно это и выполняют:

В файле ch_stac4.cpp

```
int cnt_char(char c, ch_stack s)
{
  int  count = 0;

  while (!s.empty())
    count += (c == s.pop());
  return count;
}
```

Семантика вызова по значению требует, чтобы была создана локальная копия аргумента, и чтобы она была инициализована значением выражения, переданного в качестве фактического параметра. Для этого необходим *копирующий конструктор* (copy constructor). Компилятор предоставляет копирующий конструктор по умолчанию. Его сигнатура такова:

```
ch_stack::ch_stack(const ch_stack&);
```

Компилятор выполняет копирование с помощью *почленной инициализации* (memberwise initialization). Она может не работать при некоторых обстоятельствах для сложных агрегатов с членами, являющимися указателями. В таких случаях указатель может оказаться связанным с объектом, который уже автоматически удален, так как вышел за пределы области видимости. Процесс дублирования значения указателя, а не объекта, на который он указывает, может привести к аномалиям. Удаление объекта повлияет на остальные экземпляры, которые все еще полагают, что объект существует. Для классов желательно явно определять копирующий конструктор.

В файле ch_stac4.cpp

```
//Копирующий конструктор для ch_stack из символов
ch_stack::ch_stack(const ch_stack& str) :
  max_len(str.max_len), top(str.top)
{
  s = new char[str.max_len];
  assert(s != 0);
  memcpy(s, str.s, max_len);
}
```

Процедура `memcpy()` из *stdlib.h* копирует `max_len` символов начиная с базового адреса `str.s` в память, начинающуюся с базового адреса `s`. Это называется *глубокой копией* (deep copy). Символьные массивы представляют собой особый случай, потому что они ссылаются на различные области памяти. Если бы вместо приведенного выше тела процедуры было

```
s = str.s;
```

то результатом была бы *поверхностная копия* (shallow copy); при этом переменные `ch_stack` разделяли бы общее представление.[1] Любое изменение одной переменной привело бы к изменению других.

6.3. Классы с деструкторами

Деструктор — это функция-член, имя которой представляет собой начинающееся с тильды имя класса. Почти всегда он вызывается неявно — при выходе из блока, в котором был объявлен объект. Деструктор также вызывается, когда оператор `delete` применяется к указателю на имеющий деструктор объект, или когда он необходим для уничтожения подобъекта удаляемого объекта.

Давайте добавим в наш пример `ch_stack` деструктор.

В файле ch_stac4.h

```
//ch_stack с конструктором и деструктором
class ch_stack {
public:
  ch_stack(); //конструктор по умолчанию
```

[1] То есть при глубоком копировании массива копируется сам массив, а при поверхностном — указатель на него (базовый адрес). — *Примеч. перев.*

```
    explicit ch_stack(int size) : max_len(size), top(EMPTY)
    { assert(size > 0); s = new char[size];
      assert(s != 0);}
    ch_stack(const ch_stack& str);   //копирующий конструктор
    ch_stack(int size, const char str[]);
    ~ch_stack() { delete [] s; }     //деструктор
    .....
  private:
    enum  { EMPTY = -1 };
    char*  s;
    int  max_len;
    int  top;
  };
```

Добавление деструктора позволяет классу вернуть во время выполнения программы ненужную более память, выделенную из кучи. Все открытые функции-члены работают совершенно также, как раньше. Отличие состоит в том, что при выходе из блока или функции для очистки памяти будет неявно вызван деструктор. Это хорошая программистская практика, благодаря которой приложение требует меньше памяти.

6.4. Пример: динамически размещаемые строки

Языку C++ недостает собственного строкового типа. Стандартная библиотека предоставляет шаблонный строковый класс — тип, который используется все чаще и чаще. Старое строковое представление выглядело как указатель на `char`. В этом представлении конец строки обозначался нулевым символом `'\0'`. Такое соглашение имеет существенный недостаток: затраты на многие основные строковые манипуляции пропорциональны длине строки. Использование строк с нулевым завершающим символом отражено в библиотеке *string.h* (или *cstring* в современном C++). В этой библиотеке для вычисления длины символьного массива, ограниченного нулевым символом, используется стандартная функция `int strlen(const char*)`. В современном C++ стандартная библиотека *string* предоставляет строковый тип, который хранит длину строки как часть его скрытой реализации.

В этом разделе мы разработаем несколько способов, которыми такой тип может быть реализован, а также полезный строковый АТД, в котором длина объявлена как `private`. Мы хотим, чтобы тип размещался динамически и мог представлять строки произвольной длины. Для инициализации и размещения строк будут написаны различные конструкторы, а набор строковых операций будет запрограммирован как функции-члены. Реализация будет использовать строковые функции из библиотеки *string.h* для оперирования вышеупомянутым представлением строк с помощью указателей.

В файле string5.cpp

```
//Реализация динамически распределяемых строк

class my_string {
public:
```

```
    my_string() : len(0)
      { s = new char[1]; assert(s != 0); s[0] = 0; }
    my_string(const my_string& str);   //копирующий конструктор
    my_string(const char* p);          //преобразующий конструктор
    ~my_string() { delete [] s; }
    void assign(const my_string& str);
    void print() const { cout << s << endl; }
    void concat(const my_string& a, const my_string& b);
  private:
    char*  s;
    int  len;
  };

  my_string::my_string(const my_string& str) : len(str.len)
  {
    s = new char[len + 1];
    assert(s != 0);
    strcpy(s, str.s);
  }

  void my_string::assign(const my_string& str)
  {
    if (this == &str)  //a == a; ничего не делаем
      return;
    delete [] s;
    len = str.len;
    s = new char[len + 1];
    assert(s != 0);
    strcpy(s, str.s);
  }

  void my_string::concat(const my_string& a,
                         const my_string& b)
  {
    char* temp = new char[a.len + b.len + 1];

    len = a.len + b.len;
    strcpy(temp, a.s);
    strcat(temp, b.s);
    delete [] s;
    s = new char[len + 1];
    assert(s != 0);
    strcpy(s, temp);
  }
```

Этот тип позволяет объявлять строки `my_string`, присваивать с помощью копирования одну `my_string` другой, печатать строки и объединять две строки типа `my_string`. Скрытое представление является указателем на `char` и содержит переменную `len`, в которой хранится текущая длина строки типа `my_string`.

Разбор класса my_string

- ```
  my_string() : len(0)
     { s = new char[1]; assert(s != 0); s[0] = 0; }
  my_string(const my_string& str); //копирующий конструктор
  my_string(const char* p);        //преобразующий конструктор
  ```

Это три перегруженных конструктора. Первый является конструктором по умолчанию, необходимым для объявления массива строк `my_string`. Второй — копирующий конструктор. У третьего конструктора есть аргумент — указатель на `char`, который можно использовать для преобразования `char*`-представления строк к нашему типу `my_string`. Этот конструктор содержит две библиотечные функции: `strlen` и `strcpy`. Мы размещаем один дополнительный символ для хранения символа конца строки `\0`, хотя он и не подсчитывается функцией `strlen`. Копирующий конструктор рассмотрим ниже.

- ```
  ~my_string() { delete [] s; }
  ```

Деструктор автоматически освобождает память, занятую `my_string`, для повторного использования. Используется форма `delete` с пустыми квадратными скобками, так как было выполнено размещение массива. Оператору `delete[]` известно количество памяти, связанное с указателем `s`.

- ```
  my_string::my_string(const my_string& str) : len(str.len)
  {
    s = new char[len + 1];
    assert(s != 0);
    strcpy(s, str.s);
  }
  ```

Это копирующий конструктор. Такая форма конструктора используется для копирования одной `my_string` в другую.

Использование копирующего конструктора

1. `my_string` инициализуется другой `my_string`.
2. `my_string` передается (по значению) функции в качестве аргумента.
3. `my_string` возвращается в качестве значения функции.

В C++, если конструктор не присутствует явно, компилятор его создает, при этом конструктор инициализует один член за другим.

- ```
  void my_string::assign(const my_string& str)
  {
    if (this == &str)  //a == a; ничего не делаем
      return;
    delete [] s;
    len = str.len;
    s = new char[len + 1];
    assert(s != 0);
    strcpy(s, str.s);
  }
  ```

Семантика присваивания основана на семантике глубокого копирования. Напомним, что при глубоком копировании должен воспроизводиться весь агрегат и все значения данных должны копироваться в их представление. Такое копирование нуждается в проверке: не копируем ли мы в ту же my_string (то есть не копируем ли мы строку в саму себя). Именно это имеет место, когда a == a. Если бы мы не выполнили такую проверку и удалили бы левый аргумент[1], значение a исчезло бы. Каждый раз, когда копируется значение my_string, оно физически дублируется с помощью strcpy(). Это отличается от последующей реализации, в которой мы покажем, как использовать семантику поверхностного копирования. Напомним, что поверхностное копирование устанавливает указатель на существующее значение без воспроизводства агрегата. Как мы увидим, оно может быть очень эффективно.

```
void my_string::concat(const my_string& a, const my_string& b)
{
  char* temp = new char[a.len + b.len + 1];

  len = a.len + b.len;
  strcpy(temp, a.s);
  strcat(temp, b.s);
  delete [] s;
  s = new char[len + 1];
  assert(s != 0);
  strcpy(s, temp);
}
```

Это форма конкатенации. Ни один из аргументов типа my_string не изменяется. Неявный аргумент, скрытыми членами которого являются переменные s и len, модифицируется и, в результате, представляет конкатенацию заданных строк: за my_string a следует my_string b. Заметьте, что в этой функции-члене возможно использование len, a.len и b.len. Функции-члены имеют доступ не только к закрытым членам неявного аргумента, но и к закрытому представлению любого из аргументов типа my_string.

Следующая программа тестирует класс my_string, выполняя конкатенацию нескольких строк типа my_string.

В файле string5.cpp

```
int main()
{
  char* str = "Громче всех скрипит то колесо,\n";
  my_string  a(str), b, author("Джош Биллингс\n"),
        both, quote;
```

---

[1] `my_string s1, s2("Строка 2");`
`s1.assign(s2);`
Здесь s1 считается левым аргументом операции assign(), а s2 — ее правым аргументом. — *Примеч. перев.*

```
    b.assign("которое смазывают\n");
    both.concat(a, b);
    quote.concat(both, author);
    quote.print();
}
```

Эта программа распечатает следующее:

```
Громче всех скрипит то колесо,
которое смазывают
Джош Биллингс
```

Мы намеренно использовали несколько объявлений, чтобы показать, как будут вызываться разные конструкторы. Переменные `my_string` b, both и `quote` используют конструктор по умолчанию. Объявление для `author` использует конструктор с аргументом типа `char*`. Конкатенация выполняется в два этапа. Сначала `my_string` строки a и b объединяются в both. Потом выполняется конкатенация строк both и `author`. Затем печатается вся цитата.

Конструктор `my_string::my_string(const char*)` вызывается для создания и инициализации объектов a и `author`. Этот же конструктор вызывается неявно как операция преобразования при вызове `my_string::assign()` для литерала `"которое смазывают\n"`.

6.5. Класс vect

В C++ одномерный массив — очень полезный и эффективный агрегатный тип данных. Во многих отношениях он может рассматриваться как своеобразный контейнер-прототип, простой в использовании и высокоэффективный. Однако он не защищен от ошибок. Обычная ошибка — доступ к элементам за границами массива. C++ позволяет контролировать эту проблему, определяя аналогичный контейнерный тип, в котором границы можно проверять:

В файле vect1.h

```
//Реализация типа безопасного массива vect

class vect {
public:
   explicit vect(int n = 10);
   ~vect() { delete [] p; }
   int&  element(int i);     //доступ к p[i]
   int ub() const {return (size - 1);}//верхняя граница
private:
   int*  p;
   int  size;
};

vect::vect(int n) : size(n)
{
   assert(n > 0);
```

```
    p = new int[size];
    assert(p != 0);
}

int& vect::element(int i)
{
    assert (i >= 0 && i < size);
    return p[i];
}
```

Конструктор `vect::vect(int n)` позволяет пользователю строить динамически размещаемые массивы. Такие массивы значительно более гибки, чем массивы в языках Pascal или C, где размеры должны быть постоянными выражениями. Конструктор также инициализует переменную `size`, значение которой — число элементов массива. Заметьте, что мы объявили этот конструктор одного аргумента как `explicit`, поскольку он не предназначен для неявного преобразования из `int` в `vect`.

Доступ к отдельным элементам осуществляется с помощью безопасной индексирующей функции-члена

```
int& vect::element(int i)
```

Индекс, находящийся за ожидаемыми пределами массива от 0 до `ub`, вызовет провал проверки утверждения. Эта безопасная индексирующая функция-член возвращает ссылку на `int`, которая является адресом `p[i]` и может быть использована в качестве левого операнда в присваивании (в качестве lvalue). Подобная техника часто применяется в C++ и служит эффективным механизмом при оперировании сложными типами.

Например, объявления

```
vect a(10), b(5);
```

создадут массивы из десяти и пяти целых соответственно. Доступ к отдельным элементам может быть получен с помощью функции-члена `element`, которая проверяет, не вышел ли индекс за границы. Все следующие инструкции

```
a.element(1) = 5;
b.element(1) = a.element(1) + 7;
cout << a.element(1) - 2;
```

допустимы. В результате мы получили безопасный динамический тип массива.

Классы с конструкторами по умолчанию используют их для инициализации более сложных массивов. Например, инструкция

```
vect a[5];
```

является объявлением, в котором конструктор по умолчанию применяется для создания массива `a` из пяти объектов, каждый из которых имеет тип `vect` и размер 10. Адрес *i*-го элемента в *j*-ом массиве можно задать с помощью выражения `a[j].element(i)`.

В главе 11, «Исключения» мы обсудим, как можно использовать исключения для проверки на возникновение ошибок. С помощью такой более мощной методологии строка

```
assert(n > 0);
```

заменяется на

```
if (n < 1)
   throw(vect_allocation_error(n));
```

6.6. Члены данных, являющиеся классами

В данном разделе мы будем использовать тип vect в качестве члена класса pair_vect. В объектно-ориентированной методологии это известно как отношение *включать как часть* (HASA relationship)[1]. Сложные объекты могут быть созданы из более простых путем объединения их отношениями *включать как часть*.

В файле pairvect.cpp

```
#include "vect1.h"

class pair_vect {
public:
   pair_vect(int i) : a(i), b(i), size(i){ }
   int& first_element(int i);
   int& second_element(int i);
   int ub()const {return size - 1;}
private:
   vect a, b;
   int size;
};

int& pair_vect::first_element(int i)
{ return a.element(i); }
int& pair_vect::second_element(int i)
{ return b.element(i); }
```

Заметьте, что конструктор pair_vect сводится к вызову трех инициализаторов. Инициализаторы vect-членов a и b вызывают vect:vect(int). Используем этот тип данных для построения таблицы соответствия возраста и веса.

```
int main()
{
   int i;
   pair_vect age_weight(5);  //возраст и вес

   cout << "таблица возраста и веса\n";
   for (i = 0; i <= age_weight.ub(); ++i) {
      age_weight.first_element(i) = 21 + i;
      age_weight.second_element(i) = 135 + i;
```

[1] Отношение *HASA* (иногда *has-a*) — отношение «включать как часть». То есть один объект является частью другого. Это значит, что объявление одного класса включает член данных, тип которого — другой класс. — *Примеч. перев.*

```
        cout << age_weight.first_element(i) << ", "
             << age_weight.second_element(i) << endl;
    }
}
```

6.7. Пример: односвязный список

В этом разделе мы разработаем тип данных для односвязного списка. Он является прототипом для многих динамических АТД, которые называются *ссылающимися на себя структурами* (self-referential structure). Такие типы данных содержат члены-указатели, которые ссылаются на объекты своего собственного типа. Они служат основой для многих полезных контейнерных классов.

Следующее объявление реализует подобный тип:

В файле slist.cpp

```
class slist {    //односвязный список
public:
   slist() : h(0) { } //0 означает пустой список
   ~slist() { release(); }
   void  prepend(char c); //добавление в голову списка
   void  del();
   slistelem*  first() const { return h; }
   void  print() const;
   void  release();
private:
   slistelem*  h;   //голова списка
};

struct slistelem {
   char data;
   slistelem* next;
};
```

Операции над списком

1. `prepend`: добавляет в голову списка
2. `first`: возвращает указатель на первый элемент
3. `print`: печатает содержимое списка
4. `del`: удаляет первый элемент
5. `release`: уничтожает список

Связующий член `next` указывает на следующий `slistelem` в списке. Переменная `data` в этом примере является простой переменной, но она может быть заменена на сложный тип, способный хранить целую порцию информации. Конструктор инициализует голову списка `slist` (указатель `h`) значением 0, которое называется *постоянной нулевого указателя* (null pointer constant) и может быть присвоено указа-

телю любого типа. В связных списках она обычно обозначает пустой список или значение «конец списка». Функция-член `prepend()` строит структуру списка:

```
void slist::prepend(char c)
{
   slistelem*  temp = new slistelem;  //создание элемента
   assert(temp != 0);
   temp -> next = h;     //связь с slist
   temp -> data = c;
   h = temp;      //изменение головы списка
}
```

Новый элемент списка размещается в свободной памяти, а его член данных инициализуется единственным аргументом `c`. Его связующий член `next` устанавливается на прежнюю голову списка. Затем голова (указатель `h`) изменяется так, чтобы указывать на только что созданный элемент как на новый первый элемент списка.

Функция-член `del()` выполняет обратную задачу:

```
void slist::del()
{
   slistelem*  temp = h;

   h = h -> next;//предполагается не пустой slist
   delete temp;
}
```

Она освобождает память, занимаемую первым элементом списка. Это делается с помощью оператора `delete`, направленного на голову `slist` (указатель `h`). Новая голова списка — это значение члена `next` прежней головы. Эту функцию можно модифицировать так, чтобы она работала и с пустым списком, не вызывая прерывания программы (см. упражнение 18 на стр. 188).

Обработка списков в основном связана с последовательным проходом по списку, пока не найдено значение нулевого указателя. Две следующие функции используют эту технику.

В файле slist.cpp

```
void slist::print() const //объект не изменяется
{
   slistelem*  temp = h;

   while (temp != 0) {     //обнаружение конца списка
      cout << temp -> data << " -> ";
      temp = temp -> next;
   }
   cout << "\n###" << endl;
}

//освобождение памяти, занимаемой элементами
void slist::release()
```

```
{
  while (h != 0)
    del();
}
```

Разбор функций print() и release()

- `void slist::print() const //объект не изменяется`

```
{
  slistelem*  temp = h;
```

Вспомогательный указатель `temp` будет обеспечивать прохождение по списку. Он инициализуется адресом головы `h` списка `slist`. Указатель `h` нельзя использовать, потому что его значение будет потеряно, в результате чего будет потерян доступ к списку.

- `while (temp != 0) {  //обнаружение конца списка`

```
  cout << temp -> data << " -> ";
  temp = temp -> next;
}
```

Значение 0 отвечает за представление конца списка, поскольку конструктор `slist::slist()` инициализовал его именно этим значением, а функция `slist::prepend()` обрабатывает его как значения указателя конца списка. Обратите внимание, что содержимое этого цикла можно изменить с тем, чтобы обрабатывать весь список каким-нибудь другим образом.

- `void slist::release()`

Функция `release()` используется для освобождения памяти, занимаемой всеми элементами списка. Она проходит по списку, выполняя удаление.

- `while (h != 0)`

```
  del();
```

Память, занимаемая каждым элементом списка, должна освобождаться последовательно. Для одного элемента это делается с помощью `slist::del()`, которая манипулирует указателем `h`. Поскольку список уничтожается, нам не нужно сохранять начальное значение указателя `h`. Основное применение этой функции — в теле деструктора `slist::~slist()`. Деструктор, записанный в виде

```
slist::~slist()
{
  delete h;
}
```

использовать нельзя, поскольку он удаляет только первый элемент списка.

Продемонстрируем использование этого типа в следующей программе, в которой деструктор изменен и выводит сообщение:

В файле slist.cpp

```
slist:: ~slist()
{
   cout << "вызван деструктор" << endl;
   release();
}

int main()
{
   slist*  p;
   {
      slist  w;

      w.prepend('A');
      w.prepend('B');
      w.print();
      w.del();
      w.print();
      p = &w;
      p -> print();
      cout << "выход из внутреннего блока" << endl;
   }
   //p -> print();  ведет себя в зависимости от системы
   cout << "выход из внешнего блока" << endl;
}
```

Обратите внимание, что в `main` есть внутренний блок. Этот блок вставлен для проверки того, что деструктор вызывается при выходе из блока, освобождая память, связанную с `w`. Вот что выведет эта программа:

```
B -> A ->
###
A ->
###
A ->
###
выход из внутреннего блока
вызван деструктор
выход из внешнего блока
```

Первый вызов `print()` выведет двухэлементный список `slist`, содержащий `B` и `A`. После выполнения операции `del` список содержит один элемент, хранящий `A`. Находящемуся во внешнем блоке указателю на `slist` `p` присваивается адрес переменной `slist` `w`. Если к списку обращаются через `p` из внутреннего блока, печатается `A`. Этот вывод показывает, что деструктор удаляет переменную `w` при выходе из блока.

Второй вызов функции `slist::print()` закомментирован, поскольку результат зависит от системы. Разыменование `p` здесь вызовет ошибку этапа выполне-

ния, потому что объект по адресу, на который он ссылается, разрушен процедурой удаления.

6.8. Двумерные массивы

В стандартном C нет подлинных многомерных массивов. Вместо этого программист должен проявлять осторожность при представлении подобной абстрактной структуры данных указателем на указатель, на . . ., на базовый тип. В C++ программист может реализовывать гибкие, безопасные динамические многомерные массивы. Продемонстрируем это, разработав тип двумерного динамического массива `matrix` (матрица). Обратите внимание на его сходство с классом `vect`.

В файле matrix1.cpp

```
//Безопасный тип двумерного массива matrix

class matrix {
public:
   matrix(int d1, int d2);
   ~matrix();
   int  ub1() const { return(s1 - 1); }
   int  ub2() const { return(s2 - 1); }
   int&  element(int i, int j);
private:
   int**  p;
   int   s1, s2;
};
```

Тип `matrix` имеет по каждому измерению размер и соответствующую верхнюю границу, доступ к которой открыт. Скрытое представление использует указатель на указатель на тип `int`. Здесь будет храниться базовый адрес массива указателей на `int`, в которых в свою очередь содержатся базовые адреса для каждой строки.

```
matrix::matrix(int d1, int d2) : s1(d1), s2(d2)
{
   assert(d1 > 0 && d2 > 0);
   p = new int*[s1];
   assert(p != 0);
   for (int i = 0; i < s1; ++i){
      p[i] = new int[s2];
      assert(p[i] != 0);
   }
}
matrix::~matrix()
{
   for (int i = 0; i <= ub1(); ++i)
      delete p[i];
```

```
    delete [] p;
}
```

Так же как и в типе twod из раздела 5.9, «Контейнеры и доступ к их содержимому», на стр. 147, конструктор размещает массив указателей на int. Числом элементов этого массива служит значение s1. Затем конструктор итеративно размещает массив целых, на которые указывает каждый из элементов p[i]. Таким образом, из свободной памяти выделяется пространство для s1*s2 целых, и плюс к тому, пространство для s1 указателей. Деструктор освобождает память в обратном порядке. Подобная схема распространяется и на массивы с более высокой размерностью.

Для получения lvalue элемента в этом двумерном массиве необходимы два индексных аргумента:

```
int& matrix::element(int i, int j)
{
   assert(i >= 0 && i <= ub1() && j >= 0 && j <= ub2());
   return p[i][j];
}
```

Оба аргумента проверяются на то, находятся ли они в пределах границ. Это — обобщение случая с одним индексом.

6.9. Многочлен как связный список

Многочлен (полином) называется *разбросанным* (sparse), если он содержит относительно мало (по сравнению со своей степенью) ненулевых коэффициентов. Степень многочлена — это просто самая высокая степень ненулевого члена. Например, многочлен тысячной степени $P(x)=x^{1000}+x^1+1$ имеет только три ненулевых члена. При манипуляциях с большими разбросанными многочленами часто бывает полезно основывать их представление на связном списке. В таком представлении каждый элемент списка содержит ненулевой член многочлена.

Напишем процедуру, обрабатывающую многочлены и выполняющую их сложение; процедура допускает лишь по одному члену каждой степени. Список отсортирован в порядке убывания степеней членов многочлена.

В файле poly1.cpp

```
//Полином, представленный как односвязный список

struct term {              //член полинома
   int     exponent;       //степень
   double coefficient;    //коэффициент
   term*  next;
   term(int e, double c, term* n = 0)
      : exponent(e), coefficient(c), next(n) { }
   void print()
     {cout << coefficient << "x^" << exponent << " ";}
};
```

```
class polynomial {
public:
   polynomial() : h(0), degree(0) { }
   polynomial(const polynomial& p);
   polynomial(int size, double coef[], int expon[]);
   ~polynomial() { release(); }
   void print() const;
   void plus(polynomial a, polynomial b);
private:
   term* h;
   int  degree;
   void prepend(term*  t);      //добавление члена в начало
   void add_term(term*& a, term*& b);
   void release();              //сборка мусора
   void rest_of(term* rest);    //добавление остальных членов
   void reverse();              //обращение членов
};
```

В этом представлении полином кодируется как список членов. Каждый член является парой коэффициент-степень. Полином будет содержать свои члены в виде списка в порядке убывания степени. Такая каноническая форма упрощает сложение и другие операции. Полином будет пустым, инициализованным копирующим конструктором, или составленным из двух массивов, содержащих надлежащим образом отсортированную последовательность пар коэффициент-степень.

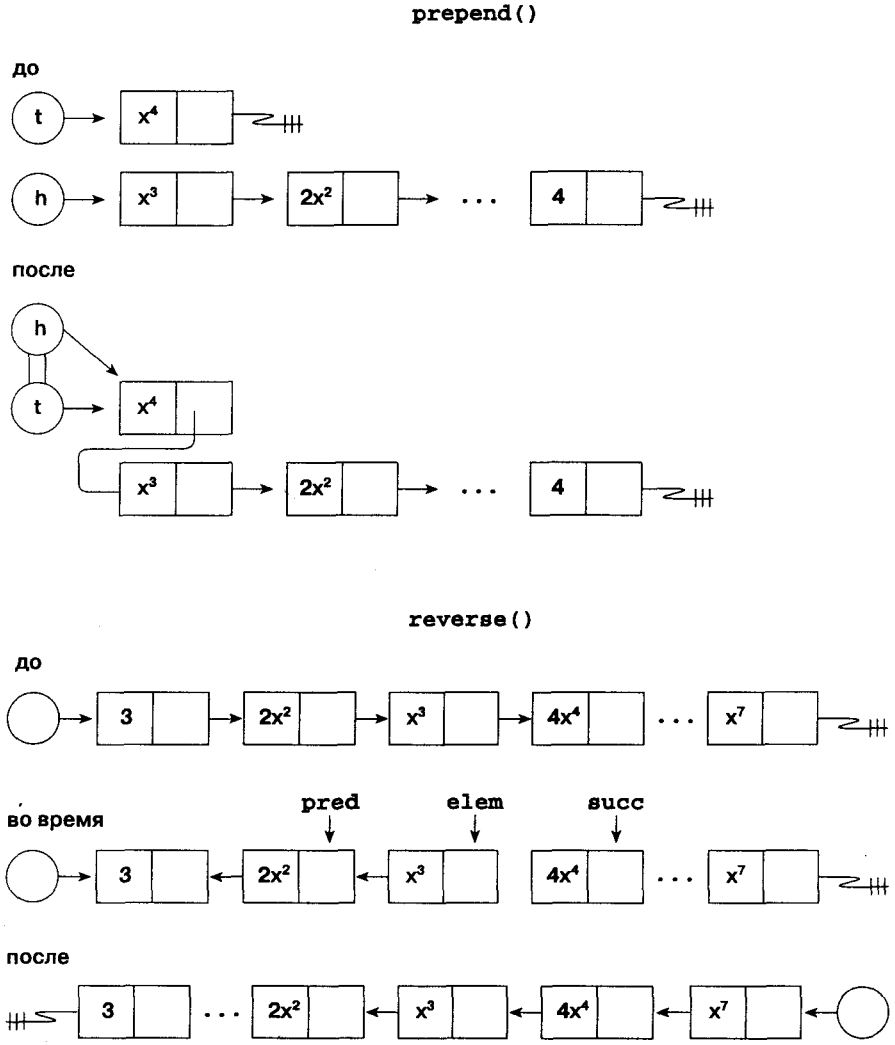
Есть несколько важных функций-членов, оперирующих приведенным ниже представлением списка. Функция `prepend()` связывает член с головой списка. Функция `reverse()` обращает список «на месте». Функция `add_term()` используется функцией `plus()` для добавления следующего члена и соответствующим образом продвигает указатели в двух складываемых полиномах.

```
inline void polynomial::prepend(term* t)
{ t -> next = h;  h = t; }

void polynomial::reverse()  //на месте
{
   term *pred, *succ, *elem;

   if (h && (succ = h -> next)) {
      pred = 0;
      elem = h;
      while (succ) {
         elem -> next = pred;
         pred = elem;
         elem = succ;
         succ = succ -> next;
      }
      h = elem;
      h -> next = pred;
   }
}
```

На следующем рисунке показано графическое представление операций `prepend` и `reverse`.



Конструкторы строят явный список для каждого полинома. Было бы неверно полагаться на генерируемый компилятором копирующий конструктор по умолчанию.

```
//Предполагается правильный порядок: expon[i] < expon[i+1]
polynomial::polynomial(int size, double coef[], int expon[])
{
   term* temp = new term(expon[0], coef[0]);
   assert(temp != 0);
   h = 0;
```

```
    prepend(temp);          //создание начального члена
    for (int i = 1; i < size; ++i) {
        assert(expon[i - 1] < expon[i]);
        temp = new term(expon[i], coef[i]);
        assert(temp != 0);
        prepend(temp);      //добавление члена
    }
    degree = h -> exponent;
}

polynomial::polynomial(const polynomial& p) : degree(p.degree)
{
    term* elem = p.h, *temp;

    h = 0;
    while (elem) {          //почленное копирование
      temp = new term(elem -> exponent, elem -> coefficient);
      assert(temp != 0);
      prepend(temp);
      elem = elem -> next;
    }
    reverse();
}
```

Следующий набор функций реализует сложение многочленов методом сортировки со слиянием.

```
void polynomial::add_term(term*& a, term*& b)
{
    term*  c;

    if (a -> exponent > b -> exponent) {      //добавление a
      c = new term(a -> exponent, a -> coefficient) ;
      assert(c != 0);
      a = a -> next;
      prepend(c);
    }
    else if (a -> exponent < b -> exponent){  //добавление b
      c = new term(b -> exponent, b -> coefficient) ;
      assert(c != 0);
      b = b -> next;
      prepend(c);
    }
     else {        //проверка на отмену
        if (a -> coefficient + b -> coefficient != 0) {
           c = new term( a -> exponent,
                        a -> coefficient + b -> coefficient);
           assert(c != 0);
           prepend(c);
        }
```

```
            a = a -> next;
            b = b -> next;
        }
    }
```

Здесь объединяются члены в головах двух списков. Степени могут иметь разные или одинаковые значения. Если степени разные, результатом является наибольший член, и продвигается указатель только его списка. Если степени совпадают, продвигаются указатели обоих списков. Отмена происходит когда обе степени одинаковы и сумма их коэффициентов равна нулю; тогда никакой член не добавляется. В противном случае количество нулевых членов быстро разрослось бы, что свело бы на нет нашу попытку создать эффективное представление разбросанного многочлена.

Когда один список членов исчерпан слиянием, члены из оставшегося списка добавляются в начало итогового списка с помощью `rest_of()`.

```
void polynomial::rest_of(term* rest)
{
    term* temp;

    while (rest) {
        temp  = new term(rest -> exponent, rest -> coefficient);
        assert(temp != 0);
        prepend(temp);
        rest = rest -> next;
    }
}

//c.plus(a,b) означает c = a + b;
void polynomial::plus(polynomial a, polynomial b)
{
    term *aterm = a.h, *bterm = b.h;

    release();                          //сборка мусора c,
                                        //но не a или b
    h = 0;
    while (aterm && bterm)              //шаг слияния
        add_term(aterm, bterm);
    if (aterm)
        rest_of(aterm);
    else if (bterm)
        rest_of(bterm);
    reverse();
    degree = ((h) ? h -> exponent: 0);
}
```

Функция `polynomial::plus()` использует `add_term()` и `rest_of()` для перестановки членов в обратном порядке (по сравнению с ожидаемым представлением), а также функцию `reverse()` для исправления порядка. Для тестирования необходимы функции `print()` и `release()` (см. упражнение 21 на стр. 189).

6.10. Строки, использующие семантику ссылок

Размещение больших агрегатов на этапе выполнения может легко исчерпать ресурсы памяти. В нашем примере списка из раздела 6.7, «Пример: односвязный список», на стр. 170, показана одна схема для разрешения данной проблемы: системе возвращается память путем прохождения по всему списку и уничтожения каждого его элемента. Эта модель восстановления памяти является формой *сборки мусора* (garbage colletction). В таких языках как LISP и Smalltalk, сама система отвечает за подобное восстановление. Эти системы периодически выполняют сборку мусора, задача которой — определить все ячейки памяти, доступные в настоящий момент, и очистить те ячейки, которые недоступны. Большинство подобных схем нуждаются в отслеживании и пометке ячеек, доступных через указатели, с помощью ресурсоемких вычислений.

Схема очистки, которая избегает этого, называется *подсчетом ссылок* (reference counting). В ней каждый динамически размещаемый объект отслеживает свои активные ссылки. При создании объекта его счетчик ссылок устанавливается в единицу. Каждый раз, когда на объект ссылаются снова, его счетчик ссылок увеличивается; при каждой потере ссылки счетчик уменьшается. Когда счетчик ссылок становится равен нулю, память, занимаемая объектом, очищается.

В следующем примере мы создадим класс `my_string`, имеющий ссылочную семантику копирования. Этот класс использует поверхностное копирование, поскольку копирование заменено присваиванием указателей. Показанная техника обычна для агрегатов данных подобного типа. Будем использовать класс `str_obj` для создания значений фактических объектов. Тип `str_obj` является необходимой деталью реализации `my_string`. Данная деталь не может быть помещена непосредственно в `my_string` без разрушения отношения (возможно, многие-к-одному) между объектами типа `my_string` и ссылочными значениями типа `str_obj`. Значения `my_string` содержатся в `str_obj`, который является дополнительным классом только лишь для использования классом `my_string`. Открыто используемый класс `my_string` управляет экземплярами `str_obj`, и иногда называется *управляющим классом* (handler class).

В файле string6.cpp

```
//Строки my_string с подсчетом ссылок

#include  <string.h>
#include  <iostream.h>
#include  <assert.h>

class str_obj {
public:
   int  len, ref_cnt;
   char*  s;
   str_obj() : len(0), ref_cnt(1)
     { s = new char[1]; assert(s != 0); s[0] = 0; }
   str_obj(const char* p) : ref_cnt(1)
     { len = strlen(p); s = new char[len + 1];
       assert(s != 0); strcpy(s, p); }
```

```
    ~str_obj() { delete []s; }
};
```

Класс `str_obj` объявляет объекты, которые используются `my_string`. Позднее мы объясним, как можно закрыть доступ к ним и получать его с помощью механизма `friend` (см. раздел 7.3, «Дружественные функции», на стр. 195). Обратите внимание, что класс `str_obj` применяется в основном для создания и уничтожения объектов, использующих свободную память. При создании `str_obj` переменная `ref_cnt` инициализуется единицей.

```
class my_string {
public:
   my_string() { st = new str_obj; assert(st != 0);}
   my_string(const char* p)
      { st = new str_obj(p); assert(st != 0);}
   my_string(const my_string& str)
      { st = str.st; st -> ref_cnt++; }
   ~my_string();
   void  assign(const my_string& str);
   void  print() const { cout << st -> s; }
private:
   str_obj*  st;
};
```

Клиент будет использовать объекты типа `my_string`. Эти объекты реализованы как указатели `st` на значения типа `str_obj`. Обратите внимание на копирующий конструктор этого класса и на то, как он применяет семантику ссылок для создания копии.

Семантика функции `assign()` демонстрирует некоторые тонкости использования счетчика ссылок:

```
void my_string::assign(const my_string& str)
{
   if (str.st != st) {
      if (--st -> ref_cnt == 0)
         delete st;
      st = str.st;
      st -> ref_cnt++;
   }
}
```

Присваивание происходит, если `my_string` не пытаются присвоить самому себе. Присваивание значения переменной влечет за собой потерю ее предыдущего значения. Это равнозначно уменьшению счетчика ссылок указываемого `str_obj`. Каждый раз, когда счетчик ссылок объекта уменьшается, он проверяется на предмет удаления.

Преимущество такого подхода перед обычным копированием очевидно. Очень большие агрегаты копируются по ссылке с помощью всего нескольких операций.

Для хранения счетчика ссылок требуется небольшое количество дополнительной памяти. Коме того, каждое возможное изменение указателя влечет операцию со счетчиком ссылок. Деструктор также должен проверять счетчик ссылок перед фактическим удалением.

```
my_string:: ~my_string()
{
   if (--st -> ref_cnt == 0)
     delete st;
}
```

6.11. В отсутствии конструктора, копирующий конструктор и другие тайны

Создание объектов собственных типов обычно является задачей компилятора. Автор класса желает достичь такой же легкости при использовании определяемых им АТД. Давайте вернемся к некоторым связанным с этим вопросам.

Каждый ли класс нуждается в явно определенном конструкторе? Конечно, нет. Там где никакой конструктор не был написан программистом, компилятор предоставляет конструктор *по умолчанию, если это необходимо:*

В файле tracking.cpp

```
//Отслеживание личных данных

struct pers_data {
   int  age;        //возраст в годах
   int  weight;     //вес в килограммах
   int  height;     //рост в сантиметрах
   char name[20];   //фамилия
};

void print(pers_data d)
{
   cout << "Возраст " << d.name << "составляет "<< d.age
        << " лет(год,года)\n";
   cout << "Вес: " << d.weight << "кг,  рост: "
        << d.height << "см" << endl;
}

int main()
{
   pers_data  laura = {3, 14, 88, "POHL"};
                  //создание в стеке

   print(laura); //вызывает копирующий конструктор
}
```

Что если мы применяем конструкторы и позволяем копирующему конструктору быть копирующим конструктором по умолчанию? Часто мы получаем неверную семантику — а именно, семантику поверхностного копирования. При такой семантике никакого нового значения не создается, вместо этого переменной-указателю присваивается адрес существующего значения. Возьмем случай с семантикой ссылок, где копирование подразумевает, что счетчик ссылок увеличивается. Этого не происходит в случае с копирующим конструктором по умолчанию. То есть объекты, копиру-

емые таким образом, не будут посчитаны и память будет освобождена преждевременно. На практике поставщик класса должен явно записать копирующий конструктор, если только не очевидно, что почленное копирование безопасно. Всегда будьте осторожны, если агрегат данных содержит любые члены, основанные на указателях.

Существуют ли специальные правила для объединений? Да. Это не удивительно, поскольку объединение — это способ, позволяющий разным объектам разделять общую память. Объединения не могут содержать члены, являющиеся конструкторами или деструкторами, а также статические члены данных. Безымянные объединения могут иметь только открытые члены данных, а глобальные безымянные объединения должны объявляться как `static`.

6.11.1. Деструкторы: некоторые детали

Деструктор вызывается неявно, когда объект выходит за пределы области видимости. Обычно это происходит при выходе из блока или из функции.

```
my_string sub_str(char c, my_string b)
{
   my_string temp;
   .....
   return temp;
}
```

В функции `sub_str()` имеется `b` — вызываемый по значению аргумент типа `my_string`. Поэтому при вызове функции для создания локальной копии задействуется копирующий конструктор. Соответственно, при выходе из функции вызывается деструктор. Имеется локальная переменная `temp` типа `my_string`, которая создается (конструируется) при входе в блок функции, и следовательно, у нее должен быть деструктор, вызываемый при выходе из блока. Наконец, аргумент инструкции `return` должен быть создан и передан в вызывающее окружение. Соответствующий деструктор будет вызван в зависимости от того, какую область видимости имеет объект, которому присваивается возвращаемое значение.

Деструктор можно вызвать явно:

```
p = new my_string("Ты мне больше не нужна");
     //вызывает my_string::my_string(const char*)
.....
p -> ~my_string();
     //но намного предпочтительнее будет delete p
.....
```

6.12. Практические замечания

В конструкторах инициализация предпочтительнее присваивания. Например, конструктор

```
ch_stack::ch_stack(int size)
   { s = new char[size]; assert(s != 0);
     max_len = size; top = EMPTY; }
```

лучше записать как

```
ch_stack::ch_stack(int size) :
     max_len(size), top(EMPTY)
  { s = new char[size]; assert(s != 0); }
```

Как уже упоминалось, члены данных, являющиеся ссылками или объявленные как const, должны инициализоваться именно так. Кроме того, компилятор часто более эффективен по части подобной инициализации.

В классах, использующих new для создания объектов, копирующий конструктор должен быть задан явно. Предоставляемый компилятором копирующий конструктор по умолчанию обычно имеет неправильную семантику. Вообще, для объектов любого класса, использующих в своей реализации указатели, обычно надо задавать копирующий конструктор и конструктор по умолчанию. В следующей главе мы также увидим, что такие классы обычно содержат собственное определение operator=(). Это обеспечивает безопасность копирования и присваивания. В C++ имеется недавно добавленное ключевое слово explicit, которое используется для отключения семантики преобразования, когда конструктор с одним аргументом не предназначен для преобразования типов.

```
class ch_stack {
public:
   explicit stack(int n);  //не используется
                           //для преобразования
.....
};
```

Резюме

1. Конструктор — это функция-член, имя которой совпадает с именем класса. Он создает объекты типа своего класса. Этот процесс может включать в себя инициализацию членов данных и распределение свободной памяти с помощью оператора new. Конструктор вызывается, когда связанный с ним тип используется в определении:

   ```
   TYPE_foo y(3);//вызывает TYPE_foo::TYPE_foo(int)
   extern TYPE_foo x; //объявление, но не определение
   ```

   Еще раз: не все объявления являются определениями. Если объявление не является определением конструктор не вызывается.

2. Деструктор — это функция-член, имя которой представляет собой имя класса, предваряемое символом «тильда»: ~. Обычно деструктор предназначен для уничтожения значений типа класса, как правило, с помощью delete.

3. Конструктор, не требующий аргументов, называется конструктором по умолчанию. Это может быть конструктор с пустым списком аргументов или конструктор, у которого все аргументы имеют значения по умолчанию. Он имеет специальное назначение при инициализации массивов объектов своего класса.

4. Копирующий конструктор вида

тип`::`*тип*`(const` *тип*`&` *x*`)`

используется для выполнения копирования одного значения типа *тип* в другое, если:

- Переменная типа *тип* инициализуется значением типа *тип*.
- Значение типа *тип* передается (по значению) в качестве аргумента функции.
- Значение типа *тип* возвращается функцией.

В C++, если копирующий конструктор отсутствует, эти операции по умолчанию выполняются почленно.

5. Класс, содержащий члены, тип которых нуждается в конструкторе, использует инициализаторы — разделенный запятыми список вызовов конструкторов, стоящий после двоеточия. Конструктор вызывается с помощью имени члена, за которым в круглых скобках идет список аргументов. Члены инициализуются в порядке объявления (а не в порядке следования их инициализаторов).

6. Эффективной схемой уничтожения больших агрегатов является подсчет ссылок. Здесь каждый динамически размещаемый объект отслеживает свои активные ссылки. Когда объект создается, счетчик его ссылок устанавливается в единицу. Каждый раз, когда объект получает новую ссылку, его счетчик ссылок увеличивается. При потере ссылки счетчик уменьшается. Когда счетчик ссылок становится равным нулю, память, занимаемая объектом, очищается.

7. Конструкторы с одним параметром автоматически являются функциями преобразования. Они преобразуют тип параметра к типу класса. Например, `my_type::my_type(int);` является преобразованием `int` к `my_type`. Это свойство может быть отключено с помощью объявления конструктора как `explicit`.

Упражнения

1. Обсудите, почему конструкторы почти всегда являются открытыми функциями-членами. Что произойдет, если они будут объявлены `private`?

2. Напишите функцию-член для класса `mod_int`:

```
void add(int i); //добавление i к v по модулю 60
```

3. Запустите следующую программу и объясните ее поведение. Помещение отладочной информации внутрь конструкторов и деструкторов — весьма полезный шаг к разработке эффективных и правильных классов.

```
//Вызванные конструкторы и деструкторы

#include  <iostream.h>

class A {
public:
  A(int n) : xx(n)
    { cout << "вызван " << "A(int " <<n << ')' << endl;}
```

```
    A(double y) : xx(y + 0.5)
      { cout << "вызван " << "A(fl " << y ')' " << endl;}
    ~A()
      { cout << "~A(), значение A::xx = " << xx << endl; }
  private:
    int  xx;
  };

  int main()
  {
    cout << "вход в main\n";
    int   x = 14;
    float y = 17.3;
    A z(11), zz(11.5), zzz(0);

    cout << "\nСХЕМА РАЗМЕЩЕНИЯ ОБЪЕКТОВ\n";
    cout << "\nx расположен по адресу " << &x;
    cout << "\ny расположен по адресу " << &y;
    cout << "\nz расположен по адресу " << &z;
    cout << "\nzz расположен по адресу " << &zz;
    cout << "\nzzz расположен по адресу " << &zzz;
    cout << "\n_____\n";
    zzz = A(x);
    zzz = A(y);
    cout << "выход из main" << endl;
  }
```

Добавьте в класс A конструктор по умолчанию:

```
A::A() : xx(0) { cout << "вызван A()" << endl; }
```

Теперь измените программу, объявив массив типа A:

```
A d[5];
```

Присвойте значения 0, 1, 2, 3 и 4 члену данных xx каждого `d[i]`. Запустите программу и объясните ее поведение.

4. Используйте тип `ch_stack`, обсуждавшийся в разделе 6.2, «Создание динамического стека», на стр. 159, и включите в него конструктор по умолчанию для размещения `ch_stack` из 100 элементов. Напишите программу, которая обменивает содержимым два стека `ch_stack`; используйте массив стеков `ch_stack` для выполнения этой работы. Обмениваемые стеки будут двумя первыми стеками `ch_stack` в массиве. Один из способов может состоять в использовании четырех `ch_stack` — `st[0]`, `st[1]`, `st[2]` и `st[3]`. Поместите содержимое `st[1]` в `st[2]`, `st[0]` в `st[3]`, `st[3]` в `st[1]`, а `st[2]` в `st[0]`. Реализовав функции печати, которые выводили бы все элементы `ch_stack`, убедитесь, что после обмена содержимым порядок элементов в стеках не изменился. Можно ли решить задачу с помощью только трех `ch_stack`?

5. Добавьте к типу ch_stack конструктор со следующим прототипом:

```
ch_stack::ch_stack(const char* c);
//инициализация символьным массивом
```

6. Используйте тип my_string, обсуждавшийся в разделе 6.4, «Пример: динамически размещаемые строки», на стр. 162, и напишите следующие функции-члены:

```
//strcmp: отрицательна, если s < s1,
//        равна 0, если s == s1,
//        положительна, если s > s1
//        здесь s – неявный аргумент
int my_string::strcmp(const my_string& s1);

//strrev обращает my_string
void my_string::strrev();

//перегруженная print для печати первых n символов
void my_string::print(int n);
```

7. Напишите функцию, которая обменивает содержимое двух my_string. Используйте ее и my_string::strcmp из предыдущего упражнения при написании программы, которая будет сортировать массив из my_string.

8. В этом упражнении, используя тип vect из раздела 6.5, «Класс vect», на стр. 166, напишите следующие функции:

```
//складывает значения всех элементов и возвращает их сумму
int vect::sumelem();

//печатает все элементы
void vect::print();

//складывает два вектора в третий
//v(неявный_аргумент)=v1+v2
void vect::add(const vect& v1, const vect& v2);

//складывает два вектора и возвращает
//v(неявный_аргумент)+v1
vect vect::add(const vect& v1);
```

9. Напишите еще один конструктор для класса vect. Он принимает массив целых и его размер и создает вектор vect:

```
vect::vect(const int* d, int sz);
```

10. Попробуйте измерить разницу в скорости обработки безопасных массивов (вроде класса vect) и обычных массивов целых. Многократно выполните процедуру суммирования элементов массива для массива целых int a[10000] и для вектора vect a(10000). Засеките время. Полезны функции для измерения времени находятся в *time.h*.

11. Определите класс multi_v как:

```
class multi_v {
public:
   multi_v(int i) : a(i), b(i), c(i), size(i) {}
   void assign(int ind, int i, int j, int k);
   void retrieve(int ind, int& i, int& j, int& k) const;
   void print(int ind) const;
   int ub const { return (size - 1); }
private:
   vect a, b, c;
   int size;
};
```

Напишите и проверьте код для функций-членов assign(), retrieve() и print(). Функция assign() должна присваивать i, j и k элементам a[ind], b[ind] и c[ind], соответственно. Функция retrieve() выполняет обратное по отношению к функции assign(). Функция print() должна печатать три значения a[ind], b[ind] и c[ind].

12. Используйте тип slist, обсуждавшийся в разделе 6.7, «Пример: односвязный список», на стр. 169, для написания следующих функций-членов:

```
//конструктор slist с инициализатором – массивом символов
slist::slist(const char* c);

//length возвращает длину slist
int slist::length();

//возвращает число элементов со значением c
int slist::count_c(char c);
```

13. Напишите функцию-член append(), которая будет добавлять список в конец неявно заданного аргумента-списка, а затем очищать добавленный slist, обнуляя голову:

```
void slist::append(slist& e);
```

14. Напишите функцию-член copy(), которая будет копировать список:

```
//неявный аргумент принимает копию e
void slist::copy(const slist& e);
```

Не забудьте уничтожить неявный список перед тем, как делать копию. Вам нужна специальная проверка, чтобы избежать копирования списка в самого себя.

15. Используя тип slist, добавьте функции-члены, эквивалентные аналогичным стековым функциям:

```
reset   push   pop   top_of   empty
```

Применение одной структуры данных в качестве реализации для другой известно как *адаптирование* структуры данных. Этот подход широко используется в стандартной библиотеке шаблонов.

16. Рассмотрите `c_pair::increment()`, измененную так, чтобы возвращать `c_pair`.

```
inline c_pair c_pair::increment()
{
  c1++;
  c2++;
  return(*this);
}
```

Обсудите, в каком контексте эта функция отличается в поведении от версии в разделе 5.7, «Указатель `this`», на стр. 143. Какой вариант более эффективен при выполнении на вашей системе? Проверьте это и объясните результат. Если можете, попробуйте сгенерировать машинный код или (на AT&T и подобных компиляторах) промежуточный C-код для этих двух функций и их вызовов.

17. Создайте тип для безопасного трехмерного массива под названием `v_3_d`.

```
//Реализация безопасного трехмерного массива
class v_3_d {
public:
  int ub1(), ub2(), ub3();
  v_3_d(int l1, int l2, int l3);
  ~v_3_d();
  int& element(int i, int j, int k) const;
  void print() const;
private:
  int*** p;
  int    s1, s2, s3;
};
```

Инициализуйте и выведите трехмерный массив.

18. Как уже говорилось, процедура `slist::del()` ожидает непустой список. Что произойдет, если ей будет передан пустой список? Посмотрите результат на вашей системе. Модифицируйте процедуру так, чтобы проверялось это условие. Заметьте, что оно может проверяться как утверждение, но тогда на пустом списке программа прервется.

19. Добавьте конструктор в `slistelem` и используйте его для упрощения записи функции-члена `slist::prepend(char c)`.

20. Мы хотим определить класс C++, который будет похож на множество (set) в Pascal. Приведенное ниже представление является 32-разрядным машинным словом:

```
//Реализация АТД для типа set
const unsigned long int  masks[32] = {
```

```
    0x80000000, 0x40000000, 0x20000000, 0x10000000,
    0x8000000, 0x4000000, 0x2000000, 0x1000000,
    0x800000, 0x400000, 0x200000, 0x100000,
    0x80000, 0x40000, 0x20000, 0x10000,
    0x8000, 0x4000, 0x2000, 0x1000,
    0x800, 0x400, 0x200, 0x100,
    0x80, 0x40, 0x20, 0x10, 0x8, 0x4, 0x2, 0x1};

class set {
public:
  set(unsigned long int i) { t = i; }
  set() { t = 0x0; }
  void  u_add(int i) { t |= masks[i]; }
  void  u_sub(int i) { t  &= ~masks[i]; }
  bool  in(int i) const
    { return bool( (t  & masks[i]) != 0); }
  void  pr_mems() const;
  set  set_union(const set& v) const
    { return (set(t | v.t)); }
private:
  unsigned long int  t;
};
```

Напишите код `pr_mems` для вывода всех элементов множества. Напишите код для функции-члена `intersection`, возвращающей пересечение множеств.

21. Дополните пакет обработки полиномов, написав код для процедур `void polynomial::release()` и `void polynomial::print()`, которого нет в тексте (см. раздел 6.9, «Многочлен как связный список», на стр. 174).

22. Напишите код для процедуры сложения многочленов `void polynomial::plus()`.

23. Перепишите `void polynomial::plus()` так, чтобы `c.plus(c, c)` работала корректно (см. раздел 6.9, «Многочлен как связный список», на стр. 178).

24. Сделайте конструктор для `polynomial` более надежным. Предположим, что пары коэффициент-степень не обязательно отсортированы. Соответственно, конструктор должен быть написан с учетом этого обстоятельства (см. раздел 6.9, «Многочлен как связный список», стр. 176).

25. Усовершенствуйте форму класса `my_string` с подсчетом ссылок, вставив в соответствующие функции-члены проверку утверждения, что `ref_cnt` неотрицателен. Зачем это может понадобиться (см. раздел 6.10, «Строки, использующие семантику ссылок», на стр. 179)?

26. Измените класс `matrix`, чтобы получить конструктор, выполняющий транспонирование (см. раздел 6.8, «Двумерные массивы», на стр. 173). Он должен содержать перечислимый тип в качестве второго аргумента, который показывает, какая трансформация должна выполняться с массивом.

```
enum transform { transpose, negative, upper };

matrix::matrix(const matrix& a, transform t)
{
//транспонирование           base[i][j] =  a.base[j][i]
//смена знака                base[i][j] = -a.base[i][j]
//верхнетреугольная матрица  base[i][j] =  a.base[i][j]
//                                         i <= j, иначе 0
   .....
}
```

27. *(Проект)* Создайте класс ассоциативных массивов, который использует строковый тип в качестве индекса для поиска и извлечения второго строкового значения. Например:

```
address["Пол"] = "Аптос, Калифорния";
```

Подобный контейнер доступен как шаблонный класс в STL.

Глава 7

Ad hoc полиморфизм

Полиморфизм (polymorphism) — это способ присваивать различные значения (смыслы) одному и тому же сообщению. Смысл зависит от типа обрабатываемых данных. «Объектно-ориентированность» использует преимущества полиморфизма, привязывая поведение к типу объекта. Такие операторы, как + или << могут иметь различный смысл в зависимости от типов операндов.

Преобразование — это явное или неявное изменения типа значения. Преобразования представляют собой вид полиморфизма. Перегрузка функций придает функции с одним именем различные значения. Имя получает несколько интерпретаций, зависящих от выбора функции. Все это называется *ad hoc*[1] *полиморфизм*. В этой главе обсуждается перегрузка, в особенности перегрузка операторов и преобразование типов данных.

Операторы перегружаются и выбираются на основе алгоритма соответствия сигнатуре. Перегрузка операторов придает им новые значения. Например, выражение `a + b` имеет различные значения, зависящие от типа переменных `a` и `b`. Перегрузка оператора + для определяемых пользователем типов дает возможность применять их в выражениях сложения почти таким же образом, как собственные типы. Выражение `a + b` может означать конкатенацию строк, сложение комплексных чисел или сложение целых, в зависимости от того, являются ли переменные абстрактными типами данных `my_string` или `complex`, или собственным типом `int`. Аналогично, благодаря функциям преобразования возможны выражения смешанных типов. В этой главе также обсуждаются дружественные функции (friend function), и почему они критичны для перегрузки операторов.

Один из принципов ООП состоит в том, что определяемые пользователем типы должны обладать теми же привилегиями, что и собственные типы. Клиент ожидает удобства при использовании таких типов, не обращая внимания на то, собственные они или нет. Способность производителя достичь такого результата является тестом на соответствие языка объектно-ориентированному применению. В базовом языке собственные типы могут смешиваться в выражениях, потому что это удобно. В противном случае было бы сложно определить привычные преобразования.

[1] Ad hoc (лат.) — специальный, устроенный для данной цели, предназначенный для данного случая. — *Примеч. перев.*

7.1. Преобразования АТД

Явное преобразование типов в выражениях необходимо, если неявное преобразование нежелательно или если без такого преобразования выражение будет недопустимым. Одна из задач ООП на C++ состоит в интеграции определяемых пользователем АТД и встроенных типов. Для этого существует механизм, позволяющий функциям-членам обеспечивать явные преобразования.

В предыдущей главе мы обсуждали, в каких случаях конструктор с одним аргументом фактически является преобразователем типа аргумента к типу класса конструктора.

```
my_string::my_string(const char* p);
```

Этот конструктор будет автоматически преобразовывать тип `char*` к типу `my_string`. Преобразование может быть как явным, так и неявным. Явно оно используется как операция преобразования в функциональной форме или в виде приведения. Так, и приведение

```
my_string s;
char* logo = "Рога и Копыта Лтд.";

s = static_cast<my_string>(logo);
```

и преобразование

```
s = logo;       //неявный вызов преобразования
```

будут работать. Обратите внимание, что такое использование требует перегрузки оператора присваивания (см. раздел 7.7, «Перегрузка операторов присваивания и индексирования», стр. 202).

Здесь показаны преобразования уже определенного типа к пользовательскому типу. Однако пользователь не может добавить конструктор во встроенный тип, такой как `int` или `double`. С другой стороны, можно определить специальную функцию преобразования внутри класса. В общем виде подобная функция-член выглядит так:

```
operator тип() { ····· }
```

Данная функция-член должна быть нестатической. Она не может иметь параметры и не имеет объявленного возвращаемого типа. Она должна возвращать выражение указанного *типа*.

В примере с `my_string` может понадобиться преобразование `my_string` в `char*`. Это можно сделать следующим образом:

В файле string7.cpp

```
my_string::operator char*()
{
   char*  p = new char[len + 1];
   assert(p != 0);
   strcpy(p, s);
   return p;
}
```

Заметьте, что мы не просто возвращаем значение закрытого члена s. Делая так, мы нарушили бы целостность объектов `my_string`.

7.2. Перегрузка и выбор функций

Перегруженные функции являются важным дополнением C++. Конкретная функция выбирается в зависимости от соответствия списка аргументов при вызове функции списку параметров в объявлении функции. Когда вызывается перегруженная функция, компилятор должен иметь алгоритм для выбора надлежащей функции. Алгоритм, который выполняет этот выбор, зависит от того, преобразования какого типа присутствуют. Наилучшее соответствие должно быть уникальным. Оно должно быть лучшим по крайней мере для одного аргумента и так же хорошо, как остальные соответствия, для всех других аргументов.

Ниже приведен алгоритм соответствия для каждого аргумента.

Алгоритм выбора перегруженной функции

1. Использовать строгое соответствие (если возможно).
2. Попробовать стандартное повышение типа (см. раздел 2.5 на стр. 46).
3. Попробовать стандартное преобразование типа.
4. Попробовать определяемое пользователем преобразование.
5. Использовать, если возможно, соответствие эллипсису (подразумеваемому аргументу).

Стандартное повышение типа (promotion) лучше, чем остальные стандартные преобразования. Повышение — это преобразования `float` в `double`, а также `bool`, `char`, `short` или `enum` в `int`. Кроме того, к стандартным преобразованиям относятся преобразования указателей.

Несомненно, строгое соответствие является наилучшим. Для достижения такого соответствия могут использоваться приведения (cast). Компилятор не справится с двусмысленной ситуацией. Поэтому не следует полагаться на тонкие различия в типах и неявные преобразования, которые делают перегруженную функцию неясной. Если вы сомневаетесь, используйте явные преобразования для обеспечения строгого соответствия.

Напишем перегруженную функцию `greater()` (больше) и будем следовать нашему алгоритму для разных вызовов. В этом примере имеется пользовательский тип `rational` (рациональный).

В файле rational.cpp

```
//Перегруженные функции

class rational{
public:
   rational(int n = 0) : a(n),q(1){}
   rational(int i, int j) : a(i), q(j){}
   rational(double r) : q(BIG), a(r * BIG){}
   void  print() const { cout << a << " / " << q ; }
   operator double(){return static_cast<double>(a)/q;}
```

```
private:
   long  a, q;
   enum {BIG = 100};
};

inline int greater(int i, int j)
     { return ( i > j ? i : j); }
inline double  greater(double x, double y)
     { return ( x > y ? x : y); }
inline rational greater(rational w, rational z)
     { return ( w > z ? w : z); }

int main()
{
   int i = 10, j = 5;
   float x = 7.0;
   double y = 14.5;
   rational w(10), z(3.5), zmax;

   cout << "\ngreater(" << i << ", " << j << ") = "
        << greater(i, j);
   cout << "\ngreater(" << x << ", " << y << ") = "
        << greater(x, y);
   cout << "\ngreater(" << i << ", " ;
   z.print();
   cout << ") = " << greater(static_cast<rational>(i), z);
   zmax = greater(w, z);
   cout << "\ngreater(";
   w.print();
   cout << ", ";
   z.print();
   cout << ") = ";
   zmax.print();
}
```

Вот что выведет эта программа:

```
greater(10, 5) = 10
greater(7, 14.5) = 14.5
greater(10, 350 / 100) = 10
greater(10 / 1, 350 / 100) = 10 / 1
```

Применялись различные правила преобразования — и явные, и неявные.

Разбор программы rational

- `rational(double r) : q(BIG), a(r * BIG){}`

Этот конструктор преобразует `double` в `rational`.

- `operator double(){return static_cast<double>(a)/q;}`

А эта функция-член преобразует `rational` в `double`.

- ```
  inline int greater(int i, int j)
       { return ( i > j ? i : j); }
  inline double  greater(double x, double y)
       { return ( x > y ? x : y); }
  inline rational greater(rational w, rational z)
       { return ( w > z ? w : z); }
  ```

Три различные функции перегружены. Наиболее интересная из них имеет переменные типа `rational` в списке аргументов и такой же возвращаемый тип. Преобразующая функция-член `operator double` необходима для выполнения сравнения `w > z`. Позже мы покажем, как перегрузить `operator>()`, чтобы он мог напрямую принимать типы `rational`.

- ```
  cout << "\ngreater(" << i << ", " << j << ") = "
       << greater(i, j);
  cout << "\ngreater(" << x << ", " << y << ") = "
       << greater(x, y);
  ```

Первая инструкция выбирает первое определение `greater`, следуя правилу точного соответствия. Вторая инструкция выбирает второе определение `greater`, потому что используется стандартное расширяющее преобразование `float` в `double`. Тип значения переменной `x` расширяется до `double`.

- `cout << ") = " << greater(static_cast<rational>(i), z);`

Выбирается второе определение `greater` следуя правилу точного соответствия. Явное преобразование переменной `i` к типу `rational` необходимо для того, чтобы избежать неопределенности.

- `zmax = greater(w, z);`

Это точное соответствие для третьего определения.

См. упражнение 3 на стр. 220, которое относится к программе *rational*.

7.3. Дружественные функции

Ключевое слово `friend` (друг) служит спецификатором, уточняющим свойства функции. Оно дает функции-не-члену доступ к скрытым членам класса и предоставляет способ для обхода ограничений сокрытия данных в C++. Однако должна быть веская причина для обхода этих ограничений, поскольку они важны для надежного программирования.

Одна из причин использования дружественных функции состоит в том, что некоторые функции нуждаются в привилегированном доступе к более чем одному классу. Вторая причина в том, что дружественные функции передают все свои аргументы через список аргументов, при этом каждое значение аргумента должно допускать ав-

томатическое преобразование. Преобразования выполняются для переменных класса, передаваемых явно, и особенно полезны в случае перегрузки операторов, как мы увидим в следующем разделе.

Дружественная функция должна быть объявлена внутри объявления класса, по отношению к которому она является дружественной (с которым она дружит). Функция предваряется ключевым словом `friend` и может встречаться в любой части класса; это не влияет на ее смысл. Мы предпочитаем размещать объявление `friend` в открытой части класса. Функция-член одного класса может быть дружественной другому классу. В этом случае для указания ее имени в дружественном классе используется оператор разрешения области видимости. То, что все функции-члены одного класса являются дружественными функциями другого класса, может быть указано как `friend class` *имя_класса*.

Следующие объявления иллюстрируют синтаксис

```
class tweedledee {
   .....
   friend void alice();       //дружественная функция
   int cheshire();            //функция-член
   .....
};

class tweedledum {
   .....
   friend int tweedledee::cheshire();
   .....
};

class tweedledumber {
   .....
   friend class tweedledee;   //все функции-члены
   .....                      //из tweedledee получают доступ
};
```

Рассмотрим класс `matrix` (см. раздел 6.8, «Двумерные массивы», на стр. 173) и класс `vect` (см. раздел 6.5, «Класс vect», на стр. 165). Функция, умножающая вектор на матрицу, как представлено в этих двух классах, могла бы быть более эффективной, если бы имела доступ к закрытым членам обоих классов. Это могла бы быть дружественная для обоих классов функция. Как обсуждалось в разделе 6.5, «Класс vect», на стр. 165, безопасный доступ к элементам `vect` и `matrix` обеспечивался функцией-членом `element()`. Используя ее, можно написать функцию умножения, не нуждающуюся в статусе дружественной. Однако затраты в виде накладных расходов на вызов функции и проверку границ массива сделали бы такое умножение матриц весьма неэффективным.

В файле matrix2.cpp

```
class matrix;    //предварительное объявление
class vect {
```

```
public:
   friend vect mpy(const vect& v, const matrix& m);
   .....
private:
   int* p;
   int size;
};

class matrix {
public:
   friend  vect mpy(const vect& v, const matrix& m);
   .....
private:
   int**  p;
   int s1, s2;
};

vect mpy(const vect& v, const matrix& m)
{
   assert(v.size == m.s1);   //проверка размеров
   //использует привелегированный доступ к p в обоих классах
   vect  ans(m.s2);
   int i, j;

   for (i = 0; i <= m.ub2(); ++i) {
      ans.p[i] = 0;
      for (j = 0; j <= m.ub1(); ++j)
         ans.p[i] += v.p[j] * m.p[j][i];
   }
   return ans;
}
```

Небольшой нюанс: необходимо предварительное объявление класса `matrix`, поскольку функция `mpy()` должна присутствовать в обоих классах и она использует каждый из классов как тип аргумента.

Парадигма ООП состоит в том, что доступ к объектам (в C++ ими являются переменные класса) должен осуществляться через открытые члены. Только функции-члены класса должны иметь доступ к скрытой реализации АТД. Это четкий и строгий принцип разработки. Дружественные функции создают некую двойственность. Они имеют доступ к закрытым членам, не являясь при этом функциями-членами. Их можно использовать для быстрых исправлений в коде, которому необходим доступ к реализации деталей класса. Но такой механизм легко неправильно использовать.

7.4. Перегрузка операторов

Ключевое слово `operator` используется для того, чтобы определить функцию-член, осуществляющую преобразование типа. Оно также используется для перегрузки встроенных операторов C++. Также как имени функции, такому как `print`, можно

придать множество различных смыслов, зависящих от аргументов, так и оператору, например, +, можно приписать дополнительные значения. *Перегруженные операторы* можно использовать для привычной записи выражений как для встроенных типов, так и для АТД. Это очень удобный способ записи, во многих случаях он делает программы короче и читабельнее.

Унарные и бинарные операторы могут быть перегружены как нестатические функции-члены. Они неявно действуют на объект класса. Большинство унарных операторов можно перегружать как обычные функции, принимающие единственный аргумент класса или ссылку на тип класса. Большинство бинарных операторов можно перегружать как обычные функции, принимающие один или оба аргумента класса или ссылку на тип класса. Операторы =, (), [] и -> должны быть перегружены нестатическими функциями-членами.

```
class foo {
public:
   foo operator-();           //унарный минус: - foo
   foo operator-(int);        //бинарный минус: foo - int
   foo operator-(foo);        //бинарный минус: foo - foo
};

foo operator-(int, foo);      //бинарный минус: int - foo
foo operator-(int, foo*);     //неверно!
                              //должно быть foo или foo&
```

Функция `mpy()` из предыдущего раздела могла бы быть записана так:

```
vect operator*(const vect& v, const matrix& m)
   .....
```

Если бы это было сделано, и при этом `r` и `s` — это `vect`, а `t` — `matrix`, то выглядящее более естественно выражение

```
r = s * t;
```

вызвало бы функцию умножения. Все это заменяет форму записи в виде функции

```
r = mpy(s, t);
```

Хотя операторы могут приобретать новые значения, их приоритет и порядок выполнения остаются прежними. Например, приоритет оператора умножения остается более высоким, чем приоритет оператора сложения. Таблица приоритетов операторов C++ включена в приложение B, «Приоритет и порядок выполнения операторов». Почти все операторы могут быть перегружены. Исключения составляют операторы выбора члена «`.`» и «`.*`», тернарный условный оператор «`? :`», оператор `sizeof` и оператор разрешения области видимости «`::`» (см. раздел C.12.5, «Перегрузка операторов», на стр. 388).

Доступные для перегрузки операторы включают все арифметические и логические операторы, операторы сравнения, равенства, присваивания и битовые операторы. Более того, операторы автоинкремента и автодекремента (`++` и `--`) могут иметь различные префиксные и постфиксные значения (см. упражнение 18, на стр. 224). Операторы индексации массива `[]` и вызова функции `()` также можно перегружать.

Это возможно и для операторов доступа к члену через указатель на объект ->, и обращения к члену через указатель на член ->* (см. упражнение 19 на стр. 224). Можно перегружать new и delete. Операторы присваивания, вызова функции, индексирования и оператор доступа к члену через указатель на объект -> можно перегружать только нестатическими функциями-членами.

7.5. Перегрузка унарных операторов

Продолжим обсуждение перегрузки операторов демонстрацией того, как можно перегрузить унарные операторы, такие как !, ++, ~ и []. Для этого разработаем класс clock (часы), который можно использовать для хранения значений времени в виде дней, часов, минут и секунд. Разработаем привычные операции для часов.

В файле clock.cpp

```
class clock {
public:
   clock(unsigned long i); //конструктор и преобразование
   void  print() const; //отформатированный вывод
   void  tick(); //добавляет одну секунду
   clock operator++(){ tick(); return *this; }
   void reset (const clock& c);
   friend clock operator +(clock c1, clock c2);
   clock operator -(clock c);
   friend clock operator *(unsigned long m, clock c);
   friend clock operator *(clock c, unsigned long m);
private:
   unsigned long  tot_secs, secs, mins, hours, days;
};
```

Этот класс перегружает префиксный оператор автоинкремента. Последний является функцией — членом и может быть вызван со своим единственным неявным аргументом. Функция-член tick() прибавляет одну секунду к неявному аргументу перегруженного оператора ++.

```
inline clock::clock(unsigned long i)
{
   tot_secs = i;
   secs = tot_secs % 60;
   mins = (tot_secs / 60) % 60;
   hours = (tot_secs / 3600) % 24;
   days = tot_secs / 86400;
}

void clock::tick()
{
   clock  temp = clock(++tot_secs);
   secs = temp.secs;
```

```
    mins = temp.mins;
    hours = temp.hours;
    days = temp.days;
}
```

Конструктор выполняет обычные преобразования из `tot_secs` в дни, часы, минуты и секунды. Например, день состоит из 86 400 секунд, следовательно, целое деление на эту константу дает общее число дней. Функция-член `tick()` создает переменную `temp` типа `clock`, которая добавляет одну секунду к общему времени. Конструктор действует как функция преобразования, которая надлежащим образом обновляет время.

Перегруженный `operator++()` также обновляет неявную переменную типа `clock` и возвращает обновленное значение. Его можно было бы записать таким же образом, как и `tick()`, за исключением того, что была бы добавлена инструкция

```
return temp;
```

Дополнив программу следующим кодом, можно проверить наши функции:

```
void clock::print() const
{
   cout << days << " д :" << hours << " ч :"
        << mins << " м :" << secs << " с" << endl;
}

//Часы clock и прегруженные операторы

int main()
{
   clock  t1(59), t2(172799);  //172799 = 2 дня - 1 сек

   cout << "начальное время" << endl;
   t1.print();
   t2.print();
   ++t1;  ++t2;
   cout << "время через секунду" << endl;
   t1.print();
   t2.print();
}
```

Вот что будет выведено:

```
начальное время
0 д : 0 ч : 0 м : 59 с
1 д : 23 ч : 59 м : 59 с
время через секунду
0 д : 0 ч : 1 м : 0 с
2 д : 0 ч : 0 м : 0 с
```

Можно перегрузить префиксный оператор ++, используя и обычную функцию:

```
clock operator++(clock& cl)
{
```

```
    c1.tick();
    return c1;
}
```

Заметьте, что переменная в часах должна увеличиваться на одну секунду. Используется вызов по ссылке.

Выбор между представлением в виде функции-члена или обычной функции (функции-не-члена) обычно зависит от того, допустимо и желательно ли неявное преобразование. Явная передача аргумента допускает его автоматическое приведение, если это необходимо и возможно. Если перегрузка осуществлена функцией-членом, то

```
++c    равнозначно    c.operator++()
```

Если перегрузка выполнена функцией-не-членом, то

```
++c    равнозначно    operator++(c)
```

7.6. Перегрузка бинарных операторов

Вернемся к нашему примеру `clock` и покажем, как перегрузить бинарные операторы. В основном, поддерживаются те же принципы. Когда бинарный оператор перегружается с помощью функции-члена, он получает в качестве первого аргумента неявно передаваемую переменную класса, а в качестве второго аргумента — единственный аргумент списка параметров. Дружественные функции и обычные функции получают оба аргумента из списка параметров (отсутствует неявный аргумент). Конечно, обычные функции не имеют доступа к закрытым членам.

Создадим операцию для типа `clock`, которая складывает два значения.

В файле clock.cpp

```
class clock {
   .....
   friend clock operator +(clock c1, clock c2);
};

clock operator+(clock c1, clock c2)
{
   return (c1.tot_secs + c2.tot_secs);
}
```

Целое выражение неявно преобразуется к типу `clock` с помощью преобразующего конструктора `clock::clock(unsigned long)`. Оба значения типа `clock` передаются как аргументы функции, и оба они являются кандидатами для преобразования при присваивании. Поскольку `operator+()` является симметричным бинарным оператором, аргументы должны трактоваться одинаково. Таким образом, обыкновенно симметричные бинарные операторы перегружаются дружественными функциями.

Теперь, напротив, перегрузим бинарный минус с помощью функции-члена.

```
class clock {
   .....
   clock operator -(clock c);
};

clock clock::operator-(clock c)
{
   return (tot_secs - c.tot_secs);
}
```

Напомним, что существует неявный первый аргумент. Это соглашение требует некоторой привычки. Для бинарного минуса было бы лучше применить дружественную функцию из-за симметричной обработки обоих аргументов.

Определим операцию умножения как бинарную операцию, первый аргумент которой — типа `unsigned long`, а второй — переменная типа `clock`. Для этой операции необходима дружественная функция, так как тип первого аргумента функции-члена, которая перегружает оператор, должен совпадать с типом своего класса.

```
class clock {
   .....
   friend clock operator*(unsigned long m, clock c);
};

clock operator*(unsigned long m, clock c)
{
   return (m * c.tot_secs);
}
```

При таком подходе множители должны иметь строгий порядок в зависимости от типов. Чтобы обойти это ограничение обычно пишут вторую перегруженную функцию:

```
clock operator*(clock c, unsigned long m)
{
   return (m * c.tot_secs);
}
```

А вот альтернатива, здесь вторая функция определена через первую:

```
clock operator*(clock c, unsigned long m)
{
   return (m * c);
}
```

Сведя вторую реализацию к первой, мы тем самым уменьшим избыточность кода и сделаем его более последовательным.

7.7. Перегрузка операторов присваивания и индексирования

Оператор присваивания для типа класса по умолчанию генерируется компилятором для выполнения почленного присваивания. Это прекрасно, когда подходит поверхностное копирование. Для типов вроде `my_string` или `vect`, которые

нуждаются в глубоком копировании, это некорректно. В качестве практического правила можно принять следующее: каждый раз, когда класс нуждается в явно определенном копирующем конструкторе, он нуждается и в определении оператора присваивания.

Оператор индексирования обычно перегружается, когда тип класса представляет агрегат, для которого предназначено индексирование. Операция индексирования должна возвращать ссылку на элемент, содержащийся в агрегате. Перегрузка присваивания, также как и перегрузка индексирования, обладают некоторыми похожими свойствами. Обе они выполняются как нестатические функции-члены и обе обычно содержат возвращаемый тип-ссылку.

Переделаем класс `vect`, расширив его функциональность с помощью перегрузки операторов (см. раздел 6.5, «Класс vect», на стр. 166). Обновленный класс будет содержать несколько улучшений, делающих его более безопасным и полезным. Будет добавлен конструктор, преобразующий обычный целый массив в безопасный массив. Это позволит нам написать код, использующий безопасные массивы, а затем эффективно применять этот код с обычными массивами. Наконец, оператор индексирования перегружается и заменяет функцию-член `element`.

В файле vect2.h

```
//Безопасный массив типа vect с перегруженным []

class vect {
public:
//конструкторы и деструктор
   explicit vect(int n = 10);
   vect(const vect& v);          //инициализация вектором vect
   vect(const int a[], int n); //инициализация массивом
   ~vect() { delete [] p; }
//другие функции-члены
   int  ub() const { return (size - 1); }  //верхняя граница
   int&  operator[](int i) ;               //с проверкой
                                           //границ
private:
   int*  p;
   int size;
};

vect::vect(int n) : size(n)
{
   assert(n > 0);
   p = new int[size];
   assert(p != 0);
}

vect::vect(const int a[], int n) : size(n)
{
   assert(n > 0);
   p = new int[size];
```

```
    assert(p != 0);
    for (int i = 0; i < size; ++i)
      p[i] = a[i];
}

vect::vect(const vect& v) : size(v.size)
{
    p = new int[size];
    assert(p != 0);
    for (int i = 0; i < size; ++i)
      p[i] = v.p[i];
}

int& vect::operator[](int i)
{
    assert(i >= 0 && i < size);
    return p[i];
}
```

Перегруженный оператор индексирования может иметь любой возвращаемый тип и любой тип списка аргументов. Однако хорошим стилем является обеспечение согласованности между смыслом, заданным пользователем, и стандартным применением. Так, обычно прототип функции имеет вид:

```
имя_класса& operator[](целый_тип);
```

Такие функции могут использоваться с обеих сторон присваивания.

Было бы также удобно иметь возможность присваивать один массив другому. Пользователь с помощью перегрузки может задать поведение такого присваивания. При этом хорошим стилем является соответствие стандартному использованию присваивания. Следующая функция-член перегружает присваивание для класса `vect`:

```
vect& vect::operator=(const vect& v)
{
    if (this != &v){ //в случае присваивания самому
                     //себе – ничего не делаем
      assert(v.size == size);
    for (int i = 0; i < size; ++i)
      p[i] = v.p[i];
    }
    return *this;
}
```

Разбор функции vect::operator=(const vect& v)

- `vect& vect::operator=(const vect& v)`

Функция `operator=()` возвращает ссылку на `vect` и имеет один явный аргумент типа ссылка на `vect`. Первый аргумент оператора присваивания является неявным

аргументом. Можно было бы записать функцию, как возвращающую void, но тогда она не допускала бы повторного присваивания.[1]

- `if (this != &v){`

Ничего не делать, если пытаются присвоить переменную самой себе.

- `assert(v.size == size);`

Гарантия того, что размеры совпадают.

- ```
  for (int i = 0; i < size; ++i)
    p[i] = v.p[i];
  return (*this);
  ```

Явный аргумент `v.p[]` будет правой стороной присваивания; неявный аргумент `p[]` — левой стороной. Указатель на себя разыменовывается и передается в качестве значения выражения. Это позволяет использовать повторные присваивания с порядком выполнения справа налево.

Выражения с типом vect могут вычисляться с помощью надлежащей перегрузки различных арифметических операторов. В качестве примера давайте перегрузим бинарный +, подразумевая поэлементное сложение двух переменных типа vect.

```
vect vect::operator+(const vect& v)
{
  assert(size == v.size);
  vect  sum(s);
  for (int i = 0; i < s; ++i)
    sum.p[i] = p[i] + v.p[i];
  return sum;
}
```

Теперь, имея в виду класс vect, посмотрим, что означают следующие выражения:

```
a = b;                    //a, b — типа vect
a = b = c;                //a, b, c — типа vect
a = vect(data, DSIZE);    //преобразует массив data[DSIZE]
a = b + a;                //присваивание и сложение
a = b + (c = a) + d;      //сложное выражение
```

Класс vect — *вполне оформившийся АТД. Он ведет себя и проявляется в клиентском коде в точности как любой встроенный тип.* Заметьте, что перегрузка присваивания и перегрузка плюса вовсе не означает, что operator+= также перегружен. В самом деле, ответственность за то, что различные операторы имеют согласованную семантику, лежит на разработчике класса. Когда перегрузка связана с наборами операторов, их обычно перегружают соответственно.

В перегруженном операторе присваивания строк my_sting длины строк (если они разной длины) подгоняются (в отличие от перегрузки присваивания vect, в которой векторы должны быть одинакового размера).

[1] То есть нельзя было бы писать a=b=c;. — *Примеч. перев.*
```

В файле string7.cpp

```
my_string my_string::operator=(my_string& a)
{
  if (this != &a) {  //если a = a, ничего не делаем
    if (a.len != len) {  //строки разного размера?
      delete []s;
      len = a.len;
      s = new char[len + 1];
      assert(s != 0);
    }
    strcpy(s, a.s);  //копирование поверх старой строки
  }
  return *this;
}
```

7.8. Многочлены: что можно ждать от типа и языка

Поведение типа в значительной степени диктуется тем, как он понимается в сообществе, которое использует его.[1] То, как ведет себя многочлен, определяется установками математического сообщества. Когда мы собираемся написать тип для многочлена, мы ожидаем, что основные математические операции, такие как +, –, * и /, доступны и работают надлежащим образом. Более того, мы ожидаем, что предоставленные операторы присваивания, равенства, инкремента и декремента согласуются с представлениями сообщества C++. Класс обеспечивает открытый интерфейс, который легко использовать, если только он соответствует обоим представлениям. Перегрузку не следует применять в случае, если отсутствует определенное ожидаемое поведение оператора.

Более реалистичный класс для многочлена основан на представлении из раздела 6.9, «Многочлен как связный список», на стр. 174. Он может содержать следующие объявления:

В файле poly2.cpp

```
//Многочлены с перегруженными арифметическими операторами
class polynomial {
public:
  polynomial();
  polynomial(const polynomial& p);
  polynomial(int size, double coef[], int expon[]);
  ~polynomial() { release(); }
  void print() const;
  double operator()(double x) const;  //вычисляет P(x)
  polynomial& operator=(const polynomial& a);
  friend polynomial& operator+(const polynomial& a,
                               const polynomial& b);
```

[1] В данном контексте тип рассматривается не как понятие языка программирования, а как категория окружающего мира. — *Примеч. перев.*

```
    friend polynomial& operator-(const polynomial& a,
                                 const polynomial& b);
    friend polynomial& operator*(const polynomial& a,
                                 const polynomial& b);
    friend polynomial& operator/(const polynomial& a,
                                 const polynomial& b);
    friend polynomial& operator-(const polynomial& a);
    friend polynomial& operator+=(const polynomial& a,
                                  const polynomial& b);
    friend bool operator==(const polynomial& a,
                           const polynomial& b);
    friend bool operator!=(const polynomial& a,
                           const polynomial& b);
  private:
    term* h;
    int degree;
    void prepend(term*  t);
    void add_term(term*& a, term*& b);
    void release();
    void rest_of(term* rest);
    void reverse();
  };
```

Мы полагаем, что все математические операции будут работать и сохранятся все основные отношения между операторами C++. Было бы нехорошо, если бы при перегрузке `operator=()`, `operator+()` и `operator+=()` выражение `a = a + b` не давало бы тот же результат, что и `a += b`.

Вот код для перегрузки `operator=()`:

```
polynomial& polynomial::operator=(const polynomial& a)
{
  if (h != a.h) {  //избегаем случая a = a
    release();  //сборка мусора от старых значений
    polynomial* temp = new polynomial(a);
    h = temp -> h;
    degree = temp -> degree;
  }
  return *this;
}
```

Реализация прочих операторов оставлена для упражнений (см. упражнение 22, на стр. 226).

7.9. Перегруженные операторы ввода-вывода << и >>

Сохраняя дух ООП, важно перегрузить оператор <<, чтобы выводить пользовательские типы, так же как и собственные. Оператор << имеет два аргумента: `ostream&` и АТД, и должен создавать `ostream&`. Используется ссылка на поток и возвращается

ссылка на поток (вне зависимости от того, что перегружается — << или >>), потому что мы не хотим копировать объект потока. Напишем эти функции для типа rational.

В файле rational.cpp

```
class rational{
public:
   friend ostream& operator<<(ostream& out, rational x);
   friend istream& operator>>(istream& in, rational& x);
   .....
private:
   long  a, q;
};

ostream& operator<<(ostream& out, rational x)
{
    return (out << x.a << " / " << x.q << '\t');
}
```

Перегрузка оператора >> для вывода пользовательского типа обычно выглядит так:

```
istream& operator>>(istream& p, пользовательский_тип& x)
```

Если функции необходим доступ к закрытым членам x, ее надо сделать дружественной к этому классу. Ключевой момент здесь — сделать x параметром-ссылкой, чтобы его значение можно было изменять. Для этого в классе rational необходимо поместить объявление friend для этого оператора и обеспечить определение его функции:

```
istream& operator>>(istream& in, rational& x)
{
   return (in >> x.a >> x.q);
}
```

7.10. Перегрузка оператора () для индексирования

Матричный тип, предоставляющий динамически размещаемые двумерные массивы, может быть разработан с оператором вызова функции для обеспечения выбора элемента. Это хороший пример контейнерного класса, который полезен в научных и других расчетах.

Оператор вызова функции () перегружается как нестатическая функция-член. Он может быть перегружен для различных сигнатур. Он часто используется в операциях итератора (см. упражнения с 12, стр. 222 по 14, стр. 223) или в тех операциях, где нужны множественные индексы (см. упражнение 8, на стр. 228, для операции с подстроками).

В файле matrix3.cpp

```
//Тип для динамической матрицы
class matrix {
```

```
public:
   matrix(int c, int r);
   ~matrix();
   double& operator()(int i, int j);
   matrix& operator=(const matrix& m);
   matrix& operator+=(matrix& m);
private:
   int c_size, r_size;
   double  **p;
};

matrix::matrix(int c, int r) : c_size(c), r_size(r)
{
   p = new double*[c];
   assert(p != 0);

   for (int i = 0; i < c; ++i){
      p[i] = new double[r];
      assert(p[i] != 0);
   }
}

matrix:: ~matrix()
{
   for (int i = 0; i < c_size; ++i)
      delete [] p[i];
   delete [] p;
}

inline double& matrix::operator()(int i, int j)
{
   assert( i >= 0 && i < c_size && j >= 0 && j < r_size);
   return p[i][j];
}

matrix& matrix::operator=(const matrix& m)
{
   assert(m.c_size == c_size && m.r_size == r_size);
   int i, j;

   for (i = 0; i < c_size; ++i)
      for (j = 0; j < r_size; ++j)
         p[i][j] = m.p[i][j];
   return (*this);
}

matrix& matrix::operator+=(matrix& m)
{
   assert(m.c_size == c_size && m.r_size == r_size);
   int i, j;
```

```
    for (i = 0; i < c_size; ++i)
      for (j = 0; j < r_size; ++j)
        p[i][j] += m.p[i][j];
    return *this;
}
```

Разбор класса matrix

- ```
  inline double& matrix::operator()(int i, int j)
  {
    assert( i >= 0 && i < c_size && j >= 0 && j < r_size);
    return p[i][j];
  }
  ```

Данная функция-член дает удобную многоаргументную нотацию для доступа к элементам. Это приводит к тому, что в клиентском коде можно использовать выражения вида `m(i, j)` для доступа к элементам матрицы. Отметьте, что индексы матрицы проверяются на нахождение в пределах границ с помощью утверждений.

- ```
  matrix& matrix::operator+=(matrix& m)
  {
    assert(m.c_size == c_size && m.r_size == r_size);
  ```

Для проверки аргументов функции-члена на предусловие используется макро проверки утверждений. Присваиваемая матрица должна быть такого же размера, как и та, которой она присваивается. Этот код заменил инструкцию `if-else`, отвечающую за выход по ошибке. Сравните с кодом, написанным для класса `vect` (см. раздел 6.5, «Класс vect», на стр. 166).

- ```
  for (i = 0; i < c_size; ++i)
    for (j = 0; j < r_size; ++j)
      p[i][j] += m.p[i][j];
  ```

Этот вложенный цикл прозрачен и эффективен. Почленное сложение выполнено без накладных расходов.

- ```
  return (*this);
  ```

Возвращаемый тип является *ссылкой* на `matrix`. Разыменовывая указатель `this`, мы получаем lvalue объекта `matrix`. Это обычный прием, позволяющий выполнять множественные (порторные) присваивания.

Обсуждение данной программы продолжается в упражнении 24 на стр. 226.

7.11. Операторы указателей

Оператор указателя структуры `->` перегружается как нестатическая функция-член класса. Перегруженный оператор указателя структуры является унарным оператором, действующим на свой левый операнд. Аргумент должен быть либо объектом

класса, либо ссылкой на этот тип. Оператор может возвращать либо указатель на объект класса, либо объект класса, для которого определен `operator->`, либо ссылку на класс, для которого определен `operator->`.

В следующем примере мы перегрузим оператор указателя структуры внутри класса `t_ptr`. Объекты типа `t_ptr` выступают в качестве указателей с управляемым доступом к объектам типа `triple`.

В файле triple.cpp

```
//Перегружаем оператор указателя структуры

#include <iostream.h>

class triple {
public:
   triple(int a, int b, int c) { i = a; j = b; k = c; }
   void print() { cout << "\ni = " << i << ", j = "
              << j << ", k = " << k; }
private:
   int i, j, k;
};

triple unauthor(0, 0, 0);

class t_ptr {
public:
   t_ptr(bool f, triple* p) { access = f; ptr = p; }
   triple* operator->();
private:
   bool access;
   triple* ptr;
};

triple* t_ptr::operator->()
{
   if (access)
      return (ptr);
   else {
      cout << "\nНесанкционированный доступ";
      return &unauthor;
   }
}
```

Перегруженный оператор -> проверяет переменную `t_ptr::access`; в том случае, если результат теста — истина, доступ предоставляется. Следующий код иллюстрирует это:

```
int main()
{
   triple a(1, 2, 3), b(4, 5, 6);
   t_ptr ta(false, &a), tb(true, &b);
```

```
ta -> print();   //доступ запрещен
tb -> print();   //доступ предоставлен
}
```

7.11.1. Указатель на член класса

Указатель на член класса отличается от указателя на класс. Указатель на тип члена класса выглядит как *T*::*, где *T* — имя класса. В C++ есть два оператора, служащих для разыменования указателя на член класса. Вот они:

```
.*
->*
```

В конструкциях *объект*.**указатель_на_член* и *указатель*->**указатель_на_член* сначала производится доступ к объекту, а затем доступ к указанному члену и его разыменование.

В следующем фрагменте показано применение этих операторов.

В файле showhide.cpp

```
//Указатель на член класса

#include <iostream.h>

class X {
public:
   int  visible;    //видимая
   void print()
      { cout << "\nhide = " << hide
           << " visible = " << visible; }
   void reset() { visible = hide; }
   void set(int i) { hide = i; }
private:
   int  hide;       //спрятанная
};

typedef void (X::*pfcn)();

int main()
{
   X  a, b, *pb = &b;
   int X::*pXint = &X::visible;
   pfcn pF = &X::print;

   a.set(8); a.reset();
   b.set(4); b.reset();
   a.print();
   a.*pXint += 1;
   a.print();
   cout << "\nb.visible = " << pb ->*pXint;
   (b.*pF)();
```

```
    pF = &X::reset;
    (a.*pF)();
    a.print();
    cout << endl;
}
```

Вот что будет выведено:

```
hide = 8 visible = 8
hide = 8 visible = 9
b.visible = 4
hide = 4 visible = 4
hide = 8 visible = 8
```

Разбор программы showhide

- `typedef void (X::*pfcn)();`

Здесь сказано, что `pfcn` является именем типа указателя на член класса `X`, базовый тип которого — функция без аргументов, возвращающая `void`. Функции-члены `X::print` и `X::reset` соответствуют этому базовому типу.

- ```
  int X::*pXint = &X::visible;
  pfcn pF = &X::print;
  ```

Здесь объявляется переменная `pXint`, которая будет указателем на член класса `X`, базовый тип которого — `int`. Она инициализуется так, чтобы указывать на член `X::visible`. Указатель `pF` инициализуется указателем на функцию-член `X::print`.

- `a.*pXint += 1;`

Это то же самое, что и `++a.visible`.

- ```
  cout << "\nb.visible = " << pb ->*pXint;
  (b.*pF)();
  ```

Выражение указателя равносильно `pb -> visible`. Вызов функции равносилен `b.print()`.

- ```
  pF = &X::reset;
  (a.*pF)();
  ```

Указателю `pF` присвоен адрес `X::reset`. Вызов функции равнозначен `a.reset()`.

Рассмотрим распределение памяти для представления некоторого объекта. Объект имеет базовый адрес и какие-то нестатические члены, которые смещены относительно этого базового адреса. То есть указатель на член класса используется как смещение, он не является истинным указателем; истинный указатель имеет в качестве значения обычный адрес памяти. Статические члены не являются смещениями, а раз так, то указатель на статический член — истинный указатель.
  ```

7.12. Перегрузка new и delete

Большинство классов содержит операции по распределению и освобождению памяти. Иногда для обеспечения эффективности или надежности требуется более изощренное использование памяти, чем то, которое предоставляется с помощью простого вызова операторов `new` и `delete`.

Оператор `new` имеет следующий общий вид:

```
::opt new размещениеopt тип инициализаторopt
```

Здесь $_{opt}$ обозначает необязательные (optional) части. Вот некоторые примеры:

```
::new char[10];  //настаивает на глобальном new
new(buff) X(a);  //вызов с buff используя X::X(a)
```

До сих пор для выделения свободной памяти мы использовали глобальный оператор `new()`. Система неявно предоставляет аргумент `sizeof(`*тип*`)` для этой функции. Вот прототип функции:

```
void* operator new(size_t size);
```

Операторы `new` и `delete` могут быть перегружены. Это свойство предоставляет механизм, позволяющий пользователю манипулировать свободной памятью. Например, в традиционном программировании на C для доступа к свободной памяти используется `malloc()`, она возвращает `void*` указатель на выделенную память. В этой схеме память освобождается с помощью функции `free()` из *stdlib.h*. Используем перегрузку операторов `new` и `delete`, чтобы позволить объекту `X` управлять свободной памятью традиционным для C образом.

```
#include <stdlib.h>  //определены malloc() и free()

class X {
public:
   void* operator new(size_t size)
      { return (malloc(size)); }
   void operator delete(void* ptr) { free(ptr); }
   X(unsigned size) { new(size); }
   ~X() { delete(this); }
   .....
};
```

В этом примере класс `X` предоставляет перегруженные формы `new()` и `delete()`. Если класс перегружает `operator new()`, глобальный оператор по прежнему доступен с помощью оператора разрешения области видимости `::`.

Одна из причин по которой перегружаются эти операторы состоит в стремлении придать им дополнительную семантику, например, обеспечение диагностической информации или устойчивости к сбоям. Кроме того, класс может обеспечивать более эффективную схему распределения памяти чем та, которую предоставляет система.

Синтаксис *размещения* (placement) предоставляет разделенный запятыми список аргументов, используемый для выбора перегруженного оператора `new()` с соответ-

ствующей сигнатурой. Эти дополнительные аргументы часто применяются для того, чтобы поместить создаваемый объект по конкретному адресу. Такая форма оператора new использует заголовочный файл *new.h*.

В файле over_new.cpp

```
//Синтаксис размещения и перегруженный new

#include <iostream.h>
#include <new.h>

char* buf1 = new char[1000];//вместо свободной памяти
char* buf2 = new char[1000];

class object {
public:
   .....
private:
   .....
};

int main()
{
   object *p = new(buf1) object;  //размещение в buf1
   object *q = new(buf2) object;  //размещение в buf2
   .....
}
```

Синтаксис размещения позволяет пользователю иметь произвольную сигнатуру для перегруженного оператора new. Эта сигнатура отличается от аргументов инициализатора, используемых в вызове new, выбирающем соответствующий конструктор.

Оператор delete бывает двух видов. Он может иметь следующие сигнатуры:

```
void operator delete(void* p);
void operator delete(void* p, size_t);
```

Первая сигнатура не заботится о количестве байт, возвращаемых оператором delete. В этом случае программист поставляет код, предоставляющий это значение. Вторая сигнатура включает аргумент size_t, передаваемый вызову delete. Он передается компилятору как размер объекта, на который указывает p. В каждом классе только одна форма delete может быть задана как статическая функция-член.

В файле *new.h* имеется указатель _new_handler на функцию, которая вызывает обработчик ошибок для оператора new. Если память исчерпана, указатель _new_handler используется для вызова системной процедуры. С помощью set_new_handler() пользователь может указать явную процедуру *«недостаточно памяти»*, которая заменит системную. Мы возвратимся к эти вопросам в разделе 11.2, «Использование signal.h», на стр. 315, когда будем обсуждать исключения, обработку ошибок и корректность программ.

В файле new_hdlr.cpp

```
//Простая отказоустойчивость с применением _new_handler

#include <new.h>
#include <stdlib.h>  //для выхода

void heap_exhausted()//пользовательский
                     //обработчик ошибок
{
   cerr << "КУЧА ИСЧЕРПАНА" << endl;
   exit(1);
}

int main()
{
   set_new_handler(heap_exhausted);
   ..... //истощение памяти как heap_exhausted()
}
```

Функции-члены `new()` и `delete()` всегда неявно статические. Функция `new()` вызывается до того, как объект существует и, следовательно, у него еще нет `this`. Функция `delete()` вызывается деструктором при уничтожении объекта.

7.13. Практические замечания

Явное приведение аргументов может одновременно улучшать документируемость и служить полезным способом избежания расплывчатых последовательностей преобразований. Заключение аргументов или выражений в скобки и использование явных приведений несмотря на то, что выражения и так могут быть правильно вычислены и преобразованы, вовсе не свидетельствует о неграмотности или невежестве.

Перегрузку операторов легко применить неправильно. Не перегружайте операторы, когда это может привести к неправильной интерпретации. Необходимо, чтобы в предметной области существовал широко используемый способ записи, соответствующий вашей перегрузке. Кроме того, перегружайте операторы отношения в духе, ожидаемом в сообществе C++. Например, операторы отношения <, >, <= и >= должны быть осмысленными и взаимно-противоположными[1].

Вообще говоря, симметричные бинарные операторы, такие как +, *, ==, != и &&, следует перегружать с помощью дружественных функций. В этом случае оба аргумента передаются как обычные параметры. Это подчиняет оба аргумента одним и тем же правилам передачи параметров. Напомним, что в случае использования функции-члена для обеспечения перегрузки симметричных бинарных операторов, первый аргумент должен передаваться посредством указателя `this`.

Каждый раз когда класс использует `new` для создания объектов, он должен предоставлять явно перегруженный `operator=()`. Этот совет аналогичен нашему правилу, по которому такой класс предоставляет копирующий конструктор (см. раздел 6.2.1, «Копирующий конструктор», на стр. 160). Семантика оператора присваивания по

[1] То есть если верно, что a<b, то должно быть верно и b>a, но не b<a. — *Примеч. перев.*

умолчанию предоставляемого компилятором в большинстве случаев приводит к случайному поведению. Отсюда вытекает обычная форма классов с памятью, выделяемой из кучи.

```
//Показана обычная форма класса с памятью из кучи

class vect {
public:
   vect();   //конструктор по умолчанию
   vect(const vect&); //копирующий конструктор
   .....
   vect& operator=(const vect&);  //возвращает lvalue
   .....
};
```

Это правило относится также и к классам с подсчетом ссылок, таких как тип `my_string` (см. раздел 6.4, «Пример: динамически размещаемые строки», на стр. 162). Присваивание должно быть эффективным, поэтому `operator=()` возвращает ссылку. Это требует семантику lvalue.

7.13.1. Соответствие сигнатуре

Правила соответствия сигнатуре в упрощенной форме даны в разделе 7.2, «Перегрузка и выбор функции», на стр. 193. Дополнительное разъяснение этих правил будет дано на примерах в этом разделе.

Для данного аргумента наилучшее соответствие сигнатуре — это всегда точное соответствие. Точное соответствие также включает в себя *тривиальные преобразования* (trivial conversion). Для типа `T` они показаны в следующей таблице.

Тривиальные преобразования

из	в
`T*`	`const T*`
`T*`	`volatile T*`

Модификатор `volatile` имеет специальный смысл. Он предупреждает, что переменная может быть изменена извне (по отношению к коду программы). Так, представление переменной с адресом, по которому записываются данные из внешнего устройства, например часов реального времени, должно быть `volatile`.

Такие дополнительные модификаторы можно использовать при разрешении перегрузки. Так, функции

```
void print(int i);
void print(const int& i);
```

могут быть недвусмысленно перегружены.

Важно помнить, что к преобразованиям, определяемым пользователем, относятся и конструкторы с одним аргументом. Такой конструктор может быть вызван неявно для выполнения преобразования из типа аргумента в тип класса. Это может происхо-

дить с преобразованиями при присваивании, как в алгоритме соответствия аргументов. Вот пример, измененный по сравнению с примером из раздела 7.5, «Перегрузка унарного оператора», на стр. 199:

В файле clock.cpp

```
//Изменим программу clock

class clock {
public:
   clock(unsigned long i); //создание и преобразование
   void  print() const;    //отформатированный вывод
   void  tick();           //добавление одной секунды
   clock operator++()
      { this -> tick(); return (*this); }
   void reset (const clock& c);
private:
   unsigned long  tot_secs, secs, mins, hours, days;
};

void clock::reset(const clock& c)
{
   tot_secs = c.tot_secs;
   secs = c.secs;
   mins = c.mins;
   hours = c.hours;
   days = c.days;
}

int main()
{
   clock c1(900), c2(400);
   .....
   c1.reset(c2);
   c2.reset(100);
   .....
}
```

Вызов `reset(100)` приводит к подгонке аргументов между `int` и `clock`, которая является заданным пользователем преобразованием, вызывающим конструктор `clock(unsigned)`. Когда подобные преобразования не нужны, при объявлении конструктора может быть использовано новое ключевое слово `explicit`, запрещающее использование конструктора в качестве неявного преобразования.

Резюме

1. Перегрузка операторов придает им новые значения. Например, выражение `a + b` имеет различные значения, зависящие от типа переменных `a` и `b`. Это выражение может означать конкатенацию строк, сложение комплексных чисел или сложение

целых, в зависимости от того, являются ли переменные АТД `my_string`, или АТД `complex`, или собственным типом `int`.

2. В C++ существуют четыре приведения типов:

```
static_cast<тип>(выражение)
reinterpret_cast<тип>(выражение)
const_cast<тип>(выражение)
dynamic_cast<тип>(выражение)
```

Они заменяют вышедшие из употребления нотации:

```
x = float(i);    //функциональная запись в стиле C++
x = (float) i;   //приведение в стиле C
```

3. Неявный конструктор с одним аргументом фактически является преобразованием типа аргумента к типу класса этого конструктора. Преобразование определенного пользователем типа к встроенному типу может быть выполнено с помощью специальной функции преобразования. В общем виде такая функция-член выглядит так:

```
operator тип() { ····· }
```

Эти преобразования происходят неявно в выражениях присваивания, а также при передаче аргументов функций и возвращаемых функциями значений.

4. Перегруженное значение выбирается исходя из соответствия списка аргументов при вызове функции списку параметров в объявлении функции. Наилучшее соответствие должно быть уникальным. Оно должно быть лучшим по крайней мере для одного аргумента и так же хорошо как остальные соответствия для всех других аргументов. Ниже приведен алгоритм соответствия для каждого аргумента.

Алгоритм выбора перегруженной функции

1. Использовать точное соответствие (если возможно).
2. Попробовать стандартное повышение типов.
3. Попробовать стандартное преобразование типов.
4. Попробовать определяемое пользователем преобразование.
5. Использовать, если возможно, соответствие эллипсису (подразумеваемому аргументу).

5. Ключевое слово `friend` является спецификатором функции, который позволяет функции-не-члену иметь доступ к не-`public` членам класса, «другом» которого она является.

6. Ключевое слово `operator` используется, в частности, для перегрузки встроенных операторов C++. Также как имени функции, такому как `print`, можно придать множество различных смыслов, зависящих от аргументов, так и оператору, например, +, можно приписать дополнительный смысл. Перегруженные операторы можно использовать для привычной записи выражений как пользовательских, так и встроенных типов. Приоритет и порядок выполнения операторов остается неизменным.

7. При перегрузке операторов обычно используются либо функции-члены, либо дружественные функции, поскольку они имеют привилегированный доступ. Когда унарный оператор перегружается с помощью функции-члена, она имеет пустой список аргументов, так как единственный аргумент оператора является неявным аргументом. Когда функцией-членом перегружается бинарный оператор, то она в качестве первого аргумента получает неявно передаваемую переменную класса, а в качестве второго — единственный параметр списка аргументов.

8. Перегруженный оператор индексирования может возвращать любой тип и иметь любой тип списка аргументов. Однако хорошим стилем является соблюдение соответствия между значением, определенным пользователем, и стандартным применением. Обычно прототип такой функции выглядит следующим образом:

   ```
   имя_класса& operator[](целочисленный тип);
   ```

   Это lvalue, которое может быть использовано с любой стороны оператора присваивания.

Упражнения

1. В следующей таблице представлены различные выражения смешанных типов. Заполните графы для типа выражения, полученного в результате преобразования, и его значения (для тех случаев, когда выражение правильно задано).

Объявления и инициализации

```
int i = 3, *p = &i;
char c = 'b';
float x = 2.14, *q = &x;
```

Выражение	Тип	Значение
`i + c`		
`x + i`		
`p + i`		
`p == & i`		
`* p - * q`		
`static_cast<int>(x + i)`		

2. Для типа `rational` из раздела 7.2, «Перегрузка и выбор функции», на стр. 193 объясните, почему преобразования целого 7 и числа с двойной точностью 7.0 ведут к различным внутренним представлениям.

3. Что произойдет, если следующая строка кода программы *rational* из раздела 7.2, «Перегрузка и выбор функции», на стр. 193:

   ```
   cout << ") = " << greater(static_cast<rational>(i), z);
   ```

   будет заменена на

   ```
   cout << ") = " << greater(i, z);
   ```

4. Напишите конструктор для класса `rational`, который при заданных двух целых — делимом и частном, использует алгоритм нахождения наибольшего общего

делителя и сводит внутреннее представление к наименьшим значениям a и q (см. раздел 7.2, «Перегрузка и выбор функции», на стр. 193).

5. Перегрузите операторы равенства и сравнения для класса rational. Заметьте, что если два рациональных числа представлены в форме, данной в предыдущем упражнении, они равны тогда и только тогда, когда равны их делимые и частные (см. раздел 7.2, «Перегрузка и выбор функции», на стр. 193).

6. Напишите функцию, которая складывает вектор v с матрицей m. Вот прототип, который надо добавить в классы matrix и vect:

```
friend vect add(const vect& v, matrix& m);
```

Вектор v будет поэлементно складываться с каждой строкой матрицы m (см. раздел 7.3, «Дружественные функции», на стр. 195).

7. Определите класс complex как

```
class complex {
public:
  complex(double r) { real = r; imag = 0; }
  void  assign(double r, double i)
    { real = r; imag = i; }
  void  print()
    { cout << real << " + " << imag << "i "; }
  operator double()
    { return (sqrt(real * real + imag * imag));}
private:
  double real, imag;
};
```

Мы хотим дополнить этот класс, перегрузив некоторые операторы. Например, функцию-член print() можно заменить, создав дружественную функцию operator<<():

```
ostream& operator<<(ostream& out, complex x)
{
  out << x.real << " + " << x.imag << "i ";
  return out;
}
```

Напишите и проверьте код для оператора унарного минуса. Он должен возвращать complex, причем значения действительной и мнимой частей меняют знак.

8. Для типа complex напишите бинарные операторы-функции сложения, умножения и вычитания. Каждая функция должна возвращать значение complex. Запишите их как дружественные функции. Почему не следует использовать функции-члены?

9. Напишите две дружественные функции:

```
friend complex operator+(complex, double);
friend complex operator+(double, complex);
```

При отсутствии преобразований типа double к типу complex необходимы две дружественные функции для того, чтобы были допустимы смешанные выражения с complex и double. *Объясните, почему не нужно писать еще одну функцию с параметром типа* int, когда есть эти две дружественные функции?

10. Перегрузите присваивание для типа complex:

```
complex complex::operator=(complex c) { reaturn c; }
```

Будет ли оно равнозначно обычному присваиванию, которое сгенерировал бы компилятор, если бы отсутствовало данное определение? Если есть оператор преобразования complex в double, каков будет результат присваивания типа complex типу double? Попробуйте перегрузить присваивание с помощью дружественной функции в классе complex.

```
friend double operator=(double d, complex c);
//присваивание d = действительная_часть(c)
```

Почему это не будет работать?

11. Запрограммируйте класс vec_complex, который является безопасным массивом, значения элементов которого — типа complex. Перегрузите операторы + и * так, чтобы они означали соответственно поэлементное сложение комплексных векторов и скалярное произведение двух комплексных векторов. Для эффективного сложения можно сделать класс vec_complex другом класса complex.

12. Следующая функция-член является формой итератора:

```
int& vect::iterate()
{
  static int i = 0;
  i = i % size;
  return p[i++];
}
```

Она называется *итератором*[1], потому что возвращает значение каждого элемента вектора по очереди. Используйте ее для написания функции печати, которая не является функцией-членом и выводит все элементы данного вектора. Измените класс vect из раздела 7.7, «Перегрузка операторов присваивания и индексирования», на стр. 202.

13. Предыдущее упражнение демонстрирует серьезное ограничение. То, что итератор содержится внутри класса, не позволяет последовательно обращаться к элементам для нескольких различных переменных vect. То есть, если a и b — переменные vect, то при первом вызове a.iterate() будет получен первый

[1] В широком смысле этого слова. В литературе по обобщенному программированию и, в частности, по стандартной библиотеке шаблонов C++ (STL), термин «итератор» обозначает, как правило, не «что угодно, позволяющее как-нибудь перебирать элементы класса-контейнера», а объект, к которому предъявляются вполне определенные требования. Эти требования уточняют, как именно объект обеспечивает перебор элементов контейнера. — *Примеч. перев.*

элемент a, а в результате последующего вызова b.iterate() будет получен второй элемент b. Поэтому, вместо использовавшегося выше класса, определим новый класс vect_iterator:

```
class vect_iterator {
public:
   vect_iterator(vect&) : p(&v), position(0) { }
   int& iterate() const;
pivate:
   vect *p;
int position;
};
```

Этот класс должен быть другом vect. Напишите код для iterate. Тогда для каждого объявления vect необходимо соответствующее объявление итератора. Например:

```
vect          a(5), b(10);
vect_iterator it_a(a), it_b(b);
```

Используйте это для написания функции, которая находит значение наибольшего элемента вектора.

14. Определите новый класс matrix_iterator как итерирующий класс, который перебирает по очереди все элементы матрицы (см. раздел 7.10, «Перегрузка оператора () для индексирования», на стр. 208). Используйте его для нахождения наибольшего элемента матрицы.

15. Переделайте АТД my_string, используя перегрузку операторов (см. раздел 6.4, «Пример: динамические строки», на стр. 162). Функция-член assing должна быть изменена так, чтобы стать operator=. Функция-член concat должна стать operator+. Кроме того, перегрузите оператор [] так, чтобы он возвращал *i*-ый символ из my_string. Если такого символа нет, должно возвращаться значение –1.

16. Переделайте АТД списка с помощью перегрузки операторов (см. раздел 6.7, «Пример: односвязный список», на стр. 169). Функция-член prepend() должна быть заменена на operator+(), а del() — на operator--(). Кроме того, перегрузите оператор () так, чтобы он возвращал *i*-ый элемент списка.

17. Измените класс set (множество) из упражнения 20, на стр. 188, чтобы в нем были перегруженные операторы +, - и *:

```
class set {
     .....
     set operator+(set& v);  //определяет объединение
     set operator*(set& v);  //определяет пересечение
     set operator-(set& v);  //определяет разность
```

Проверьте ваш законченный АТД множества с помощью следующего кода:

```
//АТД множества
{
  set s(0x5555), t(0x10303021), w, x;
  s.pr_mems(); t.pr_mems();
  w.pr_mems(); x.pr_mems();
  w = s + t;                  //объединение множеств
  x = s * t;                  //пересечение множеств
  t = t - s;                  //разность множеств
  s.pr_mems(); t.pr_mems();
  w.pr_mems(); x.pr_mems();
}
```

Обратите внимание, что теперь у нас есть тип множества, похожий на встроенный тип set языка Pascal.

18. Постфиксные операторы ++ и -- могут быть перегружены в ином значении, нежели их префиксные варианты. Постфиксные ++ и -- можно отличать (от префиксных), определив перегружающие их функции как функции, имеющие один *неиспользуемый* целый аргумент:

```
class T {
public:
  //постфикс вызывается как t.operator++(0);
  void operator++(int);
  void operator--(int);
};
```

Между постфиксными и префиксными операторами нет никаких неявных семантических связей. Добавьте постфиксные увеличение и уменьшение к классу clock из раздела 7.5, «Перегрузка унарного оператора», на стр. 199. Пусть они добавляют и вычитают одну секунду, соответственно. Запишите эти операторы так, чтобы они использовали целый аргумент n, который будет вычитаться или прибавляться как дополнительный аргумент:

```
clock c(60);
  c++;                  //добавляет секунду
  c--;                  //вычитает секунду
  c.operator++(5);      //добавляет 1+5 секунд.
  c.operator--(5);      //вычитает 6 секунд
```

19. Оператор -> можно перегружать при условии, что он является нестатической функцией-членом, возвращающей либо указатель на объект класса, либо объект класса, для которого определен operator->. Такой перегруженный оператор указателя структуры называется *интеллектуальным оператором указателя.* Как правило, он возвращает обычный указатель после выполнения некоторых начальных вычислений. Его можно применять, например, в качестве итерирующей функции:

```
vect* vect::operator->();
//обработка целого i
//увеличение на 1 и возвращение &p[++i]
```

Измените класс `vect` из раздела 7.7, «Перегрузка операторов присваивания и индексирования», на стр. 202 для того, чтобы запрограммировать и проверить эту мысль.

20. (*Сложное*) Еще лучше создать класс (а не оператор) интеллектуального указателя:

```
class vect {
public:
   friend class smart_ptr_vect;   //добавление в vect
   .....
};

class smart_ptr_vect {
public:
   smart_ptr_vect(const vect& v);
   smart_ptr_vect& operator->();
private:
   int* ptr;
   int  position;
};

smart_ptr_vect::smart_ptr_vect(const vect& v) :
         position(0), ptr(v.p) { }
smart_ptr_vect& smart_ptr_vect::operator->()
{
   //напишите код для доступа к p[position] и проверки того,
   //что она не выходит за границы
}
```

Измените класс `vect` из раздела 7.7, «Перегрузка операторов присваивания и индексирования», на стр. 202 для того, чтобы проверить эту идею.

21. Возьмите функцию-член `polynomial::plus()` из раздела 6.9, «Многочлен как связный список», на стр. 174 и представьте ее в виде кода для перегрузки `operator+`:

```
polynomial operator+(const polynomial&, const polynomial&)
```

Эта функция должна быть дружественной классу `polynomial`.

22. (*Проект*) Напишите код для оператора умножения многочленов. Этот код может многократно вызывать процедуру сложения многочленов. Уверены ли вы, что после получения промежуточных результатов будет правильно собираться мусор? Напишите законченный пакет для обработки многочленов, отвечающий ожиданиям математического сообщества. Можно также включить в этот набор интегрирование и дифференцирование многочленов.

23. Используйте флаг условной компиляции NDEBUG для того чтобы указать компилятору, должен ли он включать проверку утверждений. Его применение дает простой механизм, позволяющий компилировать как безопасные, так и небезопасные классы из одного и того же исходного кода. Запустите приложение, например, сложение больших матриц, используя оба варианта кода и измерьте насколько дольше будет выполняться код с инструкциями проверки утверждений.

24. Напишите класс `matrix_iterator` (матричный итератор) с таким же интерфейсом, как у класса `vect_iterator` из упражнения 13 на стр. 222. Он должен содержать функции `successor()` (следующий), `predecessor()` (предыдущий), `reset()` (переустановить) и `item()` (элемент). Если хотите, можете дополнить этот список функциями-членами `int row()` (строка) и `int column()` (столбец) (обратитесь к разделу 7.3, «Дружественные функции» на стр. 195 за дополнительной информацией о классе `matrix`).

25. (*Проект*) Напишите код, пополняющий тип `rational` из раздела 7.9, «Перегруженные операторы ввода-вывода << и >> », на стр. 207. Сделайте так, чтобы он правильно работал со всеми операторами. Пусть его можно будет смешивать с другими числовыми типами, включая целые, числа с плавающей точкой и комплексные числа.

Глава 8

Перебор: итераторы и контейнеры

Контейнерные классы (container class) используются для хранения большого числа отдельных элементов. Типы `stack` и `vect` являются контейнерными классами. Многие операции с контейнерными классами предоставляют возможность удобного перебора (посещения) отдельных элементов. Кроме того, в C++ классы служат для достижения абстрактности. Абстракция — это игнорирование деталей, и класс игнорирует детали; он открыто предоставляет удобный способ для управления вычислительной задачей. В этой главе мы исследуем различные приемы для выполнения перебора и извлечения элементов класса. Один из способов состоит в создании итератора, функция которого — перебирать объекты в контейнерном классе. Итератор перемещается от элемента к элементу. Перебор послужит нам поводом для написания кода, использующего контейнерные классы, объекты-итераторы, а также (немного забегая вперед) алгоритмы из стандартной библиотеки шаблонов (STL).

8.1. Перебор

В традиционном программировании инструкция `for` используется как наиболее удачный способ для структурной итерации, особенно при обработке массивов.

```
//перебрать все a[i] и что-нибудь сделать
for (i = 0; i < size; ++i)
   sum += a[i];
```

Однородный агрегат `a[]` обрабатывается элемент за элементом. Инструкция `for` задает определенный порядок перебора и управляется индексом `i`, который «растет на глазах». Конкретные порядок перебора и индекс цикла являются обычно деталями реализации, не влияющими на вычисления. Вот еще один способ выполнить те же вычисления:

```
for (j = size - 1; j != 0; --j)
   sum += a[j];
```

Отвлеченно вычисления можно задать так:

```
пока не исчерпаны элементы
  sum += следующий элемент
```

Реализуем *следующий элемент* типом указателя, который подходит для выбора объектов в контейнере. Давайте изменим класс `vect` из раздела 7.7, «Перегрузка операторов присваивания и индексирования», на стр. 203, добавив в него объекты-итераторы. Забегая вперед, назовем новый класс `vector` — так же, как контейнер в стандартной библиотеке шаблонов.

В файле vector.h

```
class vector {
public:
  typedef int* iterator;                    //указатель
                                            //vector::int*
  explicit vector(int n = 1);
  vector(const vector& v);
  vector(const int a[], int n);
  ~vector() { delete []bp; }
  iterator begin()const { return bp; }
  iterator end()const { return bp + size; }
  int  ub()const {return size - 1;}         //верхняя граница
  int& operator[](int i);                   //проверка границ
  vector& operator=(const vector& v);
private:
  iterator bp;                              //базовый указатель
  int size;                                 //число элементов
};
```

Функции `begin()` и `end()` возвращают границы заданного вектора.

Код для суммирования такого агрегата с использованием итератора будет выглядеть так:

```
vector v(n);
vector::iterator p;
int sum = 0;
.....
  for (p = v.begin(); p != v.end(); ++p)
    sum += *p;
```

Идиома перебора здесь использует тип указателя и две границы для обработки в стандартном цикле всех элементов контейнера.

8.2. Итераторы

Объекты-указатели и идиома итерационного перебора делают очень простым написание многих стандартных алгоритмов для контейнерных классов, таких как класс `vector`. Напишем несколько подобных элементарных алгоритмов.

В файле vector.h

```
ostream& operator<<(ostream& out, const vector& v)
{
   vector::iterator p;

   for (p = v.begin(); p != v.end(); ++p)
      out << *p << '\t';
   out << endl;
   return out;
}

istream& operator>>(istream& in, vector& v)
{
   vector::iterator p;

   for (p = v.begin(); p != v.end(); ++p)
   in >> *p;
   return in;
}
```

Здесь мы перегрузили два стандартных оператора ввода-вывода. В каждом случае перебор вполне понятен. Следуя эффективной, идиоматичной схеме, программист избегает обычных ловушек, таких как ошибка, связанная с выходом за границы массива.

В файле vectacum.cpp

```
//Простой численный алгоритм

int accumulate                    //сумма значений контейнера
   ( vector::iterator first,      //начальная позиция
     vector::iterator last,       //конец
     int initial_val              //начальное значение
    )
{
   vector::iterator  p;
   int sum = initial_val;

   for (p = first; p != last; ++p)
      sum += *p;
   return sum;
}
```

Обратите внимание, что `last` обозначает конец диапазона; сама переменная `last` не разыменовывается. Вот почему функция-член `vector::end()` возвращает `p + size`.

8.3. Пример: quicksort()

Процедура сортировки *quicksort* (быстрая сортировка), которую изобрел Энтони Хор (Anthony Hoare), является высокоэффективной внутренней сортировкой. Ее сложно запрограммировать из-за многочисленных индексов, которые отслеживаются в тра-

диционной реализации (см. книгу Kelly and Pohl, *A Book on C: 3rd Edition, Benjamin/Cummings, 1995*). Мы заменим индексирование итераторами.

Алгоритм `quicksort` работает, рекурсивно разбивая неупорядоченные значения на два подмножества, разделенные неким средним значением (mid-value). Среднее значение больше, чем все элементы в первом подмножестве, и меньше или равно любого элемента второго подмножества. Такое разделение приводит ко все уменьшающимся подмножествам, которые в свою очередь разделяются до тех пор, пока не отсортируются тривиальным образом. Применение процедуры `quicksort` служит иллюстрацией эффективности метода «разделяй и властвуй» в программировании.

В файле vectsort.cpp

```
//QUICKSORT, использующая итератор
void quicksort(vector::iterator from, vector::iterator to)
{
   vector::iterator mid;
   if (from < to - 1) {
      mid = partition(from, to);     //произвести разделение
      quicksort(from, mid);          //в диапазоне от from до to
      quicksort(mid + 1, to);        //mid - "середина"
   }
}
vector::iterator partition(vector::iterator from,
                           vector::iterator to)
{
   vector::iterator front = from + 1, back = to - 1;
   int  compare = *from;
   while (front < back) {
      //поиск элемента не на месте (вперед)
      while ((front < back) &&(compare > *front))
         ++front;
      //поиск элемента не на месте (назад)
      while ((front < back) &&(compare <= *back))
         --back;
      swap(*front, *back);
   }
   //вставка среднего элемента для сравнения
   if (compare > *front) {
      swap(*from, *front);
      return front;
   }
   else {
      swap(*from, *(front - 1));
      return front - 1;
   }
}
```

Настоящая работа происходит в функции `partition()`. Она «волевым решением» использует первый элемент в качестве основы для разделения элементов на части «меньше, чем» и «больше или равно, чем». Когда она находит элемент «не на месте», расположенный на стороне «меньше, чем», она переключается на поиски элемента «не на месте», который находится на стороне «больше, чем». Она меняет эти элементы местами попарно до тех пор пока не закончит разделение, как показано на следующем рисунке.

Функция `swap()` меняет местами неотсортированные элементы. Сделав функцию `swap()` встроенной, мы придадим коду внутреннего цикла в `partition()` большую эффективность, не затрачивая дополнительных усилий со стороны программиста.

```
inline void swap(int& i, int& j)
{
   int temp = i;

   i = j;
   j = temp;
}
```

В разделе 9.5, «Параметризация quicksort()», на стр. 254 мы покажем, как превратить этот код в обобщенный алгоритм, который содержится в STL.

8.4. Дружественные классы и итераторы

Класс может определять объекты, которые необходимы в качестве внутренних деталей для других классов. Чтобы сохранить свою анонимную роль, такой класс должен быть закрытым. Класс, управляющий этими закрытыми объектами, должен быть в дружественных отношениях с ними.

Вспомним тип `my_string`, определенный с семантикой подсчета ссылок из раздела 6.10, «Строки, использующие семантику ссылок», на стр. 197. Отдельные значения `my_string` были в конечном счете ссылочными объектами типа `str_obj`. Это разъединение позволяло одному экземпляру `str_obj`, которому, возможно, требовалось много байт для хранения, быть использованным многими экземплярами `my_string`. Можно переделать этот пример, используя дружественные отношения для сохранения закрытости класса `str_obj`.

В файле string8.h

```
class str_obj {
private:
   friend class my_string;
   friend class string_iterator;
   friend ostream&
     operator<<(ostream& out, const my_string& str);
   int  len, ref_cnt;
   char* s;
   str_obj():len(0), ref_cnt(1) { s = new char[1];}
   str_obj(int n) : len(n), ref_cnt(1)
     { s = new char[n + 1]; }
   str_obj(const char* p) : ref_cnt(1)
     { len = strlen(p); s = new char[len + 1];
       strcpy(s, p); }
   ~str_obj() { delete [] s; }
};
```

Этот проект с использованием двух классов имеет преимущество — большую гибкость за счет дальнейшего разделения деталей реализации и клиентского кода.

Итераторам также обычно необходимы дружественные отношения с объектом, который они перебирают. Добавим класс-итератор и изменим наш `my_string`, чтобы перегрузить операторы присваивания и «помещения в», расширив таким образом наш пример.

В файле string8.h

```
class my_string {
public:
   my_string() { st = new str_obj; }
   my_string(int n) { st = new str_obj(n); }
   my_string(const char* p) { st = new str_obj(p); }
   my_string(const my_string& str)
     { st = str.st; st -> ref_cnt++; }
```

```
    ~my_string();
    void assign(const my_string& str);
    void print() const { cout << st -> s ; }
    my_string& operator=(my_string& str)
        { assign(str); return str; }
    friend class string_iterator;
    friend ostream&
      operator<<(ostream& out, const my_string& str);
private:
    str_obj* st;
};

//дружественная функция класса my_string
//"поместить в" - типичный способ перегрузки <<
//Так как нужен синтаксис
//ostream_переменная <<  переменная_типа,
//функция-член класса my_string недопустима.

ostream& operator<<(ostream& out,const my_string& str)
{
    out << str.st -> s;
    return out;
}
```

Как было сказано, `ostream& operator<<(ostream&, const my_string&)` нуждается в дружественных отношениях с `str_obj`. Эта функция использует закрытые детали реализации класса `my_string`. Она не может быть записана как функция-член, так как ее первым аргументом должен быть `ostream`. Поскольку возвращаемое ею значение имеет тип `ostream&`, она может быть использована для многократной подстановки в выражение. Данное употребление соответствует соглашениям *iostream.h*.

Функция-член `assign()` использовалась нами для кодирования `operator=()`. Чтобы позволить множественные присваивания возвращается ссылочное значение. *Стоит отметить, что если бы присваивание не было перегружено, обычная семантика* присваивания не работала бы — подсчет ссылок не выполнялся бы надлежащим образом.

Соответствующий класс-итератор следует общей схеме. У нас есть конструктор, связывающий объект итератора с перебираемым объектом с помощью инициализации `my_string* ptr_s`. Мы предусматриваем закрытую позиционную переменную `cur_ind`. Поскольку дружественные отношения между `my_string` и `str_obj` не транзитивны, для нашей схемы необходимо, чтобы и `string_iterator` был дружественен `str_obj`.

В файле string8.h

```
class string_iterator {
public:
    string_iterator(my_string& s) : cur_ind(0),ptr_s(&s){ }
    bool successor();
    char& item() { return ((ptr_s -> st -> s)[cur_ind]); }
```

```
    void reset(int n = 0) { cur_ind = n; }
    int position() { return cur_ind; }
  private:
    my_string* ptr_s;
    int cur_ind;
};

bool string_iterator::successor()
{
    if (cur_ind >= ptr_s -> st -> len - 1)
      return false;
    else {
      ++cur_ind;
      return true;
    }
}
```

Используем итератор для поиска в строке следующего слова.

В файле string8.cpp

```
void word(string_iterator& it_s, char* w)
{
  while (isspace(it_s.item()) && it_s.successor())
    ;                              //ищем не-пробел
  if (!isspace(it_s.item())) {
    *w = it_s.item();              //первый символ в слове
    while (it_s.successor()&& !isspace(it_s.item()))
      *++w = it_s.item();          //последующие символы
  }
  *++w = 0;                        //добавляем '\0' (терминатор)
}
```

Эта процедура пропускает символы, для которых `isspace()` принимает истинное значение. Затем она собирает слово как набор не-`isspace()` символов и завершает его нулевым символом.

8.5. Обобщенное программирование с использованием void*

Тип указателя `void*` служит в качестве обобщенного или универсального указателя. Ему разрешается присваивать указатели любых других типов. Поэтому `void*` можно использовать полиморфно, разрабатывая код, косвенно управляющий объектами любого типа. Изложенное можно продемонстрировать на примере определения стандартной функции копирования памяти `memcpy`.

В файле memcopy.cpp

```
#include <stddef.h>  //определяем size_t
#include <iostream.h>
```

```
void* memcpy(void* to, const void* from, size_t n_bytes)
{
   const char* f = reinterpret_cast<const char*>(from);
   char *t = reinterpret_cast<char*>(to);

   for (int i = 0; i < n_bytes; ++i)
      t[i] = f[i];
   return to;
}

int main()
{
   char v[4];
   int  w  = 0x00424344;

   memcpy(v, &w, 4);  //полиморфный интерфейс
   cout << w << " == " << v;
}
```

Функция memcpy допускает аргументы-указатели любого типа. Она используется для побайтового копирования символов начиная с адреса, заданного from. В приведенном фрагменте мы инициализовали четырехбайтовый символьный массив целым значением, хранящимся в w. Кроме того, эта техника может использоваться с контейнерными классами, такими как stack, для того чтобы они могли (косвенно) хранить произвольные значения.

В файле genstack.h

```
//Реализация обобщенного стека: genstack.h

typedef void* generic_ptr;

class stack {
public:
   explicit stack(int size = 1000): max_len(size), top(EMPTY)
     { s = new generic_ptr[size]; assert(s != 0);}
   ~stack() {delete [] s; }
   void reset() { top = EMPTY; }
   void push(generic_ptr c) { s[++top] = c; }
   generic_ptr pop() { return s[top--]; }
   generic_ptr top_of() { return s[top]; }
   bool empty()const { return top == EMPTY; }
   bool full()const { return top == max_len - 1; }
private:
   enum  { EMPTY = -1};
   generic_ptr* s;
   int max_len;
   int top;
};
```

Конечно, для того чтобы этот класс выполнял полезную работу, значения должны быть правильно приведены. Предположим, например, что у вас есть массив слов, хранящийся как двумерный массив символов, и вы хотите использовать алгоритм стандартного стека для вывода слов в обратном порядке.

В файле month.cpp

```
#include <iostream.h>
#include "genstack.h"

char* months[12] = { "январь", "февраль", "март",
     "апрель", "май", "июнь", "июль", "август",
     "сентябрь", "октябрь", "ноябрь", "декабрь"};

int main()
{
    stack a;
    int i;

    for (i = 0; i < 12; ++i)
      a.push(months[i]);
    for (i = 0; i < 12; ++i)
      cout << reinterpret_cast<char*>(a.pop())
          << endl;
}
```

Написание обобщенной процедуры обращения порядка следования значений, использующей стек, мы оставим до упражнения 10 на стр. 242.

8.6. Список и итератор списка

В этом разделе мы разработаем двусвязный список, интерфейс которого близок к библиотечному типу `list` из STL. Двусвязный список жертвует размером в пользу эффективности. Добавляя связь, которая явно указывает на предыдущий элемент списка, мы упрощаем такие операции, как удаление; правда, за это приходится платить хранением дополнительных указателей для всех элементов списка. Рассмотрим модель такого класса и связанных с ним определений итератора.

В файле list2.h

```
class list {
public:
   struct listelem; //предварительные объявления
   class iterator;
   friend iterator;
   list():h(0), t(0) {} //создаем пустой список
   ~list() { release(); }
   iterator begin()const { return h; }
   iterator end()const {return t;}
   void push_front(char c);
```

```
    void pop_front();
    char& front(){return h -> data;}
    char& back() {return t -> data;}
    bool empty()const{ return h == 0;}
    void release();
  private:
    listelem* h, *t; //голова и хвост списка
    struct listelem //ячейка списка
    {
      char  data;
      listelem* next, *prev;
      listelem(char c, listelem* n, listelem* p)
           :data(c), next(n), prev(p){}
      };
  }
```

Исследуем схему этого класса поэтапно, начиная со скрытой реализации. Имеется структура `listelem`, которая содержит член данных списка и два указателя: на следующий и предыдущий элементы. Сам список представлен в виде указателей на голову и хвост. Проход по элементам списка легко выполнить как в прямом направлении — начиная с `h` — с помощью члена `next`, так и в обратном направлении — начиная с `t` — с помощью члена `prev`. Такой проход останавливается, когда значение указателя становится равным 0.

Мы добавили класс-итератор, вложив его внутрь класса списка. Этот класс `list::iterator` будет использоваться для указания на текущую позицию внутри списка. Он сродни указателю-курсору. Таким образом, мы должны также уметь перемещаться по списку с помощью операторов автоинкремента и автодекремента, определенных в классе итератора. Это можно сделать следующим образом.

В файле list2.h

```
//Видимый внутри класса list

class iterator{
public:
    iterator(listelem* p = 0):ptr(p){ }
    iterator operator++();
    iterator operator--();
    iterator operator++(int);
    iterator operator--(int);
    listelem* operator->() { return ptr; };
    char& operator*() { return ptr -> data; }
    operator listelem*(){return ptr;}  //преобразование
private:
    listelem* ptr;                     //текущий listelem или 0
};
```

Где, например, `operator++()` может быть определен как:

```
list::iterator iterator::operator++()
{
   assert(ptr != 0);
   ptr = ptr -> next;
   return *this;
}
```

Используем вышеизложенное, чтобы написать перегруженный `operator<<()` для вывода списка.

```
ostream& operator<<(ostream& out, list& x)
{
   list::iterator p = x.begin();  //получает x.h

   out << "список = (";
   while (p != 0) {
      out << *p  << ", ";  //получает char&
      ++p;        //продвигает итератор с помощью next
   }
   cout << ")\n";
   return out;
}
```

При доступе к члену данных соответствующего `listelem` с помощью `list::iterator::operator*()` выполняется явное разыменование `p`.

Функций-членов для списка в STL очень много; мы покажем код лишь для самых типичных и еще несколько оставим в качестве упражнений (см. упражнения с 13 на стр. 243 по 16 на стр. 243). Основная операция со списком — это операция `push_front()`, которая помещает элемент в список. Здесь надо позаботится о специальном случае с пустым списком.

В файле list2.h

```
void list::push_front(char c)
{
   listelem* temp = new listelem(c, h, 0);

   if (h != 0) { //был непустой список
      h -> prev = temp;
      h = temp;
   }
   else      //был пустой список
      h = t = temp;
}
```

Конструктор для `listelem` создает ячейку списка, которая заполняется. Если перед этим список был пуст, то голова и хвост должны быть правильно инициализованы. Обратите внимание, что указатель `next` устанавливается на `h` конструктором `listelem`.

STL предоставляет множество конструкторов и деструкторов.

В файле list2.h

```
//конструкторы
   list():h(0), t(0) {} //0 означает пустой список
   list(size_t n_elements, char c);
   list(const list& x);
   list(iterator a, iterator b);
   ~list() { release(); }
```

Конструктор по умолчанию, создающий пустой список, уже обсуждался. Второй конструктор создает список из n элементов, инициализованных членом данных, передающимся как c.

```
list::list(size_t n_elements,  char c)
{
   assert(n_elements > 0);
   h = t = 0;
   for(size_t i = 0; i < n_elements; ++i)
      push_front(c);
}
```

Этот конструктор создает список с помощью повторяющихся вызовов `push_front()`.

В определении копирующего конструктора мы совместили итерацию списка и создание списка. Мы проходим по исходному списку, получая значения данных, используемых при создании копии.

```
list::list(const list& x)
{
   list::iterator r = x.begin();
   h = t = 0;     //необходимо для пустого списка
   while (r != 0)
      push_front(*r++);
}
```

Деструктор вызывает функцию сборки мусора `release()`.

```
void list::release()
{
  while (h != 0)
     pop_front();
}
```

Логика итератора и предложенных определений должна согласовываться с логикой обычных указателей и операций C++.

8.7. Практические замечания

Обратите внимание, что класс `list::iterator` имитирует определения, соответствующие постфиксным и префиксным автоинкременту и автодекременту. Это сделано для согласованности с поведением данных операторов по отношению к «род-

ным» типам. Постфиксные операторы возвращают значение существующего объекта вычисляемому выражению и только потом выполняют приращение. Покажем, как это выполняется для итераторов списка.

В файле list2.h

```
list::iterator iterator::operator++()
{
   assert(ptr != 0);
   ptr = ptr -> next;
   return *this;
}
```

Заметьте, что в этом префиксном операторе инкремента мы проверяем указатель на значение «конец списка» с помощью проверки утверждения. Возможна и другая схема, такая как переустановка итератора на начало списка (см. упражнение 13 на стр. 243).

В постфиксном операторе инкремента начальное значение итератора сохраняется и возвращается как `temp`.

```
list::iterator iterator::operator++(int)
{
   assert(ptr != 0);
   iterator temp = *this;
   ptr = ptr -> next;
   return temp;
}
```

Резюме

1. Перебор элементов агрегата — это фундаментальная операция. Когда агрегат является классом, элегантным решением для предоставления операций перебора является создание отдельного, но связанного класса, называемого классом итератора, в функции которого входит перебор и извлечение элементов агрегата.

2. В контейнере должны определяться члены `begin()` и `end()`, возвращающие итератор. По соглашениям STL `begin()` возвращает первую позицию в контейнере, а `end()` возвращает «запредельное» значение, используемое для прекращения итерации в контейнере.

3. Тип указателя `void*` служит в качестве обобщенного или универсального указателя. Ему можно присвоить указатель любого другого типа. Это позволяет использовать `void*` полиморфно, разрабатывая код, косвенно управляющий объектами любого типа.

4. Мы разработали двусвязный список, интерфейс которого близок к библиотечному типу `list` из STL. Двусвязный список жертвует размером в пользу эффективности. Добавляя связь, которая явно указывает на предыдущий элемент списка, мы упрощаем такие операции, как удаление; правда, за это приходится платить хранением дополнительных указателей для всех элементов списка.

5. Итераторы являются указателями или подобными указателям объектами. Их семантика должна согласовываться с семантикой указателей в базовом языке.

Упражнения

1. Напишите индексирующую функцию-член

```
int& vector::operator[];
```

Она должна использовать проверку утверждений для выяснения, не выходит ли индекс за пределы массива.

2. Напишите функцию-член

```
vector::iterator vector::find(int x);
//возвращает позицию первого элемента, содержащего x
```

3. Напишите функцию-член

```
int vector::count(int x);
//возвращает число элементов, содержащих x
```

4. Напишите процедуру сортировки методом «пузырька» для `vector`, в которой перебор использует соглашения итераторов.

5. Напишите функцию:

```
//Заполнить заданный диапазон случайными числами
void fill_rand(vector::iterator first, vector::iterator last)
{
  vector::iterator p;
  .....
}
```

которая заполняет заданный диапазон вектора случайно сгенерированными целыми значениями.

6. Объясните почему при перегрузке << для действия на объекты типа `my_string` требовались дружественные отношения к `str_obj`. Перепишите `my_string`, добавив преобразующую функцию-член `operator char*()`. Это позволит << выводить объекты типа `my_string`. Обсудите данное решение.

7. Что не будет работать в следующем клиентском коде, если исключить из `my_string` переопределение `operator=()`?

```
//Перестановка строк с подсчетом ссылок

#include <iostream.h>
class my_string {
  .....
};

void swap(my_string x, my_string y)
{
  my_string temp;
```

```
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    my_string  b("не ешь меня "), c("съешь меня ");

    cout << b << c << endl;
    swap(b, c);
    cout << b << c << endl;
}
```

8. Мы можем дополнить наш класс my_string операцией выделения подстроки перегрузив вызов функции. Вот запись: my_string(from, to). Здесь from является начальной точкой подстроки, а to — конечной.

```
my_string my_string::operator()(int from, int to)
{
    my_string temp(to - from +1);

    for (int i = from; i < to + 1; ++i)
        temp.st -> s[i - from] = st -> s[i];
    temp.st[to - from + 1] = 0;
    return temp;
}
```

Используйте эту операцию с подстрокой для поиска в строке заданной последовательности символов и возвращения значения true, если такая последовательность найдена.

9. Перепишите функцию выделения подстроки, используя конструктор char*. Хуже это или лучше? Если у вас есть профайлер, запустите вышеприведенный пример с обеими формами создания подстрок для следующего клиентского кода:

```
int main()
{
    my_string large("многословная фраза для поиска");

    for (i = 0; i < MANY; ++i)
        count += (large(i, i + 3) == "ого");}
```

Напишите для этого упражнения operator==(), работающий с my_string.

10. Напишите функцию

```
void reverse(double data[], int size);  //обращает порядок
                                        //следования
                                        //элементов
//data[size] будет обращен
//внутри объявите стек из обобщенных указателей
//помещайте значения в стек, извлекайте их в data[]
```

11. Используйте стек для вывода подстрок, в которых значения стоят в возрастающем порядке. В последовательности (7, 9, 3, 2, 6, 8, 9, 2) такими подстроками являются (7, 9), (3), (2, 6, 8, 9), (2). Используйте стек для хранения увеличивающихся значений. Извлекайте содержимое стека, когда следующее значение не больше имеющихся. Помните, что значения извлекаются из стека в обратном порядке. Переделайте это упражнение, используя очередь, чтобы избежать проблемы обращения.

12. Для стека обобщенных указателей добавьте конструктор

```
stack::stack(int size, generic_ptr[]);
```

13. Перепишите постинкрементный оператор итератора списка: .

```
list::iterator iterator::operator++();
//переустановка на начало
```

Если он «натыкается» на последний элемент списка, а именно на `end()`, он переустанавливается на первый элемент, то есть на `begin()`.

14. Напишите оба оператора декремента для итератора списка:

```
list::iterator iterator::operator--();
list::iterator iterator::operator--(int);
```

15. Напишите функции-члены списка:

```
void list::push_back(char c);  //добавляет в конец списка
void pop_back                  //извлекает из конца списка
```

16. Напишите код для конструктора:

```
list::list(iterator b, iterator e);
```

17. (*Проект*) Напишите собственный текстовый редактор. Моделью будет страница текста. Текст состоит из строк, как основной единицы. Надо создать класс-итератор, который может осмысленно перемещаться по тексту. Он может переходить к следующему слову, к следующей подстроке или строчке — то есть делать все, что вы сочтете полезным. Как минимум редактор должен уметь вводить, удалять и печатать текст с номерами строк и без них. Команды замены текста должны позволять замену одного слова другим. Постарайтесь использовать также файловый ввод-вывод (см. раздел D.5, «Файлы», на стр. 413). Текст не должен исчезать, то есть когда вы закончили работу с ним, его следует записать в файл.

Глава 9

Шаблоны, обобщенное программирование и STL

В C++ ключевое слово `template` используется для обеспечения *параметрического полиморфизма* (parametric polymorphism). Он позволяет применять один и тот же код к разным типам, причем тип является параметром тела кода. Параметрический полиморфизм — форма обобщенного программирования. Многие из наших классов использовались для хранения данных конкретного типа. Данные обрабатывались одним и тем же способом независимо от типа. Определения шаблонных классов и шаблонных функций делают возможным повторное использование кода простым, безопасным с точки зрения типов образом, что позволяет компилятору автоматизировать процесс *инстанцирования* (instntiation) типа. Оно выполняется, когда фактический тип заменяет тип-параметр, присутствующий в коде шаблона.

Полиморфизм: способность принимать различные формы

9.1. Шаблонный класс stack

Изменим класс `ch_stack`, чтобы получить параметризованный тип из раздела 6.2.1, «Копирующий конструктор», на стр. 160.

В файле stack_t1.cpp

```
//Реализация шаблона стека

template <class TYPE>
class stack {
public:
   explicit stack(int size = 100)
      : max_len(size), top(EMPTY), s(new TYPE[size])
      { assert (s != 0;); }
   ~stack() { delete []s; }
   void  reset() { top = EMPTY; }
   void  push(TYPE c) { s[++top] = c; }
   TYPE  pop() { return s[top--]; }
   TYPE  top_of()const { return s[top]; }
   bool  empty()const { return top == EMPTY; }
   bool  full()const { return top == max_len - 1;}
private:
   enum  { EMPTY = -1 };
   TYPE* s;
   int   max_len;
   int   top;
};
```

Синтаксис объявления класса предваряется конструкцией:

```
template <class идентификатор>
```

Этот *идентификатор* является аргументом шаблона, который по существу означает произвольный тип. Повсюду в определении класса аргумент шаблона может использоваться как имя типа. Этот аргумент инстанцируется фактическим объявлением. Объявление шаблона обычно имеет глобальную область видимости или область видимости пространства имен. Переменная шаблонного типа может быть членом класса; такие переменные также могут объявляться внутри другого шаблонного класса. Вот примеры объявления стека:

```
stack<char>      stk_ch;          //стек на 100 char
stack<char*>     stk_str(200);    //стек на 200 char*
stack<complex>   stk_cmplx(500);  //стек на 500 complex
```

Механизм шаблонов спасает нас от необходимости переписывать объявления классов, единственное изменение в которых заключалось бы в объявлениях типа. Эта схема служит альтернативой использованию `void*` в качестве универсального типа указателя. При обработке такого типа в качестве части объявления в коде всегда должны использоваться угловые скобки < >, как показано ниже.

В файле stack_t1.cpp

```
//Обращение набора строк, представленных как char*

void reverse(char* str[], int n)
```

```
{
  stack<char*>  stk(n);

  for (int i = 0; i < n; ++i)
    stk.push(str[i]);
  for (int i = 0; i < n; ++i)
    str[i] = stk.pop();
}

//Инициализация стека комплексных чисел из массива
void init(const complex c[], stack<complex>& stk, const int n)
{
  for (int i = 0; i < n; ++i)
    stk.push(c[i]);
}
```

Функции-члены, объявленные и определенные внутри класса, являются встроенными. При определении их вне класса вы должны использовать полное объявление с угловыми скобками. То есть, будучи объявляемой за пределами шаблонного класса, функция

```
TYPE top_of() const { return s[top]; }
```

должна быть записана как

```
template<class TYPE> TYPE stack<TYPE>::top_of() const
    { return s[top]; }
```

Да, это не очень изящно и требует некоторой привычки, но иначе компилятор не знал бы, что TYPE является аргументом шаблона. В качестве другого примера напишем определение деструктора для `template<class TYPE> stack`.

```
template<class TYPE> stack<TYPE>::~stack()
  { delete []s; }
```

9.2. Шаблоны функций

Часто несколько функций сводятся к одному и тому же коду, если отвлечься от различий в типах. Например, функция инициализации содержимого массива другим массивом такого же типа. Суть ее такова:

```
for (i = 0; i < n; ++i)
  a[i] = b[i];
```

Большинство программистов на C автоматизируют подобные вычисления с помощью простого макро:

```
#define COPY(A, B, N) \
  {int i; for(i = 0; i < (N); ++i) (A)[i] = (B)[i];}
```

Программирование, которое работает независимо от типа, является формой обобщенного программирования. Макро `define` как правило работает правильно, но это макро небезопасно с точки зрения типов. Другая проблема с макро заключается в том, что оно может привести к многократному вычислению единственного параметра (см. упражнение 3 на стр. 278). Пользователь легко может смешать типы там, где преобразования

неуместны. Программисты на C++ могут использовать различные формы преобразований и перегрузки для достижения сходных результатов. Однако при отсутствии надлежащих преобразований и сигнатур не будут производится никакие действия. Шаблоны предоставляют еще один механизм обобщенного программирования.

В файле copy1.cpp

```
template<class TYPE>
void copy(TYPE a[], TYPE b[], int n)
{
   for (int i = 0; i < n; ++i)
      a[i] = b[i];
}
```

Вызов `copy()` с конкретными аргументами заставит компилятор сгенерировать фактическую функцию на основе этих аргументов. Если он не сможет этого сделать, произойдет ошибка на этапе компиляции. Каковы будут результаты следующих вызовов?

В файле copy1.cpp

```
double  f1[50], f2[50];
char    c1[25], c2[50];
int     i1[75], i2[75];
char*   ptr1, *ptr2;
copy(f1, f2, 50);
copy(c1, c2, 10);
copy(i1, i2, 40);
copy(ptr1, ptr2, 100);
copy(i1, f2, 50);
copy(ptr1, f2, 50);
```

Два последних вызова `copy()` не откомпилируются, так как их типы не могут смешиваться — произойдет ошибка компиляции. Тип фактических аргументов не соответствует шаблону. Если бы было выполнено приведение `f2` в виде

```
copy(i1, (int*)f2, 50);
```

то такая инструкция была бы откомпилирована. Однако, такое копирование нас бы не устроило. Нам нужна обобщенная процедура копирования, принимающая два аргумента различных типов классов.

В файле copy2.cpp

```
template<class T1, class T2>
void copy(T1 a[], T2 b[], int n)
{
   for (int i = 0; i < n; ++i)
      a[i] = b[i];
}
```

При такой форме копирования производится поэлементное преобразование. Обычно такое преобразование предпочтительнее, кроме того оно безопаснее.

9.2.1. Соответствие сигнатуре и перегрузка

Довольно часто обобщенные процедуры не срабатывают в некоторых особых случаях. Ниже приведена форма шаблона для перестановки, работающая с основными типами.

В файле swap.cpp

```
//Обобщенная перестановка

template <class T>
void swap(T& x, T& y)
{
   T  temp;
   temp = x;
   x = y;
   y = temp;
}
```

Любом вызов, обеспечивающий однозначное соответствие аргументам шаблона, приводит к созданию функции на основе данного шаблона.

```
int i, j;
char str1[100], str2[200], ch;
complex  c1, c2;

swap(i, j);                    //правильно,  i и j – целые
swap(c1, c2);                  //правильно, c1, c2 – комплексные
swap(str1[50], str2[33]);      //правильно, обе переменные –
                               //типа char
swap(i, ch2);                  //i int, ch char – недопустимо
swap(str1, str2);              //недопустимо: попытка поменять
                               //местами массивы
```

Чтобы `swap()` могла работать со строками, представленными в виде символьных массивов, напишем для этого особого случая следующий код:

```
void swap(char* s1, char* s2)
{
   int  max_len;

   max_len = (strlen(s1) >= strlen(s2)) ?
          strlen(s1) : strlen(s2);
   char* temp = new char[max_len + 1];

   strcpy(temp, s1);
   strcpy(s1, s2);
   strcpy(s2, temp);
   delete []temp;
}
```

С добавлением этого кода для особого случая, точное соответствие этой нешаблонной версии сигнатуре вызова `swap()` имеет приоритет перед точным соответствием при подстановке в шаблон.

Алгоритм выбора перегруженной функции

1. Строгое соответствие (при допущении некоторых тривиальных преобразований) для нешаблонных функций.
2. Строгое соответствие для шаблонов функций.
3. Обычное разрешение аргументов для нешаблонных функций.

9.3. Шаблоны классов

В примере `stack<T>` из раздела 9.1, «Шаблонный класс stack», на стр. 245, мы показали обычный случай параметризации классов. В этом разделе мы хотим обсудить различные особенности параметризации классов.

9.3.1. Друзья

Шаблонные классы могут содержать «друзей». Дружественная функция, которая не использует спецификацию шаблона, универсальна — имеется единственный ее экземпляр для всех инстанцирований шаблонного класса. Дружественная функция, которая задействует аргументы шаблона, специфична для каждого инстанцирования класса:

```
template <class T>
class matrix {
public:
   friend void foo_bar();              //универсальна
   friend vect<T> product(vect<T> v); //специфична
   .....
};
```

9.3.2. Статические члены

Статические члены не универсальны, они специфичны для каждого инстанцирования:

```
template <class T>
class foo {
public:
   static int count;
   .....
};

   .....
foo<int>    a;
foo<double> b;
```

Статические переменные `foo<int>::count` и `foo<double>::count` различны.

9.3.3. Аргументы шаблона класса

И классы, и функции могут иметь несколько аргументов шаблона. Напишем функцию, которая будет преобразовывать один тип значения к другому, при условии, что первый тип не «уже» второго.

В файле coerce.cpp

```
template <class T1, class T2>
bool coerce(T1& x, T2 y)
{
  if (sizeof(x) < sizeof(y))
    return false;
  x = static_cast<T1>(y);
  return true;
}
```

В этой шаблонной функции есть два (возможно различных) типа, заданных как аргументы шаблона.

Аргументами шаблона могут, в частности, быть выражения-константы, имена функций и символьные строки.

В файле array_tm.cpp

```
template <class T, int n>
class assign_array {
public:
  T  a[n];
};

.....
assign_array<double, 50>  x, y;
.....
x = y;          //должно работать эффективно
```

Выгода от такой параметризации состоит в том, что происходит размещение в стеке, а не в свободной памяти (в куче). На многих системах такой режим более эффективен. Допустимый диапазон ограничен конкретной целой константой, так что операции, требующие соответствия длин массивов, безопасны с точки зрения типов и проверяются на этапе компиляции.

9.4. Параметризация класса vector

Класс `vector` — естественный кандидат для параметризации. Давайте выполним параметризацию и обсудим, как `vector` используется в сочетании с итераторами и алгоритмами. Этот класс является типичным последовательным контейнерным классом, похожим на подобный класс из стандартной библиотеки шаблонов:

В файле vect_it.h

```
//Тип vector на основе шаблона

template <class T>
class vector {
public:
  typedef T* iterator;
```

```
    explicit vector(int n=100);     //создание массива
                                    //из n элементов
    vector(const vector<T>& v);     //копия вектора
    vector(const T a[], int n);     //копия массива
    ~vector() { delete []p; }
    iterator begin(){ return p;}
    iterator end(){ return p + size;}
    T& operator[](int i);           //элемент в пределах границ
    vector<T>& operator=(const vector<T>& v);
  private:
    T* p;                           //базовый указатель
    int size;                       //число элементов
};
```

Везде, где ранее класс `vector` использовал `int` как значение, которое должно храниться в отдельных элементах, определение `template` использует `T`. Так что закрытый базовый указатель `p` теперь объявлен как тип `T*`.

Помещаемое вне класса определение функции-члена должно включать метку разрешения области видимости *имя_класса*<T>. Следующие конструкторы для `vect<T>` используют `T` как спецификацию типа для `new`:

```
template <class T>
vector<T>::vector(int n = 100) : size(n)
{
  assert(n > 0);
  p = new T[size];
  assert(p != 0);
}
```

Этот конструктор является конструктором по умолчанию, поскольку имеет аргумент по умолчанию, равный 100. Ключевое слово `explicit` используется для запрета применения его в качестве преобразователя из `int` в `vector`. Проверка утверждений используется чтобы гарантировать, что конструктор выполняет свои «договорные обязательства», когда принимает соответствующие данные на вводе.

```
template <class T>
vector<T>::vector(const T a[], int n)
{
  assert(n > 0);
  size = n;
  p = new T[size];
  assert(p != 0);
  for (int i = 0; i < size; ++i)
    p[i] = a[i];
}
```

Этот конструктор преобразует обычный массив в вектор. Копирующий конструктор определяет глубокую копию (см. раздел 6.2.1, «Копирующий конструктор», на стр. 160) вектора `v`.

```
template <class T>
vector<T>::vector(const vector<T>& v)
{
  size = v.size;
  p = new T[size];
  assert(p != 0);
  for (int i = 0; i < size; ++i)
    p[i] = v.p[i];
}
```

Следующий код определяет индексирование вектора с помощью перегрузки оператора «квадратные скобки» `[]`. Возвращаемый тип для этого оператора — ссылка на `T`, поскольку он является псевдонимом для элемента, хранящегося в контейнере. Использование такого возвращаемого типа позволяет оператору `[]` иметь доступ к элементу контейнера в качестве lvalue.

```
template <class T> T& vector<T>::operator[](int i)
{
  assert (i >= 0 && i < size );
  return (p[i]);
}
```

Обратите внимание: можем проверить, что границы массива не нарушаются. Мы должны позаботиться и о перегруженном операторе присваивания (см. упражнение 7 на стр. 278).

```
template <class T>
vector<T>& vector<T>::operator=(const vector<T>& v)
{
  assert(v.size == size);
  for (int i = 0; i < size; ++i)
    p[i] = v.p[i];
  return *this;
}
```

Клиентский код почти также прост, как и при непараметризованных объявлениях. Чтобы использовать наши объявления, просто добавьте конкретный тип в угловых скобках, который подставляется в шаблон. Это может быть собственный тип, например, `int`, или тип, определенный пользователем. Следующий код использует приведенные выше шаблоны.

В файле vect_it.cpp

```
int main()
{
  vector<double> v(5);
  vector<double>::iterator p;
  int i = 0;
  for (p = v.begin(); p != v.end(); ++p)
```

```
    *p = 1.5 + i++;
  do {
    --p;
    cout << *p << " , ";
  } while (p != v.begin());
  cout << endl;
}
```

Вот что будет выведено этой программой:

```
5.5, 4.5, 3.5, 2.5, 1.5,
```

Значения выведены в порядке, обратном порядку их хранения. Это — результат итерации в обратную сторону от значения итератора `v.end()`.

9.5. Параметризация quicksort()

Используем `vector<T>` и шаблоны для построения параметризованной процедуры `quicksort()`. Будет параметризована каждая из составляющих традиционной `quicksort`.

В файле quiksort.cpp

```
//QUICKSORT с использованием класса vector

template<class T>
void swap(T& i, T& j){ T temp = i; i = j; j = temp; }
```

Сердце любой процедуры сортировки — перестановка элементов. В данном случае `swap()` (перестановка) параметризуется с тем, чтобы она могла принимать произвольные типы.

Сама процедура `quicksort` — это простая рекурсия. Она использует процедуру разбиения для того, чтобы разделить параметризованный массив `vector<T>` на две части. Элементы в диапазоне от `from` до `mid - 1` меньше элементов в диапазоне от `mid + 1` до `to`.

```
template<class T>
void quicksort(T* from, T* to)
{
  T* mid;

  if (from < to - 1) {
    mid = partition(from, to);
    quicksort(from, mid);
    quicksort(mid + 1, to);
  }
}
```

Процедура `partition()` (разбиение) параметризуется и использует итераторы для отслеживания того, где она находится при перестановке элементов «не на месте».

Итераторы front и back отвечают за текущую позицию в соответствующих частях расчленяемого массива.

```
template<class T>
T* partition(T* from, T* to)
{
  T* front = from;
  T* back = to - 1;
  T  compare;

  compare = *from;      //сравниваемый элемент
  ++front;              //переход к следующему элементу
  while (front < back) {
    //поиск элемента "не на месте" (вперед)
    while ((front < back) &&(compare > *front))
          ++front;
    //поиск элемента "не на месте" (назад)
    while ((front < back) &&(compare <= *back))
          --back;
    swap(*front, *back);
  }
  //вставка среднего элемента для сравнения
  if (compare >= *front) {
    swap(*from, *front);
    return (front);
  }
  else {
    swap(*from, *(front - 1));
    return (front - 1);
  }
}
```

Вышеизложенное очень просто использовать в клиентском коде. Мы просто инстанцируем нужный тип с угловыми скобками, например:

```
vector<int> v(n);  //создает вектор из n целых
```

и вызываем инстанцированный алгоритм сортировки как

```
quicksort(v.begin(), v.end());
```

Заметьте, что сортируемый параметризованный тип нуждается в процедуре сравнения, которая определена для него.

9.6. Параметризованное дерево двоичного поиска

Параметрический полиморфизм достигается как с помощью обобщенных указателей void*, так и с помощью шаблонов. Продемонстрируем пример использования параметрического полиморфизма для реализации двоичного отсортированого дерева.

В файле gentree1.cpp

```
//Обобщенное двоично отсториторванное дерево
template <class T>  class gen_tree;   //предварительное
                                      //объявление

template <class T>
class bnode {                         //двоичный узел
private:
   friend class gen_tree<T>;
   bnode<T>* left;                    //левый
   bnode<T>* right;                   //правый
   T         data;                    //данные
   int       count;                   //счетчик
   bnode(T d, bnode<T>* l, bnode<T>* r) :
        data(d), left(l), right(r), count(1) { }
   void  print() const
     { cout << data << " : " << count << '\t'; }
};
```

Дерево будет хранить данные `data` типа `T`. Внутренние указатели «на себя» `left` и `right` являются указателями типа `bnode<T>*`. Обозначения довольно скверные, но необходимые. Встроенная функция `bnode<T>::print()` ожидает, что имеется `operator<<()`, определенный как оператор вывода. Если это не так, инстанцирование потерпит неудачу на этапе компиляции.

```
template <class T>
class gen_tree { //обобщенное дерево
public:
   gen_tree() { root = 0; }
   void  insert(T d);
   T  find(T d) const { return (find(root, d)); }
   void  print() const { print(root); }
private:
   bnode<T>*  root;
   T  find(bnode<T>* r, T d) const;
   void  print(bnode<T>* r) const;
};
```

Тип обобщенного дерева `gen_tree<T>` инстанцируется для хранения `data` типа `T` с использованием процедуры вставки `insert()`, обеспечивающей построение двоичного отсортированного дерева.

Функция `comp()` (сравнение) является параметризованной внешней функцией. Есть две ее формы. В общем виде она использует существующий или заданный пользователем смысл операторов == и <. Когда требуется лексикографическое сравнение переменных `const char*`, необходима *специализация* этой функции.

```
#include<string.h>
template <class T> //общий случай
```

```
int comp(T i, T j)
{
  if (i == j)       //полагаем, что == и <
                    //определены для T
    return 0;
  else
    return ( (i < j) ? -1 : 1 );
}

//специализация для const char*
int comp(const char* i, const char* j)
{
  return (strcmp(i, j));
}
```

Функция `gen_tree<T>::insert()` нужна для поиска и вставки узла в отсортированном порядке. Заметьте, что для создания объекта `bnode<T>` требуется оператор `new`.

```
template <class T>
void gen_tree<T>::insert(T d)
{
  bnode<T>*  temp = root;
  bnode<T>*  old;

  if (root == 0) {
    root = new bnode<T>(d, 0, 0);
    return;
  }

  while (temp != 0) {
    old = temp;
    if (comp(temp -> data, d) == 0) {
      (temp -> count)++;
      return;
    }
    if (comp(temp -> data, d) > 0)
      temp = temp -> left;
    else
      temp = temp -> right;
  }
  if (comp(old -> data, d) > 0)
    old -> left = new bnode<T>(d, 0, 0);
  else
    old -> right = new bnode<T>(d, 0, 0);
}
```

Для того, чтобы согласовываться с параметризацией функции-члены нуждаются в простых изменениях. Почти все изменения для конкретного случая приведены в следующих объявлениях:

```
template <class T>
T gen_tree<T>::find(bnode<T>* r, T d) const
{
   if (r == 0)
     return 0;
   else if (comp(r -> data, d) ==  0)
     return (r -> data);
   else if (comp(r -> data, d) > 0)
     return (find( r -> left, d));
   else
     return (find( r -> right, d));
}

template <class T>
void gen_tree<T>::print(bnode<T> *r) const
{
   if (r != 0) {
     print( r -> left);
     r -> bnode<T>::print();
     print ( r -> right);
   }
}
```

Клиентский код нуждается лишь в подстановке конкретного типа, который будет храниться в двоичных деревьях. Ниже приведены два применения, одно — для сортировки строк `char*`, а другое — для сортировки целых.

В файле gentree1.cpp

```
int main()
{
   char  dat[256];
   gen_tree<char*>  t;
   char* p;

   while (cin>>dat){
     p = new char[strlen(dat) + 1];
     strcpy(p, dat);
     t.insert(p);
   }
   t.print();
   cout << "EOF" << endl << endl;    //конец файла

   gen_tree<int>  i_tree;

   for (int i = 15; i > -5; --i)
     i_tree.insert(i);
   i_tree.print();
}
```

В разделе 10.3, «Повторное использование кода: класс двоичного дерева», на стр. 286 мы вернемся к рассмотрению этой структуры данных и используем наследование для достижения тех же результатов в области повторного использования кода и полиморфизма, которых мы достигли здесь с помощью шаблонов. Два этих подхода имеют различные преимущества и недостатки с точки зрения легкости использования, эффективности и расширяемости.

9.7. STL

Стандартная библиотека шаблонов (STL, Standard Template Library) является стандартной библиотекой C++, предоставляющей возможности обобщенного программирования для многих стандартных структур данных и алгоритмов. Тремя ее компонентами являются: контейнеры, итераторы и алгоритмы, которые поддерживают возможности обобщенного программирования.

Библиотека построена с использованием шаблонов, ее дизайн вполне ортогонален. Компоненты можно комбинировать друг с другом, используя в качестве параметров как собственные типы C++, так и типы, определяемыми пользователем. Необходимо только следить за правильностью инстанцирования различных элементов библиотеки STL.

В нашей первой программе, использующей STL, мы применим списочный контейнер, итератор и обобщенный алгоритм `accumulate()` (накопление).

В файле stl_cont.cpp

```
//Применение списочного контейнера

#include <iostream>
#include <list> //списочный контейнер
#include <numeric> //необходимо для accumulate
using namespace std;

void print(const list<double> &lst)
{          //используем итератор для прохода по lst
   list<double>::const_iterator p;

   for (p = lst.begin(); p != lst.end(); ++p)
      cout << *p << '\t';
   cout << endl;
}

int main()
{
   double w[4] = { 0.9, 0.8, 88, -99.99 };
   list<double> z;

   for( int i = 0; i < 4; ++i)
      z.push_front(w[i]);
   print(z);
   z.sort();
   print(z);
```

```
    cout << "сумма равна "
         << accumulate(z.begin(), z.end(), 0.0) << endl;
}
```

В этом примере списочный контейнер должен хранить переменные с двойной точностью. Массив из таких переменных помещается в список. Функция `print()` использует итератор для печати всех элементов списка по очереди. Обратите внимание, что итератор работает как указатель. Итераторы имеют стандартный интерфейс, частью которого являются функции-члены `begin()` и `end()`, определяющие начало и конец контейнера. Кроме того, интерфейс списка включает в себя надежный алгоритм сортировки — функцию-член `sort()`. Функция `accumulate()` является обобщенной функцией из пакета *numeric*. Она использует `0.0` в качестве начального значения и вычисляет сумму элементов списочного контейнера путем прохода от начальной позиции `z.begin()` до конечной `z.end()`.

Заметьте, что и саму функцию `print()` можно параметризовать, превратив ее в обобщенный алгоритм. Попробуйте выполнить это, используя наиболее общий подход (см. упражнение 14, на стр. 279).

9.8. Контейнеры

Контейнеры подразделяются на два основных семейства: последовательные контейнеры и ассоциативные контейнеры. Последовательные контейнеры включают векторы (vectors), списки (lists) и двусторонние очереди (deques). Эти контейнеры упорядочиваются заданием последовательности элементов. Ассоциативные контейнеры включают множества (sets), мультимножества (multisets), отображения (maps), мультиотображения (multimaps) и содержат ключи для поиска элементов. Контейнер-отображение является ассоциативным массивом. Ему необходимо, чтобы была определена операция сравнения для хранимых элементов. Все варианты контейнеров имеют похожий интерфейс.

Интерфейсы типичных контейнеров STL

- конструкторы, включая конструкторы по умолчанию и копирующие конструкторы
- доступ к элементу
- вставка элемента
- удаление элемента
- деструктор
- итераторы

Проход по контейнеру осуществляется с помощью итераторов. Это «указателеподобные» объекты, доступные в виде шаблонов, и оптимизированные для использования с контейнерами STL.

В файле stl_deq.cpp

```
//Типичный алгоритм контейнера
double sum(const deque<double> &dq)
{
  deque<double>::const_iterator p;
```

```
    double s = 0.0;
    for (p = dq.begin(); p != dq.end(); ++p)
      s += *p ;
    return s;
}
```

Проход по контейнеру deque (double-ended queue — двусторонняя очередь) производится с помощью const_iterator. Итератор p разыменовывается для получения по очереди каждого хранимого элемента. Такой алгоритм будет работать для последовательных контейнеров и со всеми типами, для которых определен operator+=().

В следующей таблице, описывающей интерфейс контейнерных классов, они обозначены как CAN.

Определения контейнеров STL

CAN::value_type	что содержит CAN
CAN::reference	тип-ссылка на значение
CAN::const_reference	константная ссылка
CAN::pointer	указатель на значение
CAN::iterator	тип-итератор
CAN::const_iterator	константный итератор
CAN::reverse_iterator	обратный итератор
CAN::const_reverse_iterator	константный обратный итератор
CAN::difference_type	разница между двумя значениями CAN::iterator
CAN::size_type	размер CAN

Эти определения доступны во всех контейнерных классах. Например, vector<char>::value_type расскажет, что в векторном контейнере храниться символьное значение. Проход по такому контейнеру может быть выполнен с помощью vector<char>::iterator.

Контейнеры предлагают операторы равенства и сравнения. Они также имеют обширный список стандартных функций-членов, как показано в следующей таблице.

Члены контейнера STL

CAN::CAN()	конструктор по умолчанию
CAN::CAN(c)	копирующий конструктор
c.begin()	начальная позиция контейнера c
c.end()	конечная позиция c[1]
c.rbegin()	начало для обратного итератора
c.rend()	конец для обратного итератора
c.size()	число элементов в CAN
c.max_size()	наибольший размер
c.empty()	истина, если CAN пуст
c.swap(d)	обмен элементами двух CAN

[1] Точнее, c.end() указывает не на последний элемент контейнера, а на (несуществующий) элемент, «следующий за последним». Такое соглашение C++ облегчает кодирование операций над контейнерами. — *Примеч. перев.*

9.8.1. Последовательные контейнеры

Последовательные контейнеры — это вектор, список и двусторонняя очередь. Они содержат последовательность доступных элементов. В C++ тип массива во многих случаях может рассматриваться как последовательный контейнер.

В файле stl_vect.cpp

```
//Последовательные контейнеры — вставка вектора в deque

#include <iostream.h>
#include <deque>
#include <vector>

using namespace std;

int main()
{
   int data[5] = { 6, 8, 7, 6, 5 };
   vector<int> v(5, 6);       //вектор из 5 элементов
   deque<int> d(data, data + 5);
   deque<int>::iterator p;

   cout << "\n3начения очереди " << endl;
   for (p = d.begin(); p != d.end(); ++p)
     cout << *p << '\t';      //печать:6 8 7 6 5
   cout << endl;
   d.insert(d.begin(), v.begin(), v.end());
   for (p = d.begin(); p != d.end(); p++)
   cout << *p << '\t';        //печать: 6 6 6 6 6 6 8 7 6 5
}
```

Пятиэлементный вектор v инициализуется значением 6. Двусторонняя очередь d инициализуется значениями, получаемыми из массива data. Функция-член insert() помещает значения v в диапазоне от v.begin() до v.end() с положения d.begin().

Разбор программы stl_vect

- ```
  int data[5] = { 6, 8, 7, 6, 5 };
  vector<int> v(5, 6); //вектор из 5 элементов
  deque<int> d(data, data + 5);
  deque<int>::iterator p;
  ```

Вектор v инициализует контейнер из пяти целых элементов значениями 6. Двусторонняя очередь d использует значения итератора data и data + 5 для инициализации контейнера двусторонней очереди. В качестве итераторов могут быть использованы обычные указатели массива. Итератор p объявлен, но не инициализован.

```
for (p = d.begin(); p != d.end(); ++p)
  cout << *p << '\t';  //печать: 6 8 7 6 5
```
```

Это стандартная идиома прохода при использовании контейнеров и итераторов. Заметьте, что `d.end()` используется для выхода из цикла, поскольку является значением итератора «конец контейнера». Обратите также внимание, что автоинкремент `++` имеет семантику указателя, продвигая итератор к следующей позиции в контейнере. Разыменование работает аналогично семантике указателей.

- `d.insert(d.begin(), v.begin(), v.end());`

Функция-член `insert()` помещает диапазон значений итератора от `v.begin()` до `v.end()` (исключая само `v.end()`) начиная с позиции `d.begin()`.

- ```
  for (p = d.begin(); p != d.end(); p++)
  cout << *p << '\t';     //печать: 6 6 6 6 6 6 8 7 6 5
  ```

Вследствие вставки пяти новых элементов со значением 6 в начало двусторонней очереди d, теперь цикл прохода для d выведет 10 элементов, что и показано в комментарии.

В следующей таблице последовательные классы обозначены как SEQ. Имейте в виду, что приведенные ниже возможности дополняют уже описанный интерфейс любого CAN.

Члены последовательных контейнеров STL

`SEQ::SEQ(n, v)`	`n` элементов со значением `v`
`SEQ::SEQ(b_it, e_it)`	от `b_it` до `e_it - 1`
`c.insert(w_it, v)`	вставка `v` перед `w_it`
`c.insert(w_it, v, n)`	вставка `n` копий `v` перед `w_it`
`c.insert(w_it, b_it, e_it)`	вставка значений от `b_it` до `e_it` перед `w_it`
`c.erase(w_it)`	стирает элемент, «прописанный» в `w_it`
`c.erase(b_it, e_it)`	стирает от `b_it` до `e_it`

Вот некоторые примеры использования этих членов:

```
double w[6] = {1.1, 1.2, 2.2, 2.3, 3.3, 4.4};
vector<double> v(15, 1.5);      //15 элементов со значением 1.5
deque<double> d(w + 2, w + 6); //использует от 2.2 до 4.4
d.erase(d.begin() + 2);         //стирает 3-ий элемент
v.insert(v.begin() + 1, w[3]); //вставляет w[3]
```

9.8.2. Ассоциативные контейнеры

Ассоциативные контейнеры — это множества, отображения, мультимножества и мультиотображения. Они содержат доступные по ключу элементы и упорядочивающее отношение `Compare`, являющееся сравнивающим объектом ассоциативного контейнера.
```

В файле stl_age.cpp

```
//Ассоциативные контейнеры – поиск возраста

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string, int, less<string> > name_age; //имя и возраст

  name_age["Пол,Лаура"] = 7;
  name_age["Долсберри,Бетти"] = 39;
  name_age["Пол,Таня"] = 14;
  cout << "Лауре  "
       << name_age["Пол,Лаура"]
       << " лет." << endl;
}
```

Отображение `name_age` — это ассоциативный массив, в котором ключом является тип `string`. Объект `Compare` — это `less<string>`.

В следующей таблице, описывающей интерфейс ассоциативных классов, они обозначены как `ASSOC`. Имейте в виду, что приведенные ниже возможности дополняют ужс описанный интерфейс любого CAN.

Определения ассоциативных контейнеров STL

`ASSOC::key_type`	тип ключа поиска
`ASSOC::key_compare`	тип сравнивающего объекта
`ASSOC::value_compare`	тип для сравнения `ASSOC::value_type`

Инициализация ассоциативных контейнеров обеспечивается несколькими стандартными конструкторами.

Ассоциативные конструкторы STL

`ASSOC()`	конструктор по умолчанию, используется `Compare`
`ASSOC(cmp)`	конструктор, использующий `cmp` как сравнивающий объект
`ASSOC(b_it, e_it)`	использует элементы в диапазоне от `b_it` до `e_it` применяется `Compare`
`ASSOC(b_it, e_it, cmp)`	использует элементы в диапазоне от `b_it` до `e_it` и `cmp` как сравненивающий объект

Что отличает эти конструкторы от конструкторов последовательных контейнеров, так это использование.

STL: функции-члены вставки и удаления	
`c.insert(t)`	если ни один из существующих элементов не имеет такого же ключа, как `t`, вставляет `t`; возвращает пару `<iterator, bool>` `c` `bool`, имеющим значение `true`, если `t` отсутствовал
`c.insert(w_it, t)`	вставляет `t` `c` `w_it` в качестве начальной позиции поиска; терпит неудачу в множествах и отображениях, если ключевое значение уже присутствует; возвращает позицию вставки
`c.insert(b_it, e_it)`	*вставляет диапазон элементов*
`c.erase(k)`	стирает элементы, ключевое значение которых равно `k`, возвращая количество стертых элементов
`c.erase(w_it)`	стирает указываемый элемент
`c.erase(b_it, e_it)`	стирает диапазон элементов

Вставка выполняется, если ни один из элементов с таким же ключом еще не присутствует. Вот некоторые примеры использования этих членов:

```
int m[4] = {1, 2, 3, 4};
set<int, less<int> > s                  //множество целых,
                                        //упорядоченное
                                        //с помощью less
set<int, less<int> > t(m, m + 4);       //используется 1, 2, 3, 4
s.insert(3);                            //помешает 3 в множество s
t.insert(3);                            //не вставляется, так как
                                        //в множестве t уже есть 3
s.erase(2);                             //в s нет такого элемента
t.erase(4);                             //t теперь содержит 1, 2, 3
```

Дополнительные функции-члены можно найти в разделе E.1.2, «Ассоциативные контейнеры», на стр. 423.

9.8.3. Адаптеры контейнеров

Классы *адаптеров контейнеров* (container adaptors) — это контейнерные классы, которые изменяют имеющиеся контейнеры с тем, чтобы обеспечить иное открытое поведение на основе существующей реализации. Три предлагаемых адаптераконтейнеров — это `stack` (стек), `queue` (очередь) и `priority_queue` (приоритетная очередь).

Стек может быть получен (адаптирован) из вектора, списка и двусторонней очереди. Он нуждается в реализации, поддерживающей операции `push`, `pop` и `top`.

STL: функции адаптированного стека	
`void push(const value_type& v)`	помещает `v` в стек
`void pop()`	убирает верхний элемент из стека
`value_type& top() const`	возвращает верхний элемент из стека
`bool empty() const`	возвращает истину, если стек пуст
`size_type size() const`	возвращает число элементов в стеке
`operator==` и `operator <`	равенство и лексикографическое меньше чем

Очередь может быть получена из списка или двусторонней очереди. Она нуждается в реализации, поддерживающей операции `empty`, `size`, `front`, `back`, `push_back` и `pop_front`. Это — структура данных «первый вошел — первый вышел».

STL: функции адаптированной очереди	
`void push(const value_type& v)`	помещает `v` в конец очереди
`void pop()`	удаляет первый элемент из очереди
`value_type& front() const`	возвращает первый элемент очереди
`value_type& back() const`	возвращает последний элемент очереди
`bool empty() const`	возвращает истину, если очередь пуста
`size_type size() const`	возвращает число элементов в очереди
`operator==` и `operator <`	равенство и лексикографическое меньше чем

Давайте адаптируем стек из реализации `vector`. Обратите внимание, как АТД из STL заменяют наши отдельно разработанные реализации типов.

В файле stl_stak.cpp

```
//Адаптируем стек из вектора

#include <iostream>
#include <stack>
#include <vector>
#include <string>
using namespace std;

int main()
{
  stack<string, vector<string> > str_stack;
  string quote[3] =
```

```
    {"Громче всех скрипит именно то колесо,\n",
     "которое смазывают\n",
     "Джош Биллингс \n" };

  for (int i = 0; i < 3; ++i)
    str_stack.push(quote[i]);
  while (!str_stack.empty()) {
    cout << str_stack.top();
    str_stack.pop();
  }
}
```

9.9. Итераторы

Перемещение по контейнерам производится с помощью итератора. Итератор может рассматриваться как усовершенствованный тип указателей. Итератор является шаблоном, который инстанцируется типом контейнерного класса, итерируемого им. Существует пять типов итераторов: ввода, вывода, прохода вперед, двусторонние и произвольного доступа. Не все типы итераторов могут быть доступны для данного контейнерного класса. Например, итератор произвольного доступа доступен для векторов, но не для отображений.

Итераторы ввода (input iterators) поддерживают операции равенства, разыменования и автоинкремента. Итераторы, отвечающие этим условиям, могут использоваться для однопроходных алгоритмов, которые читают значения структуры данных в одном направлении. Специальным случаем итератора ввода является `istream_iterator`.

Итераторы вывода (output iterators) поддерживают разыменование, допустимое только с левой стороны присваивания, и автоинкремент. Итераторы, отвечающие этим условиям, могут использоваться для однопроходных алгоритмов, которые пишут значения в структуры данных в одном направлении. Специальным случаем итератора вывода является `ostream_iterator`.

Итераторы прохода вперед (forward iterators) поддерживают все операции итераторов ввода-вывода и, кроме того, позволяют без ограничений применять присваивание. Благодаря этому можно сохранять позицию внутри структуры данных при многократных проходах. Таким образом, с помощью итератора прохода вперед можно написать общий однонаправленный многопроходный алгоритм.

Двусторонние итераторы (bidirectional iterators) поддерживают все операции итераторов прохода вперед, а также автоинкремент и автодекремент. Таким образом, с помощью двустороннего итератора можно написать общий двунаправленный многопроходный алгоритм.

Итераторы произвольного доступа (random acces iterators) поддерживают все операции двусторонних итераторов, а также арифметические адресные операции, такие как индексирование. Кроме того, итераторы произвольного доступа поддерживают операции сравнения. Таким образом, с помощью итераторов произвольного доступа можно написать такие алгоритмы, как quicksort (быстрая сортировка), которые требуют эффективного произвольного доступа.

Контейнерные классы и алгоритмы диктуют выбор категории доступного или необходимого итератора. Так, векторный контейнер допускает итераторы произволь-

ного доступа, а список — нет. Сортировка обычно нуждается в итераторе произвольного доступа, а для поиска нужен лишь итератор ввода.

9.9.1. Итераторы istream_iterator и ostream_iterator

Итератор `istream_iterator` происходит от итератора ввода и работает исключительно с чтением из потоков. Итератор `ostream_iterator` происходит от итератора вывода и работает исключительно с записью в потоки. Напишем программу, которая запрашивает пять чисел, считывает их и вычисляет их сумму, и в которой ввод-вывод использует эти итераторы. Шаблон для `istream_iterator` инстанцируется как <*тип, расстояние*>. Это расстояние обычно задается как `ptrdiff_t`. Как определено в *cstddef* или *stddef.h*, это целый тип, представляющий разницу между двумя значениями указателей.

В файле stl_io.cpp

```
//Использование istream_iterator и ostream_iterator

#include <iterator>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> d(5);
   int i, sum;
   istream_iterator<int, ptrdiff_t> in(cin);
   ostream_iterator<int> out(cout, "\t");

   cout << "введите 5 чисел " << endl;
   sum = d[0] = *in;      //ввод первого значения
   for(i = 1; i < 5; ++i) {
      d[i] = *++in;       //ввод остальных значений
      sum += d[i];
   }
   for(i = 0; i < 5; ++i)
      *out = d[i] ;       //вывод последовательных значений
   cout << " Сумма = " << sum << endl;
}
```

Итератор `istream_iterator` инстанцируется типом `int` и параметром `ptrdiff_t`. Тип `ptrdiff_t` является типом-расстоянием, который итератор использует для перехода к следующему элементу. В приведенном выше объявлении конструктору `in` передается входной поток `cin`. Оператор автоинкремента продвигает `in` и читает следующее значение типа `int` из указанного входного потока. Конструктору итератора вывода `out` передается выходной поток `cout` и ограничитель `"\t"` типа `char*`. То есть в поток `cout` после каждого записанного целого значения будет вставляться символ табуляции. В этой программе при разыменовании итератора `out` присвоенное целое значение записывается в `cout`.

9.9.2. Адаптеры итераторов

Итераторы могут быть адаптированы для обеспечения обратного просмотра и просмотра со вставкой. Обратные итераторы изменяют порядок итерации на противоположный; с помощью итераторов вставки вместо обычного режима перезаписи выполняется вставка. В следующем примере для прохода по последовательности использован обратный итератор.

В файле stl_iadp.cpp

```
//Использование обратного итератора

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

template <class ForwIter>
void print(ForwIter first, ForwIter last, const char* title)
{
   cout << title << endl;
   while ( first != last)
      cout << *first++ << '\t';
   cout << endl;
}

int main()
{
   int  data[3] = { 9, 10, 11};
   vector<int> d(data, data + 3);
   vector<int>::reverse_iterator p = d.rbegin();

   print(d.begin(), d.end(), "Исходный");
   print(p, d.rend(), "Обращенный");
}
```

Эта программа использует обратный итератор для изменения направления, в котором функция `print()` выводит элементы вектора `d`.

Другие алгоритмы из библиотеки *iterator* обсуждаются в разделе Е.2.4, «Адаптеры итераторов», на стр. 427.

9.10. Алгоритмы

Библиотека алгоритмов STL содержит четыре категории алгоритмов.

Категории библиотеки алгоритмов STL

- алгоритмы сортировки
- не изменяющие последовательность алгоритмы
- изменяющие последовательность алгоритмы
- численные алгоритмы

Эти алгоритмы обычно используют итераторы для доступа к контейнерам, инстанцированным заданным типом. Полученный код может состязаться в эффективности со специально созданным кодом.

9.10.1. Алгоритмы сортировки

Алгоритмы сортировки включают общую сортировку, слияние, лексикографическое сравнение, перестановку, двоичный поиск и некоторые другие сходные операции. Эти алгоритмы имеют версии, использующие либо `operator<()`, либо объект `Compare`. Часто они нуждаются в итераторе произвольного доступа.

Следующая программа использует функцию быстрой сортировки `sort()` из STL.

В файле stl_sort.cpp

```
//Использование sort() из STL

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

const int N = 5;

int main()
{
   int  d[N], i, *e = d + N;

   for (i = 0; i < N; ++i)
     d[i] = rand();
   sort(d, e);
   for (i = 0; i < N; ++i)
     cout << d[i] << '\t';
}
```

Это простое применение алгоритма библиотеки `sort`, оперирующего встроенным массивом `d[]`. Заметьте, как значения обычного указателя могут быть использованы в качестве итераторов.

Вот библиотечный прототип для алгоритма сортировки:

- ```
  template<class RandAcc>
  void sort(RandAcc b, RandAcc e);
  ```

Это алгоритм быстрой сортировки элементов от `b` до `e`. Итератор `RandAcc` должен быть итератором произвольного доступа.

Другие прототипы алгоритмов можно найти в разделе Е.3.1, «Алгоритмы сортировки», на стр. 428.

9.10.2. Не изменяющие последовательность алгоритмы

Неизменяющие алгоритмы не модифицируют содержимое контейнеров, с которыми они работают. Типичная подобная операция — поиск в контейнере конкретного элемента и возвращение его позиции.
  ```

В следующей программе неизменяющая библиотечная функция `find()` используется для обнаружения элемента `t`.

В файле stl_find.cpp

```
//Использование функции поиска

#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

int main()
{                                        //слова
   string words[5] = { "мой", "пой", "дрыг", "твой", "прыг"};
   string* where;

   where = find(words, words + 5, "пой");
   cout << *++where << endl;             //дрыг
   sort(words, words + 5);
   where = find(words, words + 5, "пой");
   cout << *++where << endl;             //прыг
}
```

Здесь `find()` используется для поиска позиции слова «пой». До и после сортировки массива `words[]` выводится слово, идущее за «пой».

Вот библиотечные прототипы для двух алгоритмов поиска:

- ```
  template<class InputIter, Class T>
  InputIter find(InputIter b, InputIter e, const T& t);
  ```

Ищется позиция `t` в диапазоне от `b` до `e`.

- ```
  template<class InputIter, Class Predicate>
  InputIter find(InputIter b, InputIter e, Predicate p);
  ```

В диапазоне от `b` до `e` ищется позиция первого элемента, который сделает предикат (predicate) истинным, в противном случае возвращается позиция `e`.

Другие прототипы неизменяющих алгоритмов даны в разделе E.3.2, «Не изменяющие последовательность алгоритмы», на стр. 431.

9.10.3. Изменяющие последовательность алгоритмы

Изменяющие алгоритмы могут модифицировать содержимое контейнеров, с которыми они работают. Типичная подобная операция — это обращение содержимого контейнера.

В следующей программе используются изменяющие библиотечные функции `reverse()` и `copy()`.

В файле stl_revr.cpp

```
//Использование изменяющих функции reverse и copy

#include <iostream>
```

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

int main()
{
   string first_names[5] = {"лаура", "айра",
        "болтушка", "дебра", "дед"};
   string last_names[5] = {"пол", "пол",
        "долсберри", "долсберри", "мороз"};
   vector<string> names(first_names, first_names + 5);
   vector<string> names2(10);
   vector<string>::iterator p;

   copy(last_names, last_names + 5, names2.begin());
   copy(names.begin(), names.end(), names2.begin() + 5);
   reverse(names2.begin(), names2.end());
   for (p = names2.begin(); p != names2.end(); ++p)
     cout << *p <<'\t';
}
```

Первый вызов изменяющей функции `copy()` помещает `last_names` в контейнер-вектор `names2`. Второе обращение к функции `copy()` копирует в `first_names` то, что было использовано при создании вектора `names`. Функция `reverse()` обращает все элементы, которые затем выводятся.

Вот библиотечные прототипы для двух алгоритмов копирования:

- `template<class InputIter, class OutputIter>`
  `OutputIter copy(InputIter b1, InputIter e1, OutputIter b2);`

Это алгоритм копирования элементов от `b1` до `e1`. Копия размещается начиная с `b2`. Возвращаемая позиция является концом копии.

- `template<class BidiIter1, class BidiIter2>`
  `BidiIter2 copy_backward(BidiIter1 b1, BidiIter1 e1, BidiIter2 b2);`

Это алгоритм копирования элементов от `b1` до `e1`. Копия размещается начиная с `b2`. Копирование, запускаемое «наоборот», от `e1` к `b2`, выполняется в обратном направлении («левее» `b2`). Возвращается позиция `b2 - (e1 - b1)`.

Другие алгоритмы даны в разделе Е.3.3, «Изменяющие последовательность алгоритмы», на стр. 432.

9.10.4. Численные алгоритмы

Численные алгоритмы включают суммирование, скалярное произведение и смежную разность.

В следующей программе численная функция `accumulate()` выполняет суммирование вектора, а `inner_product()` подсчитывает скалярное произведение.

В файле stl_numr.cpp

```
//Суммирование и перемножение векторов

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

int main()
{
  double v1[3] = {1.0, 2.5, 4.6},
         v2[3] = {1.0, 2.0, -3.5};
  double sum, inner_p;

  sum = accumulate(v1, v1 + 3, 0.0);
  inner_p = inner_product(v1, v1 + 3, v2, 0.0);
  cout << "Сумма = " << sum
       << ", скалярное произведение = " << inner_p << endl;
}
```

Эти функции ведут себя так, как должны вести себя численные типы, для которых определены операции + и *.

Вот библиотечные прототипы для двух накапливающих алгоритмов:

- `template<class InputIter, class T>`
  `T accumulate(InputIter b, InputIter e, T t);`

Это стандартный алгоритм накопления, причем за начальное значение суммы принимается `t`. К этой сумме последовательно добавляются элементы из диапазона от `b` до `e`.

- `template<class InputIter, class T, class BinOp>`
  `T accumulate(InputIter b, InputIter e, T t, BinOP bop);`

Это алгоритм накопления, причем начальное значение равно `t`. К этому значению последовательно «прибавляются» элементы из диапазона от `b` до `e`, но вместо сложения применяется заданная бинарная операция `bop`.

Другие алгоритмы даны в разделе E.3.4, «Численные алгоритмы», на стр. 434.

9.11. Функции

Для дальнейшего освоения библиотеки STL полезны объекты-функции. Например, многие из рассматривавшихся ранее численных функций предполагают применение + или *, но также имеет форму, в которой в качестве аргумента может передаваться заданный пользователем бинарный оператор. Ряд объектов-функций можно найти в библиотеке *function*, но их можно создать и самостоятельно. Объекты-функции — это классы, в которых определен `operator()`. Они являются встроенными[1] и компилируются так, чтобы получить эффективный код.

[1] Речь, разумеется, идет о встраивании функции в место вызова (inline), а не о том, что некая конструкция встроена в язык или в его реализацию (built-in). — *Примеч. перев.*

В файле stl_func.cpp

```
//Использование объекта-функции minus<int>

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

int main()
{
   double v1[3] = {1.0, 2.5, 4.6}, sum;

   sum = accumulate(v1, v1 + 3, 0.0, minus<int>());
   cout << "Сумма = " << sum << endl;    //sum = -7
}
```

Накопление выполнено с помощью «целого минуса» в качестве бинарной операции над массивом `v1[]`. Поэтому значения с двойной точностью обрезаются, и результат равен –7.

Существует три вида объектов-функций, как показано в следующем списке.

Виды объектов-функций

- Арифметические объекты
- Сравнивающие объекты
- Логические объекты

В следующей таблице кратко перечислены некоторые объекты-функции из библиотеки *function* и их назначение.

Объекты-функции STL

`template <class T>`
`struct plus<T>`
`(плюс)`
	арифметическая функция над типом `T`; также: `minus`, `times`, `devides`, `modulus` (минус, умножить, разделить, разделить по модулю)

`template <class T>`
`struct greater<T>`
`(больше)`
	функция сравнения над типом `T`; также: `less`, `equal_to`, `greater_equal`, `less_equal` (меньше, равно, больше или равно, меньше или равно)

`template <class T>`
`struct logical_and<T>`
`(И)`
	логическая функция над типом `T`; также: `logical_or`, `logical_not` (логическое ИЛИ, логическое НЕ)

Арифметические объекты часто используются в численных алгоритмах, например в `accumulate()`. Сравнивающие объекты нередко применяются в алгоритмах сортировки, таких как `merge()`. Расширенные таблицы для трех типов объектов-функций даны в разделе E.4, «Функции», на стр. 435.

9.12. Адаптеры функций

Адаптеры функций делают возможным создание объектов-функций с помощью адаптирования.

Адаптеры функций

- отрицающий адаптер[1] — для отрицания предикативных объектов
- связывающий адаптер[2] — для связывания аргументов функции
- адаптеры для указателя на функцию[3]

В следующем примере мы используем связывающую функцию `bind2nd` для превращения начальной последовательности значений в последовательность удвоенных значений.

В файле stl_adap.cpp

```
//Использование адаптера функции bind2nd

#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
using namespace std;

template <class ForwIter>
void print(ForwIter first, ForwIter last,
          const char* title)
{
   cout << title << endl;
   while ( first != last)
          cout << *first++ << '\t';
   cout << endl;
}

int main()
{
   int data[3] = { 9, 10, 11};

   print(data, data + 3, "Исходные значения");
   transform(data, data + 3, data,
           bind2nd(times<int>(), 2));
   print(data, data + 3, "Новые значения");
}
```

Адаптеры также обсуждаются в разделе E.4.1, «Адаптеры функций», на стр. 436.

---

[1] Иногда такие адаптеры называют *негаторами* — *Примеч. перев.*

[2] Иногда такие адаптеры называют *связывателями* — *Примеч. перев.*

[3] Страуструп выделяет также четвертый тип адаптеров — адаптер для функций-членов. — *Примеч. перев.*

В конце приложения Е также содержатся сведения о распределителях памяти (см. раздел Е.5, «Распределители памяти», на стр. 437) и о библиотеке *string* (см. раздел Е.6, «Строковая библиотека», на стр. 438). Здесь мы не будем обсуждать эти темы.

9.13. Практические замечания

В настоящее время во многих реализациях шаблонов C++ проводится различие между тем, что может быть параметром шаблона функции и тем, что может быть параметром шаблона класса. Функции допускают лишь аргументы классов. Более того, такие аргументы класса должны находиться в шаблоне функции как часть описания типа по крайней мере для одного из параметров функции.

Следующий код верен:

```
template <class TYPE>
void maxelement(TYPE a[], TYPE& max, int size);

template <class TYPE>
int find(TYPE* data);
```

А вот такой шаблон:

```
template <class TYPE>
TYPE convert(int i) { TYPE temp(i); return temp; }
```

раньше считался недопустимым, но теперь это тоже верно, в соответствии со стандартом, предложенным ANSI. По этому стандарту функция вызывается так:

```
convert<double>(i + j);    //недавно разрешенное явное
                           //инстанцирование функции
```

Поскольку раньше приведенный вызов функции считалсч ошибочным, подобный код может не работать на многих существующих системах. Нынешние компиляторы должны использовать аргументы в вызове функции для того, чтобы определить, какая фактическая функция будет создана. Обойти проблему можно с помощью создания класса, единственным членом которого является параметризованная статическая функция:

```
template <class TYPE>      //возможны и другие аргументы
class convert_it {
  static TYPE convert(int i)
    {TYPE temp(i); return temp; }
};
```

Резюме

1. В C++ для обеспечения параметрического полиморфизма используются шаблоны. Один и тот же код применяется с разными типами, причем тип является параметром тела кода.
2. И классы, и функции могут иметь несколько аргументов шаблона. В дополнение к аргументам-классам, определения шаблона класса могут включать в качестве аргументов шаблона выражения-константы, имена функций и символьные строки. Например, часто для задания размера чего-либо используется целый аргумент.

3. В тех случаях, когда обобщенная процедура не работает, может потребоваться специальная нешаблонная версия функции. Когда доступно несколько функций, для определения того, какую функцию использовать, применяется следующий алгоритм.

   Алгоритм выбора перегруженной функции

   1. Строгое соответствие для нешаблонных функций.
   2. Строгое соответствие для шаблонов функций.
   3. Обычное разрешение аргументов для нешаблонных функций.

4. Стандартная библиотека шаблонов (STL) является стандартной библиотекой C++, предоставляющей возможности обобщенного программирования для многих стандартных структур данных и алгоритмов.

5. Контейнеры подразделяются на два основных семейства: последовательные контейнеры и ассоциативные контейнеры. Последовательные контейнеры включают векторы (vectors), списки (lists) и двусторонние очереди (deques). Эти контейнеры упорядочиваются заданием последовательности элементов. Ассоциативные контейнеры включают множества (sets), мультимножества (multisets), отображения (maps), мультиотображения (multimaps) и содержат ключи для поиска элементов.

6. Классы адаптеров контейнеров — это контейнерные классы, которые изменяют имеющиеся контейнеры с тем, чтобы обеспечить иное открытое поведение на основе существующей реализации. Библиотека предоставляет три контейнерных адаптера — это `stack` (стек), `queue` (очередь) и `priority_queue` (приоритетная очередь).

7. Итератор может рассматриваться как усовершенствованный тип указателей. Существует пять типов итераторов: ввода, вывода, прохода вперед, двусторонние и произвольного доступа. Не все типы итераторов могут быть доступны для данного контейнерного класса. Например, итератор произвольного доступа доступен для векторов, но не для отображений.

8. Библиотека алгоритмов STL содержит четыре категории алгоритмов.

   Категории библиотеки алгоритмов STL

   - алгоритмы сортировки
   - не изменяющие последовательность алгоритмы
   - изменяющие последовательность алгоритмы
   - численные алгоритмы

   Эти алгоритмы обычно используют итераторы для доступа к контейнерам, инстанцированным заданным типом. Полученный код может состязаться в эффективности со специально созданным кодом.

Упражнения

1. Перепишите `stack<T>` из раздела 9.1, «Шаблонный класс stack», на стр. 245 так, чтобы он допускал целые значения для задания размера стека по умолчанию. Теперь в клиентском коде можно использовать такие объявления:

```
stack<int, 100> s1, s2;
stack<char, 5000> sc1, sc2, sc3;
```

Обсудите все «за» и «против» такой дополнительной параметризации.

2. Определите шаблон для стеков фиксированной длины, который размещает в памяти массив определяемого на этапе компиляции размера; в этом массиве будут храниться стековые значения.

3. Код

```
#define CUBE(X) ((X)*(X)*(X))
```

ведет себя иначе, чем код

```
template<class T> T cube (T x) { return x * x * x; }
```

Объясните различия между ними при вызове `cube(sqrt(7));`. В каких случаях два способа кодирования дадут различные результаты?

4. Используя следующую «заготовку», напишите обобщенную функцию `cycle()` и проверьте ее:

```
template<class TYPE>
void cycle(TYPE& a, TYPE& b, TYPE& c)
{
//замените значение a значением b, b — значением c,
//и c — значением a

}
```

5. Напишите обобщенную функцию, которая при заданных произвольном массиве и его размере выполняет ротацию элементов:

```
a[1] = a[0], a[2] = a[1], .....,
a[size - 1] = a[size - 2], a[0] = a[size - 1]
```

6. Напишите шаблон функции-члена

```
<class T> void vector<T>::print()
```

Эта функция печатает весь вектор.

7. Перепишите перегруженный оператор присваивания, с тем чтобы он стал более обобщенным

```
template <class T>
vector<T>& vector<T>::operator=(const vector<T>& v)
//должен допускать присваивание векторов разной длины,
//для левого аргумента должна очищаться и перераспределяться
//память, необходимо избегать случая a = a
```

8. Напишите обобщенную функцию, меняющую местами два `vector<T>` различных типов (см. раздел 9.4, «Параметризация класса vector», на стр. 251). Предполагается, что оба массива содержат элементы, допускающие преобразование при присваивании.

9. Используя `vector<T>` и связанный с ним итерирующий класс, напишите обобщенную процедуру внутренней сортировки вектора. Используйте свой метод, но не метод быстрой сортировки (quicksort). Сравните время выполнения процедуры с временем выполнения процедуры сортировки из библиотеки STL для векторов из 100, 1000 и 10000 элементов (см. раздел 9.4, «Параметризация класса vector», на стр. 251).

10. (*Проект*) Создайте параметрический строковый тип. Основной тип должен действовать в качестве контейнерного класса, который содержит объекты класса `T`. Нас интересует случай, когда в качестве `T` выступает `char`. Обычным ограничителем конца строки будет `0`. Определение класса может параметризовать и ограничитель. Стандартное поведение должно моделировать функции из *string.h*. Подобный тип существует в стандартной библиотеке *string*.

11. Функции сортировки — естественные кандидаты для параметризации. Ниже представлена обобщенная процедура сортировки методом пузырька (bubblesort).

```
template <class T>
void bubble(T d[], int how_many)
{
   T temp;

   for (int i = 0; i < how_many - 2; ++i)
      for (int j = 0; j < how_many - 1 - i; ++j)
         if (d[j] < d[j+1]) {
            temp = d[j];
            d[j] = d[j+1];
            d[j+1] = temp;
         }
}
```

Что произойдет, если `bubble` будет инстанцирована классом, в котором не определен `operator<()`?

12. Измените процедуру quicksort, используя трехэлементный образец для выбора элемента разбиения (см. раздел 9.5, «Параметризация quicksort», на стр. 254). Тем самым мы пытаемся получить в среднем лучшее разбиение за счет небольшого числа дополнительных сравнений на одну итерацию. Можете ли вы показать, что в худшем случае этот алгоритм все-таки демонстрирует поведение n^2.

13. С помощью генератора случайных чисел сгенерируйте 10000 целых от 0 до 9999. Поместите их в контейнер `list<int>` (см. раздел 9.7, «STL», на стр. 259). Вычислите и напечатайте серединное (median) значение. Какое значение ожидалось? Вычислите частоту появления каждого значения, иными словами выясните, сколько было сгенерировано нулей, сколько единиц, и так далее. Напечатайте значение, которое встречается чаще всего. Используйте вектор `vector<int>` для хранения частот.

14. Перепишите функцию `print(const list<double> &lst)` так, чтобы она стала шаблонной функцией, причем как можно более общей (см. раздел 9.7, «STL», на стр. 259).

15. Для списка `list<T>` напишите функцию-член:

```
iterator list<T>::insert(iterator w_it, T v);
```

Она вставляет `v` перед `w_it` и возвращает итератор, указывающий на вставленный элемент (см. раздел 9.7, «STL», на стр. 259).

16. Для списка `list<T>` напишите функцию-член:

```
void list<T>::erase(iterator w_it);
```

Она уничтожает элемент, на который указывает `w_it` (см. раздел 9.7, «STL», на стр. 259).

17. Напишите алгоритм для нахождения второго по величине элемента, хранящегося в произвольном контейнерном классе. Используйте контейнеры STL `vector<T>`, `list<T>` и `set<T>` для проверки того, что алгоритм работает независимо от контейнера. Напишите этот алгоритм, предполагая, что доступен итератор прохода вперед и имеется возможность сравнивать значения.

18. Мы хотим выполнить простое численное интегрирование с помощью контейнеров и алгоритмов STL. Напишите функцию, которая при заданной

```
double f(double x);
```

генерирует вектор из чисел с двойной точностью от `a` до `b` с интервалом в `s`. Затем суммируйте произведения `x` на `f(x)` в этом интервале (то есть `x` пробегает все сгенерированные значения).

Глава 10

Наследование

Наследование (inheritance) — это механизм получения нового класса из существующего. Существующий класс может быть дополнен или изменен для создания производного класса. Наследование — это мощный механизм повторного использования кода. С помощью наследования может быть создана иерархия родственных типов, которые совместно используют код и интерфейсы.

Многие полезные типы являются вариантами других, и часто бывает утомительно создавать для каждого из них один и тот же код. Производный класс наследует описание *базового* класса; затем он может быть изменен добавлением новых членов, изменением существующих функций-членов и изменением прав доступа. В полезности такой концепции можно убедиться на примере того, как таксономическая классификация компактно резюмирует большие объемы знаний. Скажем, располагая сведениями о понятии «млекопитающие» и зная, что и слон, и мышь являются млекопитающими, можно сделать их описания значительно более краткими. Основное базовое понятие содержит информацию о том, что млекопитающие — это теплокровные высшие позвоночные, вскармливающие своих детенышей молоком. Эта информация наследуется понятиями «мышь» и «слон», но подробно она изложена лишь однажды — в базовом понятии. В терминах C++ и слон, и мышь являются производными от базового класса млекопитающих.

C++ поддерживает *виртуальные функции-члены* (virtual member function). Это функции, объявленные в базовом классе и переопределенные в производных классах. Иерархия классов, которая определена открытым наследованием, создает родственный набор пользовательских типов, на все объекты которых может указывать указатель базового класса. Получая доступ к виртуальной функции с помощью этого указателя, C++ выбирает надлежащее определение функции на этапе выполнения. Объект, на который направлен указатель, должен нести в себе информацию о типе, так чтобы его можно было распознать динамически; в этом заключается характерная особенность кода на C++. Каждый объект «знает», как на него можно воздействовать. Такая форма полиморфизма называется *чистым полиморфизмом* (pure polymorphism).

Наследование полезно встраивать в программное обеспечение для того чтобы увеличить возможность повторного использования кода и обеспечить моделирование проблемы в естественных для нее понятиях. Вот ключевые элементы методологии объектно-ориентированного проектирования, связанные с наследованием:

Методология объектно-ориентированного проектирования

1. Выбор надлежащей совокупности типов.
2. Проектирование взаимосвязей между типами и применение наследования для использования общего кода.
3. Использование виртуальных функции для полиморфной обработки родственных объектов.

10.1. Производный класс

Класс можно сделать производным от существующего с использованием следующей формы:

```
class имя_класса :
      (public | protected | private)opt имя_базового_класса
{
  объявления членов
};
```

Ключевое слово `class` как всегда можно заменить ключевым словом `struct`, если подразумевается, что все члены открыты. Одним из аспектов производного класса является видимость (открытость) его членов-наследников. Ключевые слова `public`, `protected` и `private` используются для указания того, насколько члены базового класса будут доступны для производного.

Ключевое слово `protected` введено для того, чтобы сохранить сокрытие данных для членов, которые должны быть доступны из производного класса, но в других случаях действуют как закрытые (private). Это промежуточная форма доступа между `public` и `private`.

Рассмотрим создание класса, представляющего студентов колледжа или университета:

В файле student2.h

```
class student {
public:      //год обучения: новичок, второкурсник,
             //младший,старший, аспирант
  enum year { fresh, soph, junior, senior, grad };
  student(char* nm, int id, double g, year x);
             //имя, номер, средний балл, год
  void  print() const;
 protected:
  int   student_id;                    //номер студента
  double  gpa;                         //средний балл
  year  y;                             //год обучения
  char  name[30];                      //имя
};
```

Мы можем написать программу, которая помогла бы студенческому отделу вести учет студентов. Поскольку информация, хранящаяся в переменных класса `student`, описывает не-аспирантов, в ней отсутствуют важные сведения, необходимые для учета аспирантов. Эти сведения могут включать информацию о финансовой

поддержке, кафедре, теме диссертации. Наследование позволяет получить подходящий класс grad_student из базового класса student, как показано ниже.

В файле student2.h

```
class grad_student : public student {
public:   //вариант финансовой поддержки:
          //за преподавание, за исследования, стипендия, иное
  enum support { ta, ra, fellowship, other };
  grad_student( char* nm, int id, double g,
      year x, support t, char* d, char* th);
          //t – вариант финансовой поддержки,
          //d – кафедра, th – тема диссертации
  void  print() const;
protected:
  support  s;           //вариант финансовой поддержки
  char   dept[10];      //кафедра
  char   thesis[80];    //тема диссертации
};
```

В этом примере grad_student — производный класс, а student — базовый. Использование в заголовке производного класса ключевого слова public, следующего за двоеточием, означает, что защищенные и открытые (protected и public) члены класса student должны наследоваться как защищенные и открытые члены grad_student. Закрытые члены базового класса недоступны производному классу. Открытое наследование означает также, что производный класс grad_student является подтипом student. То есть аспирант является студентом, но студент не обязательно должен быть аспирантом. Такое отношение подтипов называется отношением *ISA*[1]. Иногда его также называют *интерфейсным наследованием* (interface inheritance).

Производный класс является модификацией базового класса; он наследует защищенные и открытые члены базового класса. Не могут наследоваться только конструкторы, деструктор базового класса и любые функции-члены operator=(). Так, в примере с классом grad_student наследуются следующие члены класса student: student_id, gpa, name, y и print(). Часто в производный класс добавляются новые члены в дополнение к членам базового класса. Это происходит и в случае с grad_student, который содержит три новых члена данных и переопределенную функцию-член print(). Эта функция *замещается* (override). Определения функций student::print() и grad_student::print() будут даны в следующем разделе. Производный класс может включать реализацию функций-членов, отличную от базового класса. Это не имеет ничего общего с перегрузкой, когда смысл одного и того же имени функции может быть различным для разных сигнатур.

Преимущества использования производных классов

- Код используется повторно. Тип grad_student использует существующий, проверенный код из student.

[1] Отношение *ISA* (иногда *is-a*) называется отношением «быть экземпляром». То есть всякий объект (экзе́мпляр) класса grad_student является и объектом (экземпляром) класса student. — *Примеч. перев.*

- Иерархия отражает взаимоотношения, свойственные проблемной области. Когда мы говорим о студентах, выделение аспирантов в отдельную группу заимствуется из реального мира и определяется тем, как аспиранты понимаются в этом мире.
- Различные полиморфные механизмы позволят клиентскому коду рассматривать `grad_student` в качестве подтипа `student`, что упростит клиентский код, сохраняя при этом преимущества разграничения подтипов.

10.2. Преобразования типов и видимость

Открытый производный класс является *подтипом* своего базового класса. Во многих случаях переменная производного класса может рассматриваться как переменная типа базового класса. Указатель, тип которого — «указатель на базовый класс», может указывать на объекты, имеющие тип производного класса.

Рассмотрим наш пример с классами `student` и `grad_student`. Сначала исследуем конструкторы базового и производного классов.

В файле student2.h

```
student::student(char* nm, int id, double g, year x)
   : student_id(id), gpa(g), y(x)
{
   strcpy(name, nm);
}
```

Конструктор базового класса выполняет несколько простых инициализаций. Затем он вызывает `strcpy()` для копирования имени студента.

```
grad_student::grad_student (char* nm, int id, double g,
   year x, support t, char* d, char* th)
   : student(nm, id, g, x), s(t)
{
   strcpy(dept, d);
   strcpy(thesis, th);
}
```

Заметьте, что здесь конструктор `student` вызывается как часть списка инициализаторов. Это вполне естественно и логично — для завершения создания объекта сначала надо создать объект базового класса.

Тип `grad_student` является открытым производным типом, базовый класс которого — `student`. В классе `student` члены `student_id` и `gpa` являются защищенными. Это делает их видимыми для производных классов, но для всех остальных они ведут себя как закрытые.

Ссылка на производный класс может быть неявно преобразована в ссылку на открытый базовый класс. Например:

```
grad_student gs("Моррис Пол", 200, 3.2564, grad, ta,
         "Фармакология", "Продажа лекарств");
student& rs = gs;
```

В данном случае переменная rs является ссылкой на student. Базовым классом grad_student является student. Поэтому такое преобразование ссылки справедливо.

Функции-члены print() реализованы следующим образом:

В файле student2.h

```
void student::print() const
{
   cout << name << " , " << student_id
        << " , " << y << " , " << gpa << endl;
}

void grad_student::print() const
{
   student::print();  //печатается информация базового класса
   cout << dept << " , " << s << '\n'
        << thesis << endl;
}
```

При вызове функции student::print() из grad_student::print() обязательно должен использоваться идентификатор, разрешающий область видимости. Именно так: student::print(). Иначе из-за рекурсивных вызовов grad_student::print() возникнет бесконечный цикл. Чтобы увидеть, какая из двух версий этих функций будет вызвана и продемонстрировать некоторые отношения преобразований между базовым и открытым производным классом, напишем простой тест:

В файле student2.cpp

```
//Проверим правила преобразования указателей

#include "student2.h"   //включаем наши объявления

int main()
{
   student  s("Мей Пол", 100, 3.425, student::fresh), *ps = &s;
   grad_student  gs("Моррис Пол", 200, 3.2564,
      student::grad, grad_student::ta, "Фармакология",
      "Продажа лекарств"), *pgs;
   ps -> print();          //student::print
   ps = pgs = &gs;
   pgs -> print();         //grad_student::print
   ps -> print();          //student::print
}
```

Эта программа объявляет переменные класса и указатели на них. Правило преобразования состоит в том, что указатель на открытый производный класс может быть неявно преобразован к указателю на его базовый класс. В нашем примере переменная-указатель ps может указывать на объекты обоих классов, а переменная-указатель pgs — только на объекты типа grad_student. Мы хотим выяснить, как различные присваивания указателей влияют на то, какая версия print() будет вызвана.

В первом случае инструкция

```
ps -> print();
```

вызовет `student::print()`. Переменная `ps` указывает на переменную `s` типа `student`. В инструкции множественного присваивания

```
ps = pgs = &gs;
```

оба указателя направлены на объект типа `grad_student`. Присваивание `ps` приводит к неявному преобразованию. Инструкция

```
pgs -> print();
```

вызывает `grad_student::print()`. Переменная `pgs` имеет тип «указатель на `grad_student`», и при вызове с объектом этого типа выбирается функция-член из этого класса. Второй раз инструкция

```
ps -> print();
```

вызывает `student::print()`. Тот факт, что этот указатель теперь указывает на переменную `gs` типа `grad_student` к делу не относится: в разделе 10.4, «Виртуальные функции», на стр. 289 мы объясним, как использовать виртуальные функции-члены для того, чтобы выбор вызываемой функции определялся на этапе выполнения, в зависимости от того, на что направлен указатель.[1]

10.3. Повторное использование кода: класс двоичного дерева

Закрытое наследование не носит характера отношения подтипов, или отношения *ISA*. При закрытом наследовании мы повторно используем код базового класса, но не предполагаем рассматривать объекты производного класса как экземпляры базового. Мы будем называть закрытое наследование *отношением LIKEA* (*подобный*). Оно также называется *наследованием реализации*, в противоположность наследованию интерфейса. Наследование реализации оказывается полезным при построении схем отношений классов в сложной программной системе. Поскольку закрытое (также как и защищенное) наследование не создает иерархии типов, оно имеет более ограниченное применение, нежели открытое наследование. При первом знакомстве с наследованием *не*открытое наследование (то есть защищенное или закрытое) можно пропустить.

Часто повторное использование кода — это все, что мы хотим от наследования. Покажем, как закрытое наследование используется при создании обобщенного контейнерного класса — двоичного дерева (см. раздел 9.6, «Параметризованное дерево двоичного поиска», на стр. 255). Эта древесная структура была также использована в качестве шаблонного класса. Шаблоны и наследование — это способы повторного использования кода, обладающие разными преимуществами. Обычно шаблоны проще проектировать; кроме того, исполняемый код, сгенерированный на их основе, работает быстрее, но занимает больше места.

[1] Компоновка вызовов функций (кроме виртуальных) осуществляется при компиляции. На этом этапе неизвестно, на объект какого именно класса (базового или производного) направлен полиморфный указатель (или ссылка). Соответственно, компилятор связывает вызов функции с тем ее вариантом, который отвечает классу, указанному в объявлении указателя (ссылки), а не тому, на объект которого «в данный момент» направлен указатель (ссылка). — *Примеч. перев.*

Класс будет хранить члены данных void*. Идея состоит в закрытом наследовании от этого класса, чтобы, так сказать, «разобщить» указатели void*.

В файле gentree2.h

```
//Обобщенные деревья двоичного поиска

typedef void*  p_gen;   //тип обобщенного указателя
int  comp(p_gen a, p_gen b);

class bnode {           //узел
private:
   friend  class gen_tree;
   bnode*  left;
   bnode*  right;
   p_gen   data;
   int   count;
   bnode(p_gen d, bnode* l, bnode* r) :
       data(d), left(l), right(r), count(1) {}
   friend void  print(bnode* n);
};

class gen_tree {        //дерево
public:
   gen_tree() { root = 0; }
   void  insert(p_gen d);
   p_gen  find(p_gen d) const { return (find(root, d)); }
   void  print() const { print(root); }
protected:
   bnode*  root;        //корень
   p_gen  find(bnode* r, p_gen d) const;
   void  print(bnode* r) const;
};
```

Отдельные узлы в этом двоичном дереве хранят обобщенный указатель data и целую count, которая будет подсчитывать повторяющиеся вхождения. Указатель data будет соответствовать типу указателя в производном классе. Дерево будет представлять собой дерево двоичного поиска, в котором узлы с меньшим значением будут храниться слева, а с большим значением — справа. Нам нужен способ для сравнения значений, подходящий для конкретного производного типа. Мы используем дружественную функцию comp(), которая является другом bnode и будет написана надлежащим образом для производного класса.

Функция insert() помещает узлы в дерево; она должна находить в дереве позицию для очередного узла.

```
void gen_tree::insert(p_gen d)
{
   bnode*  temp = root;
   bnode*  old;

   //template<class T> void gen_tree<T>::insert(T d)
}
```

Тело функции `insert()` — такое же как и в разделе 9.6, «Параметризованное дерево двоичного поиска», на стр. 255, за исключением того, что исчезают ссылки на <T> в инструкциях, содержащих оператор `new()`.

Функция `p_gen find(bnode* r, p_gen d)` ищет в поддереве с корнем `r` информацию, представленную `d`.

```
p_gen gen_tree::find(bnode* r, p_gen d) const
{
   .....
}
```

Тело функции `find()` *— такое же как в разделе* 9.6, «Параметризованное дерево двоичного поиска», на стр. 255.

Функция `print()` — это тоже стандартная рекурсия. В каждом узле применяется внешняя функция `::print()`.

```
void gen_tree::print(bnode* r) const
{
   if (r != 0) {
      print (r -> left);
      ::print(r);
      print (r -> right);
   }
}
```

Теперь создадим производный класс, который в качестве членов данных сможет хранить указатели на `char`.

В файле gentree2.cpp

```
#include  "gentree2.h"
#include  <cstring>  //для старых систем: string.h
using namespace std;

class s_tree : private gen_tree {  //дерево строк
public:
   s_tree() { }
   void  insert(char* d) { gen_tree::insert(d); }
   char*  find(char* d) const
     { return static_cast<char*>(gen_tree::find(d)); }
   void  print() const { gen_tree::print(); }
};
```

Функция вставки `gen_tree::insert` из базового класса принимает в качестве аргумента обобщенный указатель. Функция вставки `s_tree::insert` производного класса принимает в качестве аргумента указатель на `char`. Таким образом, в производном классе `s_tree` функция

```
void  insert(char* d) { gen_tree::insert(d); }
```

использует неявное преобразование `char*` к `void*`.

Нам нужна функция, выполняющая сравнение:

```
int comp(p_gen i, p_gen j)
{
   return (strcmp(static_cast<char*>(i),
                  static_cast<char*>(j) ));
}
```

Кроме того, нужна внешняя функция `print()` для рекурсивного использования в `s_tree::print()` выводящей дерево целиком; она должна уметь правильно печатать значения, хранящиеся в отдельном узле.

```
void print(bnode* n)
{
   cout << static_cast<char*>(n -> data) << '\t' ;
   cout << n -> count << '\t';
}
```

Заметьте, что там, где соответствующий шаблонный код использовал шаблонный параметр, в этой версии программы для `gen_tree` применяется тип обобщенного указателя `p_gen`:

Методика шаблонов проще и более эффективна на этапе выполнения. Она проще, потому что для инстанцирования необходим единственный фактический тип, помещаемый в объявление шаблона. При наследовании же необходимо наследовать весь интерфейс, подставляя надлежащие типы. Методика шаблонов более эффективна на этапе выполнения, потому что часто можно избежать ненужных «косвенностей». С другой стороны, наследование позволяет, если необходимо, разработать специфический код для каждого типа. Наследование не приводит к большим модулям объектного кода. Помните, что каждое инстанцирование шаблона компилируется в объектный код.

10.4. Виртуальные функции

Перегруженная функция-член вызывается с учетом алгоритма соответствия типов, в который входит правило соответствия неявного аргумента объекту данного типа класса. Все это известно на этапе компиляции и позволяет компилятору напрямую выбирать надлежащий член. Как станет очевидно, было бы неплохо динамически (на этапе выполнения) выбирать соответствующую функцию-член среди функций базового и производного классов. Ключевое слово `virtual` служит спецификатором функции, и как раз и предоставляет подобный механизм, но оно может применяться для изменения объявлений только функций-членов. Сочетание виртуальных функций и открытого наследования станет для нас наиболее обобщенным и гибким способом построения программного обеспечения. Это — форма чистого полиморфизма.

Обычная виртуальная функция должна представлять собой исполняемый код. При вызове семантика ее точно такая же, как и у остальных функций. В производном типе она может замещаться (переписываться, переопределяться), и прототип производной функции должен иметь сигнатуру и возвращаемый тип, совпадающие с базовой. Выбор того, какое определение функции вызвать для виртуальной функции, происходит динамически (на этапе выполнения). Типичный случай — это когда базовый класс включает виртуальную функцию, а производные классы имеют свои

версии этой функции. Указатель на базовый класс может указывать либо на объект базового класса, либо на объект производного класса. Выбор функции-члена будет зависеть от класса объекта, на который фактически (в момент выполнения) направлен указатель, а не от типа указателя. При отсутствии члена производного типа используется виртуальная функция базового класса. Обратите внимание на различие между выбором надлежащей замещенной виртуальной функции и выбором перегруженной функции-члена. Перегруженная функция-член выбирается на этапе компиляции на основе сигнатуры; перегруженные функции могут иметь различные возвращаемые типы. Виртуальная функция выбирается на этапе выполнения на основе типа объекта, который передается ей в качестве аргумента-указателя `this`. Кроме того, будучи однажды объявленной как виртуальная, функция сохраняет это свойство во всех переопределениях в производных классах. В производных классах не обязательно использовать модификатор `virtual`.

Рассмотрим следующий пример.

В файле virt_sel.cpp

```
//Выбор виртуальной функции

class B {
public:
   int  i;
   virtual void  print_i() const
      { cout << i << " внутри B" << endl; }
};

class D : public B {
public:
    //тоже виртуальная
   void  print_i() const
      { cout << i << " внутри D" << endl; }
};

int main()
{
    B  b;
    B* pb = &b;          //указывает на объект B
    Dd;

    d.i = 1 + (b.i = 1);
    pb -> print_i();     //вызов B::print_i()
    pb = &d;             //указывает на объект D
    pb -> print_i();     //вызов D::print_i()
}
```

Вот что выведет эта программа:

```
1 внутри B
2 внутри D
```

Сравните это поведение с программой *student* из раздела 10.2, «Преобразования типов и видимость», на стр. 284. Там выбор функции `print()` был основан на типе указателя, известном на этапе компиляции. Здесь же функция `print_i()` выбирается на основе того, на что направлен указатель. При этом выполняется другая версия `print_i()`. Говоря языком ООП, объекту *посылается сообщение* `print_i()`, и задействует собственную версию соответствующего метода. Так, тип базового указателя не определяет выбор метода (функции). Объекты другого класса обрабатываются другими функциями, устанавливаемыми на этапе выполнения. Средства, позволяющие реализовывать АТД, наследование, а также возможность динамической обработки объектов, являются неотъемлемой частью ООП.

Виртуальные функции и перегрузка функций-членов вызывают путаницу. Рассмотрим следующий пример.

В файле virt_err.cpp

```
class B {
public:
   virtual void foo(int);
   virtual void foo(double);
};

class D : public B {
public:
   void foo(int);
};

int main()
{
   D d;
   B b, *pb = &d;

   b.foo(9);         //выбирает B::foo(int);
   b.foo(9.5);       //выбирает B::foo(double);
   d.foo(9);         //выбирает D::foo(int);
   d.foo(9.5);       //выбирает D::foo(int);
   pb -> foo(9);     //выбирает D::foo(int);
   pb -> foo(9.5);   //выбирает B::foo(double);
}
```

Функция-член базового класса `B::foo(int)` замещается, а функция-член базового класса `B::foo(double)` скрыта в производном классе. В инструкции `d.foo(9.5)` значение с двойной точностью 9.5 преобразуется к целому значению 9. Для вызова скрытой функции-члена можно было бы использовать `d.B::foo(double)`.

Объявление идентификатора в области видимости скрывает все объявления этого идентификатора в объемлющих областях видимости. Базовый класс является объемлющей областью видимости для любого из своих производных классов. Это правило не зависит от того, были ли имена объявлены как `virtual`. Ограничения доступа (`private`, `protected`) не связаны с выбором функции. Если выбранная функция недоступна, это вызовет ошибку компиляции.

Только нестатические функции-члены могут быть виртуальными. Виртуальность наследуется. Так как функция производного класса автоматически виртуальна, наличие в нем ключевого слова `virtual` — обычно дело вкуса. Конструкторы не могут быть виртуальными, в отличие от деструкторов. Практически, каждый класс, имеющий виртуальную функцию, должен иметь виртуальный деструктор.

Виртуальные функции позволяют делать выбор на этапе выполнения. Рассмотрим приложение — систему автоматизированного проектирования, в котором должна вычисляться площадь фигур в проекте. Различные фигуры будут наследоваться из базового класса `shape` (фигура):

В файле shape2.cpp

```
class shape {                                   //фигура
public:
   virtual double  area() const { return 0; }   //площадь
   //virtual double area() задает поведение по умолчанию
protected:
   double  x, y;
};

class rectangle : public shape {                //прямоугольник
public:
   double  area() const { return (height * width); }
private:
   double  height, width;                       //высота, ширина
};

class circle : public shape {                   //окружность
public:
   double  area() const { return (PI * radius * radius);}
private:
   double radius;
};
```

В этой иерархии классов производные классы соответствуют простым и важным фигурам. Систему легко расширить созданием дополнительных производных классов. За вычисление площади отвечает производный класс.

Клиентский код, использующий полиморфное вычисление площади, выглядит так:

```
shape* p[N];

   .....
for (i = 0; i < N; ++i)
   tot_area += p[i] -> area();
```

Основное преимущество здесь заключается в том, что клиентский код не нуждается в изменении, если в систему добавляются новые фигуры. Изменение управляется локально и распространяется автоматически вследствие полиморфного характера клиентского кода.

10.5. Абстрактные базовые классы

Иерархия типов обычно имеет корневой класс, содержащий некоторое число виртуальных функций. Виртуальные функции обеспечивают динамическую типизацию. Виртуальные функции корневого класса часто являются фиктивными функциями. Они имеют пустое тело в корневом классе, но в производных классах им будет придан конкретный смысл. В C++ для этих целей введена *чисто виртуальная функция* (pure virtual function). Чисто виртуальная функция — это виртуальная функция, тело которой не определено. Она объявляется внутри класса следующим образом:

```
virtual прототип_функции = 0;
```

Чисто виртуальная функция используется для того, чтобы отложить выбор реализации функции. В терминологии ООП это называется *отложенным методом* (deferred method).

Класс, имеющий хотя бы одну чисто виртуальную функцию, называется *абстрактным классом* (abstract class). Полезно, чтобы корневой тип в иерархии классов был абстрактным классом. Он должен иметь основные общие свойства со своими производными классами, но сам не может использоваться для объявления объектов. Вместо этого он используется для объявления указателей, которые могут иметь доступ к подобъектам, произведенным из абстрактного класса.

Объясним эти идеи, разработав упрощенный вариант экологической модели. Изначально ООП разрабатывалось как методика моделирования с помощью языка моделирования Simula 67. Поэтому многие концепции ООП можно воспринимать как попытку смоделировать конкретную реальность.

Мир в нашем примере включает разные взаимодействующие формы жизни. Абстрактным базовым классом будет `living` (жизнь). Его интерфейс будет наследоваться разными формами жизни. В качестве архетипического хищника у нас будет выступать лиса, добычей ее будут кролики. Сами кролики питаются травой.

В файле predator.cpp

```
//Модель хищник-добыча с использованием класса living

const int  N = 40; //размер квадратного поля
     //состояние клетки: пусто, трава, кролик, лиса
     //STATES — количество различных состояний (пока их 4)
enum state { EMPTY , GRASS , RABBIT , FOX, STATES };

const int DRAB = 3, DFOX = 6, CYCLES = 5;

class living;                    //предварительное объявление
typedef living* world[N][N];     //мир

class living {                   //что живет на свете
public:
  virtual state  who() = 0;      //выяснение состояния (кто?)
  virtual living*  next(world w) = 0;   //что дальше?
protected:
  int  row, column;              //координаты поля
```

```
    void sums(world w, int sm[]);
};

void living::sums(world w, int sm[])
{
    int  i, j;

    sm[EMPTY] = sm[GRASS] = sm[RABBIT] = sm[FOX] = 0;
    for (i = -1; i <= 1; ++i)
        for ( j = -1; j <= 1; ++j)
            sm[w[row + i][column +j] -> who()]++;
}
```

Здесь две чисто виртуальных функции и одна обычная функция-член — `sums()`. Виртуальные функции влекут небольшие дополнительные издержки на этапе выполнения по сравнению с обычными функциями-членами. Поэтому мы используем их в наших реализациях, только когда это необходимо. В модели предусмотрены правила для определения того, кто продолжит существование в следующем цикле, в зависимости от популяцияй по соседству с заданным квадратом. Эти популяции вычисляются с помощью `sums()`. Все это напоминает «Жизнь» Конвея (Conway).

Иерархия наследования будет одноуровневой:

```
//здесь — хищник (лиса)
class fox : public living {
public:
    fox(int r, int c, int a = 0) : age(a)
        { row = r; column = c; }
    state  who() { return FOX; }   //отложенный метод для лис
    living*  next(world w);
protected:
    int  age;                      //возраст
};

//здесь — добыча (кролик)
class rabbit : public living {
public:
    rabbit(int r, int c, int a = 0) : age(a)
        { row = r; column = c; }
    state  who() { return RABBIT; }
    living*  next(world w);
protected:
    int  age;
};

//здесь — растительная жизнь (трава)
class grass : public living {
public:
    grass(int r, int c) { row = r; column = c; }
    state who() {return GRASS;}
```

```
    living* next(world w);
};

//здесь нет жизни (пусто)
class empty : public living {
public:
    empty(int r, int c) { row = r; column = c; }
    state  who() { return EMPTY; }
    living*  next(world w);
};
```

Обратите внимание, что данная схема позволяет с помощью последующих уровней наследования разрабатывать другие формы хищников, добычи и растительной жизни. Характеристики того, как каждая форма жизни будет себя вести, сосредоточены в принадлежащей ей `next()`.

Трава может поедаться кроликами. Если на соседних квадратах травы больше, чем кроликов, трава остается, в противном случае ее съедают (можете заменить это правило своими собственными, поскольку оно слишком уж ограничено и искусственно):

```
living* grass::next(world w)
{
    int  sum[STATES];

    sums(w, sum);
    if (sum[GRASS] > sum[RABBIT])     //едим траву
        return (new grass(row, column));
    else
        return (new empty(row, column));
}
```

Кролики умирают от старости, если их возраст превышает некий определенный предел DRAB, или их съедают, если по соседству имеется достаточное количество лис.

```
living* rabbit::next(world w)
{
    int  sum[STATES];

    sums(w, sum);
    if (sum[FOX] >= sum[RABBIT] )     //едим кроликов
        return (new empty(row, column));
    else if (age > DRAB)              //кролик состарился
        return (new empty(row, column));
    else
        return (new rabbit(row, column, age + 1));
}
```

Лисы же умирают от перенаселения или от старости:

```
living* fox::next(world w)
{
    int  sum[STATES];
```

```
    sums(w, sum);
    if (sum[FOX] > 5)          //ну и лис развелось!
      return (new empty(row, column));
    else if (age > DFOX)       //лиса состарилась
      return (new empty(row, column));
    else
      return (new fox(row, column, age + 1));
}
```

За пустые квадраты конкурируют разные формы жизни:

```
//как заполнить пустой квадрат
living* empty::next(world w)
{
  int  sum[STATES];

  sums(w, sum);
  if (sum[FOX] > 1)
    return (new fox(row, column));
  else if (sum[RABBIT] > 1)
    return (new rabbit(row, column));
  else if (sum[GRASS])
    return (new grass(row, column));
  else
    return (new empty(row, column));
}
```

Правила в различных версиях `next()` задают сложный (в разумных пределах) набор взаимодействий. Конечно, чтобы сделать мир более интересным, можно смоделировать другие варианты поведения, такие как половое размножение, когда животные имеют пол и могут спариваться.

Тип массива `world` является контейнером для форм жизни. Контейнер отвечает за свое текущее состояние. Он должен иметь «право собственности» на объекты `living`, чтобы размещать новые и уничтожать старые:

```
//начало: мир пуст
void init(world w)
{
  int  i, j;

  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
      w[i][j] = new empty(i,j);
}

//новый мир w_new вычисляется из старого w_old
void update(world w_new, world w_old)
{
  int  i, j;

  for (i = 1; i < N - 1; ++i) //за границы нельзя
```

```
      for (j = 1; j < N - 1; ++j)
        w_new[i][j] = w_old[i][j] -> next(w_old);
}

//очистка мира
void dele(world w)
{
  int  i, j;

  for (i = 1; i < N - 1; ++i)
    for (j = 1; j < N - 1; ++j)
      delete(w[i][j]);
}
```

Модель имеет миры even и odd (четный и нечетный), которые чередуются в качестве базиса для вычислений следующего цикла:

```
int main()
{
  world  odd, even;
  int  i;

  init(odd);  init(even);
  eden(even);                  //генерирует начальный мир - рай
  pr_state(even);              //выводит состояние райского сада
  for (i = 0; i < CYCLES; ++i) {  //моделирование
    if (i % 2) {
      update(even, odd);
      pr_state(even);
      dele(odd);
    }
    else {
      update(odd, even);
      pr_state(odd);
      dele(even);
    }
  }
}
```

Написание функций pr_state() и eden() оставим для упражнений (см. упражнение 16, на стр. 310).

10.6. Шаблоны и наследование

Вместе шаблоны и наследование являются исключительно мощной техникой повторного использования кода. Параметризованные типы могут использоваться повторно посредством наследования. Такое использование аналогично использованию наследования обычных типов. И шаблоны, и наследование — это механизмы для повторного использования кода, и оба они обеспечивают полиморфизм. Они являются двумя разными чертами C++ и комбинируются в различных фор-

мах. Шаблонный класс может быть произведен от обычного класса, обычный класс — инстанцированного шаблонного класса, шаблонный класс можно произвести от шаблонного класса. Каждая из этих возможностей ведет к различным отношениям.

В некоторых ситуациях шаблоны приводят к неприемлемым издержкам — избыточному размеру объектного модуля. Каждый инстанцированный шаблонный класс нуждается в своем откомпилированном объектном модуле. Рассмотрим класс `gen_tree` из раздела 10.3, «Повторное использование кода: класс двоичного дерева», на стр. 286. Он обеспечивал повторное использование, предоставляя код, легко преобразуемый с помощью наследования и приведения к полезным типам указателей. Недостаток типа `gen_tree` состоит в том, что каждый тип указателя нуждался в отдельном кодировании определения его класса. Этот недостаток можно устранить, используя шаблон для наследования базового класса:

```
//Базовый класс для обеспечения небольшого размера кода

template <class T>
class pointer_tree : private gen_tree {
public:
   pointer_tree() { }
   void insert(T* d) { gen_tree::insert(d); }
   T* find(T* d) const
      { return reinterpret_cast<T*>(gen_tree::find(d)); }
   void print() { gen_tree::print(); }
};
```

Объектный код для `gen_tree` относительно велик, но он используется только один раз. Интерфейс `pointer_tree<`*тип*`>` нуждается лишь в небольшом объектном модуле для каждого инстанцирования. Это существенно экономичнее по сравнению с решением на основе шаблонов, представленным ранее (см. раздел 9.6, «Параметризованное дерево двоичного поиска», на стр. 255).

Получение производного класса от инстанцированного шаблонного класса в основном не отличается от обычного наследования. В следующем примере мы повторно используем `stack<char>` в качестве базового класса для безопасного стека символов:

В файле stack_t2.cpp

```
//Безопасный стек символов

#include <assert.h>

class safe_char_stack : public stack<char> {
public:
   //проверка push и pop
   void  push(char c)
     { assert (!full()); stack<char>::push(c); }
   char  pop()
     {assert (!empty()); return (stack<char>::pop());}
};
```

Инстанцированный класс stack<char> создается и используется повторно с помощью safe_char_stack.

Этот пример можно обобщить до шаблонного класса:

В файле stack_t3.cpp

```
//Параметризованный безопасный стек
template <class TYPE>
class safe_stack : public stack<TYPE> {
public:
   void  push(TYPE c)
     { assert (!full()); stack<TYPE>::push(c); }
   TYPE  pop()
     {assert (!empty()); return (stack<TYPE>::pop());}
};
```

Важно отметить связь между базовым классом и производным классом. Оба нуждаются в одном и том же инстанцированном типе. Каждая пара базового и производного классов не зависит от всех остальных пар.

10.7. Множественное наследование

До сих пор примеры в тексте требовали лишь *одиночного наследования* (single inheritance): класс производится от единственного базового класса. Это может привести к цепочке наследований, когда класс B производится от класса A, класс C — от класса B, ... , а класс N — от класса M. В результате N завершает цепь, имея в основе A, B, ... , M. Однако эта цепочка не должна быть замкнутой — класс не может быть предком самого себя.

Множественное наследование (multiple inheritance) делает возможным получение производного класса от более чем одного базового класса. Синтаксис заголовка класса расширяется, чтобы можно было использовать список базовых классов с атрибутами доступа. Например:

```
class student {  //студент
   .....
};
class worker {   //рабочий
   .....
};
class student_worker: public student, public worker {
   .....        //работающий студент
};
```

В этом примере производный класс student_worker открыто наследует члены обоих базовых классов. Такие родительские отношения описываются *ориентированным ациклическим графом* (directed acyclic graph — DAG) наследования. Ориентированный ациклический граф наследования — это граф, узлы которого являются классами, а ориентированные ребра направлены от производных классов к базовым. Он

не может содержать циклов[1], так как никакой класс не может через цепочку наследований происходить сам от себя.

При производстве от различных классов членов с идентичными именами могут возникать неопределенности. Подобные наследования допускаются при условии, что пользователь не делает двусмысленных ссылок на такой член. Например:

```
class worker {
public:
   const int soc_sec;
   const char* name;
   .....
};

class student {
public:
   const char*name;
   .....
};

class student_worker: public student, public worker {
public:
   void print() { cout << "код соц. защиты: " << soc_sec <<
"\n";
      cout << name; ..... }        //ошибка
   .....
};
```

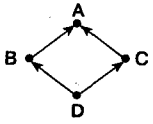
В теле функции `student_worker::print()` ссылка на `soc_sec` сработает, но ссылка на `name` двусмысленна. Двусмысленность можно преодолеть, правильно задав имя `name` с помощью оператора разрешения области видимости.

При множественном наследовании два базовых класса могут быть получены от общего предка. Если оба базовых класса используются обычным образом своим производным классом, такой класс будет иметь два подобъекта общего предка. Если подобное дублирование нежелательно, его можно избежать с помощью виртуального наследования. Вот пример:

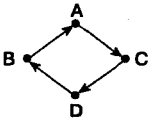
```
class student: public virtual person {
   .....
};

class worker: public virtual person {
   .....
};
```

[1] Речь идет о том, что граф множественного наследования не может включать замкнутый маршрут, то есть последовательность ребер, начав движение по которой из какой-то точки (класса) и двигаясь вдоль направлений ребер, можно вернуться в ту же точку (класс). — *Примеч. перев.*

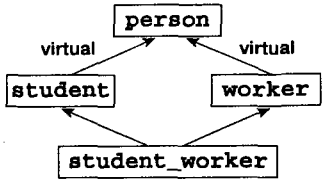


B и C наследуют от A; D наследует от B и C; все правильно, множественное наследование



A наследует от C, C — от D, D — от B, B — снова от A; ошибка — циклическое наследование

```
class student_worker: public student, public worker {
   .....
};
```



Без использования виртуальности в этом примере класс `student_worker` включал бы объекты «`class::student::person`» и «`class::worker::person`». Ниже приведен порядок выполнения конструкторов при наследовании и при инициализации членов класса:

Порядок выполнения конструкторов

1. Инициализуются (в порядке объявления) базовые классы.
2. Инициализуются (в порядке объявления) члены.
3. Выполняется тело конструктора.

Виртуальные базовые классы создаются до того, как создан любой из производных классов, и до того, как созданы невиртуальные базовые классы. Порядок их создания зависит от соответствующего графа. Это порядок «из глубины, слева направо». Деструкторы вызываются в порядке, обратном выполнению конструкторов. Несмотря на свою сложность, эти правила вполне интуитивны.

На многих системах файл *iostream.h* содержит конкретный, проработанный пример множественного наследования. Он включает класс `iostream`, который может быть произведен от `istream` и `ostream`. Однако, по поводу множественного наследования интересно заметить, что более свежие реализации вернулись к схеме одиночного наследования.

10.8. Наследование и проектирование

С одной стороны, наследование — это техника совместного использования кода. С другой стороны, оно отражает представления о решаемой задаче. Наследование выражает взаимосвязи между частями, на которые можно разбить моделируемую проблему. Открытое наследование большей части является выражением отношения ISA (быть экземпляром) между базовым и производным классами. Прямоугольник — это фигура. Такая концепция позволяет сделать фигуру суперклассом и перенести поведение, описываемое его открытыми функциями-членами, на другие объекты внутри иерархии типа. Другими словами, подклассы, полученные из этого суперкласса, используют общий с ним интерфейс.

Невозможно предложить во всех отношениях оптимальный проект. Проект — это всегда «торг» между различными целями, которые мы хотим достичь. Например, обобщенность часто противоречит эффективности. Применение иерархии классов, которая выражает ISA-отношения, помогает понять, как разделить код на разумные, логически самостоятельные части, но потенциально привносит неэффективность

кода за счет наличия «прослойки» для доступа к (скрытому) состоянию объекта. И все-таки, разумная ISA-декомпозиция может упростить весь процесс программирования. Например, пакету для рисования фигур не нужно «ожидать» будущих дополнительных фигур, которые планируется подключить к системе позднее. Посредством наследования разработчик класса импортирует интерфейс базового класса «фигура» и предоставляет код, реализующий такие операции, как «рисовать». То, что является примитивным или остается общим свойством, не изменяется. Остается неизменным также и то, как клиент использует данный пакет.

Чрезмерное увлечение разбиением на простые составляющие само по себе привносит дополнительную сложность и является игрой в свои ворота. Некоторые решения о разбиении предусматривают существование весьма специализированных классов, которые на самом деле не дают особых преимуществ, и их лучше объединять в более крупные сущности.

Одиночное наследование соответствует иерархической декомпозиции ключевых объектов в рассматриваемой проблемной области. Множественное наследование более хлопотно в качестве подхода к моделированию или решению проблемы. При множественном наследовании мы говорим, что новый объект составлен из нескольких уже существующих объектов и обычно рассматриваем его как форму каждого из них. Для того чтобы обозначить класс, полученный с помощью множественного наследования, когда все базовые классы ортогональны (независимы), используется термин *смесь*. В большинстве подобных случаев существует альтернативная формулировка *HASA*. Например, есть класс для летучих мышей (млекопитающих, которые могут летать); есть класс для чего угодно летающего (что угодно может оказаться млекопитающим); таким образом, летучая мышь — и нечто летающее, и млекопитающее. В зависимости от того, какой код доступен, разработка надлежащего класса для летучей мыши может подразумевать множественное наследование или одиночное наследование с соответствующими HASA-членами.

Множественное наследование представляет проблему для теоретика типов. Студент может быть произведен от человека. И работник может быть произведен от человека. А как быть с работающим студентом? Вообще говоря, типы лучше понимаются будучи цепочками одиночного наследования.

Ничто из вышесказанного не уменьшает привлекательность множественного наследования как техники повторного использования кода. Несомненно, оно является мощным обобщением одиночного наследования и, как таковое, вероятно вписывается в стиль работы некоторых программистов. Также как ряд программистов отдает предпочтение итерации (а не рекурсии), многие предпочитают одиночное наследование и агрегирование множественному наследованию и композиции.

10.8.1. Форма подтипов

Абстрактные типы данных успешны постольку, поскольку они ведут себя так же, как и собственные типы. Собственные типы, например целые типы в C, выступают в качестве иерархии подтипов. Это полезная модель для открыто наследуемых иерархий типов, она повышает легкость использования иерархий на основе полиморфизма. Вот рецепт для построения такой иерархии типов. Базовый класс делается абстрактным. Он используется для наследования интерфейса. Производные классы будут конкретизировать и реализовывать этот интерфейс.

```
class Abstract_Base {
public:
   //интерфейс — "совсем виртуальный"
   Abstract_Base();                    //конструктор по умолчанию
   Abstract_Base(const Abstract_Base&); //копирующий
                                        //конструктор
   virtual ~Abstract_Base() = 0;        //чисто виртуальный
   .....
protected:
   //используется вместо private из-за наследования
   .....
private:
   //часто остается пустым — иначе будет
   //мешать будущим разработкам
   .....
};

class Derived: virtual public Abstract_Base {
public:
   //конкретный экземпляр
   Derived();                          //конструктор по умолчанию
   Derived(const Derived&);            //копирующий конструктор
   ~Derived();                         //деструктор
   Derived& operator=(const Derived&);  //присваивание
   .....
protected:
   //используется вместо private если ожидается наследование
   .....
private:
   //используется для деталей реализации
   .....
};
```

Как правило, корень иерархии оставляют абстрактным. Это делает разработку более гибкой. Обычно на этом этапе никакая конкретная реализация не создается. Наличие чисто виртуальных функций запрещает объявления объектов соответствующего типа. Заметьте, что `~Abstract_Base()` — чисто виртуальная функция.

Этот уровень проекта фокусируется на открытом интерфейсе, которым являются операции, ожидаемые от любого из подтипов иерархии. Вообще, основные конструкторы, как правило, должны присутствовать, и они могут не быть виртуальными. Кроме того, большинство полезных агрегатов нуждаются в явном определении присваивания, которое отличается от семантики присваивания, принятой по умолчанию. Деструктор является виртуальным, поскольку он задействуется на этапе выполнения и его поведение зависит от размера объекта, который может отличаться от типа к типу в иерархии. Наконец, виртуальное открытое наследование (`virtual public`) позволяет быть уверенными в том, что при множественном наследовании мы не получим много копий абстрактного класса.

10.9. Идентификация типа на этапе выполнения

Идентификация типа на этапе выполнения (run-time type identification, RTTI) предоставляет механизм для безопасного выявления типа, на который направлен во время работы программы указатель базового класса. Этот механизм включает `dynamic_cast` — оператор над указателем на базовый класс, `typeid` — оператор для выяснения типа объекта и `type_info` — структуру, которая на этапе выполнения предоставляет информацию о соответствующем типе.

Оператор `dynamic_cast` имеет следующий вид:

```
dynamic_cast<тип>( v )
```

где *тип* должен быть указателем или ссылкой на тип класса, а *v* должна быть значением подхода указателя или ссылки. Это приведение используется с классами, имеющими виртуальные функции. Вот как оно применяется:

```
class Base { virtual void foo(); ····· };
class Derived : public Base { ····· };

void fcn(Base* ptr)
{
   Derived* dptr = dynamic_cast<Derived*>(ptr);
   .....
}
```

В этом примере приведение преобразует значение указателя `ptr` к типу `Derived*`. Если преобразование неприменимо, возвращается нулевой указатель NULL. Это называется *понижающим приведением* (down-cast). Динамические приведения также работают с типами-ссылками.

Оператор `typeid()` может быть применен к *имени типа* или к выражению для выяснения истинного типа аргумента. Этот оператор возвращает ссылку на класс `type_info`, который предоставляется системой и определен в заголовочном файле *type_info.h* (в некоторых компиляторах *typeinfo.h*). Класс `type_info` предоставляет функцию-член `name()`, которая возвращает строку, являющуюся именем типа. Кроме того, он предлагает перегруженные операторы проверки на равенство. Не забудьте изучить свою реализацию для выяснения деталей интерфейса этого класса.

В файле typeid.cpp

```
Base* bptr;
.....
//печатаем имя типа, на который сейчас указывает bptr
cout << typeid(bptr).name() << endl;
.....
if (typeid(bptr) == typeid(Derived)) {
   //делаем что-то, что можно делать с Derived
}
```

Можно воспользоваться «плохими» динамическими приведениями и «плохими» операциями `typeid`, чтобы возбудить исключения `bad_cast` и `bad_typeid`. Так что пользователь может выбрать, иметь ли ему дело с NULL-указателем или отлавливать исключения (см. раздел 11.10, «Стандартные исключения и их использование», на стр. 324).

10.10. Практические замечания

Сложность в изучении C++ (по сравнению с C) заключается во множестве особенностей и правил, касающихся использования функций. Многие из этих особенностей и правил связаны с наследованием. Перечислим основные особенности использования функций в C++.

Использование функций в C++

1. Виртуальная функция и ее «унаследованные» варианты, имеющие ту же сигнатуру, должны иметь одинаковый возвращаемый тип, хотя бывают несущественные исключения. Переопределение виртуальной функции называется замещением (overriding). Заметьте, что невиртуальные функции-члены базового и производного классов, имеющие одинаковые сигнатуры, могут возвращать значения различных типов (см. упражнение 14, на стр. 310).
2. Виртуальными могут быть любые функции-члены, кроме конструкторов и перегруженных `new` и `delete`.
3. Конструкторы, деструкторы, перегруженный оператор `=` и дружественность функций не наследуются.
4. Перегрузка операторов `=`, `()`, `[]` и `->` может быть выполнена только нестатическими функциями-членами. Функции преобразования вида: `operator` *тип*`()` также должны записываться с помощью нестатических функций-членов. Перегрузка операторов `new` и `delete` может выполняться только статическими функциями-членами. Все остальные перегружаемые операторы могут перегружаться с помощью дружественных функций, функций-членов или обычных функций.
5. Объединение (union) может иметь конструкторы и деструкторы, но не виртуальные функции. Оно не может служить базовым классом, равно как и иметь базовый класс. Член объединения не может требовать конструктор или деструктор.

Можно изменять права доступа при наследовании, но в случае открытого наследования изменение прав доступа к членам разрушит взаимосвязь подтипов. Изменение прав доступа не может расширять видимость. Например:

В файле acc_mod.cpp

```
//Изменение прав доступа

class B {
public:
   int  k;
protected:
   int  j, n;
private:
   int  i;
};

class D : public B {
public:
   int  m;
   B::n;  //недопустимо: защищенный доступ нельзя расширять
```

```
private:
   B::j;  //иначе по умолчанию защищенный
};
```

Резюме

1. Наследование — это механизм получения нового класса из существующего. Существующий класс может быть дополнен или изменен для создания производного класса. С помощью наследования может быть создана иерархия родственных АТД, которые совместно используют код.

2. Класс можно сделать производным от существующего с использованием следующей формы:

```
class имя_класса
     : (public | protected | private)opt имя_базового_класса
{
   объявления членов
};
```

   Ключевое слово `class` как всегда можно заменить ключевым словом `struct`, если подразумевается, что все члены открыты.

3. Модификаторами прав доступа членов класса являются ключевые слова `public`, `private` и `protected` (открытый, закрытый и защищенный). Открытый член доступен повсюду в области видимости. Закрытый член доступен только функциям-членам своего класса и дружественным функциям. Защищенный член доступен функциям-членам своего класса, дружественным функциям и внутри любого класса, унаследованного непосредственно от данного. Эти модификаторы прав доступа можно использовать в объявлении класса в любом порядке и с любой частотой.

4. Производный класс имеет свои собственные конструкторы, которые будут вызывать конструкторы базового класса. Существует специальный синтаксис для передачи аргументов конструктора производного класса конструктору базового класса:

```
заголовок_функции : имя_базового_класса (список_аргументов)
```

5. Открыто наследуемый класс является подтипом своего базового класса. Переменная производного класса во многих случаях может рассматриваться как переменная типа базового класса. Указатель, тип которого — «указатель на базовый класс», может указывать на объекты открыто наследуемого класса.

6. Ссылка на производный класс неявно может быть преобразована в ссылку на открытый базовый класс. Можно объявить ссылку на базовый класс и инициализовать ее ссылкой на объект открыто наследуемого класса.

7. Ключевое слово `virtual` является спецификатором функции и предоставляет механизм динамического выбора на этапе выполнения надлежащей функции-члена среди функций базового и производного классов. Данное ключевое слово нельзя использовать с функциями-не-членами. Переопределение виртуальной функции в производном классе называется замещением (overriding). Такая возможность динамически выбирать процедуру в зависимости от типа объекта является формой полиморфизма.

8. Наследование обеспечивает повторное использование кода. Производный класс наследует код базового класса, обычно изменяя и дополняя базовый класс. Открытое наследование создает иерархию типов. Это приводит к дальнейшему обобщению, предоставляя дополнительные неявные преобразования типов. Кроме того, наследование позволяет на этапе выполнения выбирать переопределенные виртуальные функции, правда, ценой более низкой скорости выполнения. Средства, позволяющие реализовывать АТД, наследование, а также возможность динамической обработки объектов, являются неотъемлемой частью ООП.

9. Чисто виртуальная функция — это виртуальная функция, тело которой не определено. Она объявляется внутри класса следующим образом:

```
virtual прототип_функции = 0;
```

Чисто виртуальная функция используется для того, чтобы отложить выбор реализации функции. В терминологии ООП это называется отложенным методом (deferred method). Класс, имеющий хотя бы одну чисто виртуальную функцию, называется абстрактным классом. Полезно, чтобы корневой тип в иерархии классов был абстрактным классом. Он должен определять интерфейс для своих производных классов, но при этом сам не может использоваться для объявления объектов.

Упражнения

1. Дополните код классов `student` и `grad_student` функциями-членами, которые считывали бы ввод данных для каждого члена данных в этих классах (см. раздел 10.1, «Производный класс», на стр. 282). Используйте `student::read` для реализации `grad_srudent::read`.

2. Преобразования указателей, разрешение области видимости и явные приведения типов предоставляют широкий выбор возможностей. Используйте функцию `main()`, обсуждавшуюся в разделе 10.2, «Преобразования типов и видимость», на стр. 284. Что из нижеследующего будет работать и что напечатается?

```
reinterpret_cast<grad_student*>(ps) -> print();
dynamic_cast<student*>(pgs) -> print();
pgs -> student::print();
ps -> grad_student::print();
```

Напечатайте и объясните результаты.

3. Измените класс D из раздела 10.4, «Виртуальные функции», на стр. 289, так:

```
class D2 : private B {
public:
  B::i;          //изменение прав доступа
  void print_i()
  {
    cout << i << " внутри D2: B::i равно"
         << B::i << endl;
  }
};
```

Что изменилось в выводе программы?

4. Ниже используется класс `s_tree` из раздела 10.3, «Повторное использование кода: класс двоичного дерева», на стр. 288.

```
int main()
{
  s_tree t;
  char dat[80], *p;

  cout << "\nВведите строки; выход — "
       << "по концу файла" << endl;
  while ( cin >> dat ) {
    p = new char[strlen(dat) + 1];
    strcpy(p, dat);
    t.insert(p);
  }
  t.print();
  cout << "\n\n" << endl;
}
```

Используйте этот код и перенаправление ввода-вывода, чтобы создать упорядоченный список, в котором подсчитывается, сколько раз каждая строка встречается в файле. Функция `cin >> dat` возвращает истину, если операция успешна.

5. Измените предыдущее упражнение так, чтобы на вводе использовался произвольный файл. Для этого надо считать имя файла из аргумента командной строки и использовать его для открытия `ifstream` (см. раздел D.5, «Файлы», на стр. 413).

6. Напишите деструктор для класса `gen_tree`. Не забудьте, что он должен добраться до каждого узла и удалить его.

7. Напишите деструктор для класса `s_tree`.

8. Процедура печати для класса `s_tree` из раздела 10.3, «Повторное использование кода: класс двоичного дерева», на стр. 288, является *среднеупорядоченным проходом по дереву* (inorder tree traversal).

```
void gen_tree::print(bnode* r)
{
  if (r != 0) {
    print(r -> left);
    ::print(r);
    print (r -> right);
  }
};
```

Выполните программу, используя как *предупорядоченный* (preorder), так и *поступорядоченный* (postorder) проходы. В предупорядоченном варианте инструкция `::print(r)` идет первой, а в поступорядоченном — последней.

9. Разработайте класс `gen_vect`, являющийся безопасным массивом обобщенных указателей. Унаследуйте от него класс `s_vect` — безопасный массив `char*`.

10. (*Сложное*) Используя `gen_tree`, создайте производный класс `itree`, который хранит вектор типа `int`, на который указывает член `data` каждого из узлов. Вы должны написать соответствующую функцию `comp`.

11. Перепишите код для `gen_tree::insert`, чтобы он был более эффективным. Для этого присвойте значение `comp(temp -> data, d)` временной переменной. Это позволит избежать повторных вычислений, связанных с потенциально затратными вызовами функций.

12. Создайте производный класс целого вектора из класса `vector<int>` из STL. Он должен считать 1 первым значением индекса, и `n` — последним.

```
int_vector x(n);      //вектор с диапазоном от 1 до n
```

13. Обобщите предыдущее упражнение, создав производный шаблонный класс, с диапазоном индекса от 1 до `n`.

```
vec_1<double> x(n);  //вектор с диапазоном от 1 до n
```

14. В следующей программе объясните, когда имеет место замещение, а когда — перегрузка?

```
//Замещение или перегрузка?

#include  <iostream.h>

class B {
public:
   B(int j = 0) : i(j) {}
   virtual void  print() const
      { cout << " i = " << i << endl; }
   void  print(char *s) const
      { cout << s << i << endl; }
private:
   int i;
};

class D : public B {
public:
   D(int j = 0) : B(5), i(j) {}
      void  print() const
      { cout << " i = " << i << endl; }
   int  print(char *s) const
      { cout << s << i << endl; return i; }
private:
   int  i;
};

int main()
{
   B  b1, b2(10), *pb;
   D  d1, d2(10), *pd = &d2;
```

```
    b1.print(); b2.print(); d1.print(); d2.print();
    b1.print("b1.i = "); b2.print("b2.i = ");
    d1.print("d1.i = "); d2.print("d2.i = ");
    pb = pd;
    pb -> print(); pb -> print("d2.i = ");
    pd -> print(); pd -> print("d2.i = ");
}
```

15. Определите базовый класс person (человек), который содержит универсальную информацию, включая имя, адрес, дату рождения, пол. Создайте следующие производные классы:

```
class student : virtual public person { //студент
//····· дополнительные подходящие сведения и поведение
};

class worker : virtual public person {  //рабочий
//····· дополнительные подходящие сведения и поведение
};

class student_worker : public student, public worker{
//····· работающий студент
};
```

Напишите программу, которая читала бы информацию из файла и создавала бы список людей. Обработайте этот список для создания отсортированных по фамилиям списков всех людей, студентов, работников и списка работающих студентов. Легко ли на вашей системе создать отсортированный список всех студентов, не являющихся работниками?

16. (*Проект*) Разработайте и реализуйте графический пользовательский интерфейс (GUI, graphical user interface) для модели «хищник-добыча». В задачи этой книги не входит описывать различные инструментальные пакеты для создания GUI. Упомянем лишь пакет InterViews, работающий под управлением среды X-Window и написанный на C++. Программа должна рисовать каждую итерацию модели на экране. Вы должны иметь возможность непосредственно вводить начальную ситуацию «райский сад» (игра, моделирующая жизнь, описана в разделе 10.5, «Абстрактные базовые классы», на стр. 293). Кроме того, необходимо предусмотреть возможность задавать другие параметры модели, например ее размер. Можете ли вы предоставить пользователю право определять другие формы жизни и правила для их существования, кормления и размножения? Сделайте графический интерфейс как можно более элегантным. У пользователя должна иметься возможность перемещать окно по экрану, изменять его размер и выбирать значки для разных доступных форм жизни.

Глава 11

Исключения

Эта глава описывает обработку исключений (exceptions) в C++. *Исключениями* называют нештатные (ошибочные) ситуации. Обычно такие ситуации прерывают пользовательскую программу с сообщением об ошибке, которое выдает система. Примером является деление на ноль. Как правило, в подобных случаях выполнение программы аварийно завершается системой. C++ позволяет программисту попытаться восстановить работоспособность программы при возникновении исключительных ситуаций и продолжить ее выполнение.

Проверка утверждения (assertion) – это контрольный элемент программы, который (в случае нарушения проверяемого утверждения) осуществляет аварийный выход. Одна из точек зрения состоит в том, что проверка утверждений основана на анализе гарантийного договора между поставщиком (производителем) и клиентом (пользователем) кода (см. раздел 12.3, «Клиенты и производители», на стр. 333). В этой модели клиент должен гарантировать, что условия для применения кода выполняются, а производитель должен гарантировать, что код будет работать правильно при этих условиях. В рамках такого подхода проверка утверждений обеспечивает ряд гарантий.

11.1. Использование assert.h

Корректность программы частично может рассматриваться как гарантия того, что вычисление завершится корректным выводом, зависящим от корректного ввода. Пользователь вычислений несет ответственность за корректность ввода. Это *предусловие* (precondition). Вычисление, если оно удачно, удовлетворяет *постусловию* (postcondition). Обеспечение полной проверки по всем правилам было бы идеально, но обычно она не выполняется. Но мы, по крайней мере, можем отслеживать проверку утверждений в процессе выполнения программы, что обеспечивает весьма полезную диагностику. В самом деле, дисциплина, заставляющая продумывать надлежащие утверждения, часто помогает программисту избежать ошибок и ловушек.

Сообщество программистов на C и C++ придает все большее значение использованию проверки утверждений. Стандартная библиотека *assert.h* предоставляет макро `assert`; оно вызывается так, как если бы существовала функция с сигнатурой:

```
void assert(bool выражение);
```

Если результат *выражения* — ложь, то выполнение прерывается с выводом диагностического сообщения. Проверка утверждений не выполняется, если определено макро NDEBUG.

Рассмотрим размещение безопасного массива типа `vect` (см. раздел 6.5, «Класс vect», на стр. 166).

```
vect::vect(int n) : size(n)
{
   assert(n > 0);
   p = new int[size];
   assert(p != 0);
}
```

Использование утверждений заменяет приспособленные для того же обычные проверки и является более унифицированной методикой. Это более подходящая практика. Слабой стороной является то, что проверка утверждений не позволяет повторить попытку или осуществить что-либо подобное для восстановления работоспособности и продолжения выполнения программы. Кроме того, проверка утверждений не позволяет вывести свое собственное оригинальное сообщение об ошибке, хотя добавить такую возможность было бы несложно.

Можно сделать эту схему слегка более изощренной, предоставив различные уровни проверки, как это сделано в файле *checks.h* из Borland C++. В этом пакете можно устанавливать флаг _DEBUG:

```
_DEBUG 0  без проверки
_DEBUG 1  проверять только предусловие (PRECONDITION)
_DEBUG 2  полная проверка (CHECK)
```

Суть в том, что библиотечные функции полагаются безошибочными, и уровень проверки для них можно уменьшить, проверяя только предусловия. Когда же клиентский код отлажен, можно отключить всякие проверки.

11.2. Использование signal.h

Файл *signal.h* предоставляет стандартный механизм для простой обработки определяемых системой исключений. В этом файле определены исключения, представляющие собой целые значения, зависящие от системы. Вот некоторые примеры:

```
#define SIGINT  2  /*сигнал прерывания*/
#define SIGFPE  8  /*исключение с плавающей точкой*/
#define SIGABRT 22 /*сигнал аварийного завершения*/
```

Система может возбуждать (raise) эти исключения. Например, нажатие на клавиатуре клавиш Ctrl-C на многих системах генерирует прерывание, обычно убивающее текущий процесс.

Прототип функции `raise()` находится в файле *signal.h*, ее можно использовать для генерации явного исключения.

```
raise(SIGFPE); //возбужден сигнал исключения с плавающей точкой
```

Подобные исключения могут обрабатываться с помощью функции `signal()`. Она связывает обрабатывающую функцию с сигналом. Ее также можно использовать для игнорирования сигнала или для того, чтобы заменить действие, выполняемое по умолчанию, на свою процедуру.

```
//вызвать my_abrt(), если возбуждено SIGABRT
signal(SIGABRT, my_abrt);

//действовать по умолчанию, если возбуждено SIGABRT
signal(SIGABRT, SIG_DFL);

//игнорировать SIGFPE
signal(SIGFPE, SIG_IGN);
```

Подобные действия называются *установкой* (installing) обработчика (handler). Они заменяют обычные реакции системы обработчиком, определяемым пользователем.

Используем эти идеи для написания цикла, прерывание которого управляется с клавиатуры. После прерывания обработчик просит пользователя решить, должна ли программа продолжить выполнение.

В файле signal.cpp

```
//Прерывания, обрабатываемые с помощью signal.h

#include <signal.h>
#include <time.h>
#include <iostream.h>
#include <stdlib.h>

void cntrl_c_handler(int sig);
inline double clk_psec()
{
  return static_cast<double>(clock())/CLOCKS_PER_SEC;
}

int main()
{
  int i = 0, j;

  cout << "Считаем до j миллионов , введите j: ";
  cin >> j;
  j *= 1000000;
  signal(SIGINT, cntrl_c_handler);
  cout << clk_psec() << " начальное время\n";
  while (1) {
    ++i;
    if (i > j) {
      cout << clk_psec() << " конец цикла\n";
      cout << " ЦИКЛ ИСПОЛНЕН " << j/1000000
        << "  МИЛЛИОНОВ РАЗ" << endl;
      raise(SIGINT);
```

```
            cout << "\nВведите j: ";
            cin >> j;
            j *= 1000000;
            i = 0;
            cout << clk_psec() << " начало цикла\n";
        }
    }
}

void cntrl_c_handler(int sig)
{
    char  c;

    cout << "ПРЕРЫВАНИЕ";
    cout << "\nвведите Д для продолжения: ";
    cin >> c;
    if (c == 'Д')
        signal(SIGINT, cntrl_c_handler);
    else
        exit(0);
}
```

Разбор программы signal

- `signal(SIGINT, cntrl_c_handler);`

Обработчик функции связан с прерыванием `SIGINT`. При выявлении следующего прерывания система вызывает `cntrl_c_handler` вместо системного действия по умолчанию.

- ```
  while (1) {
      ++i;
      if (i > j) {
          .....
          raise(SIGINT);
  ```

Сигнал прерывания возбуждается явным вызовом; `cntrl_c_handler()` вызывается неявно.

- `void cntrl_c_handler(int sig)`

Эта процедура обрабатывает исключение `SIGINT`.

- ```
  cout << "ПРЕРЫВАНИЕ";
  cout << "\nвведите Д для продолжения: ";
  ```

Было обнаружено прерывание и пользователю задается вопрос о том, надо ли продолжать выполнение программы.

- ```
  if (c == 'Д')
      signal(SIGINT, cntrl_c_handler);
  ```
  ```

По этому требованию продолжать выполнение обработчик исключений переустанавливается. Без этого система вернет обработчик прерываний к состоянию по умолчанию.

- ```
  else
      exit(0);
  ```

Для завершения вызывается функция `exit()` из *stdlib.h*.

Применим эту технику обработки исключений для типа `vect` (см. раздел 7.7, «Перегрузка операторов индексирования и присваивания», на стр. 203). Начнем с замены проверки утверждений на условия, которые возбуждают определенные пользователем сигналы.

В файле vect3.h

```
const int SIGHEAP = SIGUSR1;
const int SIGSIZE = SIGUSR2;

vect::vect(int n)
{
  if (n <= 0) {
    raise(SIGSIZE);
    cout << "\nВведите n, размер вектора: ";
    cin >> n;        //повтор с пользовательским значением
  }
  p = new int[n];
  if (p == 0) {
    raise(SIGHEAP);
    p = new int[n];
  }
}
```

Теперь можно определить обработчики, которые смогут выполнять надлежащие действия. Например:

```
void vect_size_handler(int sig)
{
  char c;

  cout << "\nНЕВЕРНЫЙ РАЗМЕР\nВведите Д для продолжения"
       << " с другим размером: ";
  cin >> c;
  if (c == 'Д') {
    signal(SIGSIZE, vect_size_handler);
  }
  else
    exit(0);
}
```
```

```
void vect_heap_handler(int sig)
{
   //Возможные действия по высвобождению памяти для кучи
   //или безопасный выход
   .....
}
```

Сравните эту сложную технику со стандартной техникой обработки переполнения кучи, использующей механизм `set_new_handler()`, описанный в разделе 7.12, «Перегрузка new и delete», на стр. 215.

Отметьте, что и сигнал, и проверку утверждения часто можно заменить локальной проверкой и вызовом функции, что обладает большей гибкостью. Однако, сигналы и проверка утверждений обеспечивают унифицированную методику. Кроме того, они удерживают от попыток сделать программу слишком уж отказоустойчивой. Во многих случаях излишнее рвение в создании такой программы есть неправильное программирование и дурная привычка. Проверка утверждений выясняет, выполнены ли согласованные условия. Сигналы и обработчики создают глобальный контекст восстановления на случай нежелательных обстоятельств. Сигналы ограничены набором асинхронных системно-зависимых условий. Однако эта схема легко расширяется с помощью введения глобальных данных, которые переустанавливаются, когда возникает условие для возбуждения исключения (`raise`).

11.3. Исключения C++

C++ предлагает чувствительный к контексту механизм обработки исключений. Он не предназначен для обработки асинхронных исключений, определенных в файле *signal.h*, таких как `SIGFPE` (исключение с плавающей точкой). Контекстом для возбуждения исключения является блок `try` (пробный блок[1]). Обработчики, объявленные ключевым словом `catch` (перехватчик), находятся ниже блока `try`.

В коде на C++ исключение может возбуждаться в блоке `try` с помощью выражения `throw` (запустить). Исключение обслуживается вызовом соответствующего обработчика, выбранного из списка, который идет сразу за блоком `try`. Ниже приведен пример:

В файле vect4.h

```
vect::vect(int n): size(n)
{
   if (n < 1)        //проверка предусловия
```

[1] Из многочисленных предлагаемых словарями вариантов перевода терминов, связанных с исключениями, мы избрали следующие: исключение (exception), запуск (throw), возбуждение (raise), пробный блок (try block), перехват (catch), обработчик (handler). Таким образом, программист может возбудить (запустить) исключение в пробном блоке (то есть в блоке, который «пробует» выполнить некоторые действия, зная, что может произойти нечто нештатное). Это исключение может быть перехвачено соответствующим обработчиком. Термины «запустить» и «возбудить» являются, как правило, синонимами, но первый обычно намекает на ключевое слово C++ `throw` и часто используется по отношению к выражению исключения, а последний зачастую употребляют по отношению к исключительным ситуациям вообще (в том числе и тем, которые обрабатываются без использования механизма исключений C++, см. предыдущие разделы данной главы). — *Примеч. перев.*

```
      throw (n);
   p = new int[n];
   if (p == 0)       //проверка постусловия
      throw ("СВОБОДНАЯ ПАМЯТЬ ИСЧЕРПАНА");
}
void g()
{
   try {
      vect a(n), b(n);
      .....
   }
   catch(int n) { ..... }       //отлавливает неверный размер
   catch(char* error) { ..... }//отлавливает отсутствие
}                               //свободной памяти
```

Первый `throw()` имеет целый аргумент и соответствует сигнатуре `catch(int n)`. Этот обработчик должен выполнить действия, необходимые, когда конструктору в качестве аргумента передается некорректный размер массива. Например, вывести сообщение об ошибке и прервать программу. Второй `throw()` получает в качестве аргумента указатель на символ и соответствует сигнатуре `catch(char* error)`.

11.4. Запуск исключений

Синтаксически выражение `throw` может принимать следующие две формы:

```
throw выражение
throw
```

Конструкция `throw` *выражение* возбуждает исключение. Для выбора инструкции `catch`, которая обрабатывает исключение, используется самый внутренний пробный блок (`try`-блок), в котором возбуждено исключение. А `throw` без аргумента может использоваться внутри `catch` для перезапуска (rethrow) текущего исключения. Обычно `throw` без аргумента используется, если вы хотите, чтобы для дальнейшей обработки исключения из недр первого обработчика вызвался второй.

Запущенное с помощью `throw` выражение — это временный статический объект, существующий до тех пор, пока не будет произведен выход из обработчика исключения. Выражение отлавливается обработчиком, который может использовать его значение:

В файле throw1.cpp

```
#include <iostream.h>

void foo()
{
   int  i;
   //покажем, как запускается исключение
   i = -15;
   throw i;
}
```

```
int main()
{
   try {
      foo();
   }
   catch(int n)
      { cerr << "Перехваченное выражение\n" << n << endl; }
}
```

Целое значение, запущенное `throw i`, существует, пока существует обработчик `catch(int n)` с целой сигнатурой. Это выражения доступно для использования внутри обработчика в качестве его аргумента.

Когда вложенная функция запускает исключение, стек процессов раскручивается, пока не будет найден обработчик исключений. Соответственно, выход из каждого завершенного локального процесса вызывает уничтожение автоматических объектов:

В файле throw2.cpp

```
void foo()
{
   int  i, j;
   .....
   throw i;
   .....
}

void call_foo()
{
   int  k;
   .....
   foo();
   .....
}

int main()
{
   try {
      call_foo();  //при выходе из foo уничтожаются i и j
   }
   catch(int n) { ..... }
}
```

11.4.1. Перезапуск исключений

Использование `throw` без выражения *перезапускает* перехваченное исключение. Тот `catch`, который перезапускает исключение, не может заканчивать обработку существующего исключения. Он передает управление в ближайший окружающий блок `try`, где вызывается обработчик, способный отловить все еще существующее исключение. Выражение исключения существует, пока не завершено выполнение всех об-

работчиков. Выполнение продолжается после самого внешнего блока try, который последним обработал перезапущенное выражение.

Вот пример перезапуска исключения:

```
catch(int n)
{
   .....
   throw;       //перезапуск
}
```

При условии, что запущенное выражение было целого типа, перезапущенное исключение — это тот же целый объект, который обрабатывается ближайшим обработчиком, подходящим для этого типа.

11.4.2. Выражения исключений

По замыслу, запущенное исключение «передает» информацию обработчикам. Часто обработчики не нуждаются в этой информации. Например, обработчику, выводящему сообщение и прерывающему выполнение программы, не нужна информация из его окружения. Однако пользователь может пожелать, чтобы выводилась дополнительная информация, которую можно было бы использовать как подспорье в выборе надлежащих действий. В таком случае следует упаковать нужную информацию в объект:

```
class vect_error {
private:
   //тип ошибки: границы, нет памяти, другое
   enum error { bounds, heap, other } e_type;
   //верхняя граница, индекс, размер
   int ub, index, size;
public:
   vect_error(error, int, int);   //вне границ
   vect_error(error, int);        //нет памяти
   .....
};
```

Теперь запуск выражения с помощью объекта типа vect_error будет более информативен для обработчика, чем запуск выражений простых типов:

```
   .....
throw vect_error(bounds, i, ub);
   .....
```

11.5. Пробные блоки try

Синтаксически блок try выглядит так:

```
try
составная_инструкция
список_обработчиков
```

Блок try является контекстом для определения того, какие обработчики вызываются при возбуждении исключения. Порядок, в котором определены обработчики, за-

дает порядок, в котором будут пробоваться обработчики подходящего типа для возбужденного исключения:

```
try {
   .....
   throw("SOS");
   .....
   io_condition eof(argv[i]);
   throw (eof);
   .....
}
catch(const char*) {.....}
catch(io_condition& x) {.....}
```

Выражение throw соответствует типу обработчика catch, если оно:

1. точно совпадает с типом `catch`;
2. является производным типом открытого базового класса типа обработчика;
3. является типом возбуждаемого объекта, который может быть преобразован к типу указателя, являющегося аргументом `catch`.

Ошибочно приводить список обработчиков в порядке, исключающем возможность вызова некоторых из них. Например:

```
catch(void*)              //будет соответствовать любому char*
catch(char*)

catch(BaseTypeError&)   //вызывается даже при DerivedTypeError
catch(DerivedTypeError&)
//предполагается, что класс DerivedTypeError
//является производным от BaseTypeError
```

Блоки `try` могут быть вложенными. Если нет ни одного подходящего обработчика непосредственно в данном блоке `try`, то обработчик будет выбран из ближайшего охватывающего блока `try`. Если невозможно подобрать надлежащий обработчик, используется поведение по умолчанию, которым является `terminate()` (см. раздел 11.8, «terminate() и unexpected()», на стр. 321).

11.6. Обработчики

Синтаксически обработчик имеет вид:

```
catch(формальный_аргумент)
составная_инструкция
```

Обработчик `catch` выглядит как объявление функции одного аргумента без возвращаемого типа.

В файле catch.cpp

```
catch (char* message)
{
   cerr << message << endl;
```

```
    exit(1);
}

catch( ... )    //будет выполнено действие по умолчанию
{
    cerr << "ВОТ И ВСЁ!" << endl;
    abort();
}
```

Допустима эллипсическая `( ... )` сигнатура, совпадающая с аргументом любого типа. Кроме того, формальный аргумент может быть абстрактным объявлением, то есть задавать информацию о типе без имени переменной.

Обработчик вызывается подходящим выражением `throw`. В этот момент происходит выход из блока `try`. Система вызывает функции очистки памяти, включая деструкторы для любых объектов, локальных в блоке `try`. При частично созданном объекте деструкторы будут вызываться для любой его части, являющейся созданным подобъектом. Программа продолжит выполнение с инструкции, следующей за блоком `try`.

11.7. Спецификация исключения

Синтаксически *спецификация исключения* является частью объявления и определения функции и имеет следующий вид:

заголовок_функции `throw (`*список_типов*`)`

Здесь *список_типов* — это список типов, которые может иметь выражение `throw` внутри функции. В объявлении и в определении функции спецификация исключения должна записываться одинаково.

Если список пуст, компилятор полагает, что функцией не будет выполняться никакой `throw` (ни прямо, ни косвенно).

```
void foo() throw(int, over_flow);
void noex(int i) throw();
```

Если спецификация исключения опущена, предполагается, что функцией может быть возбуждено произвольное исключение. Нарушение спецификаций исключений приводит к ошибкам на этапе выполнения. Они отлавливаются функцией `unexpected()`.

11.8. terminate() и unexpected()

Предоставляемая системой функция `terminate()` вызывается, если не было задано ни одного обработчика, «знающего, как обращаться» с исключением. По умолчанию вызывается функция `abort()`. Она немедленно завершает программу и возвращает управление операционной системе. С помощью `set_terminate()` может быть задано какое-нибудь другое действие; тем самым задается обработчик. Объявления упомянутых функций находятся в *except* или *except.h*.

Предоставляемый системой обработчик `unexpected()` вызывается, когда функция возбудила исключение, которое отсутствует в ее списке спецификации исключений. По умолчанию вызывается функция `terminate()`. Или можно воспользоваться `set_terminate()`, чтобы задать обработчик.

11.9. Пример кода с исключениями

В этом разделе мы обсудим некоторые примеры кода с исключениями. Вернемся к перехвату ошибки размера в нашем конструкторе для типа `vect`, который обсуждался в разделе 7.7, «Перегрузка операторов присваивания и индекса», на стр. 203:

В файле vect4.h

```
vect::vect(int n): size(n)
{
   if (n < 1)     //проверка предусловия
      throw (n);
   p = new int[n];
   if (p == 0)    //проверка постусловия
      throw ("СВОБОДНАЯ ПАМЯТЬ ИСЧЕРПАНА");
}
```

В файле vect4.cpp

```
void g (int m)
{
   try {
      vect a(m);
      .....
   }
   catch(int n)
   {
      cerr << "ОШИБКА РАЗМЕРА " << n << endl;
      g(10);    //пробуем g допустимого размера
   }
   catch(const char* error)
   {
      cerr << error << endl;
      abort();
   }
}
```

Обработчик заменил недопустимое значение допустимым значением по умолчанию. Такой подход может быть разумен на этапе отладки системы, когда интегрируется и тестируется множество процедур. Система пытается предоставить дополнительную диагностику. Это аналогично тому, как компилятор пытается продолжать разбор кода после отловленной им синтаксической ошибки. Часто компилятор предоставляет дополнительные сообщения об ошибках, которые оказываются полезными.

Вышеприведенный конструктор проверяет лишь одну переменную на допустимость значений. Этот конструктор выглядит искусственно: вместо того, чтобы непосредственно (прямо в теле конструктора) заменить недопустимое значение на допустимое значение по умолчанию, он запускает исключение и перекладывает заботу об «исправлении» ошибочной переменной на обработчик исключения. Тем не менее именно в приведенной выше форме наиболее очевидно отделено друг от друга то, что

является ошибкой, и то, как она должна обрабатываться. Это — ясная методология для разработки отказоустойчивого кода.

В более общем виде конструктор объекта может выглядеть так:

```
Object::Object(аргументы)
{
   if (нештатный_аргумент1)
     throw выражение1;
   if (нештатный_аргумент2)
     throw выражение2;
   .....
   //попытка создать объект
   .....
}
```

Теперь конструктор `Object` предоставляет набор запускаемых выражений для нештатной ситуации. А блок `try` может пользоваться информацией для завершения выполнения или исправления некорректного кода.

```
try {
   //..... отказоустойчивый код
   }
   catch(объявление1) { /* исправления для этого случая /* }
   catch(объявление2) { /* исправления для этого случая /* }
     .....
   catch(объявлениеK) { /* исправления для этого случая /* }
//теперь все значения допустимы
```

В том случае, когда для состояния данного объекта полезно ввести много различных ошибочных ситуаций, можно использовать иерархию классов, чтобы иметь возможность выбора родственных типов, используемых в качестве выражений `throw`.

```
Object_Error {
public:
   Object_Error(аргументы);  //фиксируем полезную инфромацию
   //члены, содержащие запускамое выражение исключения
   virtual void repair()
     { cerr << "Не удается исправить Object" << endl; abort();
}
};

Object_Error_S1 : public Object_Error {
public:
   Object_Error_S1(аргументы);
   //дополниетельные члены, в которых сосредоточена информация
   //о состоянии запускаемого выражения
   void repair();    //замещается для обеспечения исправлений
};

.....                //другие необходимые производные классы
                     //для обработки ошибок
```

Приведенные иерархии позволяют надлежащим образом упорядоченному набору перехватчиков `catch` обрабатывать исключения в логической последовательности. Напомним, что в списке объявлений перехватчиков тип базового класса должен идти *после типа производного.*

11.10. Стандартные исключения и их использование

Стандартные исключения предоставляются компиляторами C++ и поставщиками библиотек. Например, компилятором Borland запускается исключение типа `xalloc`, если оператору `new` не удается выделить место в свободной памяти.

Вот простая программа, позволяющая проверить это.

В файле except.cpp

```
#include <iostream.h>
#include <except.h> //нужно для  xalloc и xmsg

int main()
{
   int *p, n;

   try {
      while (true) {
         cout << "сколько разместить? " << endl;
         cin >> n;
         p = new int[n];
      }
   }
   catch(xalloc x){cout << "перехвачен xalloc" << endl;}
   catch(...){cout << "перехват по умолчанию\n" << endl;}
}
```

Эта программа выполняется в цикле, пока его не прервет исключение. На нашей системе требование разместить один миллиард целых чисел вызовет обработчик `xalloc`. А по стандарту ANSI для этих целей служит класс исключений `bad_alloc`.

Часто использование стандартных исключений состоит в тестировании приведений типов. Стандартное исключение `bad_cast` объявлено в файле *exception.* Вот простая программа, использующая RTTI и это исключение:

В файле bad_cast.cpp

```
#include <iostream.h>
#include <typeinfo.h>
#include <stdexception>
using namespace std;

class A {
public:
  virtual void foo(){ cout << "в A" << endl;}
};
```

```
class B: public A {
public:
  void foo(){ cout << "в B" << endl;}
};
int main()
{
  try {
    A a, *pa; B b, *pb;
    pa = &b;
    pb = dynamic_cast<B*>(pa);   //удачно
    pb -> foo();
    pa = &a;
    pb = dynamic_cast<B*>(pa);   //неудачно
    pb -> foo();
  }
  catch(bad_cast){cout << "плохое приведение" << endl;}
}
```

На системах, которые не запускают такие исключения, указатель следует тестировать с помощью проверки утверждений, чтобы убедиться, что он не был преобразован к нулю.

Исключения стандартной библиотеки наследуются от базового класса `exception`. Два производных класса — это `logic_error` (логическая ошибка) и `runtime_error` (ошибка времени выполнения). Класс логических ошибок включает: `bad_cast` (ошибка приведения), `out_of_range` (вне диапазона) и `bad_typeid` (ошибка оператора `typeid()`), которые должны запускаться в ситуациях, соответствующих названиям перечисленных классов. Класс ошибок на этапе выполнения включает: `range_error` (ошибка диапозона), `overflow_error` (переполнение) и `bad_alloc` (ошибка размещения).

Базовый класс определяет следующую виртуальную функцию:

```
virtual const char* exception::what() const throw();
```

Эта функция-член должна быть определена в каждом производном классе для того чтобы предоставить дополнительные полезные сообщения. Пустой список спецификации в `throw` показывает, что функция сама не запускает исключений.

11.11. Практические замечания

Парадоксально, но понятие «восстановления после ошибок» главным образом касается написания корректных программ. Обработка исключений связана с восстановлением после сбоев. Кроме того, это механизм передачи управления. Модель клиент-производитель налагает на производителя ответственность по созданию такого программного обеспечения, которое выдает корректный вывод при приемлемом вводе. Вопрос для производителя состоит в том, чтобы определить, сколько элементов, определяющих, и, возможно, исправляющих ошибки, должно быть встроено в программу. Часто клиент лучше «обслуживается» библиотеками, обнаруживающими сбои: он может сам решить, пытаться ли продолжать вычисление.

Любое восстановление в процессе сбоя основано на передаче управления. Неумелая передача управления ведет к хаосу. При восстановлении после сбоя подразуме-

вается, что исключительная ситуация исказила вычисление, и его опасно продолжать. Это похоже на управление автомобилем после того как выяснилось, что поврежден рулевой механизм. Весьма похвальная обработка исключений заключается в квалифицированном восстановлении при возникновении повреждений.

В большинстве случаев программа, возбуждающая исключения, должна выводить диагностическое сообщение и «плавно» завершаться. Специальные формы обработки данных, такие как обработка в реальном времени и отказоустойчивые вычисления, требуют безостановочной работы системы. В этих случаях вполне уместны самые героические попытки восстановления.

Следует согласиться с тем, что при создании классов полезно предусмотреть ошибочные ситуации. В исключительных ситуациях объект может иметь такие значения членов, которые он иметь не должен. Для подобных случаев система возбуждает исключения, и действием по умолчанию будет завершение программы. Это аналогично возбуждению системных исключений для собственных типов, таких как `SIGFPE`.

Но какое вмешательство разумно для того, чтобы продолжить (не прерывать) выполнение программы? И куда следует возвратить управление? C++ использует модель завершения, которая заставляет завершиться текущий блок `try`. В этой модели можно либо повторить попытку (начиная с момента, предшествующего возникновению недопустимого значения), либо игнорировать исключение, либо подставить результат по умолчанию и продолжить. Повтор попытки представляется наиболее подходящим для получения корректного результата.

Трудно представить себе программу, которая была бы слишком богата проверками утверждений. Утверждения и простые запуски и перехваты исключений, прерывающие вычисления, являются параллельными приемами. Тщательно продуманный набор для обработки ошибочных ситуаций, обнаруживаемых пользователем АТД, является важной частью хорошего проекта. С другой стороны, чрезмерная зависимость от обработки исключений при обычном программировании, когда исключения используются не только для выявления ошибок и завершения программы, служит признаком того, что последняя изначально была задумана неудачно, со множеством дыр.

Резюме

1. Исключениями называют нештатные (ошибочные) ситуации. Обычно такие ситуации прерывают пользовательскую программу с сообщением об ошибке, которое выдает система. Примером является деление на ноль.

2. Стандартная библиотека *assert.h* предоставляет макропроверки утверждений:

   ```
   assert(выражение);
   ```

   Если результат *выражения* — ложь, то выполнение прерывается с выводом диагностического сообщения. Проверка утверждений не выполняется, если определено макро `NDEBUG`.

3. Файл *signal.h* предоставляет стандартный механизм для простой обработки определяемых системой исключений. Вот некоторые примеры:

   ```
   #define SIGINT  2  /*сигнал прерывания*/
   #define SIGFPE  8  /*исключение с плавающей точкой*/
   ```

```
#define SIGABRT 22 /*сигнал аварийного завершения*/
```

Система может возбуждать (raise) эти исключения. Например, нажатие на клавиатуре клавиш Ctrl-C на многих системах генерирует прерывание, обычно убивающее текущий процесс. Подобные исключения могут обрабатываться с помощью функции `signal()`, связывающей функцию обработчика с сигналом.

4. В коде на C++ исключение может возбуждаться с помощью выражения `throw`. Исключение обслуживается вызовом соответствующего обработчика, выбранного из списка, который идет сразу за блоком `try`.

5. Синтаксически выражение `throw` может принимать две следующие формы:

```
throw выражение
throw
```

Конструкция `throw` *выражение* возбуждает исключение в блоке `try`. А `throw` без аргумента может использоваться внутри `catch` для перезапуска (rethrow) текущего исключения.

6. Синтаксически блок `try` выглядит так:

```
try
составная_инструкция
список_обработчиков
```

Блок `try` является контекстом для определения того, какие обработчики вызываются при возбуждении исключения. Порядок, в котором определены обработчики, задает порядок, в котором будут пробоваться обработчики подходящего типа для возбужденного исключения.

7. Синтаксически обработчик имеет вид:

```
catch(формальный_аргумент)
составная_инструкция
```

Обработчик `catch` выглядит как объявление функции одного аргумента без возвращаемого типа.

8. Синтаксически *спецификация исключения* является частью объявления функции и имеет следующий вид:

```
заголовок_функции throw (список_типов)
```

Здесь *список_типов* — это список типов, которые может иметь выражение `throw` внутри функции. Если список пуст, компилятор полагает, что функцией не будет выполняться никакой `throw` (ни прямо, ни косвенно).

9. Предоставляемый системой обработчик `terminate()` вызывается, если не было задано ни одного обработчика, «знающего, как обращаться» с исключением. Предоставляемый системой обработчик `unexpected()` вызывается, когда функция возбудила исключение, которое отсутствует в ее списке спецификации исключений. По умолчанию `terminate()` вызывает функцию `abort()`. Поведение `unexpected()` по умолчанию состоит в вызове функции `terminate()`.

Упражнения

1. Следующая сортировка методом всплывания (пузырька) работает некорректно:

```
//Некорректная сортировка методом пузырька

#include  <iostream.h>

void swap(int a, int b)
{
   int  temp = a;

   a = b;
   b = temp;
}

void bubble(int a[], int size)
{
   int  i, j;

   for (i = 0; i != size; ++i)
      for (j = i ; j != size; ++j)
         if (a[j] < a [j + 1])
            swap (a[j], a[j + 1]);
}

int main()
{
   int  t[10] = { 9, 4, 6, 4, 5, 9, -3, 1, 0, 12};

   bubble(t, 10);
   for (int i = 0; i < 10; ++i)
      cout << t[i] << '\t';
   cout << "\nОтсортировано? " << endl;
}
```

   Вставьте в код проверку утверждений, которые будут тестировать правильность работы. Используйте эту технику для написания корректной программы.

2. Используйте шаблоны для написания обобщенной версии корректной (дополненной проверкой утверждений) сортировки методом пузырька. Используйте генератор случайных чисел для получения проверочных данных.

3. Напишите функцию-член `vect::operator[](int)` так, чтобы она запускала исключение «вне диапазона», если используется некорректный индекс (см. раздел 7.7, «Перегрузка операторов присваивания и индексирования», на стр. 204). Кроме того, закодируйте подходящий `catch`, который выводит некорректное значение и останавливает программу. Для тестирования кода выполните блок `try`, в котором возникает исключение. Напишите `catch`, который позволил бы пользователю вмешиваться с клавиатуры для задания корректного индекса, чтобы продолжить (или попытаться повторить) вычисления. Можно ли это сделать разумным образом?

4. Перепишите класс `ch_stack` так, чтобы он запускал исключения для такого числа ситуаций, какое вы сочтете разумным (см. раздел 6.2, «Создание динамического стека», на стр. 159). Используйте перечислимый тип для списка условий:

```
enum stack_error { overflow, underflow, ····· };
      , //переполнение, стек пуст, ·····
```

   Напишите перехватчик, который будет использовать инструкцию `switch` для выбора надлежащего сообщения и завершения программы.

5. Напишите класс `stack_error`, заменяющий перечислимый тип из предыдущего упражнения. Сделайте его базовым классом для нескольких производных, которые инкапсулируют каждую специфическую ситуацию исключения. Перехватчики должны уметь использовать замещенные виртуальные функции для обработки различных запущенных исключений.

Глава 12

ООП на C++

C++ — гибридный язык. Ядро языка развилось из C и используется в качестве классического языка для системного программирования. Поэтому C++ весьма подходит для написания эффективного кода. Дополнения языка на основе классов соответствуют набору требований ООП. В этом качестве C++ подходит для написания многократно используемых библиотек и поддерживает полиморфный стиль программирования.

Объектно-ориентированное программирование (ООП) и C++ были очень быстро приняты индустрией программного обеспечения. C++ является гибридным языком объектно-ориентированного программирования. Он предоставляет многоцелевой подход к программированию. Сохранены традиционные преимущества C как эффективного мощного языка программирования. Новыми ключевыми ингредиентами являются наследование и полиморфизм, то есть способность принимать множество форм.

12.1. Требования к языку ООП

Характеристики языка ООП

- Инкапсуляция с сокрытием данных: возможность отделять внутреннее состояние и поведение объекта от его внешнего состояния и поведения.
- Расширяемость типов: возможность добавлять определяемые пользователем типы для расширения набора собственных типов.
- Наследование: возможность создавать новые типы, импортируя или используя повторно описания существующих типов.
- Полиморфизм и динамическое связывание: способность объектов отвечать за интерпретацию вызовов функций.

Эти особенности не могут заменить ни дисциплину программирования[1], ни соглашения, принятые сообществом разработчиков, но могут способствовать их развитию.

Типичные процедурные языки программирования, такие как Pascal и C, обладают ограниченными формами расширяемости типов и инкапсуляции. Оба языка имеют

[1] Под дисциплиной программирования обычно понимают набор неофициальных (а часто и вовсе неписаных) правил, вытекающих не из формальных требований языка, а из общепринятых представлений о том, что есть «хорошая программа». — *Примеч. перев.*

типы указателей и записей, предоставляющие данные свойства. Дополнительно в C имеется специальная ориентированная на файлы схема «закрытости» (privacy) на основе объявлений `static` в области видимости файла. Такие языки, как Modula-2 и Ada, имеют более законченные формы инкапсуляции, а именно, модули и пакеты соответственно. Эти языки позволяют пользователям легко строить АТД и предоставляют значительную библиотечную поддержку для множества прикладных областей. Такой язык, как чистый LISP, поддерживает динамическое связывание. Элементы ООП были доступны в разных языках на протяжении по крайней мере двадцати пяти лет.

LISP, Simula и Smalltalk долгое время широко использовались и в академических, и в исследовательских кругах. Эти языки во многих случаях более элегантны, чем C и C++. Однако до тех пор пока элементы ООП не были добавлены в C, не делалось никаких существенных шагов к использованию ООП в индустрии программного обеспечения. В конце восьмидесятых с триумфом был принят C++, и в этом принятии нового языка были едины компании, целые направления и прикладные области. Мы полагаем, что сфера производства программного обеспечения нуждалась в объединении ООП и возможности эффективного программирования на низком уровне.

Существенной была также легкость перехода от C к C++. В отличие от PL/1, который служил истоком для FORTRAN и COBOL, и Ada, из которого вышел Pascal, для C++ язык C является почти собственным подмножеством. А раз так, от основного установленного кода на C не надо отказываться. Для остальных языков необходим нетривиальный процесс преобразования для изменения существующего кода на языке-предке.

Традиционная академическая мудрость состоит в том, что чрезмерная забота об эффективности вредит хорошему программированию. Такая установка упускает из виду тот очевидный факт, что конкурентоспособность продукта основана на производительности. Соответственно, индустрия ценит технологии низкого уровня. Так что C++ — весьма эффективный инструмент.

12.2. АТД в не-ООП языках

Существующие языки и методики поддерживают большую часть методологии ООП с помощью комбинирования свойств языка и дисциплины программирования. Дисциплина программирования и общепринятые в сообществе разработчиков соглашения действительно работают. В не-ООП языке возможно создание и использование АТД. Тремя примерами на C служат псевдотипы: строковый, булевский и файловый. Они являются псевдотипами (а не типами) в том смысле, что не обладают теми же привилегиями, что и собственные типы. Глядя на эти примеры можно лучше понять ограничения расширяемости типов в не-ООП контексте.

Булевский тип в C представлен неявно. А именно, логические выражения принимают нулевые значения за *false*, а ненулевые — за *true*. Поскольку ноль является универсальным значением, которое доступно для всех типов, ноль по соглашению используется в качестве «охранного» (sentinel) значения. Ноль, используемый для представления конца списка, является идиомой при обработке данных, основанной на указателях.

```
while (p) {          //p == 0 нулевой указатель (NULL)
   .....             //обработка списка
   p = p -> next;    //переход
}
```

Часто для обеспечения лучшей документированности явно используются перечислимые типы:

```
enum boolean { FALSE, TRUE };

boolean search(int table[], int x, int& where)
{
  where = -1;
  for (int i = 0; i < N; ++i)
    if(x == table[i]) {
      where = i;
      break;
    }
  return boolean(where != -1);
}
```

Строковый тип — это комбинация дисциплины программирования и общепринятых соглашений, представленная в библиотеке *string.h*. Эта библиотека применима к типу указателя на символ. Конец строки — опять нулевое значение. Конкатенация, копирование, выяснение длины и другие операции представлены функциями из *string.h*. Мерой успеха библиотеки является частота фактического использования C для создания приложений, в которых применяется обработка строк.

Файловый тип основан на использовании *stdio.h*. Тип структуры (зависящий от системы) определен с именем *FILE*. В *stdio.h* представлены такие функции, как открытие, закрытие и поиск файла. Эти процедуры ожидают в качестве параметров указатели на файл. Конкретные члены структуры не подвергаются непосредственным манипуляциям, если программист придерживается данных соглашений. Опять-таки, C весьма успешно применялся для написания операционных систем и кода, манипулирующего файлами.

Подобные удачи не утверждают статус-кво. Напротив, они «выступают» за встраивание ООП в язык таким образом, чтобы гарантировалось, что библиотечные соглашения не нарушатся.

Заметьте, что в C++ имеется булевский тип, а именно `bool`; стандартный строковый тип `string`, определенный в файле стандартной библиотеки *string*; и более удачная обработка файлов, определенная с помощью `fstream`.

12.3. Клиенты и производители

Чтобы полностью понять парадигму ООП, мы должны рассмотреть весь процесс программирования (кодирования) как занятие, ответственность за которое разделяется и распределяется. Мы используем термин *клиент* (client) для обозначения пользователя класса и термин *производитель* (manufacturer) — для поставщика класса.

Клиент класса предполагает некое приближенное соответствие какой-то абстракции. Стек, чтобы его можно было использовать, должен быть приемлемого размера. Комплексное число должно быть представлено с разумной точностью. Колода должна тасоваться так, чтобы при раздаче карты распределялись случайным образом. Внутренние особенности вычисления поведения всех этих объектов не являются непосредственной заботой клиента. Клиент имеет дело с такими вещами, как стоимость, эффективность и легкость использования, но не с реализацией. Это — принцип *черного ящика*.

Черный ящик в понимании клиента

- простой в использовании, легко понимаемый и привычный
- дешевый, эффективный и мощный
- может быть составной частью системы (может восприниматься системой как компонент)

Черный ящик в понимании производителя

- легко повторно использовать и изменять, но трудно использовать неправильно и сложно воспроизвести
- дешевый, эффективный и мощный
- выгодный для производства с большой базой клиентов

Производитель сражается за клиента, реализуя АТД-продукт, имеющий приемлемую цену и эффективность. Производитель заинтересован в сокрытии деталей разработки. Такое сокрытие значительно упрощает все то, что производитель должен «объяснять» клиенту. Это дает ему свободу для дальнейших внутренних усовершенствований продукта, которые не влияют на особенности использования продукта клиентом. Оно защищает клиента от опасного, пусть даже непреднамеренного искажения продукта.[1]

Структуры и обычные функции в C позволяют строить полезные АТД, но они не поддерживают разграничения между клиентом и производителем. Клиент имеет доступ к внутренним деталям и может модифицировать их нежелательным образом. Рассмотрим стек, представленный в качестве массива с целой переменной `top`. В C клиент такого стека может извлечь внутренний член массива, используемого для представления стека. Это нарушает абстракцию LIFO (Last-In-First-Out, первым вошел — последним вышел), которую реализует стек.

Инкапсуляция объектов предотвращает подобные нарушения. Схема сокрытия данных, которая ограничивает доступ к деталям реализации для всех, кроме производителя, гарантирует клиенту соответствие абстракции АТД. Закрытые члены скрыты от клиентской программы, а открытые доступны ей. Можно изменить представление скрытых данных, но не доступ к открытым членам и их функциональность. Если сокрытие данных произведено правильно, клиентская программа не нуждается в изменениях, когда модифицируется представление скрытых данных.

Двумя ключами к выполнению условий черного ящика служат наследование и полиморфизм.

12.4. Повторное использование кода и наследование

Создание библиотек и их повторное использование являются решающими показателями успеха стратегии языка. Наследование, или производство нового класса от существующего, применяется как для совместного и повторного использования кода, так и для разработки иерархий типов. Посредством наследования может быть создана иерархия родственных АТД, которые разделяют код и общий интерфейс. Это свойство исключительно важно для обеспечения возможности повторно использовать код.

[1] А производителя — от несанкционированного использования придуманных им алгоритмов в продуктах других разработчиков. — *Примеч. перев.*

Наследование влияет на разработку программного обеспечения в целом. Оно предоставляет каркас, в котором фиксируются концептуальные элементы, ставшие предметом пристального внимания при построении и использовании систем. Например, InterView представляет собой библиотеку C++, которая поддерживает построение графического интерфейса пользователя. Главные категории объектов включают интерактивные объекты, текстовые объекты и графические объекты. Эти категории легко комбинируются, позволяя создавать различные приложения, например, системы автоматизированного проектирования — САПР (computer-aided design — CAD), броузеры или редакторы WYSIWYG[1].

Методология объектно-ориентированного проектирования

1. Выбери надлежащую совокупность АТД.
2. Спроектируй взаимосвязи между АТД, применяя наследование для использования общего кода и интерфейса.
3. Применяй виртуальные функции для обработки родственных объектов динамически.

Наследование также способствует принципу черного ящика и является важным механизмом для сокрытия деталей. Будучи иерархическим, каждый предыдущий уровень предоставляет встроенную в него функциональность следующему уровню. Ретроспективно можно отметить, что методология структурного программирования с ее процедурно-направленным взглядом опиралась на метод пошагового уточнения для организации вложенных процедур, но не придавала достаточного значения необходимости соответствующего взгляда на данные.

12.5. Полиморфизм

Полиморфизм — это джин ООП, получающий от клиента приказы и верно интерпретирующий его желания. Полиморфная функция имеет множество форм. По Карделли и Вегнеру (Cardelli and Wegner, 1985) мы различаем:

Типы полиморфизма

1. Принудительное приведение (coercion) (ad hoc полиморфизм): функция или оператор работает с несколькими различными типами, преобразуя их значения к требуемому типу. Примером в ANSI C служит преобразование при присваивании арифметических типов во время вызова функции.

```
a / b     //тип частного обуславливается
          //собственными (встроенными в ядро)
          //принудительными приведениями
```

2. Перегрузка (ad hoc полиморфизм): функция вызывается на основе ее сигнатуры, то есть списка типов аргументов. В C оператор целого деления и оператор

[1] WYSIWYG — What You See Is What You Get, «что видим, то и получим». Принцип, в соответствии с которым приложение (обычно текстовый или графический редактор) гарантирует, что пользователь, распечатав документ, получит именно то, что он видел в ходе работы на экране. Зачастую, правда, пользователи утверждают, что их приложение построено по принципу WYSIWYNG — What You See Is What You Never Get, «что видим, то никогда не получим» — *Примеч. перев.*

деления с плавающей точкой различаются. Выбор варианта оператора деления основан на типах параметров.

```
cout << a   //перегрузка функции
```

3. Включение (inclusion) (чистый полиморфизм): тип является подтипом другого типа. Функция, пригодная для базового типа, будет работать и с подтипом. Такая функция может иметь различные реализации, которые вызываются с учетом выяснения подтипа на этапе выполнения.

```
p -> draw() //вызов виртуальной функции
```

4. Параметрический полиморфизм (чистый полиморфизм): тип остается неопределенным, а позже инстанцируется. В C++ это обеспечивается обобщенными указателями и шаблонами.

```
stack <window*> win[40]
```

Полиморфизм локализует ответственность за поведение. Часто клиентский код не нуждается в пересмотре, когда в систему включается дополнительная функциональность с помощью кода, предоставляемого производителем.

Полиморфизм непосредственно содействует принципу черного ящика. Виртуальные функции, определенные для базового класса, являются интерфейсом, который повсеместно используется клиентом. Клиент знает, что замещенная функция-член несет ответственность за конкретную реализацию данного действия, относящегося к объекту. Клиенту не надо знать о разных процедурах, существующих для каждого вычисления или о различных вариантах спецификации. Эти детали скрыты.

12.6. Сложность языка

За все свои достоинства C++ платит существенную цену: сложность языка весьма ощутима. Это приводит к дополнительным затратам на обучение и едва уловимым ошибкам. Стремительное развитие C++ при столь широком его распространении практически беспрецедентно. Язык C — небольшой и элегантный. Синтаксис C++ похож на C, но семантика первого сложнее. Чтобы оценить эти трудности, в следующей таблице приведены для сравнения некоторые характеристики языков Pascal, Modula-2, Modula-3, C++ и Ada.

	Сложность языка			
Язык	**Ключевые слова**	**Инструкции**	**Операторы**	**Страницы**
Pascal	35	9	16	28
Modula-2	40	10	19	25
Modula-3	53	22	25	50
C	29	13	44	40
C++v1.0	42	14	47	66
C++v3.0 1990	48	14	52	155
Ada 1980	63	17	21	241
C++ ANSI 1995	**62**	**15**	**54**	**650**

Эти цифры наводят на размышления о сложности языка. Согласно приведенным данным, Modula-2 немного сложнее, чем ее предок, Pascal; оба они стоят на одном

уровне с языком C. C++ задумывался как расширение C и сопровождался руководством в стиле C. C++ v3.0 добавил девятнадцать ключевых слов к имевшимся в традиционном C, что увеличило его на две трети. Справочное руководство по C++ v3.0 содержало 155 страниц, в четыре раза больше, чем руководство по C. Эти две оценки наводят на мысль, что C++ v3.0 значительно сложнее, чем C. Проект стандарта ANSI C++ 1995 составлял примерно 650 страниц. Это опять в четыре раза больше по сравнению с 1990 годом. И хотя этот документ включает большое количество дополнительного материала по улучшенной стандартной библиотеке, его объем все же отражает большую сложность языка. Более того, многие конструкции C++ независимы, так что их взаимодействие значительно влияет на сложность.[1]

Подтверждающий пример можно найти на стр. 306 книги Эллис (Ellis) и Страуструпа (Stroustrup) «*C++ Annotated Reference Manual*»[2], где для показа различных вариантов и свойств разных типов функций используется таблица размером 13 на 5. Пятью характеристиками функции являются: наследуемость, виртуальность, возвращаемый тип, дружественная ли это функция или функция-член и генерируемость по умолчанию. Например, конструкторы, деструкторы и функции преобразования не могут объявлять свой возвращаемый тип, а `new()` и `delete()` должны объявлять, причем `void*` и `void` соответственно. C, в действительности, имеет лишь одну форму семантики функции. Такое увеличение в шестьдесят пять раз внушает благоговейный трепет. И хотя в этой таблице присутствует стройность и многие характеристики могут быть получены из концептуального понимания модели языка, авторы, тем не менее, считают целесообразным перечислить различия и свойства разных типов функций.

В C++ ключевые понятия перегружаются несколькими значениями (смыслами). Это приводит к путанице в понятиях. Кандидатом в самые злостные нарушители представляется ключевое слово `static`. Может существовать локальная статическая переменная, то есть переменная, сохраняющая значение при выходе из блока. Допустим статический идентификатор с областью видимости файла, то есть имя, видимость которого ограничивается этим файлом. Может существовать статический член данных класса, который является общим для всех объектов данного класса. Разрешается задать статическую функцию-член, то есть функцию-член, которая не получает аргумент-указатель `this`. Между всеми этими значениями-смыслами существует связь, но в то же время они настолько различны, что довольно трудно правильно понять тот факт, что они родились из общей концепции.

12.7. Успех ООП на C++

Объектно-ориентированное программирование на C++ завоевало ослепительный успех, несмотря на известные недостатки. Причина в том, что C++ привнес в промышленность приемлемую технологию ООП. Он основан на существующем, удачном и широко используемом языке. Он позволяет писать переносимый, компактный и эффективный код. Сохранена безопасность типов, а расширяемость типов являет-

[1] В августе 1998 года стандарт C++ был ратифицирован. Наиболее авторитетное руководство по Стандартному C++, а именно третье издание книги создателя языка Б. Страуструпа «Язык программирования C++», содержит около тысячи страниц. — *Примеч. перев.*

[2] Б. Страуструп, М. А. Эллис, Справочное руководство по языку C++ с комментариями: проект стандарта ANSI, М., «Мир», 1992. — *Примеч. перев.*

ся всеобъемлющей. C++ сосуществует с другими распространенными языками и не предъявляет особых требований к системе.

С изначально создавался как язык для разработки операционных систем, и как таковой позволяет писать код, который легко транслируется для эффективного использования машинных ресурсов. Благодаря этой эффективности программные продукты приобретают существенные преимущества. Поэтому, несмотря на жалобы на то, что традиционный С не является безопасным и надежным языком, С развивался в своей сфере применения. Сообщество программистов на С использовало структурное программирование и АТД-расширения на основе специальных соглашений и правил. ООП вторглось в это профессиональное сообщество лишь когда оно (ООП) «поженилось» с языком С в рамках концепции, которая поддерживала и традиционную точку зрения С, и преимущества ООП. Ключом к популярности C++ было понимание того, что наследование и полиморфизм дают важные дополнительные преимущества по сравнению с традиционной практикой программирования. Полиморфизм в C++ позволяет клиенту использовать АТД как черный ящик. Успех ООП характеризуется той степенью, в которой пользовательский тип может быть сделан неотличимым от собственного типа языка. Полиморфизм позволяет задать приведения типов, которые интегрируют АТД с собственными типами. Он разрешает объектам из иерархии подтипов динамически реагировать на вызов функции; в этом заключается принцип сообщений в ООП. Полиморфизм также упрощает клиентский протокол, а «размножение» имен управляется перегрузкой функций и операторов. Наличие всех четырех форм полиморфизма вдохновляет программиста на проектирование с учетом инкапсуляции и сокрытия данных.

ООП много значит для многих людей. Все попытки дать ему определение напоминают старания слепых мудрецов описать слона. Напомним нашу формулу, определяющую объектно-ориентированность:

ООП = расширяемость типов + полиморфизм.

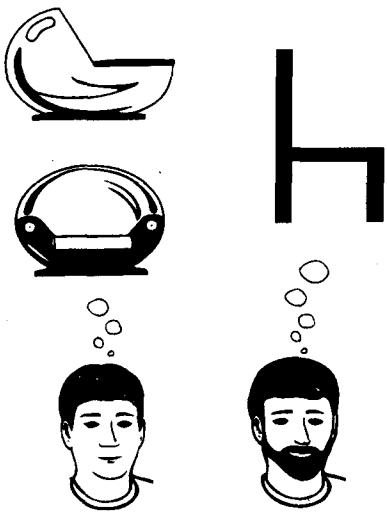
Во многих языках и системах ценой сокрытия деталей была неэффективность на этапе выполнения и чрезмерная жесткость (негибкость) интерфейса. C++ предлагает ряд вариантов, обеспечивающих и гибкость, и эффективность. Как следствие, этот язык будет иметь все больший и больший успех в промышленности.

12.8. Платонизм: проектирование «tabula rasa»

C++ дает программисту инструмент для реализации объектно-ориентированного проекта. Но как вы разрабатываете такой проект? Не бывает простой методики tabula rasa[1], то есть проекта «на ровном месте». Любой проект всегда должен быть тесно привязан к предметной области и отражать ее абстракции. Раскрыть эти абстракции позволяет философия проектирования, которую мы называем *платонизмом* (platonism).

[1] tabula rasa (лат.) — буквально: «чистая, гладкая дощечка». В философском смысле — нечто чистое, нетронутое, свободное от всяких влияний. То есть проектирование tabula rasa — это отвлеченное проектирование с нуля, без учета реалий и взаимосвязей окружающего мира. Ученый из известного анекдота, создавший для расчета вероятности выигрыша на скачках модель сферического коня в вакууме, осуществил проект tabula rasa. — *Примеч. перев.*

В соответствии с платонической парадигмой существует идеальный объект. Например, представьте идеальный стул и попытайтесь описать его характеристики. Это должны быть свойства, разделяемые всеми стульями Вселенной. Такой стул может быть подкатегорией другого идеала — мебели. Стул может иметь подкатегории, такие как вращающийся стул, шезлонг, стул с подлокотниками, кресло-качалка и тому подобное. Для толкового описания стула, возможно, потребуется устроить экспертизу по стульям, с привлечением производителей и пользователей стульев с целью прийти к соглашению о сути и природе «стульности». Платонический стул должен легко модифицироваться для описания наиболее часто встречающихся стульев. Платонический стул следует описывать в терминах, согласовывающихся с существующей «стульной» терминологией.



На C++ оказал влияние Simula 67 — язык, специально разработанный для моделирования. Платоническая парадигма является моделированием или симуляцией конкретного мира. Она несет в себе дополнительные возможности по формированию объектно-ориентированного проекта программного продукта. Объектно-ориентированное проектирование обычно предоставляет открытый интерфейс, который является удобным, обобщенным и эффективным. Эти важные моменты могут вступать в противоречия. Еще раз отметим, что нет простых правил, позволяющих разрешить подобные конфликты.

Дополнительные возможности могут оказаться очень выгодными и существенно перекроют увеличение стоимости начального проекта. Первое и главное: они налагают дополнительный уровень дисциплины на процесс программирования. Усиление такой дисциплины всегда приносит дивиденды. Второе: идеологически связанные друг с другом, наполненные смыслом куски кода инкапсулируются в классы. Инкапсуляция и декомпозиция (разбиение на составляющие) тоже всегда приносили свои плоды. Третье: совершенствуется повторное использование кода с помощью наследования и АТД. Повторное использование кода всегда выгодно. Четвертое: улучшается прототипирование посредством откладывания выбора реализации и предоставления доступа к большим и удобным общим библиотекам. Дешевые прототипы всегда окупаются.

Платоническая парадигма применения техники ООП проводит бархатную революцию во всем процессе программирования. Она не вытесняет более раннюю технику, например приемы структурного программирования, используя их для программирования «в малом», чтобы потом эффективнее управлять композицией больших программных проектов.

12.9. Принципы проектирования

Как правило, при программировании следует использовать уже имеющиеся наработки. Например, в математическом и научном сообществе существуют стандартные определения для комплексных и действительных чисел, матриц и многочленов. Каждое из них можно легко закодировать, представив в виде АТД. Ожидаемое открытое поведение этих типов принимается всеми математиками.

Сообщество разработчиков обладает обширным опытом по использованию стандартных контейнерных классов. Приемлемые соглашения существуют в отношении поведения стека, ассоциативного массива, двоичного дерева, очереди. Кроме того, у сообщества программистов есть много примеров специализированных языков программирования, ориентированных на конкретную проблемную область. Например, язык SNOBOL и выросший из него язык ICON имеют очень богатые возможности для обработки строк. Подобные возможности могут быть собраны в виде АТД в C++.

Полезным принципом проектирования служит принцип «лезвия Оккамы[1]» (Occam's razor). Этот принцип утверждает, что не следует вводить новые сущности сверх необходимости — или сверх законченности, обратимости, ортогональности, согласованности, простоты, эффективности и выразительности. Подобные идеалы могут конфликтовать, что часто приводит к неизбежной «торговле» между ними, когда дело доходит до реализации проекта.

Обратимость (invertibility) означает, что программа должна иметь функции-члены, обратные по отношению друг к другу. Для математических типов сложение и вычитание являются обратными операциями. В текстовом редакторе ими служат добавление и удаление элементов текста. Некоторые операции сами себя обращают, например, операция отрицания. В нематематическом контексте важность обратимости можно понять на примере блестящего успеха команды «Отменить» (Undo) в текстовых редакторах и команд восстановления в менеджерах файлов.

Законченность (completeness) лучше всего рассматривать на примере булевой алгебры, где единственной операции «и-не» достаточно для конструирования всех возможных булевских выражений. Но при изучении булевой алгебры в качестве основных операций обычно рассматриваются отрицание, логическое умножение (конъюнкция) и логическое сложение (дизъюнкция). Самой по себе законченности недостаточно для того, чтобы судить о проекте. Больший набор операторов часто и более выразителен.

Ортогональность (orthogonality) — это принцип, который гласит, что любой элемент проекта должен интегрироваться и работать со всеми остальными элементами без перекрытий и избыточности. Например, система, которая оперирует геометрическими фигурами, должна включать операции горизонтального и вертикального

[1] Оккам (Ockham или Occam), Уильям (ок. 1285–1349), английский философ-схоласт, логик, писатель. Главный представитель номинализма XIV в., францисканец. Согласно принципу «лезвия Оккамы», понятия, несводимые к интуитивному и опытному знанию, должны удаляться из науки. — *Примеч. перев.*

перемещения и операцию поворота. Их достаточно, чтобы расположить фигуру в любом месте экрана.

Иерархичность (hierarchy) достигается с помощью наследования. Проекты должны быть иерархичны, что является отражением двух принципов — декомпозиции и локализации. Оба они служат для сокрытия деталей — ключевой идеи в борьбе со сложностью. Однако при подобном проектировании существует серьезная проблема. Какой детализации достаточно для того, чтобы сделать сущность пригодной в качестве класса? Важно избежать разрастания излишне детализированных сущностей. Избыточные детали слишком усложняют управление проектом классов.

12.10. Схемы, диаграммы и инструменты

Процесс проектирования можно облегчить с помощью диаграмм. Существует несколько систем обозначений (нотаций) для объектно-ориентированного проектирования. Некоторые из них используются автоматизированными системами разработки программ (computer assisted software engineering — CASE-средства). Опишем две схемы, которые мы находим полезными. Первая из них — это CRC-карточки (Budd, 1991), а вторая — диаграммы Вассермана-Пирчера, (Wasserman et. al., 1990; в качестве альтернативы — Booch, 1995).

CRC означает *класс*, *ответственность*, *сотрудничество* (class, responsibility, collaboration). Ответственность — это обязательства, которые должен выполнять класс. Например, объекты комплексных чисел должны предоставлять реализацию комплексной арифметики. Сотрудник (collaborator) — это другой объект, который взаимодействует с данным для обеспечения некоего общего набора поведений. Например, целые и действительные числа сотрудничают с комплексными для предоставления исчерпывающего набора математических поведений.

CRC-карточки используются для проектирования заданного класса. Сначала описываются ответственность класса и его сотрудники. Оборотная сторона карточки применяется для описания деталей реализации. Лицевая сторона карточки соответствует открытому поведению.

CRC-карточка

лицевая сторона

имя класса: `stack`	сотрудники:
ответственность:	нет

`push`
`pop`
`empty`	

открытое поведение	

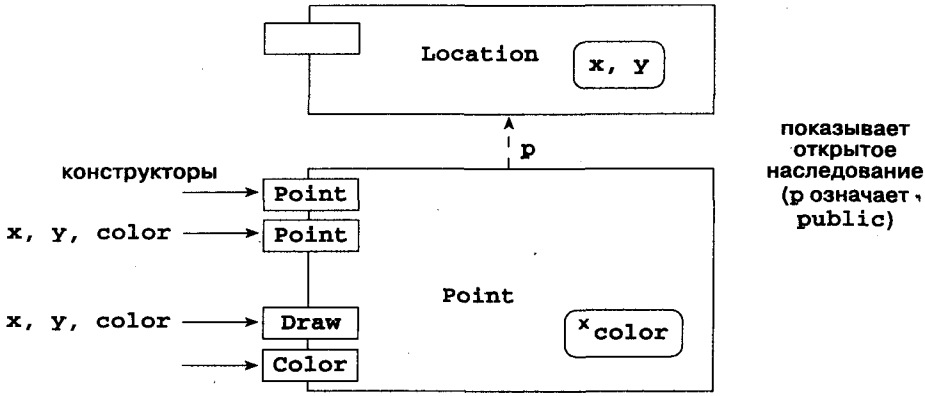
оборотная сторона

состояние/описание
`top` `base_pointer`

В процессе проектирования карточки уточняются и переписываются. Они становятся более детальными и похожими на набор заголовков функций-членов. Оборотная сторона карточки может использоваться для показа деталей реализации, включая отношения ISA, LIKEA и HASA.

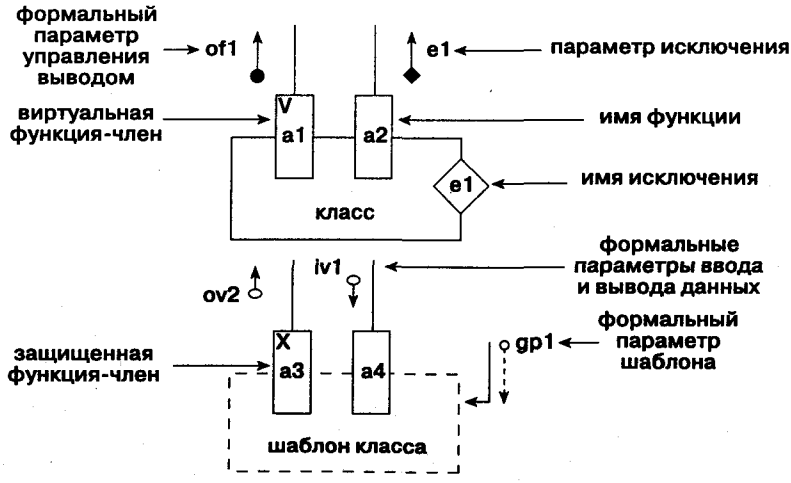
Привлекательностью подобной схемы является ее гибкость. На самом деле она представляет собой процесс уточнения псевдокода, причем этот процесс может отражать личные представления проектировщиков. Число пересмотров и уровень детализации и точности — это дело вкуса (Budd, 1991).

Диаграммы Вассермана-Пирчера восходят к модели «сущность-связь» (entity-relation) и структурному проектированию.



Комплексная интегрированная среда разработки ООП-кода, предлагаемая фирмой Interactive Development Environments (IDE), Сан-Франциско, Калифорния, использует приведенную выше схему. Проектирование программного обеспечения отображено на следующих диаграммах.

Шаблоны проектирования на C++ от IDE



Уровень детализации таков, что из проекта могут быть автоматически получены заглушки кода. Кроме того, одновременно можно проверить или сгенерировать документацию и стилевые правила.

12.11. Штампы проектирования

Повторное использование кода — вот основная тема современного программирования. На первых парах оно ограничивалось простыми библиотеками функций, например математических из *math.h* или строковых из *string.h*. В ООП основной конструкцией для повторного использования становятся класс или шаблон. Классы и шаблоны инкапсулируют код, пригодный для определенных проектов. Так, классы-итераторы из STL служат штампом проектирования (design pattern). С недавних пор концепция штампов проектирования стала очень популярной при обеспечении повторного использования в среднем масштабе (см. Gamma, 1994). Штамп проектирования имеет четыре элемента:

Элементы штампа проектирования

1. Терминология штампа. Например, *итератор*.
2. Задача и условия. Например, *перебор* в контейнере.
3. Решение. Например, подобные указателям объекты с общим интерфейсом.
4. Оценка. Например, выбор между определением итератора на основе вектора и использованием собственного типа массива C++.

Штамп проектирования — это абстракция, которая предлагает подходящее решение для конкретной задачи программирования. Повторное использование часто не требует больших затрат. Так происходит в случае с контейнерами STL и итераторными штампами проектирования, которые нуждаются лишь в инстанцировании. Иногда повторное использование все же дорого обходится, например, изобретение с нуля сбалансированного класса дерева с интерфейсом, соответствующим последовательным контейнерам STL.

Штампы проектирования в этой книге

1. *Итератор*. Например, `vector::iterator`. Организует перебор в контейнере.
2. *Композиция*. Например, класс `grad_student`. Составляет сложные объекты из более простых.
3. *Метод шаблонов*. Например, шаблон `quicksort()`.

12.12. C++: критика

В какой-то степени C++ является версией 90-х годов языков PL/1 (1965 г.) или Ada (1980 г.). C++ — результат предпринятой в среде профессиональных разработчиков попытки предложить почти универсальный язык программирования. Дефектом PL/1 было то, что он смешал в себе слишком много стилей — он объединил COBOL, FORTRAN и элементы ALGOL в одном языке. Недостатками Ada были его размер, сложность и неэффективность. У C++ есть проблемы с размером и сложностью, но очень важно то, что он основывается на существующих ресурсах и практике. Он также придает особое значение эффективности, иногда даже слишком большое.

Некоторых проблем C++, связанных со сложностью, можно избежать, придерживаясь «идеологически правильных» взглядов на смысл тех или иных свойств языка. Например, чисто виртуальная функция может быть определена с исполняемым кодом:

```
class ABC {
public:
  virtual void f() = 0;
};
void ABC::f() { cout << "чисто виртуальная f" << endl; }

//должна вызываться с уточненным именем: x.ABC::f();
```

То, что такая конструкция допускается языком, выглядит эксцентрично. По идее же, чисто виртуальная функция используется для того, чтобы отложить определение.

Другие сложности языка принципиальны с точки зрения дизайна C++, например отсутствие сборки мусора (garbage collection). Существует несколько предложений на этот счет (см. Edelson, 1991, 1992, Boehm, 1988), и их реализация подтверждает, что сборка мусора может быть выполнена для большинства прикладных задач без ущерба для производительности. Большинство основных языков ООП, такие как Smalltalk, CLOS и Eiffel, поддерживают сборку мусора. Аргументом в пользу сборки мусора служит то, что такая встроенная возможность значительно облегчает задачу программисту. Потери памяти и ошибки указателей — вполне обычные проблемы, когда каждый класс предусматривает собственное управление памятью. Это очень сложные для выявления и отладки ошибки. Сборка мусора — хорошо известная технология, так почему бы ее не использовать?

Аргументом против сборки мусора является то, что она, когда применяется универсально, означает неявные затраты для всех пользователей. Кроме того, сборка мусора управляет памятью, но не другими ресурсами. Управление остальными ресурсами все равно нуждается в деструкторах для завершения их использования, то есть для возврата ресурсов системе и для других действий, необходимых при истечении срока жизни объекта. Объектом, например, может быть файл, и для завершения его использования файл необходимо закрыть. Наконец, не в правилах сообщества программистов на C иметь автоматически управляемую свободную память.

ООП стремится придать особое значение повторному использованию кода. Оно возможно в различных масштабах. Самый крупный — это разработка библиотек, которые подходят для всей проблемной области. С одной стороны, повторное использование в конце концов способствует более простому сопровождению кода. С другой стороны, конкретное приложение не нуждается в дорогостоящей разработке библиотеки.

ООП требует от программиста искушенности. Искушенный программист — хороший программист. Однако высокая стоимость обучения и вероятность неправильного применения изощренных приемов являются их минусом.

ООП делает клиентский код легче расширяемым и более простым. Для включения локальных изменений в крупномасштабную систему без ее глобальных модификаций может быть использован полиморфизм. Минус — накладные расходы на этапе выполнения.

C++ предоставляет программисту инкапсуляцию с помощью классов, наследования и шаблонов. Инкапсуляция скрывает и локализует данные. Когда системы становятся больше и сложнее, нужда в ней все возрастает. Простой блочной структуры и

инкапсуляции функций в таких языках, как FORTRAN и ALGOL, недостаточно. 70-е годы помогли нам понять необходимость модуля как единицы программирования. В 80-х мы поняли: модули должны быть логически согласованы, что должно поддерживаться языком, причем они должны наследоваться друг от друга. Поддерживаемые языком программирования инкапсуляция и родственные отношения приводят к укреплению дисциплины программирования. Искусство программирования заключается в сочетании строгости и дисциплины с творчеством.

Резюме

1. Объектно-ориентированное программирование (ООП) и C++ были очень быстро приняты индустрией программного обеспечения. C++ является гибридным языком объектно-ориентированного программирования. Он предоставляет многоцелевой подход к программированию. Сохранены традиционные преимущества C как эффективного мощного языка программирования. Новым ключевым ингредиентом является полиморфизм, то есть способность принимать множество форм.

2. Существующие языки и методики поддерживали большую часть методологии ООП с помощью комбинирования свойств языка и дисциплины программирования. В не-ООП языке возможно создание и использование АТД. Тремя примерами из сообщества C являются: строки, булевские переменные и файлы, которые являются псевдотипами (а не типами) в том смысле, что не обладают теми же привилегиями, что и «настоящие» типы. Глядя на эти примеры можно лучше понять ограничения расширяемости типов в не-ООП контексте.

3. Черный ящик в понимании клиента должен быть прост в использовании, легко понимаем и привычен; дешев, эффективен и мощен; способен быть составной частью системы. Черный ящик в понимании производителя должен быть легок в повторном использовании и изменении, но затруднителен для неправильного использования и сложен для воспроизведения; дешев, эффективен и мощен; выгоден для производства при большой клиентской базе.

   Методология объектно-ориентированного проектирования

   - Выбери надлежащую совокупность АТД.
   - Спроектируй взаимосвязи между АТД, примененяя наследования для использования общего кода и интерфейса.
   - Используй виртуальные функции для обработки родственных объектов динамически.

4. Полиморфизм непосредственно содействует принципу черного ящика. Виртуальные функции, определенные для базового класса, являются интерфейсом, который повсеместно используется клиентом. Клиент знает, что замещенная функция-член несет ответственность за конкретную реализацию данного действия, относящегося к объекту.

5. В качестве гибридного языка ООП C++ может вызывать у программиста некое «диалектическое напряжение». Склонность программистов на C концентрироваться на эффективности и реализации конфликтует со склонностью «объективистов» сосредотачиваться на элегантности, абстрактности и обобщенности. Оба эти

требования к процессу кодирования совместимы, но нуждаются в мерах по координации и в уважении, которое должно оказываться этому процессу.

6. ООП много значит для многих людей. Вот наша формула:

   ООП = расширяемость типов + полиморфизм

   Во многих языках и системах ценой сокрытия деталей была неэффективность на этапе выполнения и чрезмерная жесткость (негибкость) интерфейса. C++ предлагает ряд вариантов, обеспечивающих и гибкость, и эффективность.

7. Не бывает простой методики tabula rasa, то есть проекта «на ровном месте». Любой проект всегда должен быть тесно привязан к предметной области и отражать ее абстракции. Раскрыть эти абстракции позволяет философия проектирования, которую мы называем *платонизмом* (platonism). В соответствии с платонической парадигмой существует идеальный объект, чьи характеристики разделяются всеми объектами данного типа.

8. Полезным принципом проектирования является лезвие Окамы. Этот принцип утверждает, что не следует вводить новые сущности сверх необходимости — или сверх законченности, обратимости, ортогональности, согласованности, простоты, эффективности и выразительности. Подобные идеалы могут конфликтовать, что часто приводит к неизбежной «торговле» между ними, когда дело доходит до реализации проекта.

Упражнения

1. Рассмотрите следующие три способа представления булевского типа:

```
//В традиционном C используется препроцессор
#define TRUE  1
#define FALSE 0
#define Boolean int

//ANSI C и C++ использует перечислимые типы
enum Boolean { false, true };

//В C++ в виде класса
class Boolean {
   .....
public:
   //различные функции-члены
   //включая перегрузку !, &&, ||, == и !=
};
```

   Обсудите преимущества и недостатки каждого способа. Примите во внимание проблемы, связанные с областью видимости, именами и преобразованиями.

2. Изначально в C++ допускалось изменение указателя `this`. Одно из применений — наличие у пользователя возможности управлять памятью посредством прямого присваивания указателю `this`. Присваивание нуля означало, что соответствующая память может быть объявлена свободной. Обсудите, в чем недостаток этой идеи.

3. Правила выбора того, какое определение перегруженной функции должно быть вызвано, изменились по сравнению с первой версией C++. Одной из причин этого стало стремление уменьшить число неопределенностей. Критике подверглись правила, допускающие установление соответствия с помощью преобразований, которые могли и не подразумеваться программистом, что могло вызвать трудности при поиске ошибок в программе на этапе выполнения. По одной из стратегий компилятор должен выдавать в таких случаях диагностическое сообщение, по другой в целях защиты от подобных ситуаций следует использовать приведения, для того чтобы информировать компилятор о задуманном выборе. Обсудите эти варианты, выяснив предварительно, как изменились правила.

4. Перечислите три вещи, которые вы исключили бы из C++. Докажите, что в результате языку не будет их не хватать. Например, допустимо защищенное наследование. Оно в данной книге не обсуждалось. Должно ли оно присутствовать в языке ради его полноты?

5. Опишите по крайней мере два понимания ключевого слова `static` в C++. Приводит ли разнообразие этих концепций к путанице?

6. Пакет *string.h* является псевдотипом. Он использует технологию традиционного C и дисциплину программирования для предоставления АТД «строка». Почему предпочтительнее стандартный библиотечный класс `string`?

Приложение А

Коды символов ASCII

Американский станадртный код обмена информацией[1]

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Некоторые замечания

- Символы с кодом от 0 до 31 и 127 не печатаются.
- Символ с кодом 32 выводит одиночный пробел.
- Коды цифр от 0 до 9 следуют подряд.
- Коды букв от A до Z следуют подряд.
- Коды букв от a до z следуют подряд.
- Код прописной буквы отличается от кода строчной на 32.

[1] ASCII (American Standard Code for Information Interchange) —американский стандартный код обмена информацией. — *Примеч. перев.*

Значение некоторых аббревиатур

`bel`	звуковой сигнал (audible bell)	`ht`	горизонтальная табуляция (horizontal tab)
`bs`	забой (backspace)	`nl`	перевод строки (newline)
`cr`	возврат каретки (carriage return)	`nul`	нуль (null)
`esc`	искейп-сивол (escape)	`vt`	вертикальная табуляция (vertical tab)

Приложение B

Приоритет и порядок выполнения операторов

Оператор	Порядок выполнения
:: (глобальная область видимости) :: (область видимости класса)	слева направо
() [] -> . (постфикс)++ (постфикс)-- typeid()	слева направо
++(префикс) --(префикс) !~ sizeof(тип) &(адрес) +(унарный) -(унарный) *(разыменование) delete new	справа налево
.* ->*	слева направо
* / %	слева направо
+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&	слева направо
^	слева направо
|	слева направо
&&	слева направо
||	слева направо
= += -= *= /= %= >>= <<= &= ^= |=	справа налево
?:	справа налево
, (оператор запятая)	слева направо

Если вы сомневаетесь в приоритете операторов, используйте скобки.

Приложение C

Руководство по языку

Это приложение является кратким руководством по C++. Здесь собрано множество ключевых элементов языка, которых не было в более ранних процедурных языках, таких как Pascal и C. Руководство задумано как удобный справочник по языку.

C.1. Структура программы

Организация программы на C++

- Ввод-вывод в C++ обеспечивается внешней стандартной библиотекой. Информация, необходимая программе для использования этой библиотеки, находится в файле *iostream.h*.
- Для преобразования программы из предварительного формата в чистый синтаксис C++ используется препроцессор, обрабатывающий набор директив, таких как директива `include`. Эти директивы предваряются символом #.
- Программа на C++ состоит из объявлений, которые могут находиться в разных файлах. Каждая функция располагается на внешнем (глобальном) уровне; объявления функций не могут быть вложенными. Файлы программы выступают в качестве модулей и могут компилироваться отдельно друг от друга.
- Функция `main()` используется как точка входа для выполнения программы. Она подчиняется правилам C++ для объявления функций. Обычно функция `main()` неявно возвращает целое значение 0, что означает нормальное завершение программы. Другие возвращаемые значения нужно задавать явно (с помощью ключевого слова `return`); они означают ошибку.
- Макро `assert` проверяет выполнение условий и прерывает программу, если «тест не сдан».

C.2. Лексические элементы

Программа представляет собой последовательность символов, которые объединяются в лексемы (token) и из которых состоит базовый словарь языка. Существует шесть категорий лексем: ключевые слова, идентификаторы, константы, строковые константы, операторы и знаки пунктуации.

Вот символы, которые можно использовать для образования лексем:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
+ - * / + () {} [] <> ' " ! # ~ % ^ & _ : ; , . ? \ |
Символы пробелов и табуляции
```

При генерации лексем компилятор выбирает наиболее длинную строку символов, которая составляет лексему.

С.2.1. Комментарии

В C++ есть символ комментария в конце строки — //. Допустимы также и пары символов комментария в стиле C, а именно, /* */. Комментарии не могут быть вложенными. Вот некоторые примеры:

```
//ООП на C++. Addison-Wesley. Программа НОД

const int N = 200; //N — это число попыток

/*   * * * * * *
   Программист:  Лаура М. Пол
   Компилятор:   Borland 5.0
   Изменения:    5-2-96   Переполнение стека
*    * * * * * * * */
```

За исключением длинных многострочных комментариев, следует использовать однострочные комментарии в конце строки. Такой стиль проще в использовании и реже приводит к ошибкам.

С.2.2. Идентификаторы

Идентификатор может состоять из одного или более символов. Первым символом должна быть буква или знак подчеркивания. Остальными символами могут быть буквы, цифры или знаки подчеркивания. Хотя, в принципе, идентификаторы могут быть произвольной длины, многие системы распознают только первые 31 символ. Идентификаторы, содержащие двойное подчеркивание или начинающиеся с символа подчеркивания с последующей прописной буквой, зарезервированы для использования системой.

Примеры идентификаторов			Замечания
multiWord	vector	flag_x	нормально
q213	sb3	abx1w	туманно
speed	Speed	speedy	различные, но путано
_Sys1	__Adriver	__C__	зарезервировано для системы
9illegal	wrong-2	il$form	недопустимо
typeid	this	register	ключевые слова нельзя использовать

C.2.3. Ключевые слова

Ключевые слова — это явно зарезервированные идентификаторы, которые имеют в C++ особый смысл. Их нельзя переопределить или использовать в другом контексте. Существуют другие ключевые слова, специфичные для конкретной реализации, например, `near` и `far` в Borland C++. Следующие ключевые слова используются в большинстве современных систем C++.

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

C.3. Константы

В C++ имеются константы для каждого из основных типов. В частности, существуют целые, символьные константы и константы с плавающей точкой. Строковые константы представляют собой последовательности символов, заключенные в двойные кавычки. Существует одна универсальная константа-указатель, а именно 0. Вот некоторые примеры:

Примеры констант				Замечания
`156`	`0156`	`0x156`		целые: десятичное, восьмеричное, шестнадцатеричное
`156l`	`156u`			целые: длинное, беззнаковое
`'A'`	`'a'`	`'7'`	`'\t'`	символы: `A`, `a`, `7`, табуляция
`3.17f`	`3.1415`	`3.14159L`		константы с плавающей точкой
`"Строка"`				строковая константа
`true`	`false`			булевские константы

Суффиксы `u` или `U`, `l` или `L` и `f` или `F` используются для обозначения беззнаковых чисел, длинных чисел и чисел с плавающей точкой (`unsigned`, `long` и `float`), соответственно. Беззнаковые константы являются неотрицательными числами. Длинные имеют большую точность, чем обычные константы. Константы с плавающей точкой `float`, как правило, менее точны, чем обыкновенные константы с двойной точностью `double`.

Символьные константы обычно даются в одинарных кавычках, например, `'s'`. Для некоторых непечатаемых и специальных символов требуются escape-последовательность.

Символьные константы	
`'\a'`	звуковой сигнал (alert)
`'\\'`	обратная косая черта (backslash)
`'\b'`	забой (backspace)
`'\r'`	возврат каретки (carriage return)
`'\"'`	двойные кавычки (double qoute)
`'\f'`	прогон листа (formfeed)
`'\t'`	табуляция (tab)
`'\n'`	перевод строки (newline)
`'\0'`	нулевой символ (null character)
`'\''`	апостроф (single quote)
`'\v'`	вертикальая табуляция (vertical tab)
`'\101'`	восьмеричный ASCII-код `'A'`
`'\x041'`	шестнадцатиричный ASCII-код `'A'`
`L'oop'`	`wchar_t` константа

Константы с плавающей точкой могут быть заданы с целой экспонентой со знаком или без таковой.

Примеры констант с плавающей точкой			Замечания
`3.14f`	`1.234F`		«узкие» константы `float`
`0.123456`	`.123456`		константы `double`
`0.12345678L`	`0.123456781`		длинные константы `long double`
`3.`	`3.0`	`0.3E1`	все обозначает `double 3.0`
`300e-2`	`.03e2`	`30e-1`	тоже `3.0`

Строковые константы рассматриваются как `static char[]` константы. Строковая константа представляет собой последовательный непрерывный массив символов. Строковые константы, разделенные только пустым местом, неявно объединяются в одну строку. Символ обратной косой черты \ в конце строки означает, что после него строка продолжается. Обратная косая черта перед двойными кавычками делает их частью строки. В качестве ограничителя (завершителя) компилятор помещает в конец строки нулевой символ.

Пример строковой константы	Замечания
`""`	пустая строка есть `'\0'`
`"Даешь ООП!"`	`'Д' 'а' 'е' 'ш' 'ь' ' ' 'О' 'О' 'П' '!' '\0'`
`"Моя \"цитата\" пропала"`	`\"` используется для вставки в строку "
`"Возможны также и \` `многострочные строки"`	\ в конце строки означает, что после него строка продолжается

`"Это одна строка,"`
`"Так как она разделена только "`
`"пробелами и переводами строки."`
	неявно объединяется

Перечисления определяют набор именованных констант, которые называются перечислимыми константами. Эти константы представляют собой список идентификаторов, которые являются неявной последовательностью целых значений, начинающейся с нуля. Перечисления могут быть безымянными или определять различные типы.

Перечислимые константы	Замечания
`enum { off, on };`	`off == 0, on == 1`
`enum color { red, blue, white, green };`	`color` — это тип
`enum { BOTTOM = 50, TOP = 100, OVER };`	`OVER == 101`
`enum grades { F = 59, D = 60, C = 70,` `B = 80, A = 90 };`	все инициализованы

В выражениях тип перечислимых констант повышается до целых.

Ключевое слово `const` используется для объявления того, что значение объекта остается неизменным в пределах его области видимости:

Использование ключевого слова const	Замечания
`const int N = 100`	N не может меняться
`double w[N]`	[использует постоянные выражения]
`const int bus_stops[5] =` `{ 23, 44, 57, 59, 83 };`	значения элементов `bus_stops[i]` являются константами

Для преобразования программы из предварительного формата в чистый синтаксис C++ используется препроцессор, обрабатывающий набор директив, таких как директива `include`. Эти директивы предваряются символом #.

Использование `const` отличается от следующего использования `#define`:

```
#define N 100
```

В случае объявления `const int N`, N является неизменяемым lvalue типа `int`. В случае макро `define`, N — это просто макро. Кроме того, замена N с помощью макро происходит как подстановка препроцессора, невзирая на другие правила области видимости.

C.4. Объявления и правила области видимости

Объявления придают смысл данному идентификатору. Синтаксис объявлений C++ весьма сложен, так как он объединяет множество разнородных элементов, которые зависят от контекста. Объявление снабжает идентификатор типом, классом памяти и областью видимости (см. раздел 2.4.1, «Инициализация», на стр. 45). Простое объявление часто является и определением. Для простых переменных это означает, что объект был создан и, возможно, инициализован. Для функции определение представляет собой тело функции, то есть стоящие в фигурных скобках инструкции, выполняемые функцией.

```
const int n = 17;  //n объявлено и определено
int sqrt(double)   //sqrt объявлена, но не определена
```

```
void foo()          //foo объявлена и определена
{
   int i = 5;       //i определена и инициализована
   .....            //i является автоматической
                    //и локальна по отношению к foo
}
```

Сложные объявления, такие как объявления классов, функций и шаблонов, обсуждаются в отдельных разделах данного приложения.

Для создания синонима типа может быть использован механизм `typedef`.

typedef	Замечания
`typedef int BOOLEAN;`	применялось до появления типа `bool`
`typedef char* c_string;`	`c_string` указатель на `char`
`typedef void (*ptr_f)();`	указатель на `void fcn()`

В C++ существуют область видимости файла, область видимости функции, область видимости класса, область видимости прототипа функции и область видимости пространства имен. Область видимости файла, известная также как глобальная область видимости, простирается от точки объявления идентификатора в файле и до конца этого файла. Область видимости прототипа функции является областью видимости идентификаторов в списке аргументов прототипа функции и простирается до конца объявления. Блоки могут быть вложенными, а функции не могут объявляться внутри других функций или блоков.

Объявления могут встречаться в любом месте блока. Объявление также может быть инициализатором в инструкции `for` (в качестве примера см. раздел 2.8.5, «Инструкция for», на стр. 56, в файле *for_test.cpp*).

Инструкции выбора, такие как `if` или `switch`, не должны непосредственно контролировать объявления. Переходы и ветвления, вообще говоря, не должны обходить инициализацию.

```
if (flag)
   int j = 6;          //недопустимо
else
   j = 19;

if (flag) {
   int j = 6;          //допустимо внутри блока
   cout << j;
}
```

В C++ имеется оператор разрешения области видимости `::`. При использовании формы `::` *переменная* он позволяет получить доступ к переменной, внешней по отношению к данной области видимости. Другие применения этой нотации важны для классов и пространств имен. Идентификатор члена класса локален по отношению к этому классу. Оператор разрешения области видимости может использоваться для того, чтобы избежать двусмысленностей. Когда он используется в виде *имя_класса* `::` *переменная*, он позволяет получить доступ к указанной переменной данного класса:

```
class A {
public:
   static void foo();
};

class B {
public:
   void foo() { A :: foo(); ····· }
};
```

Доступ к скрытому внешнему имени может быть получен с помощью оператора разрешения области видимости:

В файле scope1.cpp

```
int i;                  //внешняя i
void foo(int i);        //i – параметр
{
   i = ::i;             //параметру i присвоено внешнее i
   .....
}
```

Классы могут мыть вложенными. По правилам C++ вложенный класс принадлежит области видимости объемлющего класса. Это является источником путаницы, поскольку правила изменились и отличаются от правил C. В качестве примера смотрите раздел 5.4.2, «Вложенные классы», на стр. 139 (в файле *nested.cpp*).

Перечислимым константам, объявленным внутри класса, назначается область видимости класса, в котором они объявлены:

```
class foo {
public:
   enum button { off, on } flag;
};

int main()
{
   foo c;

   c.flag = foo::off;
   .....
}
```

С.5. Пространства имен

Традиционно в C++ имелось единственное глобальное пространство имен. При комбинировании программ, написанных разными людьми, могли (неумышленно) возникать конфликты имен. C++ поощряет использование библиотек, поставляемых различными производителями. Это приводит к идее добавления области видимости пространства имен:

```
namespace std {    //приводим в соответствие со стандартом
   #include <iostream.h>
}

namespace LMPinc { //программа от компании игрушек LMP
   class puzzles { ····· };
   class toys { ····· };
   ·····
}
```

Вообще говоря, в объявлениях используется полное (с указанием всех объемлющих идентификаторов) имя. Объявление `using` позволяет использовать эти имена без идентификаторов пространства имен:

```
using namespace std;
using namespace LMPinc;
toys top;      //LMPinc::toys
```

Объявление пространства имен, также как и объявление класса, может быть использовано в качестве части идентификатора, указанного с разрешением области видимости.

Пространства имен могут быть вложенными (в качестве примера см. раздел 3.9, «Пространства имен», на стр. 82, в файле *namespac.cpp*).

Пространства имен могут использоваться для обеспечения уникальной области видимости, что похоже на использование глобальных статических объявлений. Это делается с помощью определения безымянного пространства имен:

```
namespace { int count = 0; }   //здесь count уникален
//далее в файле доступен count
void chg_cnt(int i) { count = i; }
```

Новые удовлетворяющие ANSI заголовочные файлы библиотек больше не используют расширение *.h*. Такие файлы, как *iostream* и *complex*, теперь объявляются с пространством имен `namespace std`. Несомненно, поставщики по-прежнему будут поставлять заголовочные файлы и в старом стиле, такие как *iostream.h* и *complex.h*, так что старый код можно исполнять безо всяких изменений.

Большинство программ на C++ теперь будут начинаться с включения заголовков стандартных библиотек с последующим объявлением `using`.

```
#include <iostream>     //полным именем является std::cout
#include <vector>       //шаблон вектора STL
#include <cstddef>      //старые библиотеки C
using namespace std;
```

С.6. Правила компоновки

Большинство систем построено на основе многофайловой компиляции и компоновки. Необходимо понимать, каким образом комбинируются многофайловые программы. Компоновка отдельных модулей требует разрешения внешних ссылок. Основным правилом является то, что внешние нестатические переменные должны определяться толь-

ко в одном месте. Ключевое слово `extern` вместе с инициализатором составляет определение переменной. Использование ключевого слова `extern` без инициализатора задает объявление, но не определение. Если ключевое слово `extern` опущено, объявление является и определением, независимо от наличия инициализатора. Следующий пример, в котором файлы будут компоноваться вместе, иллюстрирует изложенные правила:

В файле prog1.cpp

```
char c;              //определение c
.....
```

В файле prog2.cpp

```
extern char c;       //объявление c
.....
```

В файле prog3.cpp

```
extern int n = 5;    //определение n
.....
```

В файле prog4.cpp

```
char c;              //повторное определение недопустимо
extern float n;      //недопустимо: несоответствие типов
extern int k;        //недопустимо: нет определения
.....
```

Определения констант и встроенные (inline) определения в области видимости файла локальны по отношению к этому файлу. Иными словами, они неявно являются статическими. Определения констант могут быть явно объявлены `extern`. Обычно такие определения помещаются в заголовочный файл, чтобы их можно было включить в любой код, в котором они необходимы.

Объявление `typedef` локально по отношению к своему файлу. Объявление перечислимых констант имеет внутреннюю по отношению к своему файлу компоновку. Перечислимые константы и `typedef`, необходимые в многофайловой программе, должны помещаться в заголовочный файл. Перечислимые константы, определенные внутри класса, локальны по отношению к этому классу, и для доступа к ним необходим оператор разрешения области видимости.

Обычно объявления помещаются в заголовочные файлы и затем используются из файлов кода:

```
//LMPstack.h
#ifndef LMP_stack  //во избежании повторного включения
#define LMP_stack
namespace LMP {
class stack { ..... };
}
#endif
```

```
//LMPstack.cpp
#include <LMPstack.h>   //в качестве источника включаем
                        //вышеприведенный файл
using namespace LMP;
.....
```

С.7. Типы

Фундаментальными типами C++ являются целые и типы с плавающей точкой. Тип `char` — самый короткий целый тип. Тип `long double` является самым длинным типом с плавающей точкой.

В следующей таблице типы перечислены от самых коротких к самым длинным. В этой таблице верхний левый элемент — самый короткий, а нижний правый — самый длинный.

Фундаментальные типы данных

`bool`		
`char`	`signed char`	`unsigned char`
`wchar_t`		
`short`	`int`	`long`
`unsigned short`	`unsigned`	`unsigned long`
`float`	`double`	`long double`

Два из приведенных типов данных, `bool` и `wchar_t`, были недавно добавлены комитетом ANSI и должны быть доступны в наиболее современных коммерческих компиляторах (см. программу *newtyp.cpp*).

Тип `wchar_t` предназначен для таких наборов символов, как японский алфавит Кана, которому требуются символы, непередаваемые `char`. Литералы этого типа являются широкими символьными константами. Этот тип является целым и в смешанных выражениях следует обычным правилам повышения целых типов.

Тип `bool` нарушает традицию С. На протяжении многих лет для получения булевского типа применялись различные схемы, новый же тип `bool` на практике устраняет несоответствия между ними. Он тоже является целым типом. Тип `bool` является типом, возвращаемым логическими выражениями и выражениями равенства и сравнения. Булевские константы `true` и `false` могут быть повышены (promote) до `1` и `0` соответственно. Ненулевые значения при присваивании допускают преобразование к `true`, а нулевые — к `false`. Предполагается, что поставщики компиляторов, при добавлении типа `bool`, включат в свои продукты опции или ключи, допускающие применение старого стиля, без использования типа `bool`.

Типы могут производиться от основных типов. Простым производным типом является перечислимый тип. Производные типы включают типы указателя, массива и структуры. Возможен тип обобщенного указателя `void*`. Допускаются как безымянные объединения, так и безымянные перечисления, существует также ссылочный тип. Безымянные объединения могут содержать только нестатические открытые члены данных. Безымянное объединение с областью видимости файла должно быть объявлено как `static`. Тип `class` и тип `struct` являются структурными типами. Имена объединения, перечисления и структуры служат именами типов.

Типы	Замечания
`void* gen_ptr;`	обобщенный указатель
`int i, &ref_i = i;`	`ref_i` служит псевдонимом для `i`
`enum button { off, on };`	перечисление
`button flag;`	теперь `button` — имя типа
`wchar_t w = L'yz';`	новый широкий символьный тип
`bool mine = false, yours = true;` `bool* p = &my_turn;`	новый булевский тип
`button set[10];`	массив
`class card {` `public:`	определенный пользователем тип
`  suit s;` `  pips p;`	открытые члены данных
`  void pr_card();` `private:`	функция-член
`  int cd;` `};`	заткрытый член данных
`suit card::* ptr_s = &card::s;`	указатель на член

Существуют пять ключевых слов классов памяти:

Классы памяти	
`auto`	локален в блоках, устанавливается неявно
`register`	применяется для оптимизации
`extern`	глобальная область видимости
`static`	внутри блоков, значения сохраняются
`typedef`	создает синонимы типов

Ключевое слово `auto` может использоваться внутри блоков, хотя это и излишне (обычно оно опускается). Автоматические переменные создаются при входе в блок и уничтожаются при выходе из него. Ключевое слово `register` можно использовать внутри блоков и для параметров функций. Оно сообщает компилятору, что в целях оптимизации программа хочет расположить переменную в быстрых регистрах. Семантически поведение регистровых переменных эквивалентно поведению автоматических.

Ключевое слово `extern` может использоваться внутри блоков и в области видимости файла. Оно означает, что переменная «подкомпоновывается» откуда-то извне. Ключевое слово `static` можно использовать внутри блоков и в области видимости файла. Внутри блока оно означает, что значение переменной сохраняется и после выхода из блока. В области видимости файла оно указывает на то, что объявления имеют внутреннюю компоновку.

Существуют два специальных ключевых слова-модификатора типов:

```
const       //неизменяемое
volatile    //подавляет оптмимизацию компилятора
```

Ключевое слово `const` используется для обозначения того, что данная переменная или параметр функции имеет неизменяемое значение. Ключевое слово `volatile` подразумевает некий неизвестный компилятору агент (источник), который может

изменять значение переменной, поэтому компилятор не должен выполнять «в лоб» оптимизацию кода, работающего с этой переменной. Переменные, получающие значения от внешних агентов, должны быть объявлены `volatile`.

```
volatile const gmt;     //ожидается внешний сигнал времени
```

С.8. Приведения и правила преобразований

В C++ имеются как явные преобразования, называемые *приведениями* (cast), так и неявные преобразования. Последние могут встречаться в выражениях, при передаче аргументов функциям, а также при возвращении функциями выражений. Многие преобразования являются неявными, что делает C++ удобным, но потенциально опасным для начинающего программиста. Из-за неявных преобразований возникают трудно отлавливаемые ошибки на этапе выполнения.

Общие правила просты.

Автоматическое преобразование выражения «x операция y»

1. Любой из `char`, `wchar_t`, `short`, `bool` или `enum` повышается до `int`. Целый тип, который нельзя представить как `int`, повышается до `unsigned`.
2. Если после первого шага выражение все еще смешанного типа, то согласно иерархии типов,

   ```
   int < unsigned < long < unsigned long
       < float < double < long double
   ```

   операнд более низкого типа повышается до более высокого, и значение выражения имеет этот тип. Заметьте, что если `long` не может содержать все значения `unsigned`, `unsigned` повышается до `unsigned long`.

Новый тип `bool` является целым типом, причем булевская константа `true` повышается до единицы, а булевская константа `false` — до нуля.

В C++ также встречаются неявные преобразования указателей. Любой тип указателя может быть преобразован в обобщенный указатель типа `void*`. Однако, в отличие от ANSI C, обобщенный указатель не совместим по присваиванию с указателем произвольного типа. Это означает, что C++ требует, чтобы обобщенный указатель был приведен к какому-либо типу для присваивания необобщенной переменной-указателю:

```
char* mem;
void* gen_p;

gen_p = mem;                          //C и C++
mem = (char*)gen_p;                   //C и (устаревшее) C++
mem = static_cast<char*>(gen_p);      //C++
mem = gen_p;              //допустимо в C и недопустимо в C++
```

Имя массива является указателем на его базовый элемент. Константа нулевого указателя может быть преобразована к любому типу указателя:

```
char* p = 0;      //p — нулевой указатель
int*  x = p;      //недопустимо, нужно static_cast
int*  y = 0;      //допустимо
```

Указатель на класс может быть преобразован к указателю на открыто унаследованный базовый класс. Это же относится и к ссылкам.

В дополнение к неявным преобразованиям, которые происходят во время присваиваний и в смешанных выражениях, существуют явные преобразования, называемые приведениями (cast). Если `i` — целое, то

```
static_cast<double>(i)
```

выполнит приведение значения `i`, так что это выражение будет иметь тип `double`. Сама переменная `i` останется неименной. Приведение `static_cast` позволяет выполнять преобразования, которые хорошо определены, переносимы и обратимы. Вот еще несколько примеров:

```
static_cast<char>('A' + 1.0)
x = static_cast<double>(static_cast<int>(y) + 1)
```

Приведения, относящиеся к представлению данных или зависящие от системы, используют `reinterpret_cast`.

```
i = reinterpret_cast<int>(&x)     //зависит от системы
```

Подобные приведения нежелательны и их лучше избегать.

В C++ существуют два других специальных приведения: `const_cast` и `dynamic_cast`. Для толкового обсуждения `dynamic_cast` необходимо понимание наследования (см. раздел 10.9, «Идентификация типа на этапе выполнения», на стр. 304). Модификатор `const` означает, что значение переменной нельзя изменить. Изредка такое ограничение надо обойти. Это называется отменой постоянства (cast away constness). В подобных случаях используется `const_cast`, как здесь:

```
foo(const_cast<int>(c_var));   //используется для вызова foo
```

Старые системы C++ предоставляют нестрогую форму приведения типов:

(тип) выражение или *тип(выражение)*

Вот некоторые примеры:

```
y = i/double(7);              //разделит с двойной точностью
ptr = (char*)(i + 88);        //целое к значению указателя
```

Такие устаревшие конструкции считаются вышедшими из употребления и в этой книге не встречаются, но они используются во многих старых компиляторах и исходных текстах. Старые системы не отличают сравнительно безопасные приведения типов, такие как `static_cast`, от менее надежных системно-зависимых, вроде `reinterpret_cast`. Кроме того, имена новых приведений типов говорят сами за себя. Например, назначение `const_cast` очевидно из названия.

В файле stcast.cpp

```
enum peer { king, prince, earl } a;      //пэры: король,
                                         //принц, граф
enum animal { horse, frog, snake } b;    //животные: конь,
                                         //лягушка, змея
```

```
.....
a = static_cast<peer>(frog);
```

Новые приведения безопаснее и могут заменить все существующие выражения приведений. Тем не менее, приведений лучше избегать, поскольку превращение лягушки в принца не всегда приемлемо.

Приведения	Замечания
`x = float(i);`	функциональная запись C++
`x = (float) i;`	запись приведения в C
`x = static_cast<float>(i);`	ANSI C++
`static_cast<char>('A' + 1.0)`	ANSI C++
`i = reinterpret_cast<int>(&x);`	ANSI C++, системно-зависимо
`foo(const_cast<int>(c_var));`	используется для отказа от постоянства при вызове `foo`

Конструктор с одним аргументом фактически является преобразованием типа аргумента к типу класса конструктора, если только этот конструктор не предваряется ключевым словом `explicit` (см. раздел 6.1.3, «Конструкторы как преобразования», на стр. 158). Рассмотрим пример конструктора `my_string`:

```
my_string::my_string(const char* p)
{
   len = strlen(p);
   s = new char[len + 1];
   assert (s != 0);
   strcpy(s, p);
}
```

Здесь происходит автоматический перевод типа `char*` в `my_string`. Это преобразование уже определенного типа к пользовательскому типу. Однако, пользователь не может добавить конструктор во встроенный тип, например в `int` или `double`. В примере с `my_string` вам также может потребоваться преобразование от `my_string` к `char*`. Его можно выполнить, определив специальную функцию преобразования внутри класса `my_string`:

```
operator char*() { return s; } //char* s является членом
```

В общем виде такая функция-член выглядит так:

```
operator тип() { ..... }
```

Такие преобразования происходят неявно в выражениях присваивания, а также в аргументах и выражениях, возвращаемых функциями. Для выполнения операций преобразования компилятор может создавать временные значения. Эти скрытые временные значения могут влиять на скорость выполнения.

В системах, которые поддерживают тип `bool`, для выражений, выполняющих проверку в инструкциях `if` или `while`, а также для первого операнда тройного оператора `?:`, необходимо неявное преобразование к `bool`. Выполняется очевидное преобразование нуля к `false`, а не-нуля — к `true`.

С.9. Выражения и операторы

C++ является богатым операторами и ориентированным на выражения языком. Операторы имеют семнадцать уровней приоритета[1]. Некоторые операторы имеют побочный эффект[2]. В следующей таблице приведены приоритет и порядок выполнения операторов.

Оператор	Порядок выполнения
:: (глобальная область видимости) :: (область видимости класса)	слева направо
() [] -> . (постфикс)++ (постфикс)-- typeid()	слева направо
++(префикс) --(*префикс*) ! ~ sizeof(тип) &(адрес) +(унарный) -(унарный) *(разыменование) delete new	справа налево
.* ->*	слева направо
* / %	слева направо
+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&	слева направо
^	слева направо
|	слева направо
&&	слева направо
||	слева направо
= += -= *= /= %= >>= <<= &= ^= |=	справа налево
?:	справа налево
, (оператор запятая)	слева направо

С.9.1. Выражения sizeof

Оператор `sizeof` может применяться к выражению или к стоящему в круглых скобках имени типа. Он возвращает размер (в байтах) того типа, к которому он был применен. Причем результат зависит от системы.

Объявления	
`int a, b[10];`	
Выражение	**Значение (GNU C++ на рабочей станции DEC)**
`sizeof(a)`	4
`sizeof(b)`	40
`sizeof(b[1])`	4
`sizeof(5)`	4
`sizeof(5.5L)`	8

[1] Б. Страуструп выделяет восемнадцать групп операторов. Отдельную группу, опущенную в приведенной здесь таблице, составляет оператор `throw`. В таблице он должен был бы располагаться между операторами `?:` и `,` (запятая). — *Примеч. перев.*

[2] То есть не только возвращают некоторое значение, но и изменяют значение своих аргументов, например операторы инкремента и декремента. — *Примеч. перев.*

С.9.2. Выражения автоинкремента и автодекремента

В C++ имеются операторы автоинкремента (++) и автодекремента (--) в префиксной и постфиксной формах. Поведение постфиксной формы отличается от префиксной тем, что в первом случае значение lvalue изменяется после того, как вычислена оставшаяся часть выражения. В качестве примера смотрите раздел 2.8.1, «Присваивание и выражения», на стр. 53 (в файле *auto.cpp*).

Автоинкремент и автодекремент	Эквивалентное выражение
`j = ++i;`	`i = i + 1; j = i;`
`j = i++;`	`j = i; i = i + 1;`
`j = --i;`	`i = i - 1; j = i;`
`j = i--;`	`j = i; i = i - 1;`

С.9.3. Арифметические выражения

Арифметические выражения согласуются с обычным ожидаемым поведением. Следующие примеры сгруппированы по приоритету выполнения, от самого высокого к более низкому.

Арифметические выражения	Замечания
`-i  +w`	унарный минус унарный плюс
`a * b  a / b  i % 5`	умножение деление деление по модулю
`a + b  a - b`	бинарные сложение и вычитание
`a = 3 / 2.0;`	переменной a присваивается `1.5`
`a = 3 / 2;`	переменной a присваивается `1`

Результатом оператора деления по модулю `%` является остаток от деления первого аргумента на второй. Этот оператор может использоваться только с целыми типами. Результат арифметического выражения зависит от правил преобразования, приведенных выше (см. раздел С.8, «Правила преобразований и приведения», на стр. 364). В приведенной таблице показано, что результат оператора деления `/` зависит от типов аргументов.

С.9.4. Выражения отношения, равенства и логические выражения

Дальнейшее обсуждение основывается на том факте, что стандартом ANSI C++ был принят тип `bool` с константами `false` и `true`. До того как был введен булевский тип, под значениями `false` и `true` понимались нулевые и ненулевые значения соответственно, они же использовались для управления порядком выполнения в инструкциях различных типов. В следующей таблице приведены операторы C++, наиболее часто применяющиеся для воздействия на порядок выполнения инструкций.

Операторы отношения, равенства и логические операторы		
Операторы отношения	меньше	<
	больше	>
	меньше или равно	<=
	больше или равно	>=

Операторы равенства	равно	==
	не равно	!=
Логические операторы	отрицание (унарное)	!
	логическое И	&&
	логическое ИЛИ	||

Оператор отрицания ! является унарным. Все остальные операторы отношения, равенства и логические операторы являются бинарными. Они действуют на выражения и вырабатывают или `true`, или `false`. Логическое отрицание может быть применено к выражению любого типа, которое затем преобразуется к типу `bool`. Если отрицание применяется к значению `true`, то результатом будет `false`, и наоборот.

При вычислении выражений, являющихся операндами && и ||, процесс вычисления прекращается, как только становится ясен результат всего логического выражения: `true` или `false`. Это называется вычислением по короткой схеме. Допустим например, что *expr1* и *expr2* являются выражениями. Если *expr1* имеет значение `false`, то в

```
expr1 && expr2
```

выражение *expr2* не будет вычисляться, поскольку уже ясно, что значение всего логического выражения будет `false`. Аналогично, если *expr1* имеет значение `true`, то в

```
expr1 || expr2
```

выражение *expr2* не будет вычисляться, поскольку уже ясно, что значение всего логического выражения будет `true`.

В системах, не поддерживающих тип `bool`, эти выражения будут давать в результате один и ноль вместо `true` и `false`.

Объявления и инициализация

```
int a = -5, b = 3, c = 0;
```

Выражение	Эквивалент	Значение
`a + 5 && b`	`((a + 5) && b)`	`false` или 0
`!(a < b) && c`	`((!(a < b)) && c)`	`false` или 0
`1 || (a != 7)`	`(1 || (a != 7))`	`true` или 1

Заметьте, что последнее выражение всегда будет иметь значение `true` и всегда будет вычислятся по короткой схеме.

C.9.5. Выражения присваивания

В C++ присваивание выполняется как часть выражения присваивания. Результатом является вычисление правой части присваивания и преобразование ее к типу значения, совместимому с переменной в левой части. Преобразование при присваивании происходит неявно и включает сужающие преобразования; простые переменные являются lvalue.

C++ допускает множественные присваивания в одной инструкции:

```
a = b + (c = 3);        равнозначно      c = 3; a = b + c;
```

C++ предоставляет операторы присваивания, сочетающие в себе присваивание и какой-либо другой оператор:

```
a опер= b;      равнозначно      a = a опер b;
```

Объявления и инициализация	
`int a, i, *p = &i; double w, *q = &w;`	
Выражение присваивания	**Замечания**
`a = i + 1;`	присваивает `(i + 1)` переменной `a`
`i = w;`	допустимо, значение `w` преобразуется к `int`
`q = i;`	*допустимо, целое значение повышается* до `double`
`*q = *p;`	допустимо
`q = p;`	недопустимое преобразование типов указателей
`q = (double*)p;`	допустимо
`a *= a + b;`	эквивалентно `a = a * (a + b);`
`a += b;`	эквивалентно `a = a + b;`

C.9.6. Выражения с запятой

Оператор «запятая» имеет самый низкий приоритет. Это бинарный оператор с выражениями в качестве операндов. В выражении с запятой вида

```
expr1, expr2
```

сначала вычисляется *expr1*, а затем *expr2*. Выражение с запятой как единое целое имеет значение и тип своего правого операнда. Оператор запятая является контрольной точкой. То есть, каждое выражение в разделенном запятыми списке вычисляется целиком, перед тем как перейти к следующему выражению, стоящему справа. Вот пример:

```
sum = 0, i = 1
```

Если `i` была объявлена как `int`, то это выражение с запятой имеет значение 1 и тип `int`. Оператор «запятая» выполняется слева направо.

C.9.7. Условные выражения

Условный оператор `? :` необычен тем, что это тройной оператор. Он принимает в качестве операндов три выражения. В конструкции

```
expr1 ? expr2 : expr3
```

первым вычисляется выражение *expr1*. Если оно истинно, то вычисляется *expr2*, и оно становится значением условного выражения в целом. Если *expr1* ложно, то вычисляется *expr3*, и уже оно становится значением условного выражения в целом. В следующем примере условный оператор используется для присваивания переменной `x` наименьшего из двух значений:

```
x = (y < z) ? y : z;
```

Скобки здесь не обязательны, так как оператор сравнения имеет более высокий приоритет, чем оператор присваивания. Однако использование скобок — это хороший стиль, поскольку с ними ясно, что с чем сравнивается.

Тип условного выражения

```
expr1 ? expr2 : expr3
```

определяется операндами *expr2* и *expr3*. Если они разных типов, то применяются обычные правила преобразования. Тип условного выражения не может зависеть от того, какое из двух выражений *expr2* или *expr3* будет вычислено. Порядок выполнения условных операторов ? : — справа налево.

C.9.8. Выражения с битовыми операторами

C++ предоставляет битовые операторы. Они действуют на машинно-зависимое битовое представление целых операндов. Обычно операторы сдвига перегружаются для осуществления ввода-вывода.

Битовый оператор	Значение
~	побитовое отрицание
<<	побитовый сдвиг влево
>>	побитовый сдвиг вправо
&	побитовое И
^	побитовое исключающее ИЛИ
|	побитовое ИЛИ

C.9.9. Выражения с определением адреса и обращением по адресу

Оператор определения адреса & является унарным; он возвращает адрес (или местоположение в памяти), по которому хранится объект. Оператор обращения по адресу * также является унарным и применяется к указателям. С его помощью можно получить значение по адресу, на который направлен указатель. Это называется *разыменованием указателя* (см. программу *lval.cpp*).

Объявления и инициализация	
`int  a = 5;`	
`int* p = &a;`	`//p указывает на a`
`int& reg_a = a;`	`//псевдоним для a`
Выражение	**Значение**
`*p = 7;`	lvalue, которым на самом деле является a, присваивается значение 7
`a = *p + 1;`	к rvalue 7 прибавляется 1 и a присваивается 8

С.9.10. Выражения с new и delete

Унарные операторы new и delete служат для управления *свободной памятью* (free store). Свободная память — это предоставляемая системой область памяти для объектов, время жизни которых напрямую управляется программистом. Программист создает объект с помощью ключевого слова new, а уничтожает его, используя delete.

В C++ оператор new принимает следующие формы:

```
new имя_типа инициализатор_opt[1]
new имя_типа[целое_выражение]
```

Первая форма размещает объект указанного типа в свободной памяти. Если присутствует инициализующее выражение, оно выполняет инициализацию. Вторая форма размещает в свободной памяти массив из объектов указанного типа. Для этих объектов должен иметься инициализатор по умолчанию.

Оператор new	Замечания
`new int`	размещает int
`new char[100]`	размещает массив из 100 char
`new int(99)`	размещает int, инициализованную 99
`new char('c')`	размещает char, инициализованную c
`new int [n][4]`	размещает массив указателей на int

В каждом случае происходят по крайней мере две вещи. Во-первых, выделяется надлежащий объем свободной памяти для хранения указанного типа. Во-вторых, возвращается базовый адрес объекта (в качестве значения оператора new). Если оператор new терпит неудачу, то либо возвращается значение 0, либо возбуждается соответствующее исключение (bad_alloc или xalloc), (см. раздел 11.10, «Стандартные исключения и их использование», на стр. 324). Последнее желательно для страховки.

Инициализатором является заключенный в скобки список аргументов. Для простых типов, таких как int, это может быть единственное выражение. Инициализаторы нельзя использовать для инициализации массивов, но они могут быть списком аргументов для соответствующего конструктора. Если размещаемый в памяти тип имеет конструктор, то размещаемый объект будет инициализован.

Оператор delete может принимать следующие формы:

```
delete выражение
delete[] выражение
```

В обеих формах выражением обычно служит переменная-указатель, использовавшаяся в предшествующем выражении new. Вторая применяется при возвращении памяти, в которой изначально размещался массив объектов. Квадратные скобки означают, что деструктор должен вызываться для каждого элемента массива. Оператор delete возвращает значение типа void.

[1] Индекс opt (от английского optional) обозначает необязательность соответствующего элемента синтаксической конструкции. — *Примеч. перев.*

Оператор delete	Замечания
`delete ptr`	удаляет объект на который указывает `ptr`
`delete p[i]`	удаляет объект `p[i]`
`delete [] p`	удаляет каждый элемент массива `p`

Оператор `delete` уничтожает объект, созданный с помощью `new`, тем самым отдавая распределенную память для повторного использования. Если удаляемый тип имеет деструктор, последний будет вызван.

Нет никаких гарантий насчет того, какие значения будут иметь объекты, размещенные в свободной памяти. Программист отвечает за надлежащую инициализацию таких объектов. В качестве примера кода см. раздел 3.19, «Операторы свободной памяти new и delete», на стр. 95 (в файле *dynarray.cpp*).

Синтаксис размещения и перегрузка new и delete

Оператор `new` имеет следующий общий вид:

`::`$_{opt}$ `new` *размещение*$_{opt}$ *имя_типа* *инициализатор*$_{opt}$

Здесь $_{opt}$ обозначает необязательные (optional) части.

Как правило, глобальный оператор `new()` используется для выделения свободной памяти. Система неявно предоставляет аргумент `sizeof(`*тип*`)` для этой функции. Вот прототип функции:

```
void* operator new(size_t size);
```

Оператор `new` может быть перегружен на глобальном уровне добавлением дополнительных параметров и вызовом его, используя синтаксис размещения (placement). Он может быть перегружен на уровне класса и использоваться для замещения (overriding) глобальных версий. Но при распределении в памяти массива объектов будет вызываться только глобальный оператор по умолчанию:

```
void* new(size_t size);
```

Оператор `delete` также может быть перегружен. Вот глобальная версия:

```
void operator delete(void* ptr);
```

Специфическая для класса версия может быть объявлена как

```
void operator delete(void* ptr);
```

или как

```
void operator delete(void* ptr, size_t size);
```

но только одна из этих форм может быть использована данным классом. При удалении массива объектов будет вызвана глобальная версия. Операторы `new` и `delete` предоставляют простой механизм для контролируемого пользователем управления свободной памятью, например:

```
#include <stddef.h>  //определен тип size_t
#include <stdlib.h>  //определены malloc() и free()
```

```
class X {
   .....
public:
   void* operator new(size_t size)
     {return (malloc(size)); }
   void operator delete(void* ptr) { free(ptr); }
   X(unsigned size) { new(size); }
   ~X() { delete(this); }
   .....
};
```

В этом примере класс X включает перегруженные формы new и delete. Когда класс перегружает оператор new(), глобальный оператор по-прежнему доступен с помощью оператора разрешения области видимости ::.

Синтаксис *размещения* (placement) обеспечивает разделенный запятыми список аргументов, используемый для выбора перегруженного оператора new() с соответствующей сигнатурой. Эти дополнительные аргументы часто применяются для того, чтобы поместить создаваемый объект по конкретному адресу. Вариант такого использования оператора new можно найти в файле *new.h*.

Функции-члены new() и delete() всегда статические. Примеры кода можно найти в разделе 7.12, «Перегрузка new и delete», на стр. 214 (в файле *over_new.cpp*).

Ошибочные ситуации

Если не реализована обработка исключений, то в случае ошибки выделения памяти new возвращает нулевое значение.

В стандартной библиотеке *new.h* имеется функция set_new_handler(), устанавливающая функцию, которая должна вызываться, если new терпит неудачу. Вызов set_new_handler() с аргументом ноль означает, что будет использоваться версия new, которая не возбуждает исключений. В противном случае будут возбуждены исключения bad_alloc или xalloc. Реализация *new.h* может зависеть от системы.

С.9.11. Другие выражения

В C++ вызов функции () и индексирование [] рассматриваются как операторы. Они имеют такой же приоритет, что и операторы выбора члена и указателя на структуру:

```
a[j + 6]          //означает *(a + j + 6)
sqrt(z + 15.5).   //возвращает значение типа double
```

Глобальный оператор разрешения области видимости имеет самый высокий приоритет. Для задания локального по отношению к классу идентификатора используется (вместе с именем класса) оператор разрешения области видимости класса.

```
::i          //доступ к глобальной i
A::foo()     //вызывает член foo(), определенный в A
```

Операторами указателя на член служат * и ->. Их приоритет ниже, чем у унарных операторов, но выше, чем у оператора умножения. Использование операторов * и -> описано в разделе 7.11.1, «Указатель на член класса», на стр. 212.

C.10. Инструкции

В C++ существует много различных типов инструкций. Инструкция завершается точкой с запятой ;. Множественные (состоящие из нескольких элементов) инструкции заключаются в фигурные скобки {}, что позволяет рассматривать их как одно целое. Инструкции служат контрольными точками. Перед выполнением новой инструкции должны быть завершены все действия предыдущей. Внутри инструкций компилятор имеет некоторую свободу в выборе того, какие части подвыражений должны вычисляться в первую очередь, например:

```
a = f(i);           //вызов f() и присваивание a
a += g(j);          //вызов g() и прибавление к a
a = f(i) + g(j);    //порядок вызова определяет компилятор
```

C++ — язык с блочной структурой, в котором объявления часто располагаются в начале блоков. В отличие от C объявления являются инструкциями и могут чередоваться с другими инструкциями. При написании кода на C++ по-прежнему следует соблюдать принципы структурного программирования. А именно, рекомендуется избегать оператора goto и добиваться, чтобы можно было легко проследить логику выполнения программы.

Поскольку в выражениях C++ возможны многие побочные эффекты, следует проявлять осторожность, чтобы избежать системно-зависимых результатов. Например, в выражениях, где порядок выполнения или возможная оптимизация компилятором могут привести к зависимости от системы, такие операторы с побочным эффектом, как автоинкремент и автодекремент, должны использоваться с осторожностью.

Во многих случаях инструкции C++ слишком уж нетребовательны, и во избежание появления ведущих к ошибкам конструкций необходимо придерживаться строгой дисциплины программирования. Например, выражение:

```
for (double x = 0.1; !(x == y); x += 0.1)
.....
```

может вызвать трудности, поскольку проблемы, связанные с машинной точностью и округлением, в большинстве случаев приведут к сбою в условии завершения цикла.

В следующей таблице обобщены основные инструкции C++.

Инструкция	C++	Замечания
пустая	`;`	
выражение	`i = i + k`	присваивание может использовать преобразования
составная	`{ ..... }`	используется для определения функций и структурирования; то же, что блок
goto	`goto l1;`	следует избегать
if	`if (p == 0)` `  cerr << "опять ошибка";`	условная без ветвления

if-else	`if (x == y)`
	`  cout << "равно\n";`
	`else`
	`  cout << "не равно\n";`
		условная с ветвлением

Продолжение на след. стр.

Продолжение

Инструкция	C++	Замечания
for	`for (i = 0; i < n; ++i)` `  a[i] = b[i] + c[i];`	в первом компоненте допустимы объявления
while	`while (x != y)`	ноль или более итераций

do-while	`do`
	`  y = y - 1;`
	`while (y >= 0);`
		одна или более итераций

switch	`switch (s) {`
	`case 1: ++i; break;`
	`case 2: --i; break;`
	`.....`
	`default: ++j;`
	`};`
		используйте `break`, чтобы избежать семантики «спуска» и `default` в качестве последней метки

break	`break;`	используется в `switch` и при итерациях
continue	`continue;`	используется в итерациях
объявление	`int i = 7;`	в блоке, файле или пространстве имен
блок try	`try { ..... }`	см. раздел 11.5, «Пробные блоки `try`», на стр. 319
помеченная	`error: cerr << "ОШИБКА";`	метка для `goto`
return	`return x * x * x;`	старайтесь использовать один `return` на функцию

C.10.1. Инструкции-выражения

В C++ присваивание выполняется как часть выражения присваивания. Не существует собственно инструкции присваивания, так как она является формой инструкции-выражения.

```
a = b + 1;      //(b + 1) присваивается a
++i;            //инструкция-выражение
a + b;          //тоже инструкция, но, по-видимому, бесполезная
```

C++ допускает несколько присваиваний в одной инструкции:

```
a = b + (c = 3);      равнозначно      c = 3; a = b + c;
```

C.10.2. Составная инструкция

Составная инструкция в C++ — это последовательность инструкций, заключенных в фигурные скобки { и }. Составные инструкции в основном применяются для группировки простых инструкций в функциональные звенья программы. Тело функции в C++ — всегда составной инструкция. В C, где объявления стоят в начале составной инструкции, такая инструкция называется *блоком*. В блоке объявления должны находиться в начале. Это правило ослаблено в C++, где объявления могут встречаться в любом месте списка инструкций. Везде, где можно вставить инструкцию, допускается и составная инструкция.

C.10.3. Инструкции if и if-else

В общем виде инструкция `if` выглядит так:

```
if (условие)
  инструкция
```

Если *условие* истинно, то выполняется *инструкция*; в противном случае *инструкция* пропускается. После того, как инструкция `if` выполнена, управление передается следующей инструкции. *Условие* — это выражение или объявление с инициализацией, с помощью которого выбирается логика выполнения. В качестве примера см. раздел 2.8.3, «Инструкции if и if-else», на стр. 55 (в файле *if_test.cpp*).

В общем виде инструкция `if-else` имеет следующую форму:

```
if (условие)
  инструкция1
else
  инструкция2
```

Если *условие* истинно, то выполняется *инструкция1*, а *инструкция2* пропускается; если *условие* ложно, то пропускается *инструкция1*, а *инструкция2* выполняется. После того, как инструкция `if-else` выполнена, управление передается следующей инструкции. Заметьте, что инструкция `else` соотносится со своим ближайшим `if`; это правило предотвращает возникновение неопределенностей, связанных с обособленным `else`. В качестве примера см. раздел 2.8.3, «Инструкции if и if-else», на стр. 55 (в файле *if_test.cpp*).

C.10.4. Инструкция while

В общем виде инструкция `while` выглядит так:

```
while (условие)
  инструкция
```

Сначала вычисляется *условие*. Если оно истинно, то выполняется *инструкция*, и управление передается обратно в начало цикла `while`. В результате тело цикла `while`, а именно *инструкция*, выполняется несколько раз, до тех пор, пока условие не станет ложно. С этого момента управление передается следующей за циклом инструкции. То есть *инструкция* может быть выполнена ноль или более раз. В качестве примера см. раздел 2.8.4, «Инструкция while», на стр. 56 (в файле *while_t.cpp*).

C.10.5. Инструкция for

Вот общий вид инструкции `for`:

```
for (начальная_инструкция; условие; выражение)
  инструкция
следующая_инструкция
```

Сначала выполняется *начальная_инструкция;* она инициализует переменную, используемую в цикле. Затем проверяется *условие*. Если оно истинно, то выполняется *инструкция*, вычисляется *выражение*, и управление передается обратно в на-

чало цикла for с той разницей, что *начальная_инструкция* уже не выполняется. Это продолжается до тех пор, пока *условие* не станет ложно, после чего управление передается *следующей_инструкции*.

Начальной_инструкцией может быть инструкция-выражение или просто объявление. Будучи объявленной, переменная имеет область видимости инструкции for. Заметьте, что это правило для области видимости изменилось по сравнению с прежним правилом, которое закрепляло за такими объявлениями область видимости, внешнюю по отношению к инструкции for.

Инструкция for является итерационной инструкцией, используемой обычно с увеличивающейся или уменьшающейся переменной. В качестве примера см. раздел 2.8.5, «Инструкция for», на стр. 56 (в файле *for_test.cpp*).

Для инициализации более одной переменной могут быть использованы выражения с запятой.

В файле forloop.cpp

```
for (factorial = n, i = n - 1; i > 1; --i)
  factorial *= i;
```

Любое или все выражения в инструкции for могут отсутствовать, но две точки с запятой должны быть обязательно. Если пропущена *начальная_инструкция*, то никакая инициализация в цикле for не выполняется. Если пропущено *выражение*, то не производится приращение, а в случае отсутствия *условия* не производится проверка. Есть специальное правило для тех случаев, когда пропущено *условие*; в такой ситуации компилятор будет считать условие выполненным всегда. Так цикл for в следующем фрагменте бесконечен:

```
for (i = 1, sum = 0 ; ; sum += i++ )
  cout << sum << endl;
```

Инструкция for — один из случаев, когда для переменной управления циклом часто используется локальное объявление, как в следующем примере.

```
for (int i = 0; i < N; ++i)
  sum += a[i];      //сумма элементов массива
                    //a[0] + ... + a[N - 1]
```

Семантика такова, что целая переменная i является локальной в предложенном цикле. В ранних системах C++ она считалась объявленной внутри объемлющего блока. Это может вносить путаницу, поэтому иногда лучше объявлять все автоматические программные переменные в начале блока.

C.10.6. Инструкция do

Вот общий вид цикла do:

```
do
  инструкция
while (условие);
  следующая_инструкция
```

Сначала выполняется *инструкция*, затем вычисляется *условие*. Если оно истинно, то управление передается обратно на начало инструкции do и процесс повторяется. Когда значение *условия* становится ложно, управление передается *следующей_инструкции*. В качестве примера см. раздел 2.8.6, «Инструкция do», на стр. 57 (в файле *do_test.cpp*).

C.10.7. Инструкции break и continue

Чтобы прервать нормальное выполнение цикла, программист может использовать две специальные инструкции:

```
break;     и     continue;
```

Инструкция `break` кроме использования в циклах может применяться в инструкции `switch`. Она вызывает выход из самого внутреннего цикла или из инструкции `switch`.

Пример из раздела 2.8.7 иллюстрирует использование инструкции `break`. Производится проверка на отрицательность значения, и если условие выполняется (значение отрицательно), инструкция `break` осуществляет выход из цикла `for`. Управление программой перескакивает к инструкции, идущей сразу за циклом. В качестве примера см. раздел 2.8.7, «Инструкции break и continue», на стр. 58 (в файле *for_test.cpp*).

Инструкция `continue` вызывает остановку текущей итерации цикла и немедленный переход к началу очередной итерации. В качестве примера см. раздел 2.8.7, «Инструкции break и continue», на стр. 58 (в файле *for_test.cpp*).

Инструкция `break` может встречаться только внутри тела инструкций `for`, `while`, `do` или `switch`. Инструкция `continue` может использоваться только внутри тела инструкций `for`, `while` или `do`.

C.10.8. Инструкция switch

Инструкция `switch` — разветвляющая условная инструкция, похожая на `if-else`. В общем виде она выглядит так:

```
switch (условие)
  инструкция
```

где *инструкция* — обычно составная инструкция, содержащая метки `case` и необязательную метку `default`. Как правило, `switch` включает несколько `case`. *Условие*, стоящее в круглых скобках за ключевым словом `switch`, определяет, какой из `case` будет выполняться, если это вообще произойдет.

Форма метки `case` такова:

```
case целое_постоянное_выражение:
```

В инструкции `switch` каждая метка `case` должна быть уникальна.

Если не была выбрана ни одна метка `case`, управление передается к метке `default`, если она есть. Метка `default` не обязательна. Если не выбрана ни одна метка `case` и нет метки `default`, произойдет выход из инструкции `switch`. В качестве примера см. раздел 2.8.8, «Инструкции switch», на стр. 59 (в файле *switch_t.cpp*).

Ключевые слова `case` и `default` не могут использоваться вне инструкции `switch`.

Как работает инструкция switch

1. Вычисляется выражение в круглых скобках, стоящее за `switch`.
2. Выполняется метка `case`, совпадающая с тем значением, которое было найдено на этапе 1; если ни одна из `case` не соответствует этому значению, выполняется метка `default`; если метки `default` нет, `switch` прерывается.
3. Выполнение `switch` прерывается, когда встречается инструкция `break` или когда достигается конец `switch`.

Инструкция `switch` не должна обходить (пропускать) инициализацию переменной, если только не пропускается целиком область видимости этой переменной:

```
switch (k) {
case 1:
   int very_bad = 3; break;
case 2:                          //недопустимо, т.к. пропускается
                                 //инициализация very_bad
   .....
}
switch (k) {
case 1:
   {
      int d = 3; break;
   }
case 2:      //допустимо, т.к. обходится область видимости d
   .....
}
```

С.10.9. Инструкция goto

Инструкция `goto` — это безусловный переход к произвольной помеченной инструкции в теле функции. В большинстве работ по современной методологии программирования инструкция `goto` рассматривается как вредная.[1]

Метка — это идентификатор. При выполнении инструкции `goto`, имеющей вид

```
goto метка;
```

управление переходит к помеченной инструкции. И инструкция `goto`, и соответствующая ей снабженная меткой инструкция, должны находиться в теле одной функции. В качестве примера см. раздел 2.8.9, «Инструкции goto», на стр. 60 (в файле *goto_tst.cpp*).

Инструкция `goto` не должна обходить (пропускать) инициализацию переменной, если только не пропускается целиком область видимости этой переменной:

```
if (i < j)
   goto max;              //недопустимо: пропущена инициализация
int crazy = 5;
max:
   .....
```

[1] Инструкция `goto` часто используется системами, автоматически генерирующими код на C++ (например, некоторыми CASE-средствами). Дело, вероятно, в том, что эти системы не читают книг по методологии программирования. — *Примеч. перев.*

С.10.10. Инструкция return

Инструкция `return` используется для двух целей. Когда она выполняется, управление программой немедленно передается обратно в вызывающее окружение. Кроме того, если за ключевым словом `return` следует какое-либо выражение, то его значение также передается в вызывающее окружение. Это значение должно допускать неявное преобразование к типу, указанному в заголовке определения функции.

Инструкция `return` имеет одну из двух форм:

```
return;
return выражение;
```

Вот некоторые примеры:

```
return;
return 3;
return (a + b);
```

С.10.11. Инструкция-объявление

Инструкция-объявление может быть помещена почти в любом месте блока. Это снимает ограничение С, согласно которому объявления переменных помещаются в начале блока, перед выполняемыми инструкциями. Инструкция-объявление имеет вид:

```
тип имя_переменной;
```

К объявленной таким образом переменной применимы обычные правила блочной структуры. Вот некоторый примеры:

```
for (int i = 0; i < N; i++) {
   a[i] = b[i] * c[i];
   int k = a[i]; //локальная k — возможно неэффективно
   .....
}
```

C++ налагает естественные ограничения на минующую объявления передачу управления в блоки. Такая передача управления запрещена, равно как и объявления, которые появляются только в одной ветви условной инструкции.

С.11. Функции

Особенности использования функций связаны с прототипами функций, перегрузкой, аргументами по умолчанию и ключевыми словами `inline`, `friend` и `virtual`. В этом разделе мы ограничимся обсуждением «обыкновенных» функций, перегрузки, вызова по значению, аргументов по умолчанию, и встраивания. Функции-члены обсуждаются в разделе С.12.2, «Функции-члены», на стр. 387. Дружественные функции рассматриваются в разделе С.12.3, «Дружественные функции», на стр. 387. Виртуальные функции — в разделе С.13.6, «Виртуальные функции», на стр. 395, а в приложении Е, «STL и строковые библиотеки», представлены обобщенные функции.

В C++ параметры функций вызываются по значению, если только они не объявлены как ссылочные типы.

Объявление функции	C++	Замечания

функция	`double cube(double x)`
	`{`
	`  return x * x * x;`
	`}`
		параметры вызываются по значению, возвращаемое выражение должно быть совместимо по присваиванию с возвращаемым типом

чистая процедура	`void pr_int_sq(int i)`
	`{`
	`  cout << i*i <<endl;`
	`}`
		возвращаемый тип `void` означает чистую процедуру[1]

пустой список аргументов	`void pr_hi()`
	`{`
	`  cout << "HI!" << endl;`
	`}`
		также возможно `void pr_hi(void)`

ссылочный аргумент	`void`
	`swap(int& i, int& j)`
	`{`
	`  int t = i;`
	`  i = j; j = t;`
	`}`
		при вызове `swap(r, s)`, `r` и `s` поменяются значениями

переменное число аргументов	`int` `scanf(const char*, ...);`	соответствует любому числу аргументов
встроенная	`inline cube(int x);`	встроенный код
аргумент по умолчанию	`int` `power(int x, int n = 2);`	`power(4)` дает 16, `power(4, 3)` дает 64
перегрузка	`double` `power(double x, int n)`	сигнатура: `double, int`

С.11.1. Прототипы

В C++ прототип имеет следующую форму:

тип имя`(`*список_объявлений_аргументов*`)`

Вот некоторые примеры:

```
double sqrt(double x);  //в math.h
double stats(const double data[], int size,
         double& max, double& min);
void print(const char* s);
int printf(char* format, ...); //в stdio.h
```

Прототипы делают функции C++ типобезопасными. Когда вызывается функция, фактические аргументы преобразуются (по присваиванию) к типу формальных параметров. В приведенном выше определении прототипа `sqrt()` вызов `sqrt()` гарантирует, что аргумент, если он это допускает, будет преобразован к типу `double`. Если требуется список аргументов неопределенной длины, используется эллипсический символ (`...`).

[1] Напомним, что под (чистой) процедурой понимается функция, не возвращающая значения. Процедура вызывается ради побочного эффекта, который может состоять в изменении фактических аргументов, выводе аргументов на печать, запуске исключения и т. д. — *Примеч. перев.*

C.11.2. Вызов по ссылке

Объявления ссылок предоставляют пользователям C++ вызываемые по ссылке аргументы. Применим этот механизм для написания функции `greater()`, которая меняет местами два значения, если первое больше второго:

```
int greater(int& a, int& b)
```

Если `i` и `j` представляют собой целые переменные, то вызов

```
greater(i, j)
```

будет использовать ссылки на `i` и `j` для перестановки местами (если это необходимо) их значений. В традиционном C такая операция должна выполняться с использованием указателей и разыменования. В качестве примера см. раздел 3.15, «Объявления ссылок и вызов по ссылке», на стр. 90 (в файле *greater.cpp*).

C.11.3. Встроенные функции

Ключевое слово `inline` предлагает компилятору преобразовать функцию во встроенный код. Данное ключевое слово используется ради эффективности, и, главным образом, с короткими функциями. Встроенность — это неявная особенность функций-членов, которые определены внутри своего класса. По ряду причин компилятор может игнорировать директиву `inline`, например, если функция слишком длинна. В таких случаях встроенная функция компилируется как обычная. Вот пример встроенной функции:

```
inline float circum(float rad) { return (pi * 2 * rad); }
```

Встроенные функции имеют внутреннюю компоновку.

C.11.4. Аргументы по умолчанию

Формальному параметру может быть придан аргумент по умолчанию. Но это допустимо лишь для идущих подряд параметров, являющихся последними в списке. Обычно аргумент по умолчанию — это подходящая константа, которая часто встречается при вызове функции. Следующая функция демонстрирует сказанное:

```
double pow(double x, int n=2);
```

При вызове:

```
r_sqrd = pow(r);      //возвратит r * r
r_5th = pow(r, 5);    //возвратит r*r*r*r*r
```

В качестве примера см. раздел 3.5, «Аргументы по умолчанию», на стр. 74 (в файле *def_args.cpp*).

C.11.5. Перегрузка

Перегрузка (overloading) использует одно и то же имя для нескольких вариантов оператора или функции. Выбор конкретного варианта зависит от типов аргументов, используемых оператором (функцией).

Рассмотрим функцию, которая находит среднее значение в массиве элементов с двойной точностью, и функцию, вычисляющую среднее для массива целых. Обе их удобно назвать `avg_arr`, как показано в следующей программе:

```
double avg_arr(const int a[], int size);
double avg_arr(const double a[], int size);
```

Список типов аргументов функции называется ее *сигнатурой*. Возвращаемый тип не является частью сигнатуры, но порядок аргументов важен. В качестве примера см. раздел 3.6, «Перегрузка функций», на стр. 75 (в файле *avg_arr.cpp*).

Рассмотрим следующие перегруженные объявления:

```
void print(int i = 0);          //сигнатура – int
void print(int i, double x);    //int, double
void print(double y, int i);    //double, int
```

При вызове функции `print()` компилятор подгоняет фактические аргументы под различные сигнатуры и выбирает наилучшее соответствие. Вообще говоря, существует три варианта: наилучшее соответствие, двусмысленное соответствие и отсутствие соответствия. Без наилучшего соответствия компилятор выдаст сообщение о синтаксической ошибке.

```
print('A');       //преобразование к int
print(str[]);     //нет соответствия, неверный тип
print(15, 9);     //двусмысленно
print(15, 9.0);   //соответствует int, double
print();          //соответствует int (по умолчанию)
```

Алгоритм подгонки к сигнатуре состоит из двух частей. Первая часть определяет наилучшее соответствие для каждого аргумента. Вторая часть смотрит, есть ли функция, которая является «единственно лучшим соответствием для». Это «единственно лучшее соответствие» определяется как наилучшее по крайней мере для одного аргумента, и «приравненное к наилучшему» для всех остальных аргументов.

Для заданного аргумента наилучшее соответствие — это всегда строгое соответствие. Строгое соответствие также включает аргумент типа константы или ссылки. Так,

```
void print(int i);
void print(const int& i);
```

является ошибкой переопределения.

Какая бы из перегруженных функций ни вызывалась, список вызываемых аргументов должен соответствовать списку объявленных параметров, как того требует алгоритм выбора функции.

Алгоритм выбора перегруженной функции

1. Использовать строгое соответствие (если возможно).
2. Попробовать стандартное повышение типа (см. раздел 2.5 на стр. 46).
3. Попробовать стандартное преобразование типа.
4. Попробовать определяемое пользователем преобразование.
5. Использовать, если возможно, соответствие эллипсису (. . .).

Стандартное повышение типа (promotion) предпочтительнее, чем остальные стандартные преобразования. Повышение — это преобразования `float` в `double`, а также `bool`, `char`, `short` или `enum` в `int`. Кроме того, к стандартным преобразованиям относятся преобразования указателей.

Несомненно, строгое соответствие является наилучшим. Чтобы гарантировать такое соответствие, можно использовать приведения (cast). В двусмысленных ситуациях компилятор будет «жаловаться».

C.11.6. Типобезопасная компоновка для функций

Правила компоновки для не-C++ функций могут быть заданы с помощью спецификации компоновки. Вот некоторые примеры:

```
extern "C" atoi(const char* nptr); //C-компоновка

extern "C" {          //C-компоновка всех функций
#include <stdio.h>
}
```

Эта спецификация существует в области видимости файла. Предоставляется ли типобезопасная компоновка для других языков — зависит от системы. В наборе перегруженных функций с одинаковым числом параметров, не более чем одна может быть объявлена с отличной от C++ компоновкой. Функции-члены класса не могут быть объявлены с указанием варианта компоновки.

C.12. Классы

Классы представляют собой форму гетерогенных агрегатных типов. Они делают возможным сокрытие данных, наследование, а также создание функций-членов, то есть предоставляют механизм, обеспечивающий определяемые пользователем типы. Вот пример:

В файле vect3.h

```
//Реализация типа для безопасного массива vect

class vect {
public:
   explicit vect(int n=10);          //конструктор по умолчанию
   vect(const vect& v);              //копирующий конструктор
   vect(const int a[], int n);       //инициализация массивом
   ~vect() { delete [] p; }          //деструктор
   int  ub() const;                  //верхняя граница
   int& operator[](int i) const;     //индексирование
   vect& operator=(const vect& v);   //присваивание
   friend ostream& operator<<(ostream& out, const vect& v);
private:
   int *p;      //базовый указатель
   int size;    //число элементов
};
```

Ключевые слова `public`, `private` и `protected` указывают на уровень доступа (открытый, закрытый и защищенный) к элементам, которые за ними следуют. Члены класса по умолчанию являются `private`, а структуры — `public`. В приведенном выше примере члены данных `p` и `size` являются `private`. Это делает их доступными исключительно для функций-членов того же класса.

C.12.1. Конструкторы и деструкторы

Конструктор (constructor) — это функция-член, имя которой совпадает с именем класса. Он создает объекты типа своего класса. Этот процесс включает в себя инициализацию членов данных и (зачастую) распределение свободной памяти с помощью `new`. Если класс имеет конструктор с пустым списком аргументов, или со списком, все аргументы которого имеют значения по умолчанию, то он может быть базовым типом для объявления массива без явной инициализации. Такой конструктор называется конструктором по умолчанию:

```
vect::vect() { ····· }  //конструктор по умолчанию
vect::vect(int i = 0) { ····· }  //конструктор по умолчанию
```

Деструктор (destructor) — это функция-член, имя которой представляет собой имя класса, предваряемое символом тильда ~. Обычно деструктор предназначен для уничтожения значений типа класса. Как правило, оно выполняется с помощью `delete`.

Конструктор вида

тип`::`*тип*`(const` *тип*`& x)`

используется для выполнения копирования одного значения типа *тип* в другое в следующем порядке:

Использование копирующего конструктора

1. Переменная типа *тип* инициализуется значением типа *тип*.
2. Значение типа *тип* передается в качестве аргумента функции.
3. Значение типа *тип* возвращается функцией.

Такой конструктор называется копирующим, и если он не задан явно, то его генерирует компилятор. По умолчанию значение данного типа инициализуется почленно.

Классы с конструктором по умолчанию могут использоваться для объявления массивов. Например,

```
vect a[5];
```

является объявлением, которое использует конструктор с пустым списком аргументов для создания массива из пяти объектов, каждый из которых является вектором из десяти элементов.

Существует специальный синтаксис для инициализации подэлементов объектов с конструкторами. Инициализаторы для членов структуры и класса могут быть указаны в разделенном запятыми списке, который следует за списком параметров и предшествует телу кода. Инициализатор имеет следующую форму:

имя_члена `(`*список_выражений*`)`

Например:

```
foo::foo(int* t) : i(7), x(9.8), z(t) //список инициализаторов
{ //здесь идет остальной исполняемый код ····· }
```

Когда члены сами представляют собой классы, *список_выражений* подгоняется к подходящей сигнатуре конструктора для вызова правильного перегруженного конструктора. Не всегда можно присвоить значения членам в теле конструктора. Список инициализаторов необходим, когда нестатический член является либо константой, либо ссылкой. В примере с классом `vect` конструкторы используют инициализатор для члена `vect::size`.

Конструкторы не могут быть виртуальными, в отличие от деструкторов. Конструкторы и деструкторы не наследуются.

Конструктор с единственным параметром автоматически является функцией преобразования. Автоматическое создание конструктора-преобразователя из конструктора с единственным параметром может быть заблокировано с помощью ключевого слова `explicit`. Для этого необходимо вставить слово `explicit` в начало конструктора с одним аргументом. В качестве примера см. раздел 6.10, «Строки, использующие семантику ссылок», на стр. 179 (в файле *string6.cpp*).

C.12.2. Функции-члены

Функции-члены — это функции, объявленные внутри класса. Следовательно, они имеют доступ к закрытым, защищенным и открытым членам своего класса. Будучи определены внутри класса они трактуются как встроенные функции; кроме того, если необходимо, они рассматриваются как перегруженные функции. В классе `vect` определена функция

```
int ub() const { return (size - 1); }   //верхняя граница
```

В этом примере функция-член `ub` имеет доступ к закрытому члену `size` и, кроме того, является встроенной.

Обычно функции-члены вызываются с помощью операторов `.` или `->`, например:

```
vect a(20), b;          //вызов нужного конструктора
vect* ptr_v = &b;
int uba = a.ub();       //вызов члена ub
ubb = ptr_v -> ub(); //вызов члена ub
```

Особый случай функций-членов — перегрузка операторов — обсуждается в разделе C.11.5, «Перегрузка», на стр. 383.

C.12.3. Дружественные функции

Ключевое слово `friend` (друг) служит спецификатором, уточняющим свойства функции. Дружественная функция должна быть объявлена внутри объявления класса, по отношению к которому она является дружественной. Дружественная функция предваряется ключевым словом `friend` и может встречаться в любой части класса. Функция-член одного класса может быть дружественной другому классу. В этом случае для указания имени функции в дружественном классе используется оператор

разрешения области видимости. Если все функции-члены одного класса являются дружественными функциями другого класса, это может быть указано с помощью записи: `friend class` *имя_класса*.

Следующие объявления вполне типичны:

```
class tweedledum {
   .....
   friend int tweedledee::cheshire();
};

class node {
   .....
   friend class tree;
   //функции-член из tree имеют доступ к node
};
```

В файле complexc.cpp

```
class complex {
   .....
   friend complex operator+(complex);
};
```

C.12.4. Указатель this

Ключевое слово `this` обозначает объявленный неявно указатель на себя. Он может использоваться только в нестатических функциях-членах. Ключевое слово `this` предоставляет встроенный не требующий объявления указатель. Это то же самое, как если бы в `clock` неявно объявлялся закрытый член `clock* const this`. Ранние системы C++ допускали управление памятью, предназначенной для объектов, при помощи присваивания указателю `this`. Подобные приемы устарели, поскольку указатель `this` теперь неизменяем. В качестве примера см. раздел 7.5, «Перегрузка унарного оператора», на стр. 199 (в файле *clock.cpp*).

C.12.5. Перегрузка операторов

Особым случаем перегрузки функций является перегрузка операторов. Для этого используется ключевое слово `operator`. Также как имени функции, такому как `print`, можно придать множество различных смыслов, зависящих от аргументов, так и оператору, например, +, можно приписать дополнительные значения. Это позволяет писать выражения в привычной инфиксной[1] форме как для встроенных, так и для пользовательских типов. Приоритет и порядок выполнения остаются неизменными.

При перегрузке операторов обычно используются либо функции-члены, либо дружественные функции, поскольку и те, и другие имеют привилегированный доступ. Для перегрузки унарного оператора используется функция-член, имеющая пус-

[1] Напомним, что инфиксная форма выражения соответствует привычной математической записи, например `c=a+b`. В префиксной форме данное выражение могло бы выглядеть как `c=+(a, b)` или, например, `c=plus(a, b)`, что, очевидно, гораздо менее «читабельно». — *Примеч. перев.*

той список аргументов, так как единственный аргумент оператора является неявным аргументом. Для бинарных операторов функция-член перегрузки оператора имеет в качестве первого аргумента неявно передаваемую переменную класса, а в качестве второго аргумента — единственный аргумент списка параметров. Дружественные функции и обычные функции получают оба аргумента из списка параметров (отсутствует неявный аргумент).

Расширим класс `vect` из раздела 7.7 «Перегрузка операторов присваивания и индексирования», на стр. 202, чтобы получить перегруженные операторы для сложения, присваивания, индексирования и вывода.

В файле vect_ovl.cpp

```
//Реализация безопасного массива типа vect

class vect {
public:
   .....
   int& operator[](int i) const;
   vect& operator=(const vect& v);
   friend vect operator+(const vect&, const vect&);
   friend ostream& operator<<(ostream&, const vect&)
private:
   int *p;      //базовый указатель
   int size;    //число элементов
};
```

Этот класс перегружает операторы присваивания и индексирования как функции-члены. Перегруженный `operator<<()` (поместить в) сделан другом `vect`, так что он может получить доступ к закрытым членам `vect`. Перегруженный `operator<<()` всегда должен возвращать тип `ostream`, так чтобы множественные операции «поместить в» могли выполняться как одно выражение. Перегруженный бинарный оператор «плюс» является дружественным, так что операции преобразования могут применяться к обоим аргументам. Заметьте, что перегруженный оператор присваивания проверяет, не происходит ли присваивание аргумента самому себе. В качестве примера см. раздел 7.7, «Перегрузка операторов присваивания и индексирования», на стр. 202 (в файле *vect2.h*).

Нельзя перегружать тройной оператор `?:`, оператор разрешения области видимости `::` и два оператора выбора члена `.` и `.*`.

Перегруженные постфиксные автодекремент и автоинкремент можно отличить от аналогичных префиксных операторов: постфиксная перегруженная функция определяется с единственным целым аргументом, как в следующем примере:

```
class T {
public:
   //постфикс ++ вызывается как t.operator++(0);
   void operator++(int);
   void operator--(int);
};
```

Не предполагается никакой семантической взаимосвязи между префиксной и постфиксной формами.

С.12.6. Функции-члены типа static и const

Обычная функция-член, вызываемая как

```
object.mem(i, j, k);
```

имеет явный список аргументов i, j, k и неявный список аргументов, являющихся членами object. Неявный список аргументов может рассматриваться как список, доступный посредством указателя this. Напротив, статическая функция-член не имеет доступа ни к каким членам через this. Функция-член, объявленная как const, не может изменять свои неявные аргументы. Программа *salary* иллюстрирует вышеуказанные отличия.

В файле salary.cpp

```
//Вычисление оклада с использованием функций-членов static
//и const

#include <iostream.h>

class salary {
public:
static void reset_all(int a) {all_bonus = a;}
   .....
private:
   int       b_sal;
   int       your_bonus;
   static int  all_bonus;       //объявление
};

int salary::all_bonus = 100;   //объявить и определить

int main()
{
   salary  w1(1000), w2(2000);
   .....
   salary::reset_all(400);   //можно также w1.reset_all(400);
}
```

Статический член all_bonus требует определения в области видимости файла. Он существует независимо от каких бы то ни было объявленных переменных типа salary. К статическому члену можно обратиться и так:

```
salary::all_bonus
```

Модификатор const идет между концом списка аргументов и телом кода. Он указывает, что никакие члены данных не изменят своих значений, так что код становится более надежным. Таким образом, можно сказать, что константные (const) функции-члены получают неявный указатель как const salary* const this.

Статическая функция-член может вызываться с помощью оператора разрешения области видимости или используя конкретный объект, то есть следующие вызовы равносильны:

```
salary::reset_all(400);
w1.reset_all(400);
(&w2) -> reset_all(400);
```

Хотя можно вызывать статическую функцию-член (или обращаться к статическому члену данных) с помощью оператора «точка», например

```
w1.reset_all(400);
```

такой вызов не отражает тот факт, что `reset_all()` является статическим членом. Разрешение области видимости, как в

```
salary::reset_all(400);
```

предпочтительнее, так как более ясно. В качестве примера см. раздел 5.8, «Функции-члены типа static и const», на стр. 144 (в файле *salary.cpp*).

C.12.7. Изменчивость (mutable)

Ключевое слово `mutable` позволяет членам класса, переменные которого были объявлены как константы, быть, тем не менее, изменяемыми. Таким образом отпадает необходимость отказываться от постоянства с помощью `const_cast`. Это относительно новая возможность, поддерживаемая не всеми компиляторами C++. В качестве примера см. раздел 5.8.1, «Изменчивость (mutable)», на стр. 146 (в файле *mutable.cpp*).

C.12.8. Проектирование классов

Полезным принципом проектирования служит лезвие Оккамы (Occam's razor). Этот принцип утверждает, что не следует вводить новые сущности сверх необходимости — или сверх законченности, обратимости, ортогональности, согласованности, простоты, эффективности и выразительности. Подобные идеалы могут конфликтовать, что часто приводит к неизбежной «торговле» между ними, когда дело доходит до реализации проекта.

C.13. Наследование

Наследование (inheritance) — это механизм получения нового класса из существующего. Существующий класс может быть дополнен или изменен для создания производного класса. Класс может быть произведен от существующего с помощью следующей конструкции:

```
class имя_класса:
   public|protected|private)opt имя_базового_класса
{
   объявления членов
};
```

Ключевое слово `class` как всегда можно заменить ключевым словом `struct`, если подразумевается, что все члены открыты. Ключевые слова `public`, `protected` и `private` используются как модификаторы прав доступа к членам класса. Открытый

член доступен во всей области видимости, где виден класс. Закрытый член доступен другим функциям-членам своего класса. Защищенный член доступен другим функциям-членам своего класса, а также класса, унаследованного непосредственно от данного. Эти модификаторы доступа могут использоваться в объявлении класса в любом порядке и сколь угодно часто.

Производный класс должен иметь конструктор, если в его базовом классе отсутствует конструктор по умолчанию. Если в базовом классе имеются конструкторы, требующие аргументы, производный класс явно обращается к конструктору базового класса в своем инициализующем списке. Вот форма такого конструктора:

имя_класса(*список_арг*) :
  имя_баз_класса$_{opt}$ (*список_арг_баз_класса*)

Стоящий в скобках *список_арг_баз_класса* используется при вызове соответствующего конструктора базового класса и выполняется перед тем, как выполняется тело конструктора производного класса.

Открыто наследуемый класс является подтипом своего базового класса. Переменная производного класса во многих случаях может рассматриваться как переменная типа базового класса. Указатель, тип которого — «указатель на базовый класс», может указывать на объекты открыто наследуемого класса. Ссылка на производный класс неявно может быть преобразована в ссылку на открытый базовый класс. Можно объявить ссылку на базовый класс и инициализовать ее ссылкой на объект открыто наследуемого класса.

В следующем примере в качестве базового класса используется класс `vect` из раздела 7.7, «Перегрузка операторов присваивания и индексирования», на стр. 202. Единственное изменение сделанное в базовом классе — замена закрытых членов на защищенные. Следующий класс `vect_bnd` является производным:

В файле vect_bnd.cpp

```
class vect_bnd : public vect {
public:
   vect_bnd(int = 0, int = 9);     //по умолчанию массив
                                   //из 10 эл-тов
   vect_bnd(vect_bnd& v);          //копирующий конструктор
   vect_bnd(vect& v);              //преобразующий конструктор
   vect_bnd(const int a[], int ne, int lb = 0);
   int&  operator[](int) const;
   int   ub() const { return (u_bnd); }
   int   lb() const { return (l_bnd); }
   vect_bnd& operator=(const vect_bnd& v);
private:
   int   l_bnd, u_bnd;
};

//конструктор по умолчанию
vect_bnd::vect_bnd(int lb, int ub) :
        vect(ub - lb + 1), l_bnd(lb), u_bnd(ub) { }
```

```
//преобразующий конструктор
vect_bnd::vect_bnd(vect& v) :
       vect(v), l_bnd(0), u_bnd(size - 1) { }

//копирующий конструктор
vect_bnd::vect_bnd(vect_bnd& v) :
        vect(v), l_bnd(v.l_bnd), u_bnd(v.u_bnd) { }

vect_bnd::vect_bnd(const int a[], int n, int lb) :
        vect(a, n), l_bnd(lb), u_bnd(lb + n) { }
```

В данном примере конструкторы производного класса вызывают конструктор базового класса (из списка аргументов после двоеточия).

С.13.1. Множественное наследование

Множественное наследование (multiple inheritance) делает возможным получение производного класса от более чем одного базового класса. Синтаксис заголовка класса расширяется, чтобы можно было использовать список базовых классов с атрибутами доступа. Например:

```
class shape {
  //класс для интерфейса фигуры
};

class tview {
  //класс, реализующий просмотр текста
};

class tshape : public shape, private tview {
  //адаптер текстового просмотра,
  //позволяющий просматривать фигуры
};
```

В этом примере производный класс `tshape` открыто наследует базовый класс `shape` — интерфейс, и закрыто наследует `tview` — реализацию просмотра текста. Подобная схема создания классов называется адаптерной схемой (adaptor pattern). Она использует множественное наследование для объединения интерфейса с реализацией, поэтому также говорят, что такая техника использует *класс-примесь* (mixin class).

Вообще, родительские отношения между классами описываются *ориентированным ациклическим графом* (directed acyclic graph — DAG) наследования. Ориентированный ациклический граф наследования — это граф, узлы которого являются классами, а ориентированные ребра направлены от производных классов к базовым.

При производстве от различных классов, имеющих члены с идентичными именами, могут возникать неопределенности. Подобное наследование допускается при условии, что пользователь не делает двусмысленных ссылок на такой член.

При множественном наследовании два базовых класса могут быть получены от общего предка. Если оба базовых класса используются обычным образом своим производным классом, такой класс будет иметь два подобъекта общего предка. Подобного дублирования можно избежать с помощью виртуального наследования. В качестве примера см. раздел 10.4, «Виртуальные функции», на стр. 289 (в файле *shape2.cpp*).

С.13.2. Вызов конструктора

Порядок выполнения конструкторов при наследовании и при инициализации членов класса таков:

Порядок выполнения конструкторов

1. Базовые классы инициализуются в порядке объявления.
2. Члены инициализуются в порядке объявления.

Виртуальные базовые классы создаются до того, как создан любой из производных классов, и до того, как созданы невиртуальные базовые классы. Порядок их создания зависит от соответствующего графа (от DAG). Это порядок «из глубины, слева направо». Деструкторы вызываются в порядке, обратном выполнению конструкторов.

С.13.3. Абстрактные базовые классы

Чисто виртуальная функция — это виртуальная функция-член, тело которой, вообще говоря, не определено. Она объявляется внутри класса следующим образом:

```
virtual прототип_функции = 0;
```

Класс, имеющий хотя бы одну чисто виртуальную функцию, называется *абстрактным базовым классом* (abstract base class). Нельзя объявлять переменные абстрактного класса, но можно определять и полиморфно использовать указатели на такой класс. В качестве примера см. раздел 10.5, «Абстрактные базовые классы», на стр. 293 (в файле *predator.cpp*).

Чисто виртуальный деструктор должен иметь определение.

С.13.4. Указатель на член класса

В C++ указатель на член класса отличают от указателя на класс. Указатель на член класса имеет тип `T::*`, где `T` — имя класса. В C++ есть два оператора, используемые для разыменования указателя на член класса. Вот они:

```
.*   и   ->*
```

Выражение `x.*ptr_mem` следует понимать так: сначала разыменовывается указатель для получения переменной-члена, а затем происходит доступ к члену объекта `x`. В качестве примера см. раздел 7.11.1, «Указатель на член класса», на стр. 212 (в файле *showhide.cpp*).

С.13.5. Идентификация типа на этапе выполнения

Идентификация типа на этапе выполнения (run-time type identification, RTTI) предоставляет механизм для безопасного выявления типа, на который направлен во время работы программы указатель базового класса. Этот механизм включает в себя `dynamic_cast` — оператор над указателем на базовый класс, `typeid` — оператор для выяснения типа объекта и `type_info` — структуру, которая на этапе выполнения предоставляет информацию о соответствующем типе.

Оператор `dynamic_cast` имеет следующий вид:

```
dynamic_cast<тип>( v )
```

где *тип* должен быть типом указателя или ссылки на класс, а *v* должна быть значением соответствующего указателя или ссылки. Данное приведение используется с производными классами, имеющими виртуальные функции. Вот как оно применяется:

```
class Base { ····· };
class Derived : Base { ····· };

void fcn(Base* ptr)
{
   Derived* dptr = dynamic_cast<Derived*>(ptr);
   .....
}
```

В этом примере приведение преобразует значение указателя `ptr` к типу `Derived*`. Если преобразование неприменимо, возвращается значение `0` или возбуждается исключение `bad_cast`. Динамические приведения работают также с типами-ссылками. По сути, объект производного типа имеет подобъект, который соответствует базовому типу. Преобразование замещает значение указателя производного типа (или ссылки) на подходящее значение указателя базового типа (или ссылки).

Оператор `typeid()` может быть применен к *имени типа* или к выражению для выяснения истинного типа аргумента. Этот оператор возвращает ссылку на класс `type_info`, который предоставляется системой и определен в заголовочном файле *type_info.h* или *typeinfo.h*. Класс `type_info` предоставляет функцию-член `name()`, которая возвращает строку, являющуюся именем типа. Кроме того, он предлагает перегруженные операторы проверки на равенство. Не забудьте изучить свою реализацию для выяснения деталей интерфейса этого класса. Можно воспользоваться «плохими» динамическими приведениями и «плохими» операциями `typeid` чтобы возбудить исключения `bad_cast` и `bad_typeid`, так что пользователь может выбрать, иметь ли ему дело с NULL-указателем или отлавливать исключения. В качестве примера см. раздел 10.9, «Идентификация типа на этапе выполнения», на стр. 304 (в файле *typeid.cpp*).

С.13.6. Виртуальные функции

Ключевое слово `virtual` служит спецификатором функции, который обеспечивает механизм для динамического выбора на этапе выполнения подходящей функции-члена среди функций базового и производного классов. Оно может применяться для изменения объявлений только функций-членов. Виртуальная функция, как и обычная функция, должна иметь исполняемое тело. При вызове семантика ее точно такая же, как и у остальных функций. Виртуальная функция может замещаться в производном классе. Выбор того, какое определение функции вызвать для виртуальной функции, происходит динамически (на этапе выполнения). Типичный случай — когда базовый класс содержит виртуальную функцию, а производные классы имеют свои версии этой функции. Указатель на базовый класс может указывать либо на объект базового класса, либо на объект производного класса. Выбор вызываемой функции-члена будет произведен на этапе выполнения и будет зависеть от типа объекта, а не от типа указателя. При отсутствии члена производного типа по умол-

чанию используется виртуальная функция базового класса. В качестве примера см. раздел 10.4, «Виртуальные функции», на стр. 289 (в файле *shape2.cpp*).

Одна из причин, определяющих сложность C++, заключается в большом количестве типов функций и в столь же большом числе правил, касающихся их применения. Теперь, когда мы рассмотрели наследование и виртуальные функции, мы познакомились с большинством вариантов функций. Еще существуют функции, которые генерируются из шаблонов, а также обработчики `catch()`, которые похожи на функции, но в то же время являются частью механизма обработки исключений. Полезно свести различные типы функций и соответствующие правила в таблицу. Например, встроенная функция может быть членом и не-членом, может возвращать или не возвращать значение. Кроме того, встраивание приводит к локальной компоновке.

Характеристики функций

Категория	Член	Виртуальная	Возвращаемый тип	Особенности
конструктор	да	нет	нет	не наследуется; может генерироваться по умолчанию
деструктор	да	да	нет	не наследуется; может генерироваться по умолчанию
присваивание	да	да	да	не наследуется
`-> [] ()`	да	да	да	
`operator`	возможно	да	да	
преобразование	да	да	нет	без аргументов
`new`	статическая	нет	`void*`	
`delete`	статическая	нет	`void*`	
`inline`	возможно	да	возможно	локальная компоновка
`catch`	нет	нет	нет	один аргумент
`friend`	дружественная	нет	да	не наследуется

С.14. Шаблоны

Ключевое слово `template` используется для реализации параметризованных типов. Вместо того, чтобы многократно переписывать код для каждого типа, шаблоны позволяют выполнять инстанцирование и генерировать код автоматически.

В файле stack_t1.cpp

```
template <class T> //параметризация типом T
class stack {
public:
   stack();
   explicit stack(int s);
   T& pop();
   void push(T);
   .....
```

```
private:
  T*  item;
  int top;
  int size;
};

typedef stack<string> str_stack;
str_stack s(100);          //явная переменная строкового стека
```

В качестве примера см. раздел 9.1, «Шаблонный класс stack», на стр. 245 (в файле *stack_t1.cpp*).

Объявление шаблона имеет следующий вид:

```
template<шаблонные_аргументы> объявление
```

причем шаблонными аргументами могут быть:

```
class идентификатор
объявление_аргумента
```

Аргументы `class` *идентификатор* инстанцируется типом. Другие аргументы объявления инстанцируются постоянными выражениями не с плавающей точкой и могут быть функцией или адресом объекта с внешней компоновкой, как показано в следующем фрагменте.

В файле array.cpp

```
template <class T, int n>
class array_n {
  .....
private:
  T  items[n];                  //n явно инстанцируется
};

  array_n<compex, 1000>  w;     //w – массив комплексных чисел
```

Синтаксис функции-члена, при расположении ее тела вне определения класса, следующий:

```
template <class T>
T& stack<T>::pop()
{
  return(item[top--]);
}
```

Имя класса, используемое оператором разрешения области видимости, включает шаблонные аргументы, а перед объявлением функции-члена необходимо объявление шаблона.

С.14.1. Параметры шаблона

Вышеприведенный шаблон может быть переписан с параметрами по умолчанию как для целого аргумента, так и для типа. Например:

```
template <class T = int, int n = 100>
class array_n {
   .....
};
```

Параметры по умолчанию могут быть инстанцированы при объявлении переменных, а могут быть опущены. В последнем случае будут использованы значения по умолчанию.

Шаблоны могут использовать ключевое слово `typename` вместо `class`. Например:

```
template <typename T = double, double* ptr_dbl>
```

Это позволяет коду шаблона использовать в качестве аргумента указатель на `double`. Обычные аргументы с плавающей точкой не допускаются, возможен только указатель или ссылка на аргумент с плавающей точкой.

Аргументом шаблона может быть другой шаблон. Например:

```
template <typename T1, template<class T2> class T3>
```

Это позволяет создавать весьма изощренные меташаблоны — шаблоны, инстанцируемые шаблонами же. Такие библиотеки, как STL, могут использовать подобные приемы.

С.14.2. Шаблоны функций

До 1995 года компиляторы допускали параметризацию для обычных функций лишь с помощью ограниченной формы шаблонного синтаксиса. Допускалось только инстанцирование вида `class` *идентификатор*, причем *идентификатор* должен встретиться в списке аргументов функции.

В файле swap.cpp

```
//обобщенная swap
template <class T>
void swap(T& x, T& y)
{
   T  temp;

   temp = x;
   x = y;
   y = temp;
}

//возможно в ANSI C++, но не поддерживается
//некоторыми компиляторами
template <class T, int n>
T foo()
{
   T temp[n];
.....
}

foo<char, 20>(); //использует char, 20 и вызывает foo
```

Шаблон функции используется для создания подходящей функции для любого вызова, однозначно соответствующего его аргументам:

```
swap(i, j);    //при int i, j; имеем T = int; порядок!
swap(c1, c2); //при complex c1, c2; имеем T = complex; порядок!
swap(i, ch);  //недопустимо! при i; char ch; что есть T?
```

Алгоритм выбора перегруженной функции выглядит следующим образом:

Алгоритм выбора перегруженной функции

1. Строгое соответствие (при допущении тривиальных преобразований) для нешаблонных функций.
2. Строгое соответствие для шаблонов функций.
3. Обычное разрешение аргументов для нешаблонных функций.

В случае `swap(i, ch)` из примера выше была бы вызвана обычная функция с прототипом

```
void swap(char, char);
```

В качестве примера см. раздел 10.4, «Виртуальные функции», на стр. 289 (в файле *shape2.cpp*).

C.14.3. Друзья

Шаблонные классы могут содержать «друзей». Дружественная функция, которая не использует спецификацию шаблона, универсальна — имеется единственный ее экземпляр для всех инстанцирований шаблонного класса. Дружественная функция, которая задействует аргументы шаблона, специфична для каждого инстанцирования класса:

```
template <class T>
class matrix {
public:
   friend void foo_bar();              //универсальна
   friend vect<T> product(vect<T> v); //специфична
   .....
};
```

C.14.4. Статические члены

Статические члены не универсальны, они специфичны для каждого инстанцирования:

```
template <class T>
class foo {
public:
   static int count;
   .....
};

   .....
foo<int>    a, b;
foo<double> c;
```

Статические переменные `foo<int>::count` и `foo<double>::count` различны. Переменные `a.count` и `b.count` относятся к `foo<int>::count`, а `c.count` относится к `foo<double>::count`. Предпочтительнее использовать форму `foo<`*тип*`>::count`, так она подчеркивает статичность переменой.

С.14.5. Специализация

Когда шаблонный код является неудовлетворительным для конкретного типа аргумента, он может быть специализирован. Шаблонная функция может быть перегружена обычной функцией такого же типа — то есть функцией, список аргументов и возвращаемый тип которой соответствуют объявлению шаблона. Такая перегружающая шаблон функция является специализацией шаблона. Когда специализация соответствует вызову, то вызывается именно она, а не код, сгенерированный из шаблона.

```
void maxelement<char*>(char*a[], char* &max, int size);
//специализована с использованием strcmp()
//для возвращения max string
```

Это — специализация объявленного ранее шаблона `template<class T> maxelement()`. Возможна и специализация класса, например:

```
class stack<foobar_obj> {/* специализация для foobar_obj */ };
```

В качестве примера см. раздел 9.2.1, «Соответствие сигнатуре и перегрузка», на стр. 249 (в файле *swap.cpp*).

С.15. Исключения

Обычно исключение — это возникающая в программе нештатная ситуация, с которой программа не может справиться. Примером является деление на ноль. Как правило, выполнение программы аварийно завершается системой.

Код на C++ позволяет напрямую возбуждать исключения в пробном блоке с помощью запускающего выражения `throw`. Исключения обрабатываются с помощью вызова надлежащего обработчика, выбираемого из списка обработчиков, который следует сразу за пробным блоком. Вот простой пример:

В файле vect_ex2.cpp

```
vect::vect(int n)
{ //устойчивый к сбоям конструктор
  try {
    if (n < 1)
      throw(n);
    p = new int[n];
    if (p == 0)
      throw ("СВОБОДНАЯ ПАМЯТЬ ИСЧЕРПАНА");
  }
  catch (int n) { ..... }                    //отлавливает
                                             //неверный размер
```

```
    catch (const char* error) { ····· }   //отлавливает
                                          //исчерпание
                                          //свободной памяти
}
```

Заметьте, что new в данном примере — это традиционная new, возвращающая ноль в случае ошибки выделения памяти. Системы C++, использующие исключения внутри new, могут в случае сбоя запускать исключения xalloc или bad_alloc. Такой подход замещает new, возвращающую в случае ошибки выделения памяти ноль. Можно оставить обработку ошибок в старом стиле, использовав для этого set_new_handler(0).

С.15.1. Запуск исключений

Синтаксически выражение throw может принимать две формы:

```
throw выражение
throw
```

Конструкция throw *выражение* возбуждает исключение в пробном блоке. Для выбора инструкции catch, которая обрабатывает исключение, используется самый внутренний пробный блок (try-блок), в котором возбуждено исключение. А throw без аргумента может использоваться внутри catch для перезапуска (rethrow) текущего исключения. Обычно throw без аргумента используется, если вы хотите, чтобы для дальнейшей обработки исключения из недр первого обработчика вызвался второй.

Запущенное с помощью throw выражение — это временный статический объект, существующий до тех пор, пока не будет произведен выход из обработчика исключения. Выражение отлавливается обработчиком, который может использовать этот временный объект. Неотловленное выражение завершает программу.

В файле throw_it.cpp

```
void foo()
{
   int  i;
   //покажем, как запускается исключение
   i = -15;
   throw i;
}

int main()
{
   try {
      foo();
   }
   catch(int n)
      { cerr << "Перехваченное исключение\n" << n << endl; }
}
```

Целое значение, запущенное `throw i`, существует, пока не произойдет выход из обработчика `catch(int n)` с целой сигнатурой. Это выражение доступно для использования внутри обработчика как его аргумент. В качестве примера см. раздел 11.4, «Запуск исключений», на стр. 314 (в файле *throw1.cpp*).

Вот пример перезапуска исключения:

```
catch(int n)
{
   .....
   throw;    //перезапуск
}
```

При условии, что запущенное выражение было целого типа, перезапущенное исключение — это тот же целый объект, который обрабатывается ближайшим обработчиком, подходящим для этого типа.

С.15.2. Пробные блоки try

Синтаксически блок `try` выглядит так:

```
try
составная_инструкция
список_обработчиков
```

Блок `try` является контекстом для определения того, какие обработчики вызываются при возбуждении исключения. Порядок, в котором определены обработчики, весьма важен, ибо он задает очередность проверки обработчиков подходящего типа для возбужденного исключения:

```
try {
   .....
   throw("SOS");
   .....
   io_condition eof(argv[i]);
   throw (eof);
   .....
}

catch(const char*) {.....}
catch(io_condition& x) {.....}
```

Выражение throw соответствует перехватчику catch, если оно:

1. точно совпадает с типом `catch`;
2. является производным типом открытого базового класса типа обработчика;
3. является типом возбуждаемого объекта, который (тип) может быть преобразован к типу указателя, являющегося аргументом `catch`.

Ошибочно приводить список обработчиков в порядке, исключающем возможность вызова некоторых из них. Например:

```
catch(void*)                    //соответствует любому char*
catch(char*)
catch(BaseTypeError&)           //в том числе и DerivedTypeError
catch(DerivedTypeError&)
//предполагается, что класс DerivedTypeError
//является производным от BaseTypeError
```

С.15.3. Обработчики

Синтаксически обработчик имеет вид:

```
catch(формальный_аргумент)
составная_инструкция
```

Обработчик `catch` выглядит как объявление функции одного аргумента без возвращаемого типа.

В файле catch.cpp

```
catch (const char* message)
{
  cerr << message << endl;
  exit(1);
}

catch( ... )    //будет выполнено действие по умолчанию
{
  cerr << "ВОТ И ВСЁ!" << endl;
  abort();
}
```

Допустима эллипсическая `( ... )` сигнатура, совпадающая с аргументом любого типа. Кроме того, формальный аргумент может быть абстрактным объявлением, то есть предоставлять информацию о типе, но не задавать имя переменной. В качестве примера см. раздел 11.6, «Обработчики», на стр. 320 (в файле *catch.cpp*).

С.15.4. Спецификация исключения

Синтаксически *спецификация исключения* является частью объявления и определения функции, она имеет следующий вид:

```
заголовок_функции throw (список_типов)
```

Здесь *список_типов* — это список типов, которые может иметь выражение `throw` внутри функции. В объявлении и в определении функции спецификация исключения должна записываться одинаково.

Если список пуст, компилятор полагает, что функцией не будет выполняться никакой `throw` (ни прямо, ни косвенно).

```
void foo() throw(int, over_flow);
void noex(int i) throw();
```

Если спецификация исключения опущена, предполагается, что такой функцией может быть возбуждено произвольное исключение. Нарушение спецификаций исключений приводит к ошибкам на этапе выполнения. Они отлавливаются функцией `unexpected()`.

С.15.5. terminate() и unexpected()

Предоставляемая системой функция `terminate()` вызывается, если не было задано ни одного обработчика, знающего, как обращаться с исключением. По умолчанию вызывается функция `abort()`. Она немедленно завершает программу и возвращает управление операционной системе. С помощью `set_terminate()` может быть задано какое-нибудь другое действие; тем самым задается обработчик. Объявления упомянутых функций находятся в *except* или *except.h*.

Предоставляемый системой обработчик `unexpected()` вызывается, когда функция возбудила исключение, которое отсутствует в ее списке спецификации исключений. По умолчанию вызывается функция `terminate()`. Или можно воспользоваться `set_terminate()`, чтобы задать обработчик.

С.15.6. Стандартные библиотечные исключения

Исключения стандартной библиотеки наследуются от базового класса `exception`. Два производных класса — это `logic_error` (логическая ошибка) и `runtime_error` (ошибка выполнения). Типы логических ошибок включают: `bad_cast` (ошибка приведения), `out_of_range` (вне диапазона) и `bad_typeid` (ошибка оператора `typeid`), которые должны запускаться в ситуациях, соответствующих названиям перечисленных классов. Типы ошибок на этапе выполнения включают: `range_error` (ошибка диапозона), `overflow_error` (переполнение) и `bad_alloc` (ошибка выделения памяти).

Базовый класс определяет следующую виртуальную функцию:

```
virtual const char* exception::what() const throw();
```

Она служит для возвращения содержательного диагностического сообщения. Эта функция-член должна быть определена в каждом производном классе для того чтобы предоставить дополнительные полезные сообщения. Пустой список спецификации исключений показывает, что сама функция не запускает исключение.

С.16. Предосторожности, связанные с совместимостью

C++ не полностью совместим с C, однако в большинстве случаев можно считать, что C++ представляет собой надмножество языка C. Кроме того, C++ не является стабильной языковой схемой. Он находится в процессе стандартизации[1]. Следующие разделы отмечают особенности языка, представляющиеся проблематичными.

[1] Стандарт языка C++ ратифицирован в августе 1998 года (ISO/IEC 14882). Язык C++, описанный в данной книге, в основном соответствует утвержденному стандарту. — *Примеч. перев.*

C.16.1. Объявления вложенных классов

Изначально правила области видимости для вложенных классов основывались на правилах C. В результате вложенность была косметической, вложенный класс был виден глобально. В C++ вложенный класс локален по отношению к объемлющему его классу. Для доступа к такому вложенному классу может потребоваться многократное использование оператора разрешения области видимости:

```
int outer::inner::foo(double w)  //foo - вложенная
   .....
```

Кроме того, можно создавать классы, вложенные в функции.

C.16.2. Совместимость типов

Вообще, C++ более строг в отношении типов, нежели ANSI C. Ниже приведены некоторые отличия.

Отличия в типах от ANSI C

- Перечисления являются отдельными типами, а их элементы (перечислители) не являются явно `int`. Это означает, что перечисления должны приводиться (cast) при присваивании целых или других перечислений. Перечисления могут повышаться до целых (см. раздел 2.6, «Перечислимые типы», на стр. 49).
- Любой тип указателя может быть преобразован к обобщенному указателю `void*`. Однако, в отличие от ANSI C, в C++ обобщенный указатель не совместим по присваиванию с произвольным типом указателя. Это означает, что в C++ обобщенный указатель для присваивания его необобщенной переменной-указателю необходимо привести к явному типу.
- Символьная константа в C++ имеет тип `char`, в то время как в ANSI C она — `int`. Тип `char` отличается от типов `signed char` и `unsigned char`. Можно перегружать функции, основываясь на этих отличиях; указатели на три эти типа несовместимы.

C.16.3. Разное

Синтаксис функций старого C, когда список аргументов оставался пустым, заменен в ANSI C заданием явного аргумента `void`. В C сигнатура `foo()` рассматривалась как эквивалент использования эллипсиса, а в C++ она рассматривается как эквивалент пустого списка аргументов.

В ранних системах C++ указатель `this` мог быть изменен. Он мог использоваться для выделения памяти для объектов класса. Хотя такое его применение является устаревшим, компиляторы могут по-прежнему допускать его (см. раздел 5.7, «Указатель this», на стр. 143).

C++ допускает, чтобы объявления смешивались (чередовались) с выполняемыми инструкциями. ANSI C допускает объявления только в начале блока или в области видимости файла. Однако в C++ не допускается, чтобы инструкции `goto`, итерации и инструкции ветвления обходили инициализацию переменных. Данное правило отличается от аналогичного в ANSI C.

В C++ глобальный объект данных должен иметь в точности одно определение. Остальные объявления должны использовать ключевое слово `extern`. В ANSI C допускалось многократные объявления без ключевого слова `extern`.

С.17. Новые возможности C++

Большинство компиляторов полностью реализуют шаблоны и исключения. При реализации исключений `new` запускает `xalloc` или `bad_alloc` (см. раздел 11.10, «Стандартные исключения и их использование», на стр. 305).

В язык вошли механизмы, динамически определяющие тип объекта. Это называется идентификацией типа на этапе выполнения (Run Time Type Identification, RTTI). Новыми операторами являются `typeid()`, который применяется либо к *имени_типа*, либо к *выражению*, и `dynamic_cast<`*тип*`>(`*указатель*`)`, который либо возвращает ноль, если приведение неудачно, либо выполняет приведение. При использовании исключений, в случае неудачи при преобразовании, запускается стандартное библиотечное исключение `bad_cast`. Вообще, такие приведения допускаются в полиморфных иерархиях классов (см. раздел 10.9, «Идентификация типа на этапе выполнения», на стр. 304).

Кроме того, добавлены операторы приведения `static_cast` и `reinterpret_cast` (см. раздел 2.5, «Традиционные преобразования», на стр. 46).

Конструктору с одним аргументом можно запретить служить преобразующим конструктором с помощью ключевого слова `expclicit` (см. раздел 6.1.3, «Конструкторы как преобразования», на стр. 158).

Ключевое слово `mutable` позволяет членам данных переменной класса, которая объявлена как `const`, оставаться изменяемыми (см. раздел 5.8.1, «Изменчивость (mutable)», на стр. 146).

Появилось два новых простых типа, `bool` и `wchar_t` (см. раздел 2.4, «Простые типы», на стр. 44).

Существование библиотек с потенциально конфликтными именами стало причиной введения области видимости пространства имен (см. раздел 3.9, «Пространства имен», на стр. 82). Стандартная библиотека инкапсулирована в `namespace std`. Частью стандартной библиотеки является стандартная библиотека шаблонов (STL), включающая стандартные контейнерные классы, итераторы и важнейшие алгоритмы.

Обратитесь к руководству по вашему компилятору для детального описания того, что именно в нем реализовано.

Приложение D

Ввод-вывод

В этом приложении описывается ввод-вывод в C++ с использованием заголовочного файла *iostream.h* и связанных с ним библиотек. В программное обеспечение C++ входит стандартная библиотека, содержащая функции, широко используемые в сообществе C++. В C++ по прежнему доступна стандартная библиотека ввода-вывода для C, описываемая заголовочным файлам *stdio.h*. Однако в C++ появился *iostream.h*, реализующий иной набор функций ввода-вывода. В системах версий до Release 2.0 использовался заголовочный файл *stream.h*; он и сейчас доступен во многих компиляторах C++.

Поток ввода-вывода описан в *iostream.h* как набор классов. Эти классы перегружают операторы << (поместить в) и >> (извлечь из). Потоки могут связываться с файлами. В этом разделе приводятся и обсуждаются примеры обработки файлов с использованием потоков. Часто управление файлами нуждается в обрабатывающих символы макро, которые находятся в *ctype.h*. Они также обсуждаются здесь.

В ООП объекты должны знать, как им печататься, так что мы часто делаем `print` функцией-членом класса. В пользовательских АТД зачастую полезно перегружать <<. В этом разделе для иллюстрации подобной техники мы воспользуемся функциями вывода для типов `card` (карта) и `deck` (колода).

D.1. Класс вывода ostream

Операция вывода помещена в объект типа `ostream`, объявленный в заголовочном файле *iostream.h*. Оператор << перегружен в этом классе для выполнения преобразований вывода над стандартными типами. Перегруженный оператор сдвига влево называется *вставкой* (insertion) или оператором *поместить в* (put to). Оператор левоассоциативен и возвращает значение типа `ostream&`. Стандартным выходным потоком `ostream`, соответствующим `stdout`, является `cout`, а стандартным выходным потоком диагностики `ostream`, соответствующим `stderr`, является `cerr`.

Результатом выполнения простой инструкции вывода, например такой:

```
cout << "x = " << x << '\n';
```

будет вывод на экран строки из четырех символов, подходящего представления `x`, и последующий перевод строки. Представление зависит от того, какая перегруженная версия << была вызвана.

Класс ostream содержит, в частности, следующие открытые члены:

```
ostream& operator<<(int i);
ostream& operator<<(long i);
ostream& operator<<(double x);
ostream& operator<<(char c);
ostream& operator<<(const char* s);
ostream& put(char c);
ostream& write(const char* p, int n);
ostream& flush();
```

Функция-член put() выводит символьное представление c. Функция-член write() выводит строку длиной n на которую указывает указатель p. Функция-член flush() принудительно выводит содержимое буфера потока. Поскольку перечисленные функции являются функциями-членами, их можно использовать следующим образом:

```
cout.put('A');                //вывод A

char* str = "АБВГДЕЖЗ";
cout.write(str + 2, 3);       //вывод ВГД
cout.flush();
```

D.2. Форматированный вывод и iomanip.h

По умолчанию оператор << использует минимальное количество символов, необходимое для представления вывода. Как следствие, вывод может быть запутанным, что видно в следующем примере:

В файле basic_o.cpp

```
int i = 8, j = 9;

cout << i << j;          //путаница: напечатается 89
cout << i << "  "<< j;   //лучше: напечатается 8  9
cout << "i=" << i << " j="<< j;  //лучше всего: i=8 j=9
```

Две схемы, которые мы использовали для правильного разделения вывода — это применение каких-либо строк, разделяющих выводимые значения, и использование \n и \t для создания новых строчек и табулирования. Кроме того, в выходном потоке можно использовать манипуляторы для управления форматированием вывода.

Манипулятор — это значение или функция, которые оказывают особый эффект на поток. Простой пример манипулятора — это endl, определенный в *iostream.h*. Он переводит строку и очищает (flush) выходной поток:

```
x = 1;
cout << "x= " << x << endl;
```

В результате немедленно напечатается:

```
x = 1
```

Другой манипулятор, flush, очищает выходной поток, например:

```
cout << "x= " << x << flush;
```

Это даст почти такой же результат, что и предыдущий пример, но не будет переведена строка.

Манипуляторы dec, hex и oct могут использоваться для изменения основания системы счисления. По умолчанию основание равно 10. Преобразованное основание остается в силе пока не будет явно изменено.

В файле manip.cpp

```
//Использование различных оснований при вводе-выводе целых

#include <iostream.h>

int main()
{
   int i = 10, j = 16, k = 24;
   cout << i << '\t' << j << '\t' << k << endl;
   cout << oct << i << '\t' << j << '\t' << k << endl;
   cout << hex << i << '\t' << j << '\t' << k << endl;
   cout << "Введите 3 целых, например 11 11 12a" << endl;
   cin >> i >> hex >> j >> k;
   cout << dec << i << '\t' << j << '\t' << k << endl;
}
```

Результат будет выглядеть так:

```
10    16    24
12    20    30
a     10    18
Введите 3 целых, например 11 11 12a
11    17    298
```

Причина, по которой в последней строке вывода за 11 следует 17, заключается в том, что второе 11 при вводе интерпретировалось как шестнадцатеричное, то есть 16 + 1.

Приведенные выше манипуляторы находятся в файле *iostream.h*. Другие манипуляторы описаны в *iomanip.h*. Например, setw(int width) является манипулятором, который изменяет стандартную ширину поля для следующей операции форматированного ввода-вывода на значение, представленное его аргументом. В следующей таблице кратко перечислены стандартные манипуляторы, их назначение и где они определены.

Манипуляторы ввода-вывода

Манипулятор	Назначение	Файл
endl	переводит строку и очищает выходной поток	*iostream.h*
ends	пишет '\0' и очищает выходной поток	*iostream.h*
flush	очищает выходной поток	*iostream.h*
dec	десятичные	*iostream.h*
hex	шестнадцатеричные	*iostream.h*

Продолжение на след. стр.

Продолжение

Манипулятор	Назначение	Файл
`oct`	восьмеричные	*iostream.h*
`ws`	пропускает пробелы при вводе	*iostream.h*
`setw(int)`	устанавливает ширину поля	*iomanip.h*
`setfill(int)`	устанавливает заполняющий символ	*iomanip.h*
`setbase(int)`	устанавливает основание системы счисления	*iomanip.h*
`setprecision(int)`	устанавливает точность чисел с плавающей точкой	*iomanip.h*
`setiosflags(long)`	устанавливает флаги потока	*iomanip.h*
`resetiosflags(long)`	переустанавливает флаги потока	*iomanip.h*

D.3. Пользовательские типы: вывод

Обычно определяемые пользователем типы печатаются с помощью функции-члена `print()`. Вспомним типы `card` и `deck` в качестве примера простых пользовательских типов. Напишем набор функций вывода для показа карт.

В файле pr_card.cpp

```
//Вывод карт

#include <iostream.h>

char pips_symbol[14] = { '?', 'A', '2', '3', '4', '5', '6',
          '7','8', '9', 'T', 'J', 'Q', 'K' };
char suit_symbol[4] = { 'c', 'd', 'h', 's' };

enum suit { clubs, diamond, hearts, spades };

class pips {
public:
   void assign(int n) { p = n % 13 + 1; }
   void print() { cout << pips_symbol[p]; }
private:
   int p;
};

class card {
public:
   suit s;
   pips p;
   void assign(int n)
        { cd = n; s = suit(n / 13); p.assign(n); }
   void pr_card()
        { p.print(); cout << suit_symbol[s] << "  "; }
private:
   int cd;      //cd от 0 до 51
};
```

```
class deck {
public:
   void init_deck();
   void shuffle();
   void deal(int, int ,card*);
   void pr_deck();
private:
   card[52];
};

void deck::pr_deck()
{
   for (int i = 0; i < 52; ++i) {
      if (i % 13 == 0) //13 карт в строчку
         cout << endl;
      d[i].pr_card();
   }
}
```

Каждая карта будет напечатана в два символа. Если `d` — переменная типа `deck`, то `d.pr_deck()` распечатает всю колоду, по 13 карт в строке.

Сохраняя верность духу ООП было бы неплохо перегрузить << для достижения тех же целей. Оператор << имеет два аргумента, `ostream&` и АТД, и он должен выдавать `ostream&`. При перегрузке << или >> всегда следует использовать именно ссылку на поток (а не сам поток) и возвращать ссылку на поток, так как мы не хотим копировать объект потока. Запишем эти функции для типов `card` и `deck`:

В файле pr_card2.cpp

```
ostream& operator<<(ostream& out, pips x)
{
   return (out << pips_symbol[x.p]);
}

ostream& operator<<(ostream& out, card cd)
{
   return (out << cd.p << suit_symbol[cd.s] << "   " );
}

ostream& operator<<(ostream& out, deck x)
{
   for (int i = 0; i < 52; ++i) {
      if (i % 13 == 0) //13 карт в строчку
         out << endl;
      out << x.d[i];
   }
   return out;
}
```

Функции, оперирующие с `pips` и `deck`, должны быть друзьями соответствующего класса, поскольку они обращаются к закрытым членам.

D.4. Класс ввода istream

Оператор >> перегружен в istream для выполнения преобразований ввода над стандартными типами. Перегруженный оператор сдвига вправо называется *извлечением* (extraction) или оператором *получить из* (get from). Стандартным входным потоком istream, соответствующим stdin, является cin.

Результатом выполнения простой инструкции ввода, например:

```
cin >> x >> i;
```

будет считывание со стандартного ввода, обычно с клавиатуры, значения для переменной x, а затем — для i. Пробелы игнорируются.

Класс istream содержит, в частности, следующие открытые члены:

```
istream& operator>>(int& i);
istream& operator>>(long& i);
istream& operator>>(double& x);
istream& operator>>(char& c);
istream& operator>>(char* s);
istream& get(char& c);
istream& get(char* s, int n, char c = '\n');
istream& getline(char* s, int n, char c = '\n');
istream& read(char* s, int n);
```

Функция-член get(char& c) помещает символьное представление в c, включая символы пробела. Функция-член get(char* s, int n, char c = '\n') помещает в строку, на которую указывает s, не более чем n - 1 символ (символы до ограничивающего символа c или до конца файла, EOF). В выходную строку помещается завершающий \0. Третий аргумент, который пользователь может не задавать, выступает как терминатор, но не помещается в выходную строку. Если не указано иначе, ввод считывается до следующей новой строки. Функция-член getline() работает как и get(char*, int, char c = '\n'), за исключением того, что она отбрасывает, а не сохраняет ограничивающий символ в соответствующем istream. Функция-член read(char* s, int n) помещает в строку, на которую указывает s, не более чем n символов. Она устанавливает бит сбоя (failbit), если конец файла достигнут до того, как были прочитаны n символов (см. раздел D.8, «Использование состояний потока», на стр. 417). В системах, в которых реализованы стандартные исключения ANSI, может быть запущено ios_base::failure.

В файле basic_i.cpp

```
cin.get(c);             //один символ
cin.get(s, 40);         //длиной в 40 или завершается '\n'
cin.get(s, 10, '*');    //длиной в 10 или завершается *
cin.getline(s, 40);     //то же, что и get, но
                        //'\n' выбрасывается из входного потока
```

Другие полезные функции-члены:

```
int gcout();            //количество только
                        //что извлеченных символов
```

```
istream& ignore(int n=1, int delimiter=EOF);
                          //пропускает символы потока
int peek();               //читает следующий символ
                          //без извлечения его из потока
istream& putback(char c); //помещает символ обратно в поток
```

При перегрузке оператора >> для выполнения ввода в пользовательский тип обычно используют следующую форму:

```
istream& operator>>(istream& p, пользовательский_тип& x)
```

Если функции необходим доступ к закрытым членам x, она должна быть сделана другом класса x. Ключевым моментом является необходимость сделать x ссылочным параметром, чтобы его значение можно было изменять.

D.5. Файлы

Системы C включают stdin, stdout и stderr в качестве стандартных файлов. Кроме того, системы могут определять другие стандартные файлы, такие как stdprn и stdaux. Отвлеченно файл можно рассматривать как поток символов, обрабатываемых последовательно.

Стандартные файлы

C	C++	Название	С чем связан
stdin	cin	файл стандартного ввода	клавиатура
stdout	cout	файл стандартного вывода	экран
stderr	cerr	файл стандартной диагностики	экран
stdprn	cprn	файл стандартного принтера	принтер
stdaux	caux	файл стандартного дополнительного порта	дополнительный порт

Потоки ввода-вывода C++ привязывают первые три из этих стандартных файлов к cin, cout и cerr соответственно. Обычно C++ привязывает cprn и caux к соответствующим стандартным файлам stdprn и stdaux. Существует также clog, который является буферизованной версией cerr. Программист может открывать или создавать другие файлы. Покажем, как это делается в контексте написания программы, которая удваивает пробелы. Имена файлов будут задаваться в командной строке и передаваться в argv.

Файловый ввод-вывод обрабатывается с помощью *fstream.h*. Этот файл содержит классы ifstream и ofstream для создания и управления входными и выходными файловыми потоками. Для того чтобы правильно открыть потоки ofstream или ifstream для файла системы и управлять ими, сначала надо объявить их с соответствующим конструктором.

```
ifstream();
ifstream(const char*, int = ios::in,
        int prot = filebuf::openprot);
ofstream();
```

```
ofstream(const char*, int = ios::out,
        int prot = filebuf::openprot);
```

Конструктор без аргументов создает переменную, которая позднее будет связана с файлом ввода. Конструктор с тремя аргументами принимает в качестве первого аргумента указанный файл. Второй аргумент задает файловый режим. Третий аргумент служит для защиты файла.

Аргументы файлового режима определены как перечислители в классе `ios`:

Файловые режимы

Аргумент	Режим
`ios::in`	режим ввода
`ios::app`	режим добавления
`ios::out`	режим вывода
`ios::ate`	открыть и искать до конца файла
`ios::nocreate`	открыть, но не создавать
`ios::trunc`	отбросить содержимое и открыть
`ios::noreplace`	если файл существует, открытие не выполняется

Таким образом, режимом по умолчанию для `ifstream` является режим ввода, а для `ofstream` — режим вывода. Если открыть файл не удается, поток переводится в плохое состояние (bad state). Поток можно проверить с помощью оператора `!`. В библиотеках, построенных с поддержкой исключений, может быть запущено исключение `failure`.

Другие важные функции-члены, находящиеся в *fstream.h* включают:

```
//открывает файл ifstream
void open(const char*, int = ios::in,
        int prot = filebuf::openprot);

//открывает файл ofstream
void open(const char*, int = ios::out,
        int prot = filebuf::openprot);

void close();
```

Эти функции могут использоваться для открытия и закрытия нужных файлов. Если вы создаете файловый поток с помощью конструктора по умолчанию, обычно следует использовать `open()` для связывания его с файлом. Затем можно применить `close()` для закрытия файла и открыть другой файл, используя тот же поток. Дополнительные функции-члены в других классах ввода-вывода предоставляют полный набор действий с файлами.

В файле dbl_sp.cpp

```
//Программа для удваивания пробелов в файле
//Использование: имя_исполняемой_программы файл1 файл2
//файл1 должен существоать и открываться
//в файл2, если он существует, должна быть разрешена запись
```

```
#include <fstream.h>    //включает iostream.h
#include <stdlib.h>

void double_space(ifstream& f, ofstream& t)
{
   char c;
   while (f.get(c)) {
      t.put(c);
      if (c == ' ')
         t.put(c);
   }
}

int main (int argc, char* argv)
{
   if (argc != 3) {
      cout << "\nИспользование: " << argv[0]
           << " входной_файл  выходной_файл" << endl;
      exit(1);
   }

   ifstream f_in(argv[1]);
   ofstream f_out(argv[2]);

   if (!f_in) {
      cerr << "Невозможно открыть " << argv[1] << endl;
      exit(1);
   }
   if (!f_out) {
      cerr << "Невозможно открыть " << argv[2] << endl;
      exit(1);
   }
   double_space(f_in, f_out);
}
```

D.6. Использование строк как потоков

Класс strstream позволяет обращаться со строками char* как с потоками iostream. При использовании потоков strstream должна быть подключена библиотека *strstream.h*. Новые библиотеки будут включать *sstream*, который предоставит istringstream и ostringstream, поддерживающие ввод-вывод с помощью стандартного библиотечного типа string.[1]

Поток istrstream используется, когда ввод осуществляется из строки, а не из потока. Перегруженный оператор << (извлечь из) может использоваться с переменными istrstream. Формы объявления переменной istrstream следующие:

```
istrstream имя (char* s);
istrstream имя (char* s, int n);
```

[1] Стандарт C++, ратифицированный в августе 1998 года, включает *sstream*.— *Примеч. перев.*

где s — это строка, используемая в качестве ввода, n — необязательная длина буфера ввода, а *имя* используется вместо cin. Если n не указано, строка должна завершаться нулевым символом. Ограничитель конца строки рассматривается как конец файла. Например:

В файле str_strm.cpp

```
char name[15];
int total;
char* scores[4] = { "Дейв 2", "Ида 5", "Джим 4", "Айра 8" };

istrstream ist(scores[3]);   //ist использует scores[3]
ist >> name >> total;        //name = 'Айра', total = 8
```

Объявления ostrstream имеют следующие формы:

```
ostrstream ();
ostrstream имя (char* s, int n, int mode = ios::out);
```

где s — это указатель на buf для получения строки, n — необязательный размер буфера, а mode указывает, должны ли данные помещаться в пустой буфер (ios::out) или добавляться к оканчивающейся нулем строке в буфере (ios::app или ios::ate). Если размер не указан, буфер размещается динамически. Переменная ostrstream может использовать для построения строки перегруженный оператор >> (поместить в). Использование ostrstream особенно полезно, когда вы хотите создать одну строку из информации, хранящейся в разных переменных. Эта техника применяется при обработке исключений для построения единственной строковой переменной, которая будет использоваться в качестве аргумента throw(). Наш пример с vect в разделе 11.9, «Пример кода с исключениями», на стр. 322 использует подобную технику. В следующем примере обратите внимание на то, что ost2 должна содержать существующую и завершающуюся нулем строку для того, чтобы добавление выполнялось корректно.

```
ostrstream ost1;
ostrstream ost2 (charbuf, 1000, ios::app);

ost1 << name << "  " << score << endl;
ost2 << address << city << endl << ends;
```

D.7. Функции и макро в *ctype.h*

В системе имеется стандартный заголовочный файл *ctype.h* или *cctype*, содержащий набор функций, которые используются для проверки символов и набор функций для преобразования символов. Они могут быть выполнены как макро или как встроенные функции. Мы вспомнили об этом здесь, так как это полезно для ввода-вывода. Те функции, которые лишь проверяют символ, возвращают значение типа int. Их аргумент — также типа int.

Функция в ctype.h	Возвращается ненулевое значение (истина), если
isalpha(c)	c является буквой
isupper(c)	c является прописной буквой
islower(c)	c является строчной буквой
isdigit(c)	c является цифрой

Функция в ctype.h	Возвращается ненулевое значение (истина), если
`isxdigit(c)`	`c` является шестнадцатеричной цифрой
`isspace(c)`	`c` является символом пробела
`isalnum(c)`	`c` является буквой или цифрой
`ispunct(c)`	`c` является знаком пунктуации
`isgraph(c)`	`c` является печатаемым символом, кроме пробела
`isprint(c)`	`c` является печатаемым символом
`iscntrl(c)`	`c` является управляющим символом
`isascii(c)`	`c` является кодом ASCII

Остальные функции обеспечивают надлежащее преобразование символьного значения. Заметьте, что эти функции не изменяют хранящееся в памяти значение `c`.

Преобразующая функция в ctype.h	Результат
`toupper(c)`	преобразует `c` из строчной в прописную
`tolower(c)`	преобразует `c` из прописной в строчную
`toascii(c)`	преобразует `c` в ASCII код

Функции с ASCII-кодами часто применяются в ASCII-системах.

D.8. Использование состояния потока

Каждый поток имеет связанное с ним состояние, которое можно выяснить. Эти состояния в существующих системах таковы:

```
enum io_state { goodbit, eofbit, failbit, badbit };
  //состояние потока: хороший бит, бит конца файла,
  //бит сбоя, плохой бит
```

Системы ANSI предлагают тип `ios_base::iostate` в качестве типа битовой маски, определяющего эти значения. Когда некая операция ввода-вывода устанавливает «нехорошие» значения, ANSI системы могут запускать стандартное исключение ввода-вывода `ios_base::failure`. Связанная с этим исключением функция-член `what()` возвращает строковое (`char*`) сообщение, которое объясняет причину сбоя.

Значения для конкретного потока могут быть проверены с помощью открытых функций-членов, представленных в следующей таблице:

Функция состояния потока	Что она возвращает
`int good();`	не-ноль, если не установлен бит `EOF` или другой бит ошибки
`int eof();`	не-ноль, если установлен `eofbit`
`int fail();`	не-ноль, если установлены `failbit` и `badbit`
`int bad();`	не-ноль, если установлен `badbit`
`int rdstate();`	возвращает состояние ошибки
`void clear(int i = 0);`	переустанавливает состояние ошибки
`int operator!();`	истина, если установлены `failbit` или `badbit`
`operator void*;`	возвращает ложь, если установлены `failbit` или `badbit`

Проверка того, не находится ли поток в «нехорошем» состоянии, может защитить программу от зависания. Состояние потока good означает, что предыдущая операция ввода-вывода сработала и следующая операция имеет шансы завершиться успешно. Состояние потока EOF означает, что предыдущая операция ввода возвратила состояние «конец файла». Состояние потока fail означает, что предыдущая операция ввода-вывода потерпела неудачу, но поток восстановит работоспособность после того, как будет очищен бит ошибки. Состояние потока bad означает, что предыдущая операция ввода-вывода была ошибочной, но поток может восстановить работоспособность после исправления ошибочной ситуации.

Можно также проверить поток непосредственно. Его значение ненулевое, если он находится в состоянии good или EOF:

```
if (cout << x )  //вывод удался
   .....
else
   .....          //вывод не удался
```

Следующая программа подсчитывает количество слов, поступающих со стандартного ввода. Обычно он связывается с существующим файлом. Программа иллюстрирует идеи, обсуждавшиеся в этом и двух предыдущих разделах.

В файле word_cnt.cpp

```
//Программа word_cnt для подсчета слов
//Использование: имя_программы < файл

#include <iostream.h>
#include <ctype.h>

int found_next_word();

int main()
{
   int word_cnt = 0;

   while (found_next_word())
      ++word_cnt;
   cout << "количество слов: " << word_cnt << endl;
}

int found_next_word()
{
   char c;
   int word_sz = 0;

   cin >> c;
   while (!cin.eof() && !isspace(c)) {
      ++word_sz;
      cin.get(c);
   }
```

```
    return word_sz;
}
```

Символ, не являющийся пробелом, поступает из входного потока и присваивается c. Цикл while проверяет, что и соседние символы не являются пробелами. Цикл завершается, когда встречается символ конца файла или символ пробела. Возвращается нулевой размер слова, если единственный найденный символ, не являющийся пробелом — символ конца файла. И последнее замечание: цикл не может быть переписан вот так

```
while (!cin.eof() && !isspace(c)) {
   ++word_sz;
   cin >> c;
}
```

потому что тогда будут пропускаться пробелы.

D.9. Совместное использование библиотек ввода-вывода

В этой книге мы везде использовали *iostream.h*. Но вы вправе предпочесть *stdio.h*. Эта библиотека является стандартом в сообществе C и хорошо известна. Ее недостаток в том, что она не типобезопасна. Функции вроде printf() используют непроверяемые компилятором списки аргументов переменной длины. Потоковый ввод-вывод требует, чтобы аргументы его функций и перегруженных операторов принадлежали типам, преобразуемым по присваиванию. Возможно, вы захотите смешивать обе формы ввода-вывода. При этом могут возникать проблемы синхронизации, поскольку две библиотеки используют различные стратегии буферизации. Этого можно избежать с помощью вызова

```
ios::sync_with_stdio();
```

В файле mix_io.cpp

```
//Программа mix_io с синхронизованным вводом-выводом

#include <stdio.h>
#include <iostream.h>

unsigned long fact(int n)
{
   unsigned long f = 1;

   for (int i = 2; i <= n; ++i)
      f *= i;
   return f;
}

int main()
{
   int n;

   ios::sync_with_stdio();
```

```
    do {
      cout <<
        "\nВведите n положительных чисел или 0 для выхода: ";
      scanf("%d", &n);
      printf("\n fact(%d) = %ld", n, fact(n));
    } while (n > 0);
    cout << "\nконец работы" << endl;
}
```

Заметьте, что для целых значений, больших 12, наступит переполнение. Можно (и это безопасно) смешивать `stdio` и `iostream`, при условии, что они не смешиваются по отношению к одному и тому же файлу.

Приложение E

STL и строковые библиотеки

Стандартная библиотека шаблонов (STL, Standard Template Library) является стандартной библиотекой C++, предоставляющей возможности обобщенного программирования для многих стандартных структур данных и алгоритмов. Библиотека предлагает контейнеры, итераторы и алгоритмы, которые поддерживают возможности обобщенного программирования. Мы представляем краткое описание, охватывающее эти три компонента.

Библиотека построена с использованием шаблонов, ее дизайн вполне ортогонален. Компоненты можно комбинировать друг с другом, используя в качестве параметров как собственные типы C++, так и типы, определяемые пользователем. Необходимо только следить за правильностью инстанцирования различных элементов библиотеки STL. В зависимости от системы требуются различные заголовочные файлы. Наши примеры соответствуют стандарту ANSI и инкапсулированы в стандартном пространстве имен (`namespace std`). В качестве примера см. раздел 9.7, «STL», на стр. 259 (в файле *stl_cont.cpp*).

E.1. Контейнеры

Контейнеры подразделяются на два основных семейства: последовательные контейнеры и ассоциативные контейнеры. Последовательные контейнеры включают векторы (vectors), списки (lists) и двусторонние очереди (deques). Эти контейнеры упорядочиваются заданием последовательности элементов. Ассоциативные контейнеры включают множества (sets), мультимножества (multisets), отображения (maps), мультиотображения (multimaps) и содержат ключи для поиска элементов. Контейнер-отображение является ассоциативным массивом. Ему необходимо, чтобы для хранимых элементов была определена операция сравнения. Все варианты контейнеров имеют похожий интерфейс.

Интерфейсы типичных контейнеров STL

- конструкторы, включая конструкторы по умолчанию и копирующие конструкторы
- доступ к элементу

- вставка элемента
- удаление элемента
- деструктор
- итераторы

Проход по контейнеру осуществляется с помощью итераторов. Это «указателеподобные» объекты, доступные в виде шаблонов, и оптимизированные для использования с контейнерами STL. В качестве примера см. раздел 9.8, «Контейнеры», на стр. 260 (в файле *stl_deq.cpp*).

В следующей таблице, описывающей интерфейс контейнерных классов, они обозначены как CAN.

Определения контейнеров STL

`CAN::value_type`	тип значения, содержащегося в CAN
`CAN::reference`	тип-ссылка на значение
`CAN::const_reference`	константная ссылка
`CAN::pointer`	указатель на значение
`CAN::iterator`	тип-итератор
`CAN::const_iterator`	константный итератор
`CAN::reverse_iterator`	обратный итератор
`CAN::const_reverse_iterator`	константный обратный итератор
`CAN::difference_type`	разница между двумя значениями `CAN::iterator`
`CAN::size_type`	размер CAN

Эти определения доступны во всех контейнерных классах. Например, если мы используем векторный контейнерный класс, то `vector<char>::value_type` расскажет, что в векторном контейнере хранится символьное значение. Проход по такому контейнеру может быть выполнен с помощью `vector<char>::iterator`.

Контейнеры предлагают операторы равенства и сравнения. Они также имеют обширный список стандартных функций-членов.

Члены контейнера STL

`CAN::CAN()`	конструктор по умолчанию
`CAN::CAN(c)`	копирующий конструктор
`c.begin()`	начальная позиция контейнера c
`c.end()`	конечная позиция c[1]
`c.rbegin()`	начало для обратного итератора
`c.rend()`	конец для обратного итератора
`c.size()`	число элементов в CAN
`c.max_size()`	наибольший размер
`c.empty()`	истина, если CAN пуст
`c.swap(d)`	обмен элементами двух CAN

[1] Точнее, `c.end()` указывает не на последний элемент контейнера, а на (несуществующий) элемент, «следующий за последним». Такое соглашение C++ облегчает кодирование операций над контейнерами. — *Примеч. перев.*

Операторы контейнера STL	
`== != < > <= >=`	операторы равенства и сравнения, использующие `CAN::value_type`

E.1.1. Последовательные контейнеры

Последовательные контейнеры — это вектор, список и двусторонняя очередь. Они содержат последовательность доступных элементов. В C++ тип массива во многих случаях может рассматриваться как последовательный контейнер. В качестве примера см. раздел 9.8.1, «Последовательные контейнеры», на стр. 262 (в файле *stl_vect.cpp*).

В следующей таблице, описывающей интерфейс последовательных классов, они обозначены как `SEQ`. Приведенные ниже возможности дополняют уже описанный интерфейс любого `CAN`.

Члены последовательных контейнеров STL	
`SEQ::SEQ(n, v)`	`n` элементов со значением `v`
`SEQ::SEQ(b_it, e_it)`	от `b_it` до `e_it - 1`
`c.insert(w_it, v)`	вставка `v` перед `w_it`
`c.insert(w_it, v, n)`	вставка `n` копий `v` перед `w_it`
`c.insert(w_it, b_it, e_it)`	вставка значений от `b_it` до `e_it` перед `w_it`
`c.erase(w_it)`	стирает элемент, «прописанный» в `w_it`
`c.erase(b_it, e_it)`	стирает от `b_it` до `e_it`

E.1.2. Ассоциативные контейнеры

Ассоциативные контейнеры — это множества, отображения, мультимножества и мультиотображения. Они содержат доступные по ключу элементы и упорядочивающее отношение `Compare`, являющееся сравнивающим объектом ассоциативного контейнера. В качестве примера см. раздел 9.8.2, «Ассоциативные контейнеры», на стр. 263 (в файле *stl_age.cpp*).

В следующей таблице, описывающей интерфейс ассоциативных классов, они обозначены как `ASSOC`. Имейте в виду, что приведенные ниже возможности дополняют уже описанный интерфейс любого `CAN`.

Определения ассоциативных контейнеров STL	
`ASSOC::key_type`	тип ключа поиска
`ASSOC::key_compare`	тип сравнивающего объекта
`ASSOC::value_compare`	тип для сравнения `ASSOC::value_type`

Для инициализации ассоциативные контейнеры имеют несколько стандартных конструкторов.

Ассоциативные конструкторы STL	
`ASSOC()`	конструктор по умолчанию, использующий `Compare`
`ASSOC(cmp)`	конструктор, использующий `cmp` как сравнивающий объект
`ASSOC(b_it, e_it)`	использует элемент в диапазоне от `b_it`, до `e_it`, применяя `Compare`
`ASSOC(b_it, e_it, cmp)`	использует элементы в диапазоне от `b_it`, до `e_it` и `cmp` как сравненивающий объект

Что отличает ассоциативные конструкторы от конструкторов последовательных контейнеров, так это использование объекта сравнения.

STL: функции-члены вставки и удаления	
`c.insert(t)`	если ни один из существующих элементов не имеет такого же ключа, как `t`, вставляет `t` и возвращает пару `<iterator, bool>` с `bool`, имеющим значение `true, если t` отсутствовал
`c.insert(w_it, t)`	вставляет `t` с `w_it` в качестве начальной позиции поиска; терпит неудачу в множествах и отображениях, если ключевое значение уже присутствует; возвращает позицию вставки
`c.insert(b_it, e_it)`	вставляет элементы в этом диапазоне
`c.erase(k)`	стирает элементы, ключевое значение которых равно `k`, возвращая количество стертых элементов
`c.erase(w_it)`	стирает указываемый элемент
`c.erase(b_it, e_it)`	стирает диапазон элементов

Вставка выполняется, если ни один из элементов с таким же ключом еще не присутствует.

Функции-члены STL	
`c.find(k)`	возвращает итератор на элемент, имеющий заданный ключ `k`
`c.count(k)`	возвращает число элементов с ключом `k`
`c.lower_bound(k)`	возвращает итератор на первый элемент со значением, большим или равным `k`
`c.upper_bound(k)`	возвращает итератор на первый элемент со значением, большим `k`
`c.equal_range(k)`	возвращает пару итераторов на `lower_bound` и `upper_bound`

E.1.3. Адаптеры контейнеров

Классы адаптеров контейнеров (container adaptors) — это контейнерные классы, которые изменяют имеющиеся контейнеры с тем, чтобы обеспечить иное открытое поведе-

ние на основе существующей реализации. Три предлагаемых контейнерных адаптера — это `stack` (стек), `queue` (очередь) и `priority_queue` (приоритетная очередь).

Стек может быть получен (адаптирован) из вектора, списка и двусторонней очереди. Он нуждается в реализации, поддерживающей операции `push`, `pop` и `top`. Это — структура данных «последний вошел — первый вышел».

STL: функции адаптированного стека

`void push(const value_type& v)`	помещает `v` в стек
`void pop()`	убирает верхний элемент из стека
`value_type& top() const`	возвращает верхний элемент из стека
`bool empty() const`	возвращает истину, если стек пуст
`size_type size() const`	возвращает число элементов в стеке
`operator==` и `operator<`	равенство и лексикографическое «меньше чем»

Очередь может быть адаптирована из списка или двусторонней очереди. Она нуждается в реализации, поддерживающей операции `empty`, `size`, `front`, `back`, `push_back` и `pop_front`. Это — структура данных «первый вошел — первый вышел».

STL: функции адаптированной очереди

`void push(const value_type& v)`	помещает `v` в конец очереди
`void pop()`	удаляет первый элемент из очереди
`value_type& front() const`	возвращает первый элемент очереди
`value_type& back() const`	возвращает последний элемент очереди
`bool empty() const`	возвращает истину, если очередь пуста
`size_type size() const`	возвращает число элементов в очереди
`operator==` и `operator <`	равенство и лексикографическое «меньше чем»

Приоритетная очередь может быть адаптирована из вектора или двусторонней очереди. Она нуждается в реализации, поддерживающей операции `empty`, `size`, `front`, `push_back` и `pop_back`. Кроме того, приоритетной очереди для ее инстанцирования необходим объект сравнения. Как определено отношением сравнения для приоритетной очереди, верхний элемент — это самый большой элемент.

STL: функции приоритетной очереди

`void push(const value_type& v)`	помещает `v` в приоритетную очередь
`void pop()`	удалает верхний элемент приоритетной очереди
`value_type& top() const`	возвращает верхний элемент приоритетной очереди
`bool empty() const`	проверяет, не пуста ли приоритетная очередь
`size_type size() const`	показывает число элементов в приоритетной очереди

Стек может быть получен, в частности, из реализации `vector`. Обратите внимание, как АТД из STL заменяют наши самостоятельно разработанные реализации этих типов. В качестве примера см. раздел 9.8.3, «Адаптеры контейнеров», на стр. 265 (в файле *stl_stak.cpp*).

E.2. Итераторы

Перемещение по контейнерам производится с помощью итератора. Итератор может рассматриваться как усовершенствованный тип указателей. Итератор является шаблоном, который инстанцируется типом контейнерного класса, итерируемого им. Существует пять типов итераторов: ввода, вывода, прохода вперед, двусторонние и произвольного доступа. Не все типы итераторов могут быть доступны для данного контейнерного класса. Например, итератор произвольного доступа доступен для векторов, но не для отображений.

Итераторы ввода и вывода не очень требовательны. Они могут использоваться для ввода и вывода и имеют для этих целей специальные реализации, называемые `istream_iterator` и `ostream_iterator` (см. разделы E.2.2, «Итератор istream_iterator», на стр. 427 и E.2.3, «Итератор ostream_iterator», на стр. 427). Итератор прохода вперед может выполнять все то же, что и итераторы ввода-вывода и, кроме того, может запоминать местоположение внутри контейнера. Двусторонний итератор может выполнять проход в прямом и обратном направлении. Итератор произвольного доступа является наиболее мощным и может получать доступ к любому элементу (при этом время доступа одинаково для всех элементов) в подходящем контейнере, таком как `vector`. В качестве примера см. раздел 9.9.1, «Итераторы istream_iterator и ostream_iterator», на стр. 268 (в файле *stl_io.cpp*).

E.2.1. Категории итераторов

Итераторы ввода поддерживают операции равенства, разыменования и автоинкремента. Итераторы, отвечающие этим условиям, могут использоваться для однопроходных алгоритмов, которые читают значения структуры данных в одном направлении. Специальным случаем итератора ввода является `istream_iterator`.

Итераторы вывода поддерживают разыменование, допустимое только с левой стороны присваивания, и автоинкремент. Итераторы, отвечающие этим условиям, могут использоваться для однопроходных алгоритмов, которые пишут значения в структуры данных в одном направлении. Специальным случаем итератора вывода является `ostream_iterator`.

Итераторы прохода вперед поддерживают все операции итераторов ввода-вывода и, кроме того, позволяют без ограничений применять присваивание. Соответственно, имеется возможность сохранять позицию внутри структуры данных при многократных проходах. Таким образом, с помощью итератора прохода вперед можно написать общий однонаправленный многопроходный алгоритм.

Двусторонние итераторы поддерживают все операции итераторов прохода вперед, а также автоинкремент и автодекремент. Таким образом, с помощью двустороннего итератора можно написать общий двунаправленный многопроходный алгоритм.

Итераторы произвольного доступа поддерживают все операции двусторонних итераторов, а также арифметические адресные операции, такие как индексирование.

Кроме того, итераторы произвольного доступа поддерживают операции сравнения. Таким образом, с помощью итераторов произвольного доступа можно написать такие алгоритмы, как quicksort (быстрая сортировка), которые требуют эффективного произвольного доступа.

Контейнерные классы и алгоритмы диктуют выбор категории доступного или необходимого итератора. Так, векторный контейнер допускает итераторы произвольного доступа, а список — нет. Сортировка обычно нуждается в итераторе произвольного доступа, а для поиска нужен лишь итератор ввода.

E.2.2. Итератор istream_iterator

Итератор `istream_iterator` происходит от итератора ввода `input_iterator` и работает исключительно с чтением из потоков. Шаблон для `istream_iterator` инстанцируется как *<тип, расстояние>*. Это расстояние обычно задается как `ptrdiff_t`. Как определено в *cstddef* или *stddef.h*, это целый тип, представляющий разницу между двумя значениями указателей. В качестве примера см. раздел 9.9.1, «Итераторы istream_iterator и ostream_iterator», на стр. 268 (в файле *stl_io.cpp*).

E.2.3. Итератор ostream_iterator

Итератор `ostream_iterator` происходит от итератора вывода `output_iterator` и работает исключительно с записью в потоки. Итератор `ostream_iterator` может быть создан с ограничителем (тип которого — `char*`), как правило — `"\t"`. То есть в поток `cout` после каждого записанного, к примеру, целого значения будет вставляться символ табуляции. В программе из указанного ниже примера при разыменовании итератора `out` присвоенное целое значение записывается в `cout`. В качестве примера см. раздел 9.9.1, «Итераторы istream_iterator и ostream_iterator», на стр. 268 (в файле *stl_io.cpp*).

E.2.4. Адаптеры итераторов

Итераторы могут быть адаптированы для обеспечения обратного просмотра и просмотра со вставкой. В качестве примера см. раздел 9.9.2, «Адаптеры итераторов», на стр. 269 (в файле *stl_iadp.cpp*).

STL: Адаптеры итераторов

- Обратные итераторы — обратный порядок итерации
- Итераторы вставки — вместо обычного режима перезаписи происходит вставка

Кратко перечислим итераторы и их назначение в данной библиотеке.

- ```
  template <class BidiIter,
    class T, class Ref = T&,
    class Distance = ptrdiff_t>
  class reverse_bidirectional_iterator;
  ```

  Этот итератор изменяет нормальное направление итерации на противоположное. Используйте `rbegin()` и `rend()` для задания диапазона.

- ```
  template <class RandAccIter,
    class T, class Ref = T&,
  ```

```
    class Distance = ptrdiff_t>
class reverse_iterator;
```

Этот итератор изменяет нормальное направление итерации на противоположное. Используйте `rbegin()` и `rend()` для задания диапазона.

- ```
  template <class Can>
    class insert_iterator;
  template <class Can, class Iter>
  insert_iterator<Can>
    inserter(Can& c, Iter p);
  ```

  Этот итератор производит вставку вместо обычного режима перезаписи. Вставка в контейнер `c` производится с позиции `p`.

- ```
  template <class Can>
    class front_insert_iterator;
  template <class Can>
  front_insert_iterator<Can>
    front_inserter(Can& c);
  ```

  Вставка спереди выполняется с начала контейнера, для чего требуется член `push_front()`.

- ```
  template <class Can>
    class back_insert_iterator;
  template <class Can>
  back_insert_iterator<Can>
    back_inserter(Can& c);
  ```

  Вставка сзади выполняется с конца контейнера, для чего требуется член `push_back()`.

E.3. Алгоритмы

Библиотека алгоритмов STL содержит четыре категории алгоритмов.

Категории библиотеки алгоритмов STL

- алгоритмы сортировки
- не изменяющие последовательность алгоритмы
- изменяющие последовательность алгоритмы
- численные алгоритмы

Эти алгоритмы обычно используют итераторы для доступа к контейнерам, инстанцированным заданным типом. Полученный код может состязаться в эффективности со специально созданным кодом.

E.3.1. Алгоритмы сортировки

Алгоритмы сортировки включают общую сортировку, слияние, лексикографическое сравнение, перестановку, двоичный поиск и некоторые другие сходные операции. Эти алгоритмы имеют версии, использующие либо `operator<()`, либо объект `Compare`. Часто они нуждаются в итераторе произвольного доступа. См. раздел 9.10.1, «Алгоритмы сортировки», на стр. 270.
  ```

Рассмотрим библиотечные прототипы алгоритмов сортировки.

- ```
  template<class RandAcc>
  void sort(RandAcc b, RandAcc e);
  ```

  Это — алгоритм быстрой сортировки элементов в диапазоне от `b` до `e`. Итераторный тип `RandAcc` должен быть итератором произвольного доступа.

- ```
  template<class RandAcc>
  void stable_sort(RandAcc b, RandAcc e);
  ```

  Это — алгоритм сохраняющей (stable) сортировки элементов в диапазоне от `b` до `e`. При сохраняющей сортировке одинаковые элементы сохраняют свое взаимное расположение.

- ```
  template<class RandAcc>
  void partial_sort(RandAcc b, RandAcc m, RandAcc e);
  ```

  Это — алгоритм частичной сортировки элементов в диапазоне от `b` до `e`. Диапазон от `b` до `m` заполняется элементами, отсортированными до позиции `m`.

- ```
  template<class InputIter, class RandAcc>
  void partial_sort_copy(InputIter b, InputIter e,
    RandAcc result_b, RandAcc result_e);
  ```

  Это — алгоритм частичной сортировки элементов в диапазоне от `b` до `e`. Сортируемые элементы принимаются из диапазона итератора ввода и копируются в диапазон итератора произвольного доступа. Используется наименьший из двух диапазонов.

- ```
  template<class RandAcc>
  void nth_element(RandAcc b, RandAcc nth, RandAcc e);
  ```

  Элемент `nth` (*n*-ый) разделяет остальные элементы (слева — меньшие, справа — большие). Например, если выбрана пятая позиция, то четыре наименьших элемента помещаются слева от пятого, а остальные элементы помещаются справа и будут больше пятого.

- ```
  template<class InputIter1, class InputIter2, class OutputIter>
  OutputIter merge(InputIter1 b1, InputIter1 e1, InputIter2 b2,
  InputIter2 e2', OutputIter result_b);
  ```

  Элементы в диапазонах от `b1` до `e1` и от `b2` до `e2` сливаются с начальной позиции `result_b`.

- ```
  template<class BidiIter>
  void inplace_merge(BidiIter b, BidiIter m, BidiIter e);
  ```

  Элементы в диапазонах от `b` до `m` и от `m` до `e` сливаются «по месту жительства»[1].

---

[1] Существует еще один вариант этой функции, позволяющий задавать критерий сравнения:

```
template<class BidiIter, class Compare>
void inplace_merge(BidiIter b, BidiIter m, BidiIter e, Compare comp);
```

Обсуждаемая функция применяется, в частности, для сортировки по нескольким критериям. См. об этом подробнее в [Страуструп, §18.7.3]. — *Примеч. перев.*
```

Используем таблицу, чтобы кратко перечислить другие алгоритмы из данной библиотеки, а также их назначение.

STL: библиотечные функции, связанные с сортировкой	
`binary_search(b, e, t)`	истина, если `t` находится между `b` и `e`
`lower_bound(b, e, t)`	поиск первой позиции, в которую можно поместить `t` не нарушая порядок сортировки (предполагается, что контейнер отсортирован)
`upper_bound(b, e, t)`	поиск последней позиции, в которую можно поместить `t` не нарушая порядок сортировки (предполагается, что контейнер отсортирован)
`equal_range(b, e, t)`	возвращает пару итераторов диапазона, в который может быть помещен `t` не нарушая порядок сортировки (предполагается, что контейнер отсортирован)
`push_heap(b, e)`	помещает элемент по адресу `e` в уже существующую кучу
`pop_heap(b, e)`	меняет местами элементы с адресами `e` и `b` и перераспределяет динамическую память (reheap)
`sort_heap(b, e)`	выполняет сортировку в куче
`make_heap(b, e)`	создает кучу
`next_permutation(b,e)`	производит следующую перестановку
`prev_permutation(b,e)`	производит предыдущую перестановку
`lexicographical_compare (b1, e1, b2, e2)`	возвращает истину, если последовательность 1 лекскикографически меньше *последовательности* 2
`min(t1, t2)`	возвращает наименьший из `t1` и `t2`, которые передаются по ссылке
`max(t1, t2)`	возвращает наибольший из `t1` и `t2`, которые передаются по ссылке
`min_element(b, e)`	возвращает позицию наименьшего элемента
`max_element(b, e)`	возвращает позицию наибольшего элемента
`includes(b1, e1, b2, e2)`	возвращает истину, если вторая является последовательность подмножеством первой
`set_union(b1, e1, b2, e2, r)`	возвращает объединение множеств в виде итератора вывода `r`
`set_intersection (b1, e1, b2, e2, r)`	возвращает пересечение множеств в виде итератора вывода `r`
`set_difference (b1, e1, b2, e2, r)`	возвращает разность множеств в виде итератора вывода `r`
`set_symmetric_difference (b1, e1, b2, e2, r)`	возвращает симметричную разность множеств в виде итератора вывода `r`

Ряд алгоритмов имеют форму, которая использует объект `Compare` вместо `operator<()`, например:

- `template<class RandAcc, class Compare>`
  `void sort(RandAcc b, RandAcc e, Compare comp);`

  Это — алгоритм быстрой сортировки элементов в диапазоне от `b` до `e`, использующий `comp` для упорядочивания.

E.3.2. Не изменяющие последовательность алгоритмы

Неизменяющие (nonmutating) алгоритмы не модифицируют содержимое контейнеров, с которыми они работают. Типичная подобная операция — это поиск в контейнере конкретного элемента и возврат его позиции. В качестве примера см. раздел 9.10.2, «Не изменяющие последовательность алгоритмы», на стр. 270 (в файле *stl_find.cpp*).

Представим библиотечные прототипы для неизменяющих алгоритмов.

- `template<class InputIter, Class T>`
  `InputIter find(InputIter b, InputIter e, const T& t);`

  Этот алгоритм находит позицию элемента `t` в диапазоне от `b` до `e`.

- `template<class InputIter, Class Predicate>`
  `InputIter find(InputIter b, InputIter e, Predicate p);`

  Этот алгоритм находит позицию первого элемента в диапазоне от `b` до `e`, который делает предикат `p` истинным; если такого элемента нет, возвращается позиция элемента `e`.

- `template<class InputIter, Class Function>`
  `void for_each(InputIter b, InputIter e, Function f);`

  Этот алгоритм применяет функцию `f` к каждому элементу в диапазоне от `b` до `e`.

В следующей таблице кратко перечислены другие алгоритмы и их назначение.

STL: библиотечные функции, не изменяющие последовательность

`next_permutation(b, e)`	производит следующую перестановку
`prev_permutation(b, e)`	производит предыдущую перестановку
`count(b, e, t, n)`	помещает в `n` количество элементов, равных `t`
`count_if(b, e, p, n)`	помещает в `n` количество элементов, которые делают предикат `p` истинным
`adjacent_find(b, e)`	возвращает первую позицию соседних равных элементов; если таких нет, возвращает `e`
`adjacent_find(b, e, binp)`	возвращает первую позицию соседних элементов, удовлетворяющих двоичному предикату `binp`; если таких нет, возвращает `e`
`mismatch(b1, e1, b2)`	ищет первую пару элементов последовательностей, которые не совпадают, и возвращает пару итераторов на эти элементы

Продолжение на след. стр.

Продолжение

STL: библиотечные функции, не изменяющие последовательность	
`mismath(b1, e1, b2, binp)`	то же, что и предыдущее, но вместо равенства используется двоичный предикат `binp`
`equal(b1, e1, b2)`	возвращает истину, если указанные посследовательности совпадают, иначе возвращает ложь
`equal(b1, e1, b2, binp)`	то же, что и предыдущее, но вместо равенства используется двоичный предикат `binp`
`search(b1, e1, b2, e2)`	проверяет, содержит ли первая последовательность вторую. Если да, то возвращает итератор на первый элемент соответствующей подпоследовательности; в противном случае возвращает `e1`
`search(b1, e1, b2, e2, binp)`	то же, что и предыдущее, но вместо равенства используется двоичный предикат `binp`

E.3.3. Изменяющие последовательность алгоритмы

Изменяющие алгоритмы могут модифицировать содержимое контейнеров, с которыми они работают. Типичная подобная операция — это обращение содержимого контейнера. В качестве примера см. раздел 9.10.3, «Изменяющие последовательность алгоритмы», на стр. 271 (в файле *stl_revr.cpp*).

Рассмотрим библиотечные прототипы для изменяющих алгоритмов.

- ```
  template<class InputIter, class OutputIter>
  OutputIter copy(InputIter b1, InputIter e1, OutputIter b2);
  ```

  Это — алгоритм копирования элементов от `b1` до `e1`. Копия размещается начиная с позиции `b2`. Возвращается позиция конца копии.

- ```
  template<class BidiIter1, class BidiIter2>
  BidiIter2 copy_backward(BidiIter1 b1, BidiIter1 e1,
                                                BidiIter2 b2);
  ```

  Это —алгоритм копирования элементов от `b1` до `e1` в обратном порядке. Копия размещается начиная с позиции `b2`. Копирование выполняется в обратную сторону: `e1` в `b2`, `e1--` в `b2--` и так далее. Возвращается позиция `b2 - (e1 - b1)`.

- ```
  template<class BidiIter>
  void reverse(BidiIter b, BidiIter e);
  ```

  Этот алгоритм обращает элементы от `b` до `e` «не сходя с места».

- ```
  template<class BidiIter, class OutputIter>
  OutputIter reverse_copy(BidiIter b1, BidiIter e1,
                          OutputIter b2);
  ```

Это — алгоритм копирования в обратном порядке элементов от b1 до e1. Копия размещается начиная с позиции b2. Копирование выполняется в обратном порядке: e1 в b2 и так далее. Возвращается позиция b2 + (e1 - b1).

- ```
  template<class ForwIter>
  ForwardIter unique(ForwIter b, ForwIter e);
  ```

  Соседние тождественные элементы в диапазоне от b до e уничтожаются (кроме первого). Возвращается позиция конца результирующего диапазона.

- ```
  template<class ForwIter, class BinaryPred>
  ForwardIter unique(ForwIter b, ForwIter e, BinaryPred bp);
  ```

  Соседние элементы в диапазоне от b до e, удовлетворяющие двоичному предикату bp, уничтожаются (кроме первого). Возвращается позиция конца результирующего диапазона.

- ```
  template<class InputIter, class OutputIter>
  OutputIter unique_copy(InputIter b1, InputIter e1,
                         OutputIter b2);

  template<class InputIter, class OutputIter, class BinaryPred>
  OutputIter unique_copy(InputIter b1, InputIter e1,
              OutputIter b2, BinaryPred bp);
  ```

  Аналогично предыдущему, но результаты копируются в b2, а исходный диапазон не изменяется.

Другие библиотечные функции описаны в следующей таблице.

STL: библиотечные функции, изменяющие последовательность

`swap(t1, t2)`	меняет местами t1 и t2
`iter_swap(b1, b2)`	меняет местами элементы, на которые направлены указатели
`swap_range(b1, e1, b2)`	меняет местами элементы в диапазоне от b1 до e1 с теми, которые хранятся, начиная с b2; возвращает b2 + (e1 - b1)
`transform(b1, e1, b2, op)`	с помощью унарного оператора op трансформирует последовательность от b1 до e1, помещает ее в b2, возвращает положение конца результирующей последовательности
`transform (b1, e1, b2, b3, bop)`	применет бинарный оператор bop к двум последовательностям, начинающимся в b1 и b2, для создания последовательности b3; возвращает положение конца результирующей последовательности
`replace(b, e, t1, t2)`	в диапазоне от b до e замещает значение t1 на t2
`replace_if(b, e, p, t2)`	в диапазоне от b до e замещает значения, удовлетворяющие предикату p, на t2

Продолжение на след. стр.
  ```

Продолжение

STL: библиотечные функции, изменяющие последовательность	
`replace_copy(b1, e1, b2, t1, t2)`	копирует диапазон от `b1` до `e1` с заменой `t1` на `t2`; результат размещается с адреса `b2`
`replace_copy_if(b1, e1, b2, p, t2)`	копирует диапазон от `b1` до `e1` с заменой элементов, удовлетворяющих предикату `p`, на `t2`; результат размещается с адреса `b2`
`remove (b, e, t)` `remove_if, remove_copy, remove_copy_if`	стирает элементы со значеним `t` подобны `replace`, но значения стираются, а не копируются
`fill(b, e, t)`	заполняет значением `t` диапазон от `b` до `e`
`fill_n(b, n, t)`	присваивает `n` значений `t`, начиная от `b`
`generate(b, e, gen)`	с помощью вызова генератора `gen` присваивает значения диапазону от `b` до `e`
`generate_n(b, n, gen)`	с помощью вызова генератора `gen` присваивает `n` значений начиная от `b`
`rotate(b, m, e)`	поворачивает влево элементы в диапазоне от `b` до `e`; элемент в позиции `i` попадает в позицию `(i+n-m)%n`, где `n` — это размер диапазона, `m` — средняя позиция, а `b` — первая позиция
`rotate_copy(b1, m, e1, b2)`	то же, что и предыдущее, но копируется в `b2`, при этом оригинал не изменяется
`random_shuffle(b, e)`	перемешивает элементы
`random_shuffle(b, e, rand)`	перемешивает элементы с помощью указанного генератора случайных чисел `rand`
`partition(b, e, p)`	Элементы в диапазоне от `b` до `e` упорядочиваются так, чтобы все элементы, удовлетворяющие условию `p`, помещались перед теми, которые не удовлетворяют этому условию
`stable_partition(b, e, p)`	то же, что и предыдущее, но сохраняется исходный относительный порядок одинаковых элементов

E.3.4. Численные алгоритмы

К численным алгоритмам относятся суммирование, скалярное произведение и смежная разность. В качестве примера см. раздел 9.10.4, «Численные алгоритмы», на стр. 272 (в файле *stl_numr.cpp*).

Вот библиотечные прототипы для численных алгоритмов.

- ```
  template<class InputIter, class T>
  T accumulate(InputIter b, InputIter e, T t);
  ```

  Это стандартный алгоритм накопления, причем за начальное значение суммы принимается `t`. К этой сумме последовательно добавляются элементы из диапазона от `b` до `e`.
  ```

- `template<class InputIter, class T, class BinOp>`
  `T accumulate(InputIter b, InputIter e, T t, BinOP bop);`

  Это алгоритм накопления, причем начальное значение равно `t`. К этому значению последовательно «прибавляются» элементы из диапазона от `b` до `e`, но вместо сложения применяется заданная бинарная операция `bop`.

В следующей таблице кратко перечислены остальные алгоритмы и их назначение.

STL: библиотечные численные функции

`iner_product(b1, e1, b2, t)`	возвращает скалярное произведение двух диапазонов, начинающихся с `b1` и `b2`, причем суммирование начинается со значения `t`, которое обычно устанавливается равным нулю
`iner_product (b1, e1, b2, t,bop1, bop2)`	возвращает обобщенное скалярное произведение, используя `bop1` для суммирования, а `bop2` для умножения
`partial_sum(b1, e1, b2)`	создает последовательность, начинающуюся в `b2`, которая включает частичные суммы членов из диапазона от `b1` до `e1`
`partial_sum(b1, e1, b2, bop)`	то же, что и предыдущее, но вместо суммирования используется `bop`
`adjacent_difference (b1, e1,b2)`	создает последовательность, начинающуюся в `b2`, которая является смежной разностью членов из диапазона от `b1` до `e1`
`adjacent_difference (b1, e1, b2, bop)`	то же, что и предыдущее, но вместо разности используется `bop`

E.4. Функции

Для дальнейшего освоения библиотеки STL полезны объекты-функции. Например, многие из рассматривавшихся ранее численных функций предполагают применение + или *, но также имеет форму, в которой в качестве аргумента может передаваться заданный пользователем бинарный оператор. Ряд объектов-функций можно найти в библиотеке *function*, но их можно создать и самостоятельно. Объекты-функции — это классы, в которых определен `operator()`. Они являются встроенными[1] и компилируются так, чтобы получить эффективный код. В качестве примера см. раздел 9.11, «Функции», на стр. 273 (в файле *stl_func.cpp*).

Существует три вида объектов-функций.

Виды объектов-функций

- Арифметические объекты
- Сравнивающие объекты
- Логические объекты

[1] Речь, разумеется, идет о встраивании функции в место вызова (inline), а не о том, что некая конструкция встроена в язык или в его реализацию (built-in). — *Примеч. перев.*

В следующей таблице кратко перечислены арифметические алгоритмы и их назначение.

STL: Арифметические объекты	
`template <class T> struct plus<T>`	складывает два операнда типа `T`
`template <class T> struct minus<T>`	вычитаетает два операнда типа `T`
`template <class T> struct times<T>`	умножает два операнда типа `T`
`template <class T> struct divides<T>`	делит два операнда типа `T`
`template <class T> struct modulus<T>`	делит по модулю два операнда типа `T`
`template <class T> struct negate<T>`	унарный минус для одного аргумента типа `T`

Арифметические объекты часто используются в численных алгоритмах, таких как `accumulate()`.

STL: Сравнивающие объекты	
`template <class T> struct equal_to<T>`	равенство двух операндов типа `T`
`template <class T> struct not_equal_to<T>`	неравенство двух операндов типа `T`
`template <class T> struct greater<T>`	сравнение двух операндов типа `T` с помощью отношения «больше» (`>`)
`template <class T> struct less<T>`	сравнение двух операндов типа `T` с помощью отношения «меньше» (`<`)
`template <class T> struct greater_equal <T>`	сравнение двух операндов типа `T` с помощью отношения «больше или равно» (`>=`)
`template <class T> struct less_equal<T>`	сравнение двух операндов типа `T` с помощью отношения «меньше или равно» (`<=`)

Сравнивающие объекты часто используются в алгоритмах сортировки, таких как `merge()`.

STL: Логические объекты	
`template <class T> struct logical_and<T>`	выполняет операцию логического И (`&&`) над двумя операндами типа `T`
`template <class T> struct logical_or<T>`	выполняет операцию логического ИЛИ (`||`) над двумя операндами типа `T`
`template <class T> struct logical_not<T>`	выполняет операцию логического отрицания (`!`) над одним аргументом типа `T`

E.4.1. Адаптеры функций

Адаптеры функций делают возможным создание объектов-функций с помощью адаптирования. В качестве примера см. раздел 9.12, «Адаптеры функций», на стр. 275 (в файле *stl_adap.cpp*).

Адаптеры функций

- отрицающий адаптер[1] — для отрицания предикативных объектов
- связывающий адаптер[2] — для связывания аргументов функции
- адаптеры для указателя на функцию[3]

В следующей таблице кратко перечислены адаптеры STL и их назначение.

STL: Адаптеры функций

`template<class Pred> unary_negate<Pred> not1(const Pred& p)`	возвращает `!p`, где `p` — унарный предикат
`template<class Pred> binary_negate<Pred> not2(const Pred& p)`	возвращает `!p`, где `p` — бинарный предикат
`template<class Op, class T> binder1st<Op> bind1st (const Op& op, const T& t)`	первый аргумент бинарной `op` привязывается к `t`; возвращается объект-функция
`template<class Op, class T> binder2nd<Op> bind2nd (const Op& op, const T& t)`	второй аргумент бинарной `op` привязывается к `t`; возвращается объект-функция
`template<class Arg, class T> ptr_fun(T (*f)(Arg1, Arg2))`	создает `pointer_to_unary_function<Arg,T>`
`template<class Arg1, class Arg2, class T> ptr_fun(T (*f)(Arg1, Arg2))`	создает `pointer_to_binary_function<Arg1,Arg2,T>`

E.5. Распределители памяти

Объекты-распределители памяти (allocators) управляют памятью для контейнеров. Они позволяют приспособить конкретную реализацию к условиям данной системы, сохраняя в то же время переносимость интерфейса контейнерного класса.

К определениям распределителей памяти относятся `value_type`, `reference`, `size_type`, `pointer` и `difference_type`.

Используем таблицу, чтобы кратко перечислить функции-члены распределителей памяти и их назначение.

STL: Функции-члены распределителей памяти

`allocator();`	конструктор и деструктор
`~allocator();`	для распределителей памяти
`pointer address(reference r);`	возвращает адрес `r`
`pointer allocate(size_type n);`	распределяет свободную память для `n` объектов размера `size_type`

[1] Иногда такие адаптеры называют *негаторами*. — *Примеч. перев.*

[2] Иногда такие адаптеры называют *связывателями*. — *Примеч. перев.*

[3] Страуструп выделяет также четвертый тип адаптеров — адаптер для функций-членов. — *Примеч. перев.*

STL: Функции-члены распределителей памяти	
`void deallocate(pointer p);`	освобождает память, связанную с `p`
`size_type max_size();`	возвращает наибольшее значение для `difference_type`, то есть наибольшее число элементов, которые можно разместить в контейнере

Выясните, нет ли в вашем компиляторе каких-либо зависящих от системы особенностей реализации.

E.6. Строковая библиотека

C++ предоставляет строковый тип в стандартном заголовочном файле *string*. Он является инстанцированием шаблонного класса `basic_string<T>` типом `char`. Строковый тип предлагает функции-члены и операторы, выполняющие манипуляции со строками, например, конкатенацию, присваивание или замещение. Вот пример программы, использующей строковый тип для простых строковых операций.

В файле stringt.cpp

```
//Переписывание предложения с помощью строкового класса

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

int main()
{
   string sentence, words[10];            //предложение, слова
   int pos = 0, old_pos = 0, nwords, i = 0;

   sentence = "Эскимосы знают 23 слова для ";
   sentence += "описания снега";

   while (pos < sentence.size()) {
        pos = sentence.find(' ', old_pos);
        words[i++].assign(sentence, old_pos, pos - old_pos);
        cout << words[i - 1] << endl;         //печать слов
        old_pos = pos + 1;
   }
   nwords = i;
   sentence = "Программисты на C++ ";
   for (i = 1; i < nwords - 1; ++i)
        sentence += words[i] + ' ';
   sentence += "Windows";
   cout << sentence << endl;
}
```

Строковый тип используется для захвата каждого слова из начального предложения, в котором слова разделяются символом пробела. Позиция пробелов вычисляется с

помощью функции-члена `find()`. Затем используется функция-член `assign()` для выбора подстроки из предложения `sentence`. Наконец, создается новое предложение `sentence` при помощи перегруженных операторов присваивания `operator+=()` и `operator+()`, выполняющих «накапливающее» присваивание и конкатенацию соответственно.

Рассмотрим представление строки символов.

Закрытые члены данных класса string

`char* ptr`	для указания на начальный символ
`size_t len`	для длины строки
`size_t res`	для распределенного в настоящий момент размера, или максимальный размер для не размещенной строки

Отметим, что обычным является также наличие инстанцирования `basic_string<wchar_t>` для широкого строкового типа `wstring`. Возможны и другие инстанцирования.

E.6.1. Конструкторы

Строки имеют семь открытых конструкторов, позволяющих легко объявлять и инициализовать строки с широким набором параметров. Два из них обеспечивают преобразование, а именно, конструктор с `char*` и конструктор с `vector<char>*`.

Конструкторы класса string

`string()`	конструктор по умолчанию, создает пустую строку
`string(const char* p)`	преобразующий конструктор (от указателя на `char`)
`string(const vect<char>* v)`	преобразующий конструктор (от векторного контейнера)
`string(const string& str, size_t pos = 0, size_t n = npos)`	копирующий конструктор; `npos` обычно равен `-1` и указывает, что память не была выделена
`string(size_t size, capacity cap)`	создается строка, содержащая `'\0'`, размера `size`, где `capacity` является перечислением, один из элементов которого — `default_size`; в противном случае создается строка нулевой длины
`string(const char* p, size_t n)`	копирует `n` символов, `p` является базовым адресом
`string(char c, size_t n = 1)`	создает строку из `n` символов `c`

Эти конструкторы позволяют довольно легко применять строковый тип, инициализованный указателями `char*`, что было традиционным методом C при работе со строками. Кроме того, поскольку многие операции удобно выполнять над вектором символов, интерфейс `string` обеспечивает соответствующий вариант инициализации.

E.6.2. Функции-члены

Строки содержат некоторые члены, которые перегружают операторы, что описано в следующей таблице.

Члены string, перегружающие операторы

`string& operator=(const string& s)`	оператор присваивания
`string& operator=(const char* p)`	присваивает `char*` строке
`string& operator=(const char c)`	присваивает `char` с строке
`string& operator+=(const string& s)`	добавляет строку `s`
`string& operator+=(const char* p)`	добавляет `char*` к строке
`string& operator+=(const char c)`	добавляет `char` с к строке
`operator vector<char>()const`	преобразует строку в вектор
`char operator[](size_t pos) const`	возвращает символ из позиции `pos`
`char& operator[](size_t pos)`	возвращает ссылку на символ из позиции `pos`

Существует обширный набор открытых функций-членов, которые позволяют манипулировать строками. Во многих случаях они перегружены для работы со `string`, `char*` и `char`. Начнем с описания функции `append()`.

- ```
  string& append(const string& s, size_t pos = 0,
       size_t n = npos);
  ```

  Добавляет `n` символов начиная от позиции `pos` из `s` в неявный строковый объект.

  ```
  //например, s1 "мне " s2 "исполнилось 7 лет "
  s1.append(s2);          //s1 "мне исполнилось 7 лет "
  //или
  s2.append(s1, 0, 4);    //s2 "исполнилось 7 лет мне"
  ```

- ```
  string& append(const char* p, size_t n);
  string& append(const char* p);
  string& append(char c, size_t n = 1);
  ```

  В каждом случае создается объект типа `string` с помощью конструктора той же сигнатуры и добавляется к неявному объекту типа `string`.

- ```
  string& assign(const string& s, size_t pos = 0,
                                  size_t n = npos);
  ```

  Начиная от позиции `pos` в `s` неявному строковому объекту присваивается `n` символов.

  ```
  //Например, s1 "мне " s2 "исполнилось 7 лет "
  s1.assign(s2);   //s1 "исполнилось 7 лет "
  ```

  Следующие сигнатуры с ожидаемой семантикой являются перегруженными:

  ```
  string& assign(const char* p, size_t n);
  string& assign(const char* p);
  string& assign(char c, size_t n = 1);
  ```

- ```
  string& insert(size_t pos1, const string& str, size_t pos2 = 0
           size_t n = npos);
  ```

Функция `insert()` представляет собой перегруженный набор определений, которые вставляют строку символов в указанную позицию. Она вставляет `n` символов, полученных из `str`, начиная с позиции `pos2`, в неявную строку с позиции `pos1`.

```
//например, s1 "Мне уже" s2 "исполнилось 7 лет "
s1.insert(4, s2);  //s1 "Мне исполнилось 7 лет уже"
```

Следующие сигнатуры с ожидаемой семантикой также являются перегруженными:

```
string& insert(size_t pos, const char* p, size_t n);
string& insert(size_t pos, const char* p);
string& insert(size_t pos, char c, size_t n = 1);
```

Для выполнения обратных действий служит функция `remove()`.

- `string& remove(size_t pos = 0, size_t n = npos);`

  Удаляются `n` символов из неявной строки начиная с позиции `pos`.

В следующей таблице кратко описаны остальные открытые функции-члены.

Открытые функции-члены string

`string& replace(pos1, n1, str, pos2 = 0, n2 = npos);`	в неявной строке начиная с позиции `pos1` заменяет `n1` символов `n2` символами из подстроки `str` с позиции `pos2`

`string& replace(pos, n, p, n2);`
`string& replace(pos, n, p);`
`string& replace(pos, n, c, rep = 1);`
	заменяет `n` символов в позиции `pos` используя `char* p` из `n2` символов, или `char* p` до завершающего нуля, или повторяя символ `c` `rep` раз

`char get_at(pos) const;`	возвращает символ из позиции `pos`
`void put_at(pos, c);`	помещает символ `c` в позицию `pos`
`size_t length() const;`	возвращает длину строки
`const char* c_str() const;`	преобразует строку в традиционное `char*` представление
`const char* data() const;`	возвращает базовый адрес строкового представления
`void resize(n, c);` `void resize(n);`	изменяет строку, делая ее длину равной `n`; в первой функции в качестве заполняющего символа выступает `c`, а во второй — символ `eos()` (end-of-string, конец строки)
`void reserve(size_t res_arg);` `size_t reserve() const;`	выделяет память под строку; первая функция переустанавливает `this`; вторая возвращает закрытый член `res` — размер выделенного фрагмента;
`size_t copy(p, n, pos = 0) const;`	`n` симолов неявной строки, начиная с позиции `pos`, копируются в `char* p`
`string substr(pos = 0, n = pos) const;`	возвращается подстрока из `n` символов неявной строки

Вы можете лексикографически сравнить две строки, используя семейство перегруженных функций-членов `compare()`.

- `int compare(const string& str, size_t pos = 0, size_t n = npos) const;`

  Эта функция сравнивает `n` символов неявной строки, начиная с позиции `pos`, со строкой `str`. Возвращается ноль, если строки равны; в противном случае возвращается положительное или отрицательное целое значение, показывающее, что неявная строка лексикографически больше или меньше чем строка `str`. Следующие сигнатуры с ожидаемой семантикой также являются перегруженными:

```
int compare(const char* p, size_t pos, size_t n) const;
int compare(const char* p, size_t pos) const;
int compare(char c, size_t pos, size_t rep = 1) const;
```

  В каждой сигнатуре указывается, как создается явная строка и как она затем сравнивается с неявной строкой.

Последний набор функций-членов выполняет операцию поиска. Обсудим одну их группу, а затем сведем в таблицу оставшиеся.

- `size_t find(const string& str, size_t pos = 0) const;`

  В неявной строке начиная с позиции `pos` производится поиск строки `str`. Если она найдена, возвращается позиция, в которой она начинается; в противном случае возвращается позиция `npos`, что означает неудачу.

  Следующие сигнатуры с ожидаемой семантикой также являются перегруженными:

```
size_t find(const char* p, size_t pos, size_t n) const;
size_t find(const char* p, size_t pos = 0) const;
size_t find(char c, size_t pos = 0) const;
```

  В каждой сигнатуре указывается, как создается явная строка и как в ней затем ищется неявная строка. Остальные функции для поиска строк и символов кратко описаны в следующей таблице.

Поисковые функции-члены string

`size_t rfind(str, pos = npos) const;`
`size_t rfind(p, pos, n) const;`
`size_t rfind(p, pos = npos) const;`
`size_t rfind(c, pos = npos) const;`
	похоже на `find()`, но при поиске первого совпадения строка сканируется в обратном направлении

`size_t find_first_of (str, pos = 0) const;`
`size_t find_first_of (p, pos, n) const;`
`size_t find_first_of (p, pos = 0) const;`
`size_t find_first_of (c, pos = 0) const;`
	производится поиск первого вхождения любого из символов в указанном образце, будь то `str`, `char*` `p` или `char c`

`size_t find_last_of (str, pos = npos) const;` `size_t find_last_of (p, pos, n) const;`	аналогично, но в обратном направлении

`size_t find_last_of (p, pos = npos) const;` `size_t find_last_of (c, pos = npos) const;`	

`size_t find_first_not_of (str, pos = 0) const;`
`size_t find_first_not_of (p, pos, n) const;`
`size_t find_first_not_of (p, pos = 0) const;`
`size_t find_first_not_of (c, pos = 0) const;`
	производится поиск первого символа, который не соответствует ни одному из символов в указанном образце, будь то `str`, `char*` `p` или `char c`

`size_t find_last_not_of (str, pos = npos) const;`
`size_t find_last_not_of (p, pos, n) const;`
`size_t find_last_not_of (p, pos = npos) const;`
`size_t find_last_not_of (c, pos = npos) const;`
	аналогично, но в обратном направлении

E.6.3. Глобальные операторы

В строковом пакете имеются перегруженные варианты операторов, которые предоставляют операции ввода-вывода, конкатенации и сравнения. Эти операторы интуитивно понятны; они кратко описаны в следующей таблице.

Глобальные операторы, перегруженные в string

`ostream& operator<<(ostream& o, const string& s);`	оператор вывода
`istream& operator>>(istream& in, string& s);`	оператор ввода
`string oparator+(const string& s1, const string& s2);`	конкатенация строк `s1` и `s2`
`bool operator==(const string& s1, const string& s2);`	истина, если строки `s1` и `s2` лексикографически равны
`<` `<=` `>` `>=` `!=`	как ожидается

Операторы сравнения и оператор конкатенации `operator+()` также перегружены с четырьмя следующими сигнатурами:

```
bool operator==(const char* p, const string& s);
bool operator==(char c, const string& s);
bool operator==(const string& s, const char* p);
bool operator==(const string& s, char c);
```

В результате может выполняться сравнение или конкатенация в любых вариантах, то есть строки и второго аргумента, которым может быть либо строка, либо символ, либо указатель на символ.

Литература

Boehm, H. and Weiser, M., «Garbage Collection in an Uncooperative Environment», *Software — Practice and Experience.* September 1988. pp. 807-820.

Booch, G., *Object-Oriented Analysis and Design, Second Edition.* 1995. Reading, MA: Addison-Wesley.

Timothy Budd, *An Introduction to Object-Oriented Programming.* 1991. Reading, MA: Addison-Wesley.

Edelson, D., «A Mark and Sweep Collector for C++», In *Proceedings of Principles of Programming Languages.* January 1992.

Edelson, D. and Pohl, I., «A Copying Collector for C++», In *Usenix C++ Conference Proceedings.* 1991. pp. 85-102.

Ellis, M. and Stroustrup, B., *The Annotated C++ Reference Manual.* 1990. Reading, MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software.* 1995. Reading, MA: Addison-Wesley.

Glass, G. and Schuchert, B., *The STL <Primer>.* 1996. Upper Saddle River, NJ: Prentice Hall.

Kelley, A. and Pohl, I., *A Book on C, Third Edition.* 1995. Reading, MA: Addison-Wesley.

Kernighan, B. and Ritchie, D., *The C Programming Language, Second Edition.* 1988. Englewood Cliffs, NJ: Prentice Hall.

Kernighan, B. and Plauger, P., *The Elements of Programming Style.* 1974. New York, NY: McGraw-Hill.

Linton, M., Vlissides, J., and Calder, P., «Composing user interfaces with Interviews», IEEE *Computer* v. 22, no. 2, pp. 8-22, 1989.

Lippman, S., Lajoio J., *The C++ Primer, Third Edition.* 1998. Reading, MA: Addison-Wesley.

Meyers, S., *Effective C++: 50 Specific Ways to Improve your Programs and Designs*. 1992. Reading, MA: Addison-Wesley.

Musser, D. and Saini, A., *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. 1996. Reading, MA: Addison-Wesley.

Pohl, I. and Edelson, D., «A-Z: C Language Shortcomings», *Computer Languages*, vol. 13, no. 2. 1988. pp. 51-64.

Pohl, I., *C++ for C Programmers, Second Edition*. 1994. Reading, MA: Addison-Wesley.

Pohl, I., *C++ for Pascal Programmers, Second Edition*. 1995. Reading, MA: Addison-Wesley.

Stroustrup, B., *The Design and Evolution of C++*. 1994. Reading, MA: Addison-Wesley.

Stroustrup, B., *The C++ Programming Language, Second Edition*. 1991. Reading, MA: Addison-Wesley.

Taligent Inc., *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*. 1994. Reading, MA: Addison-Wesley.

Teale, S., *C++ IO Streams Handbook*. 1993. Reading, MA: Addison-Wesley.

Wasserman, A., Pircher, P., and Muller, R., «The Object-oriented Structured Design Notation for Software Design Representation», IEEE *Computer* v. 23, no. 3, 1990. pp. 50-63.

Русские переводы

Т. Бадд. *Объектно-ориентированное программирование в действии*. СПб. «Питер». 1997.

Г. Буч. *Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е издание*. СПб. «Невский Диалект». 1998.

Б. Керниган, Д. М. Ричи. *Язык программирования Си*. М. «Финансы и статистика». 1985.

Б. Страуструп. *Язык программирования C++, 3-е издание*. СПб. «Невский Диалект». 1999.

М. А. Эллис, Б. Страуструп. *Справочное руководство по языку C++ с комментариями: проект стандарта ANSI*. М. «Мир». 1992.

Предметный указатель

Символы

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

A

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ч

Ш

Э

Я