А.О. Семкин, А.С. Перин

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ. ЯЗЫКИ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Министерство науки и высшего образования Российской Федерации
Томский государственный университет
систем управления и радиоэлектроники

А.О. Семкин, А.С. Перин

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ. ЯЗЫКИ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Томск
Издательство ТУСУРа
2021

УДК 004.4(075.8)
ББК 32.973.2-018я73
С307

Рецензенты:

Коханенко А.П., д-р физ.-мат. наук, проф.;
Запасной А.С., канд. физ.-мат. наук, доц.

Печатается по решению научно-методического совета ТУСУРа
(протокол № 5 от 24.06.2021 г.)

Семкин, Артем Олегович

С307 Информационные технологии. Языки и системы программирования : учеб. пособие / А.О. Семкин, А.С. Перин. – Томск : Изд-во Томск. гос. ун-та систем упр. и радиоэлектроники, 2021. – 180 с.

ISBN 978-5-86889-930-0

Раскрыты базовые принципы и даются практические советы, необходимые для программирования приложений с графическим интерфейсом при помощи средств разработки *Qt*. Каждый раздел пособия включает в себя подробное рассмотрение множества вопросов, относящихся к языку программирования *C++* и среды разработки *Qt*. Текст дополняется набором примеров. Знания материала этой части вполне достаточно для создания работоспособных приложений с графическим интерфейсом.

Для студентов всех форм обучения технических направлений подготовки и специальностей.

УДК 004.4(075.8)
ББК 32.973.2-018я73

ISBN 978-5-86889-930-0

© Семкин А.О., Перин А.С., 2021
© Томск. гос. ун-т систем упр.
и радиоэлектроники, 2021

Оглавление

Предисловие

Данное учебное пособие является частью учебно-методического комплекса и предназначено для подготовки студентов к лекционным и практическим занятиям по дисциплине «Информационные технологии».

Цель преподавания дисциплины — обеспечить базовую подготовку студентов в области использования средств вычислительной техники, а также развить навыки работы на персональных компьютерах для решения инженерных задач, сбора, передачи, обработки и хранения информации. Лекционный курс по «Информационным технологиям» знакомит студентов с назначением и принципом действия современных персональных компьютеров, основами алгоритмизации и технологиями программирования научно-технических задач, языками программирования высокого уровня, технологиями обработки и отладки программ, современным прикладным программным обеспечением, методами решения типовых инженерных задач и их программной реализацией.

Процесс изучения дисциплины направлен на формирование следующих компетенций:

– способности понимать сущность и значение информациив развитии современного информационного общества, сознавать опасности и угрозы, возникающие в этом процессе, соблюдать основные требования информационной безопасности;

– способности владеть основными методами, способами и средствами получения, хранения, переработки информации;

– способности сформировать навыки самостоятельной работы на компьютере и в компьютерных сетях, осуществлять компьютерное моделирование устройств, систем и процессов с использованием универсальных пакетов прикладных компьютерных программ.

Список литературы включает источники, рекомендуемые для самостоятельного и более углубленного изучения вопросов, выносимых на практические занятия и лабораторные работы.

Введение

Язык *C++*, так же как и язык *C*, является детищем компании AT&T Bell Laboratories. Бьярни Страуструп (Bjarne Stroustrup) разработал этот язык в начале 80-х годов XX в. на основе языка *C*, так как язык *C* был кратким, хорошо подходил для системного программирования, широко доступен и тесно связан с операционной системой UNIX. Объектно-ориентированная часть языка *C++* возникла под влиянием языка моделирования Simula67. Страуструп добавил в язык *C* элементы объектно-ориентированного программирования (ООП), не изменяя при этом существенно сам язык *C*. Таким образом, язык *C++* является расширением языка *C*, а это означает, что любая корректно составленная программа *C* является также корректно составленной программой *C++*. Имеются лишь некоторые незначительные различия, но они не столь существенны. Программы *C++* могут использовать существующие библиотеки языка *C*. Библиотеки — это совокупности программных модулей, которые вызываются из программ. Они предоставляют готовые решения различных широко распространенных задач программирования, экономя таким образом много времени и усилий. Это помогло распространению языка *C++*. Название *C++* происходит от обозначения оператора инкремента ++ в языке *C*, который добавляет единицу к значению переменной. Название *C++* подразумевает, что этот язык является усовершенствованной (++) версией языка *C*. Когда язык *C++* получил некоторое признание, Страуструп добавил в него шаблоны, обеспечивая тем самым возможность обобщенного программирования. И только после того как шаблоны были использованы на практике и усовершенствованы, он стал понимать, что они имеют такое же значение, как и ООП, или даже большее. Тот факт, что язык *C++* включает в себя как ООП, так и обобщенное программирование, показывает, что в *C++* упор делается на утилитарный, а не идеологический подход, и это одна из причин успеха этого языка [1, 2].

Основная цель этого пособия — раскрыть базовые принципы и дать практические советы, необходимые для программирования приложений с графическим интерфейсом при помощи средств

разработки *Qt* [1] с использованием языка программирования *C++* [2]. Каждый раздел пособия включает в себя подробное рассмотрение множества вопросов, относящихся к языку программирования C++ и среде разработки *Qt*, глубокое понимание которых необходимо специалистам-разработчикам технических направлений и специальностей.

1 НАЧАЛЬНЫЕ СВЕДЕНИЯ О ЯЗЫКЕ *C++*

Истоки языка *C++*: немного истории

В начале семидесятых годов прошлого столетия Деннис Ритчи (*Dennis Ritchie*), сотрудник компании *Bell Laboratories*, участвовал в проекте по разработке операционной системы (ОС) *Unix*. В своей работе Ритчи нуждался в языке программирования, который был бы лаконичным, с помощью которого можно было бы создавать компактные и быстро выполняющиеся программы и посредством которого можно было бы эффективно управлять аппаратными средствами.

По сложившейся на то время традиции программисты использовали для решения этих задач язык ассемблера, тесно связанный с внутренним машинным языком. Однако язык ассемблера является языком низкого уровня, т.е. он работает непосредственно с оборудованием (например, напрямую обращается к регистрам центрального процессора и ячейкам памяти). Таким образом, язык ассемблера является специфическим для каждого процессора компьютера. Поэтому если вы хотите, чтобы ассемблерная программа, написанная для компьютера одного типа, могла работать на компьютере другого типа, то, возможно, вам придется полностью переписать программу на другом языке ассемблера.

Однако ОС *Unix* предназначалась для работы на самых разнообразных типах компьютеров (или платформ). Таким образом, предполагалось использование языка программирования высокого уровня. Такой язык ориентирован на решение задач, а не на обслуживание определенного оборудования. Специальные программы, называемые компиляторами, переводят язык программирования высокого уровня на внутренний язык определенного компьютера. Следоваиельно, одну и ту же программу, написанную на языке программирования высокого уровня, можно запускать на различных платформах, применяя разные компиляторы. Ритчи необходим был язык, который сочетал бы в себе эффективность языка низкого уровня и возможность доступа к аппаратным средствам с универсальностью и переносимостью языка высокого уровня. Поэтому, основываясь на старых языках программирования, он создал язык *C*.

Название языка *C++* происходит от операции инкремента (++) в языке *C*, которая увеличивает на единицу значение переменной. Таким образом, имя *C++* в точности отражает расширенную версию языка *C*. Компьютерная программа переводит практическую задачу в последовательность действий, которые должен выполнить компьютер.

Поскольку *C++* прививает языку *C* новые принципы программирования, мы должны сначала изучить философию программирования на языке *C*.

В большинстве старых программ было настолько много запутанных переходов, что при прочтении программу практически невозможно было понять и попытка модифицировать такую программу сулила одни неприятности. Чтобы выйти из ситуации такого рода, специалисты по вычислительной технике разработали более дисциплинированный стиль программирования, называемый структурным программированием.

Язык программирования *C* обладает всеми возможностями для реализации этого подхода. Например, структурное программирование ограничивает ветвление (выбор следующей инструкции для выполнения) небольшим набором удобных и гибких конструкций. В языке *C* эти конструкции (циклы *for,while*, *do while* и оператор *if else*) включены в его словарь.

Другим новшеством было так называемое нисходящее проектирование. В языке *C* эта идея состояла в том, чтобы разделить большую программу на небольшие, поддающиеся управлению задачи. Если после разбиения одна из задач все равно остается крупной, этот процесс продолжается до тех пор, пока программа не будет разделена на небольшие модули, с которыми будет просто работать. Этот подход вполне осуществим в языке *C*, т.к. он позволяет разрабатывать программные модули, называемые функциями, которые отвечают за выполнение конкретной задачи. Как вы могли заметить, в структурном программировании отображено процедурное программирование, в том смысле, что программа представляется в виде действий, которые она должна выполнить.

Комментарии в языке *C++*

В *C++* комментарий обозначается двумя косыми чертами (//). Комментарий — это примечание, написанное программистом для пользователя программы, которое обычно идентифицирует ее раздел или содержит пояснения к определенному коду. Компилятор игнорирует комментарии. Хотя он знает *C++* не хуже вас, понимать комментарии он не умеет. Поэтому для него листинг будет выглядеть следующим образом:

```
#include<iostream>
int main()
{
using namespace std;
cout<< "Come up and C++ me some time.";
cout<<endl;
cout<< "You won't regret it!" « endl;
return 0;
}
```

Комментарий в *C++* начинается с символов «//» и простирается до конца строки. Комментарий может занимать одну строку целиком, а может находиться в строке вместе с кодом.

На рисунке 1.1 представлен листинг, где для символьного вывода применяется объект *cout*. Исходный код содержит для читателя строки комментариев, которые отмечаются парой символов «//»; эти символы компилятор игнорирует. Язык программирования *C++* чувствителен к регистру символов; это означает, что символы в верхнем и нижнем регистре считаются разными. Поэтому должен использоваться в точности тот же регистр, что применяется в примерах. Например, в приведенной далее программе используется *cout*, поэтому если вы введете *Cout* или *COUT*, компилятор отклонит это и сообщит о наличии неизвестных идентификаторов.

Программы на *C++* конструируются из строительных блоков, называемых функциями. Обычно программа систематизируется в набор главных задач, для обработки которых проектируются отдельные функции.

```
// программа myfirst.cpp - отображает на экране сообщение

#include <iostream>                              // директива препроцессора
using namespace std;                             // включает в программу определения
int main()                                       // заголовок функции
{                                                // начало тела функции
    cout << "Come up and C++ me some time.";     // сообщение
    cout << "\n";                                // начать новую строку
    return 0;                                    // завершить функцию main
}                                                // конец тела функции
```

Рисунок 1.1 – Листинг программы *myfirst.cpp*

Пример *myfirst.cpp* содержит следующие элементы:

– комментарии, обозначаемые с помощью «//»;

– директиву препроцессора *#include***;**

– заголовок функции: *int main*();

– директиву *using namespace***;**

– тело функции, ограниченное фигурными скобками { и };

– операторы, которые используют объект *C++ cout* для отображения сообщения;

– оператор возврата для прекращения выполнения функции *main()*.

Функция *main* (). Пример программы из рисунка 1.1 имеет следующую фундаментальную структуру:

intmain()
}
Операторы
*Return*0;
}

В этих строках говорится о том, что существует функция по имени *main*(), и они описывают ее поведение. Вместе эти строки образуют определение функции. Это определение состоит из двух частей: первой строки *int main*(), которая называется заголовком функции, и частью, заключенной в скобки, которая называется телом функции. На рисунке 1.2 приведено графическое представление функции *main* (). В заголовке функции вкратце описан ее интерфейс с остальной частью программы, а в теле функции содержатся компьютерные инструкции о том, что функция должна делать. В языке *C++* каждая полная инструкция называется оператором. Каждый оператор

должен завершаться точкой с запятой, поэтому не забывайте ставить ее при наборе примеров.

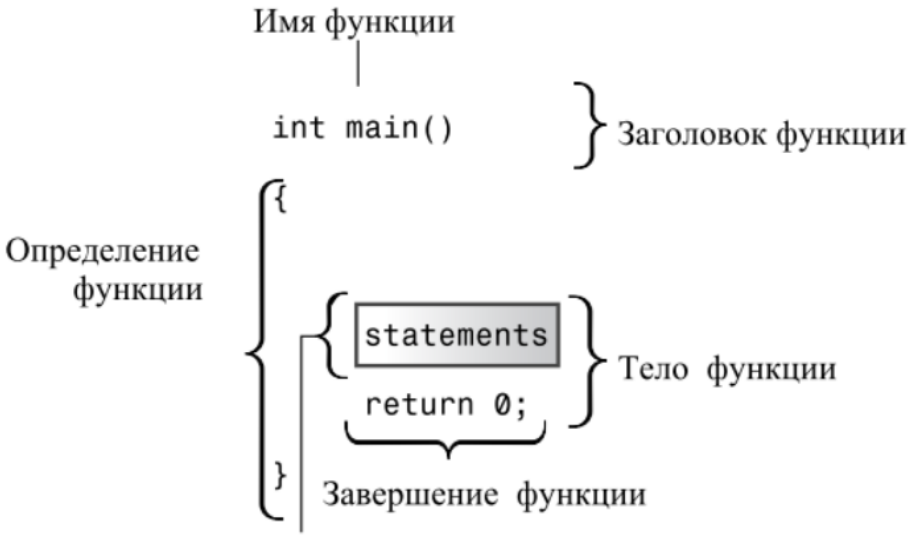

Рисунок 1.2 — Функция *main* ()

Финальный оператор функции *main* () называется оператором возврата. Его назначение — завершить выполнение функции.

Вывод в *C*++ с помощью *cout.* Для вывода сообщения на экран в программе *my first .cpp* используется следующий оператор *C*++:

cout « "*Come up and C++ me some time.*"»

Часть, заключенная в двойные кавычки, — это сообщение, которое необходимо вывести на экран. В *C*++ любая последовательность символов, заключенных в двойные кавычки, называется символьной строкой, по-видимому, из-за того, что она состоит из множества символов, собранных в одну большую конструкцию. Запись («) означает, что оператор отправляет строку в *cout*; символы указывают на направление передачи информации. А что такое *cout* — это предопределенный объект, который знает о том, как отображать разнообразные элементы, включая строки, цифры и индивидуальные символы.

Объект *cout* имеет простой интерфейс. Если *string* представляет строку, то для ее вывода на экран можно сделать следующее:

cout <<*string*;

Вот и все, что требуется для отображения строки на экране.

В этом представлении вывод рассматривается как поток, т.е. последовательность символов, передаваемых из программы. Этот поток отображает объект *cout,* свойства которого определены в файле *iostream*. Свойства объекта *cout* включают операцию вставки («), которая добавляет в поток данные, указанные в правой части. Рассмотрим следующий оператор (обратите внимание на завершающую точку с запятой):

cout « "*Come up and C++ me some time.*";

В выходной поток будет помещена строка "*Come up and C++ me some time*. ". Таким образом, вы можете сказать, что ваша программа не выводит на экран сообщение, а вставляет строку в поток вывода. В чем-то это звучит более выразительно (рисунок 1.3).

Рисунок 1.3 — Использование *cout* для отображения строки

Манипулятор *endl.* Рассмотрим еще одну строку:

cout<<*endl*;

Здесь *endl* — это специальное обозначение в *C++*, которое представляет важное понятие начала новой строки. Вставка *endl* в поток вывода означает, что курсор на экране будет перемещен на начало следующей строки. Специальные обозначения наподобие *endl*, которые имеют определенное значение для *cout,* называются манипуляторами. Как и *cout*, манипулятор *endl* определен в заголовочном файле *iostream*и является частью пространства имен *std*.

Символ новой строки. Обозначить новую строку в выводе в *C++* можно и старым способом — посредством символов «*n*» — это обозначение начала новой строки, принятое в языке *C*:

cout << "*What's next*?*n*";

Комбинация символов «\n» рассматривается как один символ, называемый символом новой строки. При отображении строки использование «\n» сокращает количество печатаемых символов по сравнению с манипулятором *endl*:

cout << "*Jupiter is a large planet*.\n"; // отображает текст,
// переходит на следующую строку;
cout<< "*Jupiter is a large planet*." <<*endl*; // отображает текст,
// переходит на следующую строку.

Операторы в языке *C++*

В *C++* имеется несколько разновидностей операторов, рассмотрим некоторые из них. На рисунке 1.4 можно увидеть операторы двух новых видов. Первый из них, оператор объявления, создает переменную. Второй, оператор присваивания, присваивает этой переменной определенное значение. Кроме этого, в программе продемонстрирована новая возможность объекта *cout*.

```
// fleas.cpp - отображает на экране
// значение переменной
#include <iostream>
using namespace std;
int main()
{
    int fleas;              // создает целочисленную переменную
    fleas = 3;             // присваивает значение этой переменной
    cout << "My cat has ";
    cout << fleas;          // отображает на экране значение переменной
    cout << " fleas.\n";
    return 0;
}
```

Рисунок 1.4 — Листинг программы с операторами обновления и присваивания

Раздел объявления отделен от остальной части программы пустой строкой. Этот прием часто можно было встретить в программах на *C*, но в программах на *C++* это редкость. В результате выполнения программы получается следующий результат:

My cat has 3 fleas.

Операторы объявления и переменные. Для того чтобы сохранить элемент информации в компьютере, вы должны идентифицировать как ячейку памяти, так и объем памяти, требуемый для хранения

этой информации. В языке *C++* проще всего это можно сделать с помощью оператора объявления, который идентифицирует тип памяти и предоставляет метку для ячейки. Например, в программе, представленной на рисунке 1.4, имеется следующий оператор объявления (обратите внимание на наличие точки с запятой):

int fleas;

Этот оператор предоставляет два вида информации: необходимый тип хранения и метку, привязанную к этой ячейке. В частности, оператор объявляет, что программа требует объема памяти, достаточного для хранения целого числа, для которого в *C++* используется метка *int*. Еще одной выполненной задачей является именование ячейки памяти. В этом случае оператор объявления говорит, что с этого момента программа будет использовать имя *fleas* для обозначения значения, хранящегося в указанной ячейке памяти, *fleas* — это переменная, поскольку ее значение можно изменять.

В общем случае, объявление указывает тип сохраняемых данных и имя программы, которая будет использовать данные, хранящиеся в этой переменной. В этой конкретной ситуации программа создает переменную по имени *carrots*, в которой хранится целое число (рисунок 1.5).

Оператор объявления в программе называется оператором определяющего объявления, или кратко — определением. Его присутствие означает, что компилятор выделит пространство памяти для хранения значений переменной. В более сложных ситуациях можно применять ссылочное объявление.

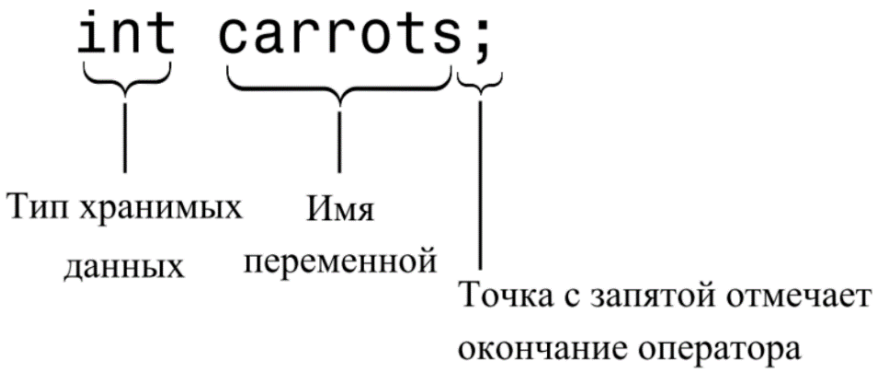

Рисунок 1.5 — Объявление переменной

Операторы присваивания. Оператор присваивания присваивает значение ячейке памяти. Например, следующий оператор присваивает целое значение «3» ячейке памяти, обозначаемой переменной *fleas*:

fleas = 3;

Символ «=» называется операцией присваивания. Одной из необычных особенностей языка *C++* (как и *C*) является то, что вы можете использовать операцию присваивания несколько раз подряд, т. е. следующий код программы будет действительным:

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

Операция присваивания выполняется поочередно, справа налево. Сначала значение 88 присваивается переменной *steinway*, затем это же значение присваивается переменной *baldwin* и, наконец, переменной *yamaha*.

Базовые типы данных

Сущность объектно-ориентированного программирования заключается в проектировании и расширении собственных типов данных. Встроенные типы данных *C++* разделены на две группы: фундаментальные типы и составные типы.

Имена, назначаемые переменным. В *C++* необходимо придерживаться следующих простых правил именования:

– в именах разрешено использовать только алфавитные символы, цифры и символы подчеркивания (_);

– первым символом имени не должна быть цифра;

– символы в верхнем и нижнем регистре рассматриваются как разные;

– в качестве имени нельзя использовать ключевое слово *C++*;

– имена, которые начинаются с двух символов подчеркивания или с одного подчеркивания и следующей за ним буквы в верхнем регистре, зарезервированы для использования реализациями *C++*, т. е. с ними имеют дело компиляторы и ресурсы. Имена,

начинающиеся с одного символа подчеркивания, зарезервированы для применения в качестве глобальных идентификаторов в реализациях;

– на длину имени не накладывается никаких ограничений, и все символы в имени являются значащими. Однако некоторые платформы могут вводить свои ограничения на длину.

Целочисленные типы *short, int, long*

Целыми являются числа без дробной части, например 2, 98, 5286 и 0. Память компьютера состоит из единиц, называемых битами. Используя различное количество битов для хранения значений, типы *short, int, long* в *C++* могут представлять до четырех различных целочисленных ширин.

В *C++* предлагается гибкий стандарт, позаимствованный у *C*, с некоторыми гарантированными минимальными размерами типов:

– целочисленный тип *short* имеет ширину не менее 16 битов;

– целочисленный тип *int* как минимум такой же, как *short*;

– целочисленный тип *long* имеет ширину не менее 32 битов и как минимум такой же, как *int*;

– целочисленный тип *long long* имеет ширину не менее 64 битов и как минимум такой же, как *long*.

В настоящее время во многих системах используется минимальная гарантия: тип *short* имеет 16 битов, а тип *long* — 32 бита. Это по-прежнему оставляет несколько вариантов для типа *int.* Он может иметь ширину в 16, 24 или 32 бита и соответствовать стандарту. Он может быть даже 64 бита, предоставляя минимальную ширину *long* и *long long.* Тип *int* обычно имеет 16 битов (столько же, сколько и *short*) в старых реализациях для *IBM* совместимых ПК и 32 бита (столько же, сколько и *long*) для *Windows XP*, *Windows Vista, Windows* 7, *Mac OS X*, *VAX* и многих других реализаций для миникомпьютеров.

Тип *char*: символы и короткие целые числа. Тип *char*, исходя из своего названия (сокращение от *character* — символ), предназначен для хранения символов, таких как буквы и цифры. Хранение чисел в памяти компьютера не представляет сложности, тогда как хра-

нение букв связано с рядом проблем. В языках программирования принят простой подход, при котором для букв используются числовые коды.

Тип данных *wchar_t.* Традиционный 8-битовый тип *char* может представлять базовый набор символов, а тип по имени *wchar t* (от *wide character type* — расширенный тип символов) — расширенный набор символов, *wchar t* — это целочисленный тип с объемом, достаточным для представления самого большого расширенного набора символов в системе. Этот тип имеет такой же размер и знак, как и один из остальных целочисленных типов, и называется лежащим в основе типом. Выбор лежащего в основе типа зависит от реализации, поэтому в одной системе это может быть *unsigned short*, а в другой — *int*.

Тип данных *bool.* Целочисленный тип данных, так как диапазон допустимых значений — целые числа от 0 до 255. *Bool* используется исключительно для хранения результатов логических выражений. У логического выражения может быть один из двух результатов *true* или *false*: *true*, если логическое выражение истинно, *false*, если логическое выражение ложно.

Когда *C++* оценивает выражение, значения *bool*, *char, unsigned char, signed char* и *short* преобразуются в *int*. В частности, значение *true* преобразуется в 1, а *false* — в 0. Такие преобразования называются целочисленными расширениями.

Числа с плавающей точкой. Типы с плавающей точкой позволяют представить такие числа, как 2,5; 3,14159 и 122442,32 — т. е. числа с дробными частями. Такие значения хранятся в памяти компьютера в виде двух частей. Одна часть представляет значение, а другая – масштабный коэффициент, который увеличивает или уменьшает значение.

Запись чисел с плавающей точкой. В *C++* поддерживаются два способа записи чисел с плавающей точкой. Первый из них заключается в использовании стандартной нотации, применяемой в повседневной жизни:

12,34 // число с плавающей точкой;
939001,32 // число с плавающей точкой;
0,00023 // число с плавающей точкой;
8,0 // тоже число с плавающей точкой.

Даже если дробная часть равна 0 как в 8,0, наличие десятичной точки гарантирует, что число представлено в формате с плавающей точкой и не является целочисленным. (Стандарт *C++* позволяет реализациям учитывать различные локали, например использовать для представления десятичной точки запятую. Однако это влияет только на внешний вид чисел при вводе и выводе, но не на их представление в коде.)

Требования в *C* и *C++* относительно количества значащих разрядов следующие: тип *float* должен иметь как минимум 32 бита, *double* — как минимум 48 битов и естественно быть не меньше чем *float*, а *long double* должен быть минимум таким же, как и тип *double.* Все три типа могут иметь одинаковый размер. Однако обычно *float* занимает 32 бита, *double* — 64 бита, а *long double* — 80, 96 или 128 битов. Кроме того, диапазон порядка для каждого из этих трех типов — как минимум от минус 37 до плюс 37.

Константы с плавающей точкой. Когда в программе записывается константа с плавающей точкой, то с каким именно типом она будет сохранена? По умолчанию константы с плавающей точкой, такие как 8.24 и 2.4E8, имеют тип *double*. Если константа должна иметь тип *float*, необходимо указать суффикс *f* или *F*. Для типа *long double* используется суффикс *l* или *L*. (Поскольку начертание буквы *l* в нижнем регистре очень похоже на начертание цифры 1, рекомендуется применять *L* в верхнем регистре.) Ниже приведены примеры использования суффиксов:

1,234*f* // константа *float*;
2,45E20*F* // константа *float*;
2,345324E28 // константа *double*;
2,2*L* // константа long *double*;

Арифметические операции в *C++*

В *C++* предоставляются операции для выполнения пяти базовых арифметических действий: сложения, вычитания, умножения, деления и получения остатка от деления. Каждая из этих операций использует два значения (называемые операндами) для вычисления конечного результата. Операция и ее операнды вместе образуют выражение.

Например, рассмотрим следующий оператор:

int wheels = 4 + 2;

Значения 4 и 2 — это операнды, знак «+» обозначает операцию сложения, а 4 + 2 — это выражение, результатом которого является 6.

Ниже перечислены пять базовых арифметических операций в *C++*.

Операция «+» выполняет сложение операндов. Например, 4 + 20 дает 24.

Операция «–» вычитает второй операнд из первого. Например, 12–3 дает 9.

Операция «*» умножает операнды. Например, 28*4 дает 112.

Операция «/» выполняет деление первого операнда на второй.

Например, 1000/5 дает 200. Если оба операнда являются целыми числами, то результат будет равен целой доли частного. Например, 17/3 дает 5, с отброшенной дробной частью.

Операция «%» находит остаток от деления первого операнда на второй. Например, 19 % 6 равно 1, поскольку 6 входит в 19 три раза, с остатком 1.

Порядок выполнения операций: приоритеты операций и ассоциативность. Можно ли доверить *C++* выполнение сложных вычислений. Да, но для этого необходимо знать правила, которыми руководствуется *C++*. Например, во многих выражениях присутствует более одной операции. Тогда возникает логичный вопрос: какая из них должна быть выполнена первой? Например, рассмотрим следующий оператор:

int flyingpigs =3+4*5; // каким будет результат: 35 или 23?

Получается, что операнд 4 может участвовать и в сложении, и в умножении. Когда над одним и тем же операндом может быть выполнено несколько операций, *C++* руководствуется правилами старшинства или приоритетов, чтобы определить, какая операция должна быть выполнена первой.

В том случае, когда две операции имеют одинаковый уровень приоритета, *C++* анализирует их ассоциативность: слева направо или справа налево. Ассоциативность слева направо означает, что если две операции, выполняемые над одним и тем же операндом, имеют одинаковый приоритет, то сначала выполняется операция слева от операнда. В случае ассоциативности справа налево первой будет выполнена операция справа от операнда.

Различные результаты, получаемые после деления. Давайте продолжим рассмотрение особенностей операции деления (/). Поведение этой операции зависит от типа операндов. Если оба операнда являются целочисленными, то *C++* выполнит целочисленное деление. Это означает, что любая дробная часть результата будет отброшена, приводя результат к целому числу. Если один или оба операнда являются значениями с плавающей точкой, то дробная часть остается, поэтому результатом будет число с плавающей точкой.

Операция нахождения остатка от деления. На практике чаще применяются операции сложения, вычитания, умножения и деления, нежели операция нахождения остатка от деления, поэтому рассмотрим ее более подробно. В результате выполнения этой операции возвращается остаток, полученный от целочисленного деления. В комбинации с целочисленным делением операция нахождения остатка особенно полезна при решении задач, в которых интересуемую величину необходимо разделить на целые единицы, например при преобразовании дюймов в футы и дюймы или долларов в двадцатипятицентовые, десятицентовые, пятицентовые и одноцентовые эквиваленты.

Преобразования типов. Большое разнообразие типов в *C++* позволяет выбрать тип, удовлетворяющий необходимым требованиям. Однако помимо пользы это разнообразие еще и усложняет компьютеру жизнь. Например, при сложении двух значений, имеющих

тип *short*, могут использоваться аппаратные инструкции, отличные от применяемых при сложении двух значений *long*. При наличии 11 целочисленных типов и 3 типов чисел с плавающей точкой компьютер сталкивается с множеством случаев их обработки, особенно если смешиваете различные типы. Чтобы справиться с потенциальной неразберихой в типах, многие преобразования типов в *C++* осуществляются автоматически.

C++ преобразует значения во время присваивания значения одного арифметического типа переменной, относящейся к другому арифметическому типу.

C++ преобразует значения при комбинировании разных типов в выражениях.

C++ преобразует значения при передаче аргументов функциям.

2 ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ. МАССИВЫ. СТРОКИ

Введение в массивы

Массив — это структура данных, которая содержит множество значений, относящихся к одному и тому же типу. Например, массив может содержать 60 значений типа *int*, которые представляют информацию об объемах продаж за 5 лет, 12 значений типа *short*, представляющих количество дней в каждом месяце, или 365 значений типа *float*, которые указывают ежедневные расходы на питание в течение года. Каждое значение сохраняется в отдельном элементе массива, и компьютер хранит все элементы массива в памяти последовательно, друг за другом.

Для создания массива (рисунок 2.1) используется оператор объявления. Объявление массива должно описывать три аспекта:

– тип значений каждого элемента;

– имя массива;

– количество элементов в массиве.

В *C++* это достигается модификацией объявления простой переменной, к которому добавляются квадратные скобки, содержащие внутри количество элементов. Например, следующее объявление создает массив по имени *ragnar*, имеющий 7 элементов, каждый из которых может хранить одно значение типа *int*:

int ragnar [7]; // создает массив из 7 элементов типа *int*

В сущности, каждый элемент – это переменная, которую можно трактовать как простую переменную.

Польза от массивов определяется тем фактом, что к его элементам можно обращаться индивидуально. Способ, который позволяет это делать, заключается в применении индекса для нумерации элементов. Нумерация массивов в *C++* начинается с нуля. Для указания элемента массива в *C++* используется обозначение с квадратными скобками и индексом между ними. Например, *ragnar* [0] — это первый элемент массива *ragnar*, а *ragnar* [6] — его последний элемент.

Рисунок 2.1 – Создание массива

Строки. Строка — это серия символов, сохраненная в расположенных последовательно байтах памяти. В *C++* доступны два способа работы со строками информации, такой как сообщения для пользователя или его ответы. Строки в стиле *C* обладают специальной характеристикой: последним в каждой такой строке является нулевой символ. Этот символ, записываемый как «\0», служит меткой конца строки.

Например, рассмотрим два следующих объявления:

char dog [8] ={ '*b*', 'e', 'a', 'и', 'x', ' ', 'I', 'I'}; // это не строка;
char cat [8] = { '*f* ', '*a*', '*t*', '*e*', '*s*', '*s*', '*a*', * \ 0 * }; // а это — строка.

Обе эти переменные представляют собой массивы *char*, но только вторая из них является строкой. Нулевой символ играет фундаментальную роль в строках стиля *C*. Например, в *C++* имеется множество функций для обработки строк, включая те, которые спользуются *cout*. Все они обрабатывают строки символ за символом до тех пор, пока не встретится нулевой символ.

Существует простой способ инициализации массива с помощью строки (рисунок 2.2). Для этого просто используйте строку в двойных кавычках, которая называется строковой константой или строковым литералом, как показано ниже:

char bird [11] = "*Mr. Cheeps*"; // наличие символа «\0» подразумевается;

char fish [] = "*Bubbles*"; // позволяет компилятору подсчитать количество элементов.

```
char boss[8] = "Bozo";
```

Рисунок 2.2 — Инициализация массива строкой

Строчно-ориентированный ввод с помощью *getline* (). Функция *getline*() читает целую строку, используя символ новой строки, который передан клавишей «*Enter*», для обозначения конца ввода. Этот метод инициируется вызовом функции *cin.getline*(). Функция принимает два аргумента. Первый аргумент — это имя места назначения (т. е. массива, который сохраняет введенную строку), а второй — максимальное количество символов, подлежащих чтению. Если, скажем, установлен предел 20, то функция читает не более 19 символов, оставляя место для автоматически добавляемого в конец нулевого символа. Функция-член *getline*() прекращает чтение, когда достигает указанного предела количества символов или когда читает символ новой строки, смотря, что произойдет раньше.

Например, предположим, что вы хотите воспользоваться *getline*() для чтения имени в 20-элементный массив *name*. Для этого следует указать такой вызов:

cin.getline(*name*,20);

Он читает полную строку в массив *name*, предполагая, что строка состоит не более чем из 19 символов.

Строчно-ориентированный ввод с помощью *get* (). Теперь попробуем другой подход. Класс *istream* имеет функцию-член *get*(), которая доступна в различных вариантах. Один из них работает почти так же, как *get line*(). Он принимает те же аргументы, интерпретирует их аналогичным образом и читает до конца строки. Но вместо того, чтобы прочитать и отбросить символ новой строки, *get*() оставляет

его во входной очереди. Предположим, что используются два вызова *get* () подряд:

cin.get(*name*, *ArSize*);
cin.get(*dessert*, *Arsize*); // проблема.

Поскольку первый вызов оставляет символ новой строки во входной очереди, получается, что символ новой строки оказывается первым символом, который видит следующий вызов. Таким образом, второй вызов *get*() заключает, что он достиг конца строки, не найдя ничего интересного, что можно было бы прочитать. Без посторонней помощи *get*() вообще не может преодолеть этот символ новой строки.

К счастью, на помощь приходят различные варианты *get*(). Вызов *cin.get*() без аргументов читает одиночный следующий символ, даже если им будет символ новой строки, поэтому можно использовать его для того, чтобы отбросить символ новой строки и подготовиться к вводу следующей строки. То есть следующая последовательность будет работать правильно:

cin.get(*name*, *ArSize*); // чтение первой строки;
cin.get(); // чтение символа новой строки;
cin.get(*dessert*, *Arsize*); // чтение второй строки.

Другой способ применения *get*() состоит в конкатенации, или соединении, двух вызовов функций-членов класса, как показано в следующем примере:

cin.get(*name*, *ArSize*).*get*(); // конкатенация функций-членов.

Такую возможность обеспечивает то, что *cin.get* (*name*, *ArSize*) возвращает объект *cin*, который затем используется в качестве объекта, вызывающего функцию *get*(). Аналогично приведенный ниже оператор читает две следующие друг за другом строки в массивы *name*1 и *name*2, что эквивалентно двум отдельным вызовам:

cin.getline ():
cin.getline (*name*1, *ArSize*) .*getline*(*name*2, *ArSize*);

Структуры

Структура представляет собой определяемый пользователем тип с объявлением, описывающим свойства данных типа. После определения типа можно создавать переменные этого типа. То есть создание структуры — процесс, состоящий из двух частей. Вначале определяется описание структуры, в котором перечисляются и именуются хранящиеся в структуре типы данных. Затем создаются структурные переменные, или, иначе говоря, структурные объекты данных, которые следуют плану, заданному объявлением.

Например, предположим, что компания *Bloataire. Inc*. желает создать тип данных, описывающий линейку ее продуктов — различного рода надувных предметов. В частности, тип должен включать наименование продукта, его объем в кубических футах, а также розничную цену. Вот описание структуры, отвечающее этим потребностям (рисунок 2.3):

```
struct inflatable // объявление структуры
{
char name[20];
float volume;
ouble price;
};
```

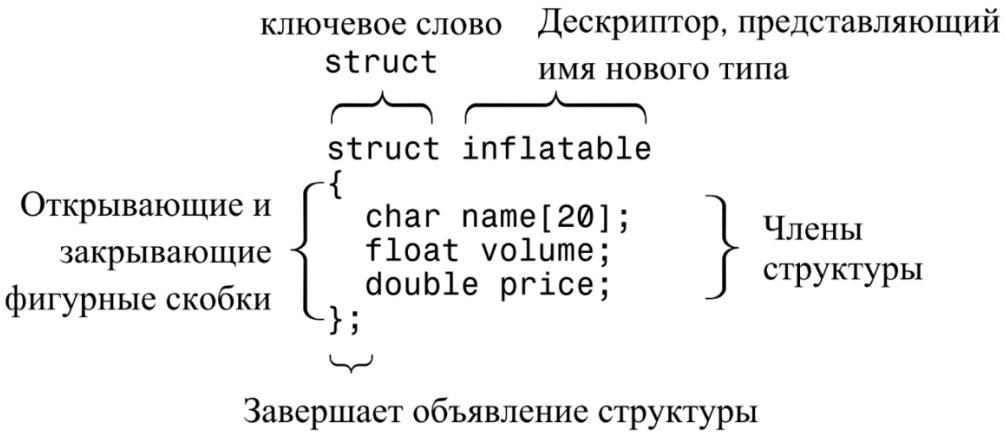

Рисунок 2.3 — Части описания структуры

Ключевое слово *struct* указывает на то, что этот код определяет план структуры. Идентификатор *inflatable* — имя, или дескриптор, этой формы, т. е. имя нового типа. Таким образом, теперь можно

создавать переменные типа *inflatable* точно так же, как создаются переменные типа *char* или *int.* Далее между фигурными скобками находится список типов данных, которые будут содержаться в структуре. Каждый элемент списка — это оператор объявления. Здесь допускается использовать любые типы *C++*, включая массивы и другие структуры. В этом примере применяется массив *char*, который подходит для хранения строки, затем идет один элемент типа *float* и один типа *double*. Каждый индивидуальный элемент в списке называется *членом* структуры, так что структура *inflatable* имеет три члена. Выражаясь кратко, определение структуры описывает характеристики типа — в рассматриваемом случае типа *inflatable*.

Объединения. Объединение — это формат данных, который может хранить в пределах одной области памяти разные типы данных, но в каждый момент времени только один из них. То есть, в то время как структура может содержать, скажем, *int, long* и *double*, объединение может хранить либо *int*, либо *long*, либо *double*. Синтаксис похож на синтаксис структур, но смысл отличается. Например, рассмотрим следующее объявление:

```
union one4all {
int int_val; long long_val; double double val;
```

Переменную *one4all* можно использовать для хранения *int, long* или *double*, если только делать это не одновременно:

```
one4all pail;
pail.int_val = 15; // сохранение int
cout<<pail.int_val;
pail.double_val =1.38; // сохранение double, int теряется
cout<<pail.double_val;
```

Таким образом, *pail* может служить в качестве переменной *int* в одном случае и в качестве переменной *double* — в другом. Имя члена идентифицирует роль, в которой в данный момент выступает переменная. Поскольку объединение хранит только одно значение в единицу времени, оно должно иметь достаточный размер, чтобы вместить самый большой член. Поэтому размер объединения определяется размером его самого большого члена.

Причиной применения объединения может быть необходимость сэкономить память, когда элемент данных может использовать два или более форматов, но никогда – одновременно.

Перечисления. Средство *C++ enum* представляет собой альтернативный по отношению к *const* способ создания символических констант. Он также позволяет определять новые типы, но в очень ограниченной манере. Синтаксис *enum* подобен синтаксису структур. Например, рассмотрим следующий оператор:

enum spectrum {*red, orange, yellow, green, blue, violet, indigo, ultraviolet*};

Этот оператор делает две вещи:

– объявляет имя нового типа — *spectrum*; при этом *spectrum* называется перечислением, почти так же, как переменная *struct* называется структурой;

– устанавливает *red*, *orange*, *yellow* и другие в качестве символических констант для целочисленных значений 0–7. Эти константы называются перечислителями.

По умолчанию перечислителям присваиваются целочисленные значения, начиная с 0 для первого из них, 1 — для второго и т. д. Это правило по умолчанию можно переопределить, явно присваивая целочисленные значения.

Имя перечисления можно использовать для объявления переменной с этим типом перечисления:

spectrum band; // *band* – переменная типа *spectrum*.

Указатели и динамическая память

Новая стратегия хранения данных изменяет трактовку местоположения как именованной величины, а значения — как производной величины. Для этого предусмотрен специальный тип переменной — указатель, который может хранить адрес значения. Таким образом, имя указателя представляет местоположение. Применяя операцию «*», называемую косвенным значением или операцией разыменования, можно получить значение, хранящееся в указанном месте. Предположим, например, что *manly* — это указатель. В таком случае *manly*

представляет адрес, а **manly* — значение, находящееся по этому адресу. Комбинация **manly* становится эквивалентом простой переменной типа *int*.

Объявление и инициализация указателей. Рассмотрим процесс объявления указателей. Компьютеру нужно отслеживать тип значения, на которое ссылается указатель. Например, адрес *char* обычно выглядит точно так же, как и адрес *double*, но *char* и *double* используют разное количество байт и разный внутренний формат представления значений. Поэтому объявление указателя должно задавать тип данных указываемого значения.

Предположим, пример содержит следующее объявление:

*int * p_updates*;

Этот оператор устанавливает, что комбинация **p_updates* имеет тип *int*. Поскольку вы используете операцию «*», применяя ее к указателю, сама переменная *p_updates* должна быть указателем. *p updates* — это указатель (адрес), а **p_updates* — это *int*, а не указатель. Знак «*» должен быть помещен возле каждой переменной типа указателя.

Выделение памяти с помощью операции *new*. Переменные — это именованная память, выделенная во время компиляции, и каждый указатель, до сих пор использованный в примерах, просто представлял собой псевдоним для памяти, доступ к которой и так был возможен по именам переменных. Реальная ценность указателей проявляется тогда, когда во время выполнения выделяются неименованные области памяти для хранения значений. В этом случае указатели становятся единственным способом доступа к такой памяти. В языке *C* память можно выделять с помощью библиотечной функции *mall oc*(). Ее можно применять и в *C++*, но язык *C++* также предлагает лучший способ — операцию *new*.

Давайте испытаем этот новый прием, создав неименованное хранилище времени выполнения для значения типа *int* и обеспечив к нему доступ через указатель. Ключом ко всему является операция *new*. Вы сообщаете *new*, для какого типа данных запрашивается память; *new* находит блок памяти нужного размера и возвращает его

адрес. Вы присваиваете этот адрес указателю, и на этом все. Ниже показан пример:

*int * pn = new int*;

Освобождение памяти с помощью операции *delete.* Использование операции *new* для запрашивания памяти, когда она нужна, – одна из сторон пакета управления памятью *C++*. Второй стороной является операция *delete*, которая позволяет вернуть память в пул свободной памяти, если работа с ней завершена. Это — важный шаг к максимально эффективному использованию памяти. Память, которую вы возвращаете или освобождаете, затем может быть повторно использована другими частями программы. Операция *delete* применяется с указателем на блок памяти, который был выделен операцией *new*:

*int * ps = new int*; // выделить память с помощью операции;
new ... // использовать память;
delete ps; // по завершении освободить память;
// с помощью операции *delete*.

Это освобождает память, на которую указывает *ps*, но не удаляет сам указатель *ps*. Можно повторно использовать *ps*, например, чтобы указать на другой выделенный *new* блок памяти. Нужно всегда обеспечивать сбалансированное применение *new* и *delete*; в противном случае есть риск столкнуться с таким явлением, как утечка памяти, т. е. ситуацией, когда память выделена, но более не может быть использована. Если утечки памяти слишком велики, то попытка программы выделить очередной блок может привести к ее аварийному завершению.

Динамические переменные. Динамические массивы. Распределение массива во время компиляции называется статическим связыванием и означает, что массив встраивается в программу во время компиляции. Но с помощью *new* вы можете создать массив, когда это необходимо во время выполнения программы, либо не создавать его, если потребность в нем отсутствует. Или же вы можете выбрать размер массива уже после того, как программа запущена. Это называется динамическим связыванием и означает, что массив будет создан во

время выполнения программы. Такой массив называется динамическим массивом. При статическом связывании вы должны жестко закодировать размер массива во время написания программы. При динамическом связывании программа может принять решение о размере массива во время своей работы.

Создание динамического массива с помощью операции *new*. Создать динамический массив на *C++* легко; нужно сообщить операции *new* тип элементов массива и требуемое количество элементов. Синтаксис, необходимый для этого, предусматривает указание имени типа с количеством элементов в квадратных скобках. Например, если необходим массив из 10 элементов *int*, следует записать так:

*int * psome = new int* [10] ; // получение блока памяти из 10 элементов типа *int*

Операция *new* возвращает адрес первого элемента в блоке. В данном примере это значение присваивается указателю *psome*.

Как всегда, вы должны сбалансировать каждый вызов *new* соответствующим вызовом *delete*, когда программа завершает работу с этим блоком памяти. Однако использование *new* с квадратными скобками для создания массива требует применения альтернативной формы *delete* при освобождении массива:

delete [] *psome*; // освобождение динамического массива

При использовании *new* и *delete* необходимо придерживаться перечисленных ниже правил:

– не использовать *delete* для освобождения той памяти, которая не была выделена *new*.

– не использовать *delete* для освобождения одного и того же блока памяти дважды.

– использовать *delete* [], если применялась операция *new* [] для размещения массива.

– использовать *delete* без скобок, если применялась операция *new* для размещения отдельного элемента.

– помнить о том, что применение *delete* к нулевому указателю является безопасным (при этом ничего не происходит).

Указатели, массивы и арифметика указателей. *C++* позволяет добавлять целые числа к указателю. Результат добавления к указателю единицы равен исходному адресу плюс значение, эквивалентное количеству байт в указываемом объекте. Можно также вычесть один указатель из другого, чтобы получить разницу между двумя указателями. Последняя операция, которая возвращает целочисленное значение, имеет смысл только в случае, когда два указателя определяют элементы одного и того же массива (указание одной из позиций за границей массива также допускается); при этом результат означает расстояние между элементами массива.

Ниже приведен ряд примеров:

int tacos[10] = (5,2,8, 4,1,2,2,4,6,8};
*int * pt = tacos*; // предположим, что *pf* и *tacos* указывают на адрес 3000 *pt = pt* + 1; // теперь *pt* равно 3004, если *int* имеет размер 4 байта
*int *pe = &tacos*[9]; // *pe* равно 3036, если *int* имеет размер 4 байта
pe = pe – 1; // теперь *pe* равно 30 32 – адресу элемента *tacos* [8]
int diff = pe – pt; // *diff* равно 7, т. е. расстоянию между *tacos* [8] и *tacos* [1]

Динамическое и статическое связывание для массивов. Объявление массива можно использовать для создания массива со статическим связыванием, т. е. массива, размер которого фиксирован на этапе компиляции:

int tacos [10]; // статическое связывание, размер фиксирован во время компиляции

Для создания массива с динамическим связыванием (динамического массива) используется операция *new* []. Память для этого массива выделяется в соответствии с размером, указанным во время выполнения программы. Когда работа с таким массивом завершена, выделенная ему память освобождается с помощью операции

delete[]:
int size; *cin>>size;*
*int * pz = new int* [*size*]; // динамическое связывание, размер
// устанавливается во время выполнения
delete [] *pz*; // освобождение памяти по окончании работы
// с массивом

Указатели и строки. Функция *strlen*(), которую вы уже применяли ранее, возвращает длину строки. Функция *strcpy*() копирует строку из одного места в другое. Обе функции имеют прототипы в файле заголовков *cstring*.

Использование операции *new* для создания динамических структур. Динамические здесь снова означает выделение памяти во время выполнения, а не во время компиляции.

Применение *new* со структурами состоит из двух частей: создание структуры и обращение к ее членам (рисунок 2.4). Для создания структуры вместе с операцией *new* указывается тип структуры. Например, чтобы создать безымянную структуру типа *inflatable* и присвоить ее адрес соответствующему указателю, можно поступить следующим образом:

*inflatable * ps = new inflatable*;

Это присвоит указателю *ps* адрес участка памяти достаточного размера, чтобы вместить тип *inflatable*.

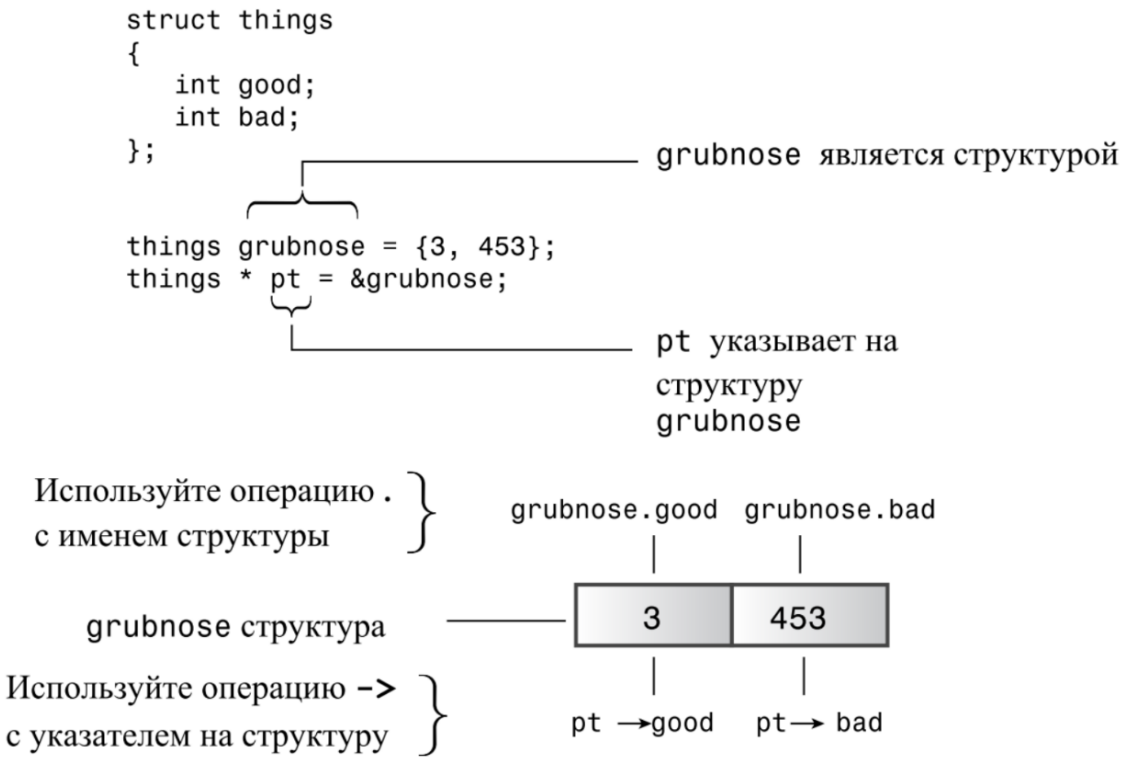

Рисунок 2.4 — Идентификация членов структуры

Автоматическое, статическое и динамическое хранилище. В *C++* предлагаются три способа управления памятью для данных в зависимости от метода ее выделения: автоматическое хранилище, статическое хранилище и динамическое хранилище, иногда называемое свободным хранилищем или кучей. Объекты данных, выделенные этими тремя способами, отличаются друг от друга тем, насколько долго они существуют.

Автоматическое хранилище. Обычные переменные, объявленные внутри функции, используют автоматическое хранилище и называются автоматическими переменными. Этот термин означает, что они создаются автоматически при вызове содержащей их функции и уничтожаются при ее завершении.

Автоматические переменные обычно хранятся в стеке. Это значит, что, когда выполнение программы входит в блок кода, его переменные последовательно добавляются к стеку в памяти и затем освобождаются в обратном порядке, когда выполнение покидает данный блок. (Этот процесс называется *UFO* (*last-in*, *first-out* — «последним пришел — первым ушел».) Таким образом, по мере продвижения выполнения стек растет и уменьшается.

Статическое хранилище. Статическое хранилище — это хранилище, которое существует в течение всего времени выполнения программы. Доступны два способа для того, чтобы сделать переменные статическими. Один заключается в объявлении их вне функций. Другой предполагает использование при объявлении переменной ключевого слова *static*:

static double fee = 56.50;

Динамическое хранилище. Операции *new* и *delete* предлагают более гибкий подход, нежели использование автоматических и статических переменных. Они управляют пулом памяти, который в *C++* называется свободным хранилищем, или кучей. Этот пул отделен от области памяти, используемой статическими и автоматическими переменными.

Совместное применение *new* и *delete* предоставляет возможность более тонко управлять использованием памяти, чем в случае обычных переменных. Однако управление памятью становится более сложным. В стеке механизм автоматического добавления и удаления приводит к тому, что части памяти всегда являются смежными. Тем не менее, чередование операций *new* и *delete* может привести к появлению промежутков в свободном хранилище, усложняя отслеживание места, где будут распределяться новые запросы памяти.

3 ЦИКЛЫ И ВЫРАЖЕНИЯ СРАВНЕНИЯ. ОПЕРАТОРЫ ВЕТВЛЕНИЯ И ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Циклы и выражения сравнения

Обстоятельства часто требуют от программ выполнения повторяющихся задач, таких как сложение элементов массивов один за другим или 20-кратная распечатка похвалы за продуктивность. Цикл *for* облегчает выполнение задач подобного рода.

Цикл *for* представляет собой средство пошагового выполнения повторяющихся действий. Давайте рассмотрим более подробно, как он устроен. Обычно части цикла *for* выполняют следующие шаги;

– установка начального значения;

– выполнение проверки условия для продолжения цикла;

– выполнение действий цикла;

– обновление значения (значений), используемого в проверочном условии.

В структуре цикла *C++* эти элементы расположены таким образом, чтобы их можно было охватить одним взглядом. Инициализация, проверка и обновление составляют три части управляющего раздела, заключенного в круглые скобки. Каждая часть является выражением, отделяемым от других частей точкой с запятой. Оператор, следующий за управляющим разделом, называется теломцикла, и он выполняется до тех пор, пока проверочное условие остается истинным:

for (инициализация; проверочное выражение**;** обновляющее выражение)

В синтаксисе *C++* полный оператор *for* считается одним оператором, несмотря на то что он может заключать в своем теле один или более других операторов.

Цикл выполняет инициализациютолько однажды. Как правило, программы используют это выражение для установки переменной в некоторое начальное значение, а потом применяют эту переменную в качестве счетчика цикла.

Проверочное выражение определяет, должно ли выполняться тело цикла. Обычно это выражение представляет собой выражение сравнения, т. е. выражение, сравнивающее два значения.

Цикл *for* является циклом с входным условием (рисунок 3.1). Это значит, что проверочное условие выполняется перед каждым шагом цикла. Цикл никогда не выполняет тело, если проверочное условие возвращает *false*.

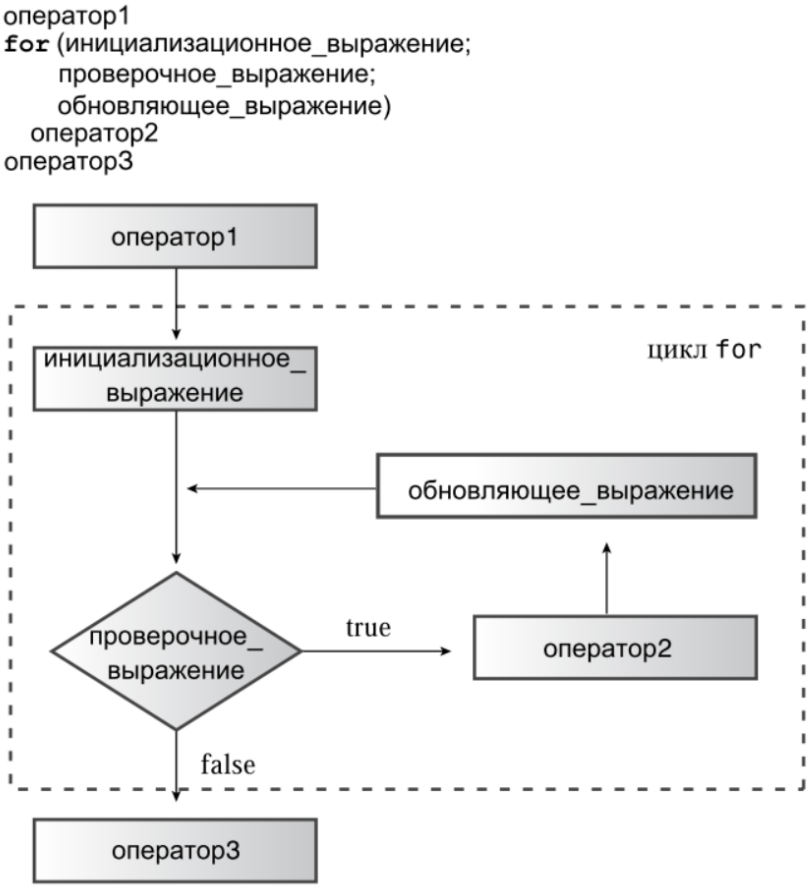

Рисунок 3.1 — Структура циклов *for*

Оператор цикла *for* в чем-то похож на вызов функции, потому что использует имя, за которым следует пара скобок. Однако статус *for* как ключевого слова *C++* предотвращает восприятие его компилятором как функции. Это также предохраняет вас от попыток назвать функцию именем *for*.

Операции инкремента и декремента. Язык *C++* снабжен несколькими операциями, которые часто используются в циклах, две из них: операция инкремента (++), которая получила отражение в самом

названии *C++*, а также операция декремента. Эти операции выполняют два чрезвычайно часто встречающихся действия в циклах: увеличивают и уменьшают на единицу значение счетчика цикла. Префиксная версия операции указывается перед операндом, как в (*++X*). Постфиксная версия следует после операнда, как в (*X++*). Эти две версии имеют один и тот же эффект для операнда, но отличаются в контексте применения.

В листинге 3.1 демонстрируется разница на примере операции инкремента.

Листинг 3.1. *plus_one. cpp*

```
// plus_one.cpp -- операция инкремента
#include<iostream>
int main()
{
using std::cout;
int a = 2 0;
intb = 20;
cout « "a = " « a<< ": b = " « b « "\n";
cout « "a++ = " « a++ « ++b = " « ++b « "\n";
cout « "a = " « a « " : b = " « b « "\n";
return 0;
```

Результат выполнения этой программы показан ниже:

```
a =20: b = 20
a++ = 20: ++b = 21
a =21: b = 21
```

Грубо говоря, нотация «*a++*» означает «использовать текущее значение *a* при вычислении выражения, затем увеличить *a* на единицу». Аналогично, нотация «*++a*» означает «сначала увеличить значение *a* на единицу, затем использовать новое значение при вычислении выражения».

Операции инкремента и декремента представляют собой простой удобный способ решения часто возникающей задачи увеличения или уменьшения значений на единицу.

Комбинированные операторы присваивания. Для обновления счетчика цикла используется следующее выражение:

i = *i* + *by*

В *C++* предусмотрена комбинированная операция сложения с присваиванием, которая позволяет получить тот же результат, но более кратко:

i += *by*

Операция «+=» складывает значения своих двух операндов и присваивает результат левому операнду. Это предполагает, что левый операнд должен быть чем-то таким, чему можно присваивать значения, вроде переменной, элемента массива, члена структуры либо элемента данных, полученного через разыменование указателя.

Каждая арифметическая операция имеет соответствующую операцию присваивания, как показано на рисунке 3.2. Каждая такая операция работает аналогично «+=». То есть, например, следующий оператор заменяет текущее значение *k* в 10 раз большим значением: *k* *= 10.

Операция	Эффект (L — левый операнд, R — правый операнд)
+=	Присваивает L + R операнду L
-=	Присваивает L - R операнду L
*=	Присваивает L * R операнду L
/=	Присваивает L / R операнду L
%=	Присваивает L % R операнду L

Рисунок 3.2 — Комбинированные операции присваивания

Составные операторы, или блоки. Формат, или синтаксис, оператора *for* может показаться чересчур ограниченным, поскольку тело цикла должно состоять всего лишь из одного оператора. Это весьма неудобно, если вы хотите, чтобы тело цикла включало несколько операторов. К счастью, *C++* предлагает синтаксис, позволяющий поместить в тело цикла любое количество операторов. Трюк заключается в использовании пары фигурных скобок, с помощью которых конструируется составной оператор, или блок. Блок состоит

из пары фигурных скобок с заключенными между ними операторами и синтаксически воспринимается как один оператор.

Составные операторы обладают еще одним интересным свойством. Если вы определяете новую переменную внутри блока, она будет существовать только во время выполнения операторов этого блока. Когда поток выполнения покидает блок, такая переменная уничтожается. Это значит, что переменная известна только внутри блока.

Дополнительные синтаксические трюки: операция запятой. Блок позволяет помещать два и более оператора там, где синтаксис *C++* разрешает лишь один. Операция запятой (,) делает то же самое с выражениями, позволяя вставлять два выражения туда, где синтаксис *C++* допускает только одно. Например, предположим, что имеется цикл, в котором на каждом шаге одна переменная увеличивается на единицу, а вторая — на единицу уменьшается. Было бы удобно сделать то и другое в обновляющей части цикла *for*, но синтаксис цикла разрешает там только одно выражение. Решение состоит в применении операции запятой для комбинации двух выражений в одно:

```
++j, –i // два выражения воспринимаются как одно;
// для удовлетворения требований синтаксиса.
```

Запятая — это не всегда операция. Например, запятая в следующем объявлении служит для разделения имен в списке объявляемых переменных:

```
int i, j; // здесь запятая — разделитель, а не операция
```

Выражения отношений. В *C++* доступны шесть операций отношений для сравнения чисел. Каждое сравнивающее выражение возвращает булевское (типа *bool*) значение *true*, если сравнение истинно, и *false* — в противном случае, поэтому данные операции хорошо подходят для применения в проверочных условиях циклов. (Старые реализации оценивали истинные выражения как 1 и ложные — как 0.)

На рисунке 3.3 представлен список операций отношений.

Операция	Описание
<	Меньше чем
<=	Меньше или равно
==	Равно
>	Больше чем
>=	Больше или равно
!=	Не равно

Рисунок 3.3 — Операции отношений

Этими шестью операциями отношений исчерпываются все возможности, предусмотренные в *C++* для сравнения чисел. Если хотите сравнить два значения на предмет того, какое из них более красивое или более удачное, вам придется поискать в другом месте.

Вот некоторые примеры сравнений:

$for(x = 20; x > 5; x--)$ // продолжать, пока *x* больше чем 5;

$for(x = 1; y != x; ++x)$ // продолжать, пока *y* не равно *x*;

$for(cin >> x; x == 5; x--)$ // продолжать, пока *x* равно 0.

Присваивание, сравнение и вероятные ошибки. Не следует путать операцию проверки равенства (==) с операцией присваивания (=). Следующее выражение задает вопрос «Равно ли значение *musicians* четырем?»:

musicians == 4 // сравнение

Выражение может принимать значение *true* или *false*. Приведенное ниже выражение присваивает *musicians* значение 4:

musicians = 4 // присваивание

Полное выражение в данном случае имеет значение 4, потому что таково значение левой части.

Сравнение строк класса *string*. Для сравнения нужно использовать строки класса *string*, поскольку этот класс позволяет применять операции отношений для выполнения сравнений. Это становится возможным благодаря определению функций класса, которые «перегружают» или переопределяют операции.

Цикл *while.* Цикл *while* — это цикл *for*, у которого удалены инициализирующая и обновляющая части; в нем имеется только проверочное условие и тело.

Сначала программа вычисляет выражение ”проверочное условие” в скобках. Если выражение дает в результате *true*, программа выполняет оператор (или операторы), содержащийся в теле цикла. Как и в случае с циклом *for*, тело состоит из единственного оператора либо блока, определенного фигурными скобками. После того как завершено выполнение тела, программа возвращается к проверочному условию и заново вычисляет его. Если условие возвращает ненулевое значение, программа снова выполняет тело. Этот цикл проверки и выполнения продолжается до тех пор, пока проверочное условие не вернет *false* (рисунок 3.4). Понятно, если вы хотите в конечном итоге прервать цикл, то в теле цикла должно происходить нечто такое, что повлияет на выражение проверочного условия. Например, цикл может увеличивать значение переменной, используемой в проверочном условии, либо читать новое значение, вводимое с клавиатуры. Подобно *for*, цикл *while* является циклом с входным условием. То есть если проверочное условие оценивается как *false* в самом начале, то программа ни разу не выполнит тело цикла.

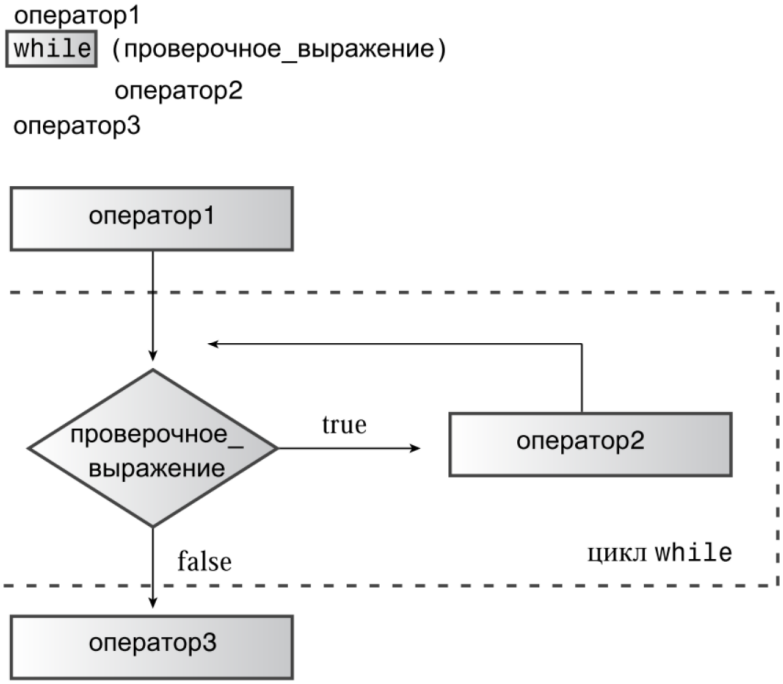

Рисунок 3.4 — Структура циклов *while*

Цикл *do while.* К этому моменту познакомились с двумя циклами — *for* и *while*. Третьим циклом в *C++* является *do while* (рисунок 3.5). Он отличается от двух других тем, что осуществляет проверку на выходе. Это значит, что такой цикл сначала выполнит свое тело и только потом оценит проверочное условие, чтобы узнать, нужно ли продолжать дальше. Если условие оценивается как *false*, цикл завершается; в противном случае выполняется новый шаг с последующей проверкой условия. Такой цикл всегда выполняется как минимум один раз, потому что поток управления программы проходит через его тело до того, как достигает проверочного условия. Синтаксис цикла *do while* показан ниже:

do
тело цикла
while (проверочное_выражение)*;*

Часть “тело цикла” может быть единственным оператором либо блоком операторов, заключенным в фигурные скобки. На рисунке 3.5 показан поток управления в цикле *do while*.

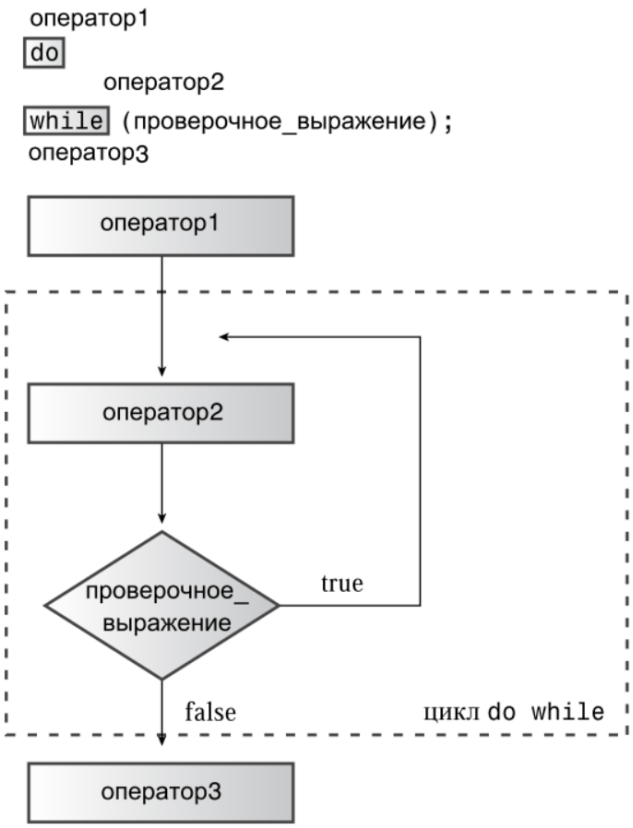

Рисунок 3.5 — Структура циклов *do while*

Вложенные циклы и двумерные массивы. Ранее уже видели, что цикл *for* — это естественный инструмент для обработки массивов. Теперь сделаем еще один шаг и посмотрим, как цикл *for*, внутри которого находится еще один цикл *for* (вложенные циклы), может применяться для обработки двумерных массивов (рисунок 3.6).

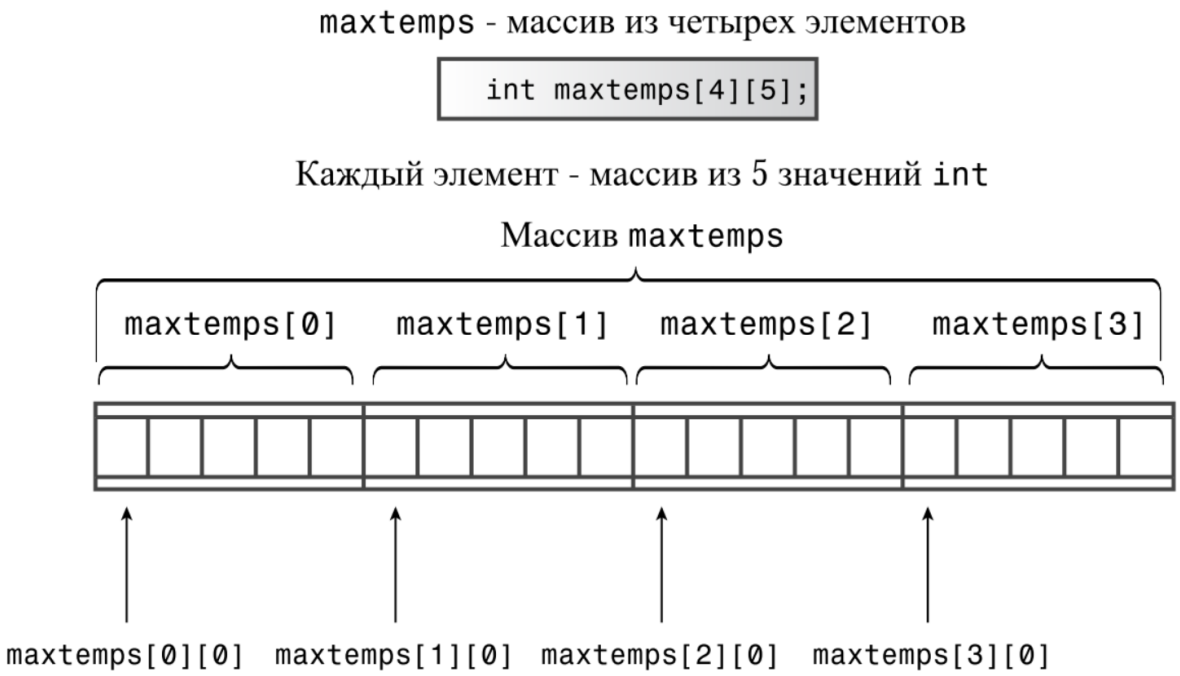

Рисунок 3.6 — Массив массивов

Инициализация двумерного массива. При создании двумерного массива имеется возможность инициализировать каждый его элемент. Прием основан на способе инициализации одномерного массива. Это делается указанием разделенного запятыми списка элементов, заключенного в фигурные скобки:

// Инициализация одномерного массива *int btus* [5] = (23, 26, 24, 31, 28};

В двумерном массиве каждый элемент сам по себе является массивом, поэтому инициализировать каждый элемент можно так, как показано в предыдущем примере. То есть инициализация состоит из разделенной запятыми последовательности одномерных инициализаций, каждая из которых заключена в фигурные скобки:

int maxtemps[4][5] = // двумерный массив
{

{96, 100, 87, 101, 105} // значение для *maxtemps* [0]
{96, 98, 91, 107, 104} // значение для *maxtemps* [1]
{97, 101, 93, 108, 107} // значение для *maxtemps* [2]
{98, 103, 95, 109, 108} // значение для *maxtemps* [3]
};

Массив *maxtemps* содержит четыре строки по пять чисел в каждой.

Выражение {96, 100, 87, 101, 105} инициализирует первую строку, представляемую как *maxtemps* [0]. Стиль размещения каждой строки данных в отдельной строке кода улучшает читабельность.

Операторы ветвления и логические операции

Оператор *if*. Когда программа *C++* должна принять решение о том, какое из альтернативных действий следует выполнить, такой выбор обычно реализуется оператором *if*. Этот оператор имеет две формы: просто *if* и *if else*. Сначала исследуем простой *if*. Оператор *if* разрешает программе выполнять оператор или блок операторов при условии истинности проверочного условия и пропускает этот оператор или блок, если проверочное условие оценивается как ложное. Таким образом, оператор *if* позволяет программе принимать решение относительно того, нужно ли выполнять некоторую часть кода.

Синтаксис оператора *if* подобен *while*:

if (проверочное_выражение) оператор

Истинность выражения «проверочное_выражение» заставляет программу выполнить оператор, который может быть единственным оператором или блоком операторов. Ложность выражения «проверочное_выражение» заставляет программу пропустить оператор (рисунок 3.7). Как и с проверочными условиями циклов, тип проверочного условия *if* приводится к *bool*, поэтому ноль трактуется как *false*, а все, что отличается от нуля — как *true*. Вся конструкция *if* рассматривается как одиночный оператор.

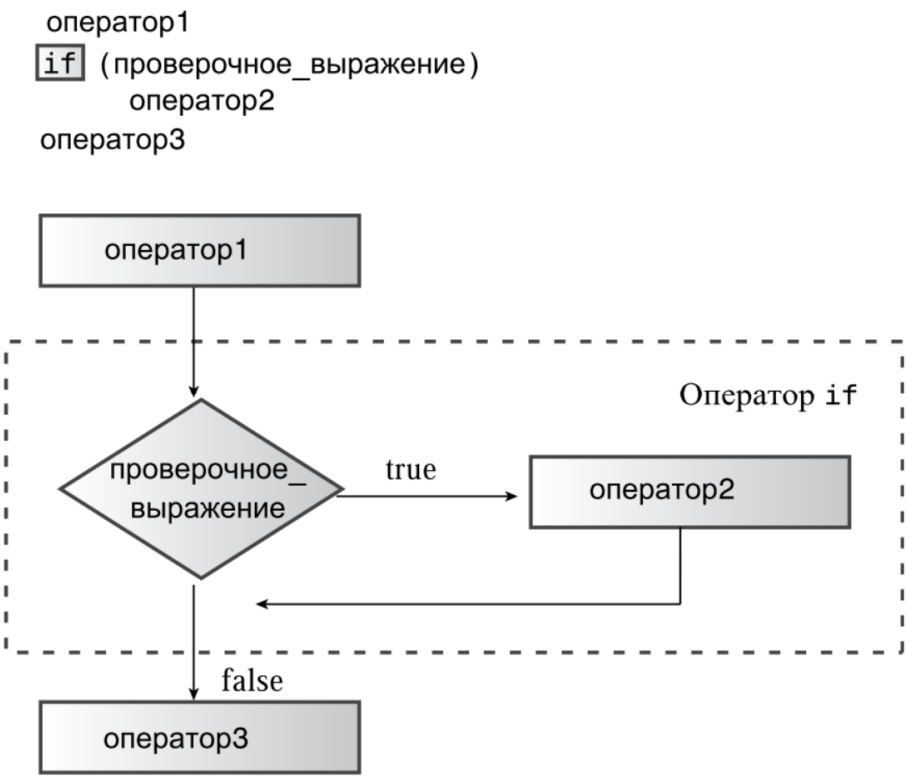

Рисунок 3.7 — Структура оператора *if*

Оператор *if else.* В то время как оператор *if* позволяет программе принять решение о том, должен ли выполняться определенный оператор или блок, *if else* позволяет решить, какой из двух операторов или блоков следует выполнить. Это незаменимое средство для программирования альтернативных действий.

Оператор *if else* имеет следующую общую форму:

if (проверочное_выражение) оператор1
else
оператор2

Если «проверочное_выражение» равно *true* или не ноль, то программа выполняет «оператор1» и пропускает «оператор2». В противном случае, когда «проверочное_выражение» равно *false* или ноль, программа выполняет «оператор2» и пропускает «оператор1» (рисунок 3.8).

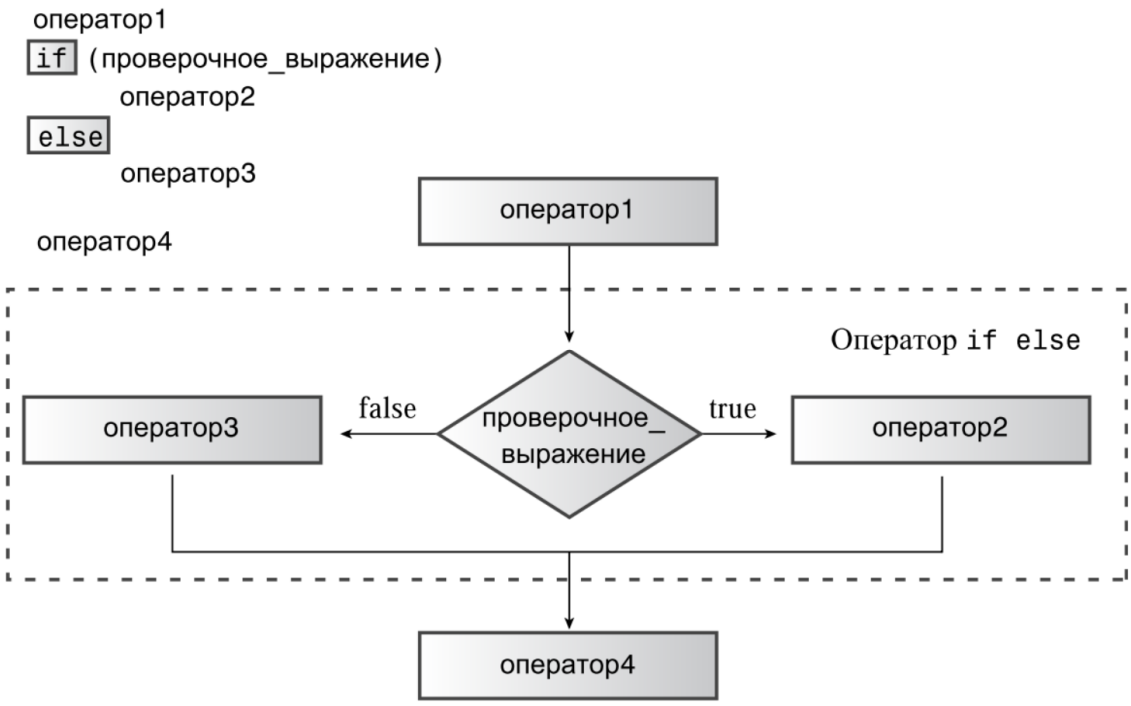

Рисунок 3.8 — Структура оператора *if else*

Конструкция *if else if else.* Оператор *C++ if else* можно расширить, чтобы он отвечал большим потребностям. Как уже говорилось, за *else* должен следовать единственный оператор, который может быть и блоком операторов. Поскольку конструкция *if else* сама является единым оператором, она может следовать за *else*:

```
if (ch == 'A' ) a_grade++; else
if (ch == 'B') b_grade++; else
soso++;
// альтернатива # 1
// альтернатива # 2 // подальтернатива # 2a
// подальтернатива # 2b
```

Если значение *ch* не равно «*A*», программа переходит к *else*. Там второй оператор *if else* разделяет эту альтернативу еще на два варианта. Свойство свободного форматирования *C++* позволяет расположить эти элементы в более читабельном виде:

```
if (ch == 'A')
a_grade++; // альтернатива # 1
else if (ch == 'B')
b_grade++; // альтернатива # 2
else
soso++; // альтернатива # 3
```

Это выглядит как совершенно новая управляющая структура — *if else if else*. Но на самом деле это один оператор *if else*, вложенный в другой. Пересмотренный формат выглядит намного яснее и позволяет даже при поверхностном взгляде оценить все альтернативы. Вся эта конструкция по-прежнему трактуется как единственный оператор.

Логические выражения. Часто приходится проверять более одного условия. Например, чтобы символ относился к прописным буквам, его значение должно быть больше или равно ***a*** и меньше или равно ***z***. Либо же если попросить пользователя ответить ***y*** или ***n***, то наряду с прописными должны приниматься и заглавные буквы (**Y** или **N**). Чтобы обеспечить такие возможности, *C++* предлагает три логические операции, с помощью которых можно комбинировать или модифицировать существующие выражения:

– логическое «ИЛИ» (которое записывается как «||»);

– логическое «И» (записывается как «&&»);

– логическое «НЕ» (записывается как «!»).

Логическая операция «ИЛИ». В английском языке слово «*or*» (или) означает, что одно из двух условий либо оба сразу удовлетворяют некоторому требованию. Эквивалент логической операции «ИЛИ» в *C++* записывается как «||». Эта операция комбинирует два выражения в одно. Если одно или оба исходных выражения возвращают *true* или не ноль, то результирующее выражение имеет значение *true* (истина). В противном случае выражение имеет значение *false*. Ниже показаны некоторые примеры:

– 5 == 5 || 5 == 9 // истинно, потому что первое выражение истинно;

– 5 > 3 || 5 > 10 // истинно, потому что первое выражение истинно;

– 5 > 8 || 5 < 10 // истинно, потому что второе выражение истинно;

– 5 < 8 || 5 > 2 // истинно, потому что оба выражения истинны;

– 5 > 8 || 5 < 2 // ложно, потому что оба выражения ложны.

Поскольку «||» имеет более низкий приоритет, чем операции сравнения, нет необходимости использовать в этих выражениях скобки. На рисунке 3.9 показано, как работает операция «||».

Значение expr1 || expr2		
	expr1 == true	expr1 == false
expr2 == true	true	true
expr2 == false	true	false

Рисунок 3.9 — Операция «||»

В *C++* предполагается, что операция «||» является точкой следования. Это значит, что любое изменение, проведенное в левой части, вычисляется прежде, чем вычисляется правая часть.

Логическая операция «И». Логическая операция «И», которая записывается как «&&», также комбинирует два выражения в одно. Результирующее выражение имеет значение *true* только в том случае, когда оба исходных выражения также равны *true*.

Ниже приведены некоторые примеры:

– 5==5&&4==4 // истинно, потому что оба выражения истинны;
– 5==3&&4==4 // ложно, потому что первое выражение ложно;
– 5>3&&5>10 // ложно, потому что второе выражение ложно;
– 5>8&&5<10 // ложно, потому что первое выражение ложно;
– 5<8&&5>2 // истинно, потому что оба выражения истинны;
– 5>8&&5<2 // ложно, потому что оба выражения ложны.

Поскольку «&&» имеет меньший приоритет, чем операции сравнения, нет необходимости использовать скобки. Подобно «||», операция «&&» действует как точка следования, а потому возможны любые побочные эффекты перед тем, как будет вычислено правое выражение. Если левое выражение ложно, то и все составное выражение также ложно, поэтому в таком случае *C++* можно не беспокоиться о вычислении правой части. Работа операции «&&» показана на рисунке 3.10.

Значение expr1 && expr2		
	expr1 == true	expr1 == false
expr2 == true	true	false
expr2 == false	false	false

Рисунок 3.10 — Операция «&&»

Логическая операция «НЕ». Операция «!» выполняет отрицание, или обращает истинность выражения, следующего за ней. То есть если *expression* равно *true*, то !*expression* равно *false*, и наоборот. Точнее говоря, если *expression* имеет значение *true* или ненулевое, то !*expression* будет равно *false*.

Обычно выражение отношения можно представить яснее без применения операции «!»:

$if\left(!\left(x>5\right)\right)$// в этом случае $if\left(x\le 5\right)$ яснее.

Однако операция «!» может быть полезна с функциями, которые возвращают значения *true*/*false* либо значения, которые могут интерпретироваться подобным образом.

Библиотека символьных функций *cctype.* Язык *C++* унаследовал от *C* удобный пакет функций для работы с символами, прототипы которых находятся в заголовочном файле *cctype*, и это упрощает решение таких задач, как выяснение, является ли символ символом верхнего регистра, десятичной цифрой или знаком препинания.

Например, функция *isaplha (ch*) возвращает ненулевое значение, если *ch* — буква, и ноль — в противоположном случае. Аналогично *ispunct (ch)* возвращает значение *true*, только если *ch* — знак препинания, такой как запятая или точка.

Использовать эти функции намного удобнее, чем операции «И» и «ИЛИ». Например, вот как пришлось бы с помощью «И» и «ИЛИ» проверять, что *ch* является буквенным символом:

$if\left(\left(ch\ge 'a'\&\&ch\le 'z'\right)\|\left(ch\ge 'A'\&\&ch\le 'Z'\right)\right);$

А вот так с применением функции *isalpha* ():

$if\left(isalpha\left(ch\right)\right).$

На рисунке 3.11 кратко описаны функции, доступные в пакете *cctype*. В некоторых системах присутствуют не все эти функции, а в некоторых могут существовать дополнительные функции.

Имя функции	Возвращаемое значение
`isalnum()`	Возвращает `true`, если аргумент — буква или десятичная цифра
`isalpha()`	Возвращает `true`, если аргумент — буква
`isblank()`	Возвращает `true`, если аргумент — пробел или знак горизонтальной табуляции
`iscntrl()`	Возвращает `true`, если аргумент — управляющий символ
`isdigit()`	Возвращает `true`, если аргумент — десятичная цифра (0–9)
`isgraph()`	Возвращает `true`, если аргумент — любой печатаемый символ, отличный от пробела
`islower()`	Возвращает `true`, если аргумент — символ в нижнем регистре
`isprint()`	Возвращает `true`, если аргумент — любой печатаемый символ, включая пробел
`ispunct()`	Возвращает `true`, если аргумент — знак препинания
`isspace()`	Возвращает `true`, если аргумент — стандартный пробельный символ (т.е. пробел, прогон страницы, новая строка, возврат каретки, горизонтальная табуляция, вертикальная табуляция)
`isupper()`	Возвращает `true`, если аргумент — символ в верхнем регистре
`isxdigit()`	Возвращает `true`, если аргумент — шестнадцатеричная цифра (т.е. 0–9, a–f или A–F)
`tolower()`	Если аргумент — символ верхнего регистра, возвращает его вариант в нижнем регистре, иначе возвращает аргумент без изменений
`toupper()`	Если аргумент — символ нижнего регистра, возвращает его вариант в верхнем регистре, иначе возвращает аргумент без изменений

Рисунок 3.11 — Символьные функции *cctype*

Операция «?». Язык *C++* включает операцию, которая часто может использоваться вместо оператора *if else*. Она называется условной операцией, записывается как «?:» и является единственной операцией *C++*, которая требует трех операндов. Ее общая форма выглядит следующим образом:

«выражение! ? выражение2 : выражение3»;

Если «выражение!» истинно, то значением всего условного выражения будет значение «выражение2». В противном случае значением всего выражения будет «выражение3». Ниже приведены два примера, демонстрирующие ее работу:

5>3?10:12//5>3 истинно, поэтому значением всего выражения будет 10;

3==9?25:18//3==9 ложно, поэтому значением всего выражения будет 18.

Первый пример можно перефразировать так: если 5 больше, чем 3, то выражение оценивается как 10; иначе оно оценивается как 12. В реальных ситуациях программирования выражения, конечно же, могут включать в себя переменные.

Оператор *switch.* Предположим, что вы создаете экранное меню, которое предлагает пользователю на выбор один из четырех возможных вариантов, например «дешевый», «умеренный», «дорогой», «экстравагантный» и «непомерный». Вы можете расширить последовательность *if else if else* для обработки этих пяти альтернатив, но оператор *C++ switch* упрощает обработку выбора из большого списка. Ниже представлена общая форма оператора *switch:*

Switch (целочисленное-выражение)
{
case метка1 : оператор (ы)
case метка2 : оператор (ы)
……
default : оператор (ы)
}

Оператор *switch* действует подобно маршрутизатору, который сообщает компьютеру, какую строку кода выполнять следующей. По достижении оператора *switch* программа переходит к строке, которая помечена значением, соответствующим текущему значению целочисленное выражение. Например, если целочисленное выражение имеет значение 4, то программа переходит к строке с меткой *case* 4. Как следует из названия, выражение целочисленное выражение должно быть целочисленным. Также каждая метка должна быть целым константным выражением. Чаще всего метки бывают константами типа *char* или *int*, либо же перечислителями. Если целочисленное выраженииене соответствует ни одной метке, программа переходит к метке *default*. Метка *default* не обязательна. Если она опущена, а соответствия не найдено, программа переходит к оператору, следующему за *switch* (рисунок 3.12).

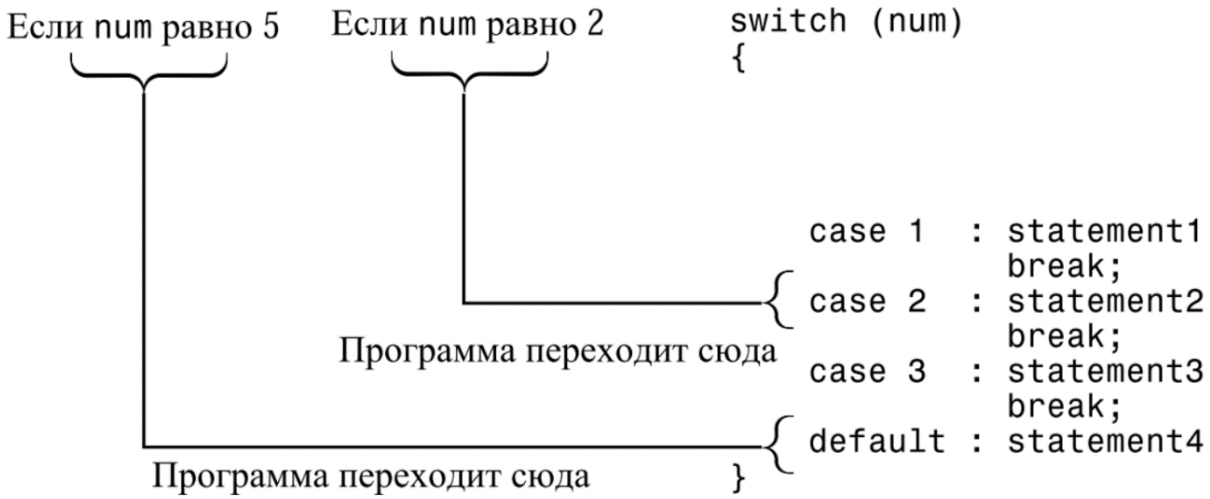

Рисунок 3.12 — Структура оператора *switch*

То есть после того как программа перейдет на определенную строку в *switch*, она последовательно выполнит все операторы, следующие за этой строкой внутри *switch*, если только вы явно не направите ее в другое место. Выполнение *не* останавливается автоматически на следующем *case*. Чтобы прекратить выполнение в конце определенной группы операторов, вы должны использовать оператор *break*. Это передаст управление за пределы блока *switch*.

Операторы *break* и *continue.* Операторы *break* и *continue* позволяют программе пропускать часть кода. Оператор *break* можно использовать в операторе *switch* и в любых циклах. Он вызывает немедленную передачу управления за пределы текущего оператора *switch* или цикла. Оператор *continue* применяется только в циклах и вынуждает программу пропустить остаток тела цикла и сразу начать следующую итерацию (рисунок 3.13).

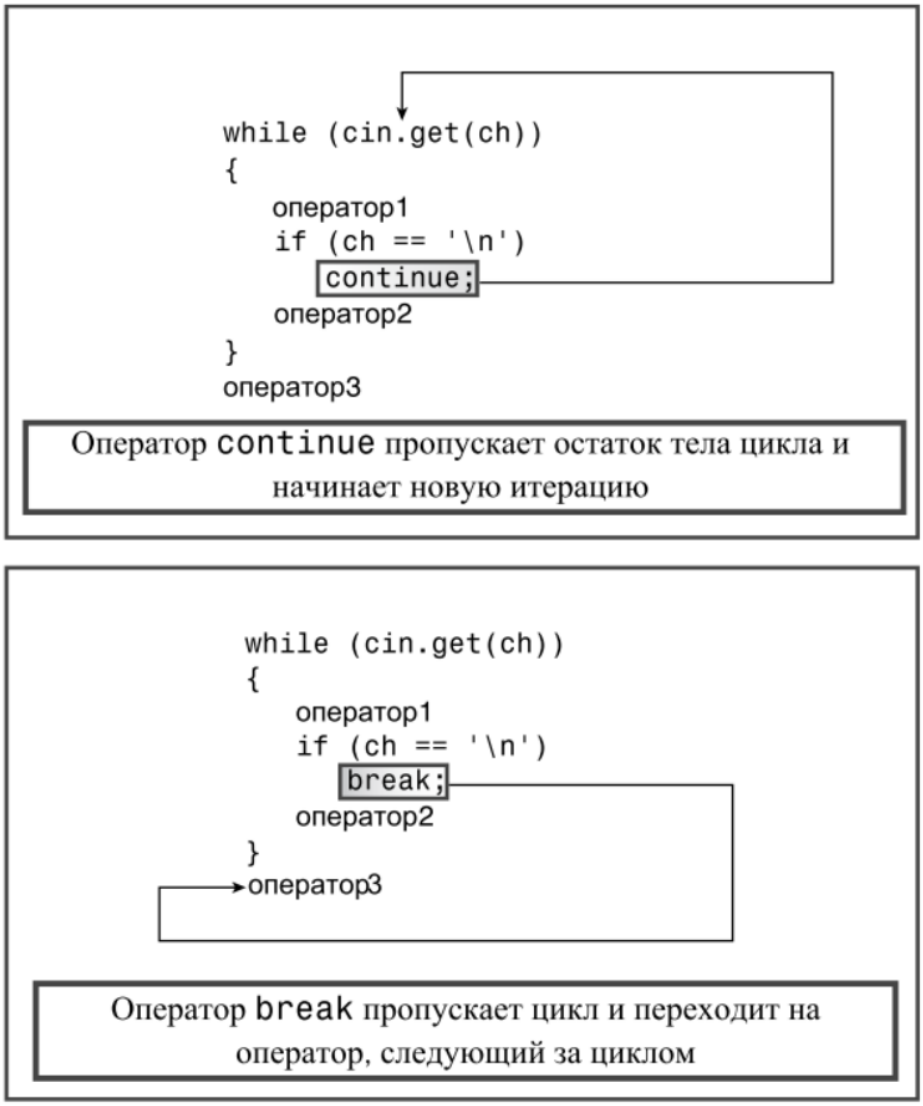

Рисунок 3.13 — Структура операторов *break* и *continue*

4 ФУНКЦИИ ЯЗЫКА *C++*

Функции в *C++* можно разбить на две категории: функции, которые возвращают значения, и функции, значения не возвращающие. Для каждой разновидности функций можно найти примеры в стандартной библиотеке функций *C++*. Кроме того, можно создавать собственные функции обеих категорий.

Функция, имеющая возвращаемое значение, генерирует значение, которое можно присвоить переменной или применить в каком-нибудь выражении. Например, стандартная библиотека *C/C++* содержит функцию *sqrt* (), которая возвращает квадратный корень из числа. Предположим, что требуется вычислить квадратный корень из 6,25 и присвоить его переменной *x*. В программе можно использовать следующий оператор:

x = *sqrt* (6.25); // возвращает значение 2,5 и присваивает его переменной *x*

Выражение *sqrt* (6.25) активизирует, или вызывает, функцию *sqrt* (). Выражение *sqrt* (6.25) называется вызовом функции, активизируемая функция — вызываемой функцией, а функция, содержащая вызов функции — вызывающей функцией (рисунок 4.1).

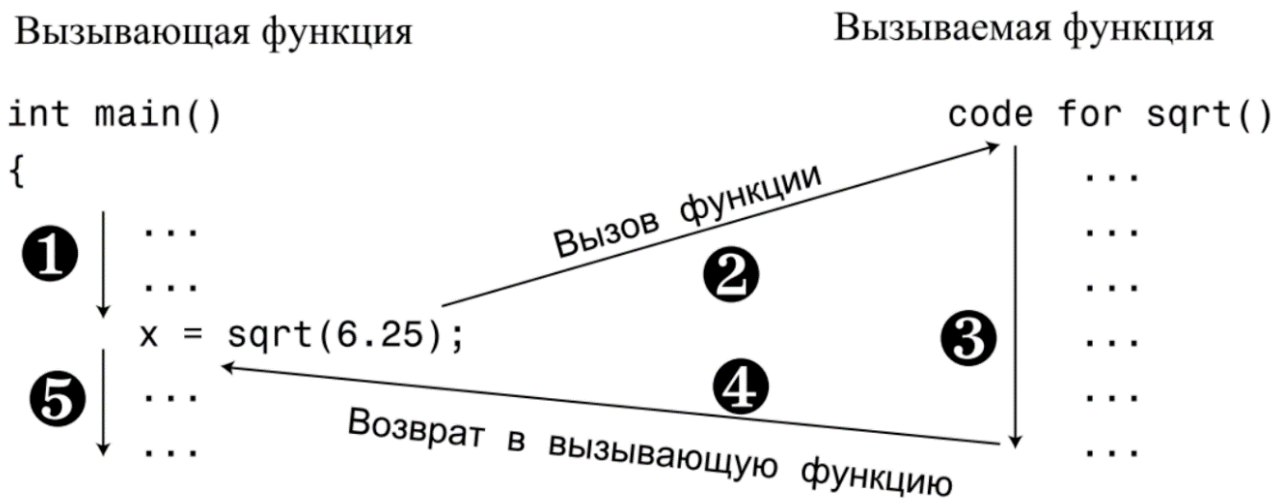

Рисунок 4.1 — Вызов функции

Значение в круглых скобках (в нашем случае — 6.25) — это информация, которая отправляется функции; говорят, что это значение передается функции. Значение, которое отсылается функции подобным образом, называется аргументом, или параметром. Функция *sqrt* () вычисляет ответ, равный 2.5, и отправляет его обратно вызывающей функции; отправленное обратно значение называется возвращаемым значением функции. Возвращаемое значение можно представлять как значение, которое подставляется вместо вызова функции в операторе после того, как выполнение функции завершено. Так, в рассматриваемом примере возвращаемое значение присваивается переменной *x*. Говоря кратко, аргумент — это информация, которая отправляется функции, а возвращаемое значение — это значение, которое отправляется обратно из функции (рисунок 4.2).

Рисунок 4.2 — Синтаксис вызова функции

Прототип функции является для функции тем же, чем для переменной является объявление переменной, он информирует программу о типах данных. Например, в библиотеке *C++* определено, что функция *sqrt*() принимает в качестве аргумента число с дробной частью и возвращает число того же самого типа.

Формат определения функции. На рисунке 4.3 представлен листинг программы *outfunk.cpp*:

```
// ourfunc.cpp - определяет вашу
// собственную функцию

#include <iostream>
using namespace std;
void simon(int); //прототип функции
simon();
int main()
{
    simon (3); //вызов функции simon
    cout << "Pick an integer";
    int cout;
    cin >> cout;
    simon (cout); //еще один вызов этой функции
    return 0;
}

void simon(int n)
{
    cout << "Simon says touch your toes "
         << n << " times. \n";
} // в функциях без возвращаемого значения
  // не требуются операторы return
```

Рисунок 4.3 — Листинг программы *outfunk.cpp*

Функция *main* () вызывает функцию *simon* () два раза: один раз с аргументом 3, а другой раз — с аргументом-переменной *count*. Между этими вызовами пользователь вводит целое число, которое присваивается переменной *count*. В этом примере в приглашении на ввод значения для *count* символ новой строки не используется. В результате пользователь будет вводить значение в той же строке, где располагается приглашение. Ниже показан пример выполнения этой программы:

Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.

Форма функции. Определение функции *simon* () в листинге на рисунке 4.3 следует той же общей форме, что и определение *main* (). Сначала идет заголовок функции. Затем в фигурных скобках следует тело функции. Форма определения функции может быть обобщена следующим образом:

type имя_функции (список_аргументов)
{
Операторы
}

Заголовки функций. Функция *simon* () из рисунка 4.3 имеет следующий заголовок:

void simon (*int n*)

Здесь *void* означает, что функция *simon* () не имеет возвращаемого значения. Поэтому при ее вызове не генерируется значение, которое можно было бы присвоить какой-то переменной в *main* ().

Определения функций располагаются в файле программы последовательно, как показано на рисунке 4.4.

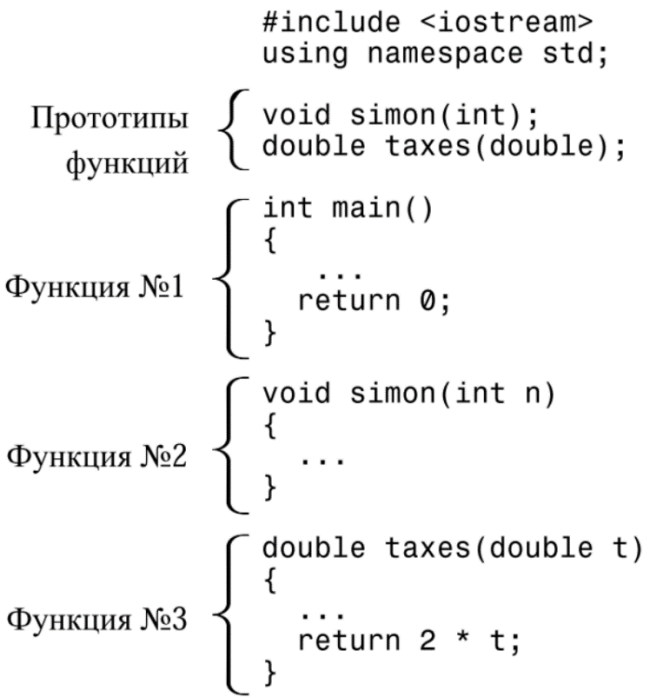

Рисунок 4.4 — Определения функций в файле программы

Функции, не возвращающие значений, называются функциями типа *void* и имеют следующую общую форму:

void ИмяФункции (Список Параметров)
{
оператор (ы)
return; // не обязательно
}

Здесь «Список Параметров» указывает типы и количество аргументов (параметров), передаваемых функции. Необязательный оператор *return* отмечает конец функции. При его отсутствии функция завершается на закрывающей фигурной скобке.

Функции с возвращаемыми значениями требуют использования оператора *return* таким образом, чтобы вызывающей функции было возвращено значение (рисунок 4.5).

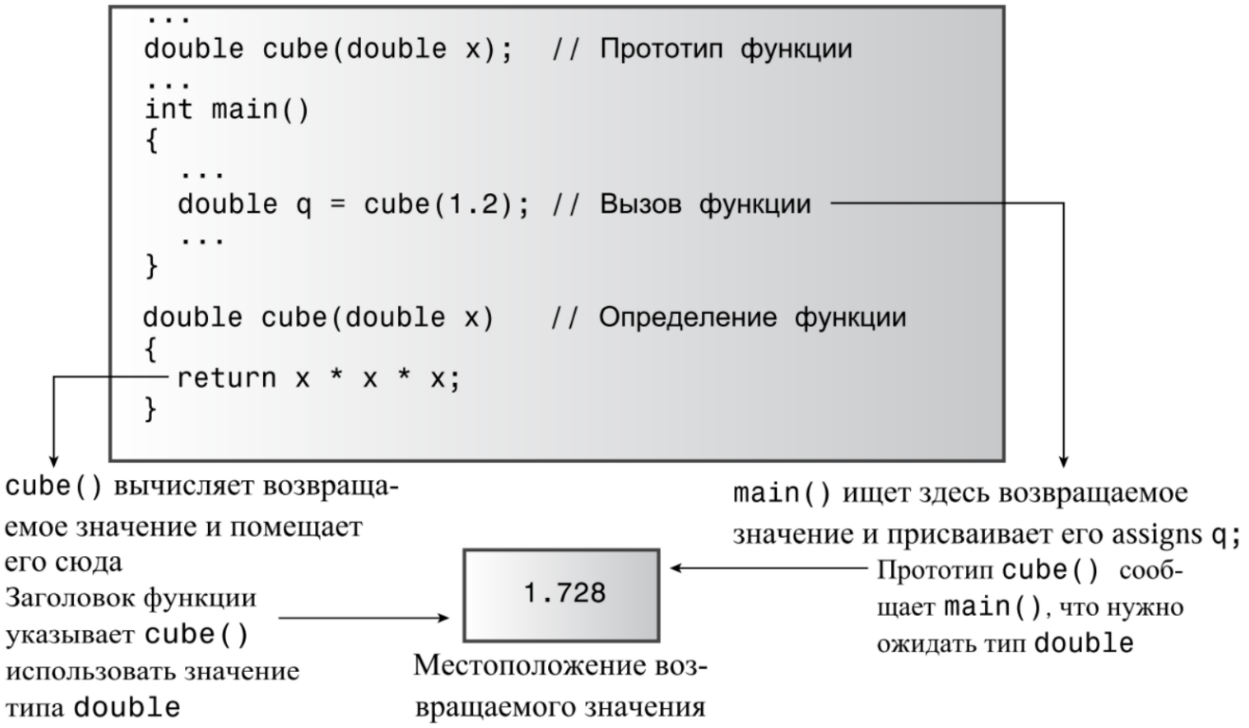

Рисунок 4.5 — Типичный механизм возврата значений

Прототипирование и вызов функции. Прототип описывает интерфейс функции для компилятора. Это значит, что он сообщает компилятору, каков тип возвращаемого значения, если оно есть у функции, а также количество и типы аргументов данной функции.

Рассмотрим для примера, как влияет прототип на вызов функции:

double volume = cube (*side*);

Во-первых, прототип сообщает компилятору, что функция *cube* () должна принимать один аргумент типа *double*. Если программа не предоставит этот аргумент, то прототипирование позволит

компилятору перехватить такую ошибку. Во-вторых, когда функция *cube* () завершает вычисление, она помещает возвращаемое значение в некоторое определенное место, возможно, в регистр центрального процессора, а может быть и в память. Затем вызывающая функция — *main* () в данном случае — извлекает значение из этого места. Поскольку прототип устанавливает, что *cube* () имеет тип *double*, компилятор знает, сколько байт следует извлечь и как их интерпретировать. Без этой информации он может только предполагать, а это то, чем заниматься он не должен.

Прототип функции является оператором, поэтому он должен завершаться точкой с запятой. Простейший способ получить прототип — скопировать заголовок функции из ее определения и добавить точку с запятой.

Прототипы значительно снижают вероятность допущения ошибок в программе. В частности, они обеспечивают следующие моменты:

– компилятор корректно обрабатывает возвращаемое значение;

– компилятор проверяет, указано ли правильное количество аргументов;

– компилятор проверяет правильность типов аргументов. Если тип не подходит, компилятор преобразует его в правильный, когда это возможно.

Прототипирование происходит во время компиляции и называется статическим контролем типов.

Аргументы функций и передача по значению. Аргументы функций в *C++* обычно передаются по значению. Это означает, что числовое значение аргумента передается в функцию, где присваивается новой переменной.

Переменная, которая используется для приема переданного значения, называется формальным аргументом, или формальным параметром. Значение, переданное функции, называется фактическим аргументом, или фактическим параметром. Чтобы немного упростить ситуацию, в стандарте *C++* слово аргумент используется для обозначения фактического аргумента, или параметра, а слово параметр для

обозначения формального аргумента, или параметра. Применяя эту терминологию, можно сказать, что передача аргумента инициализирует параметр значением этого аргумента (рисунок 4.6).

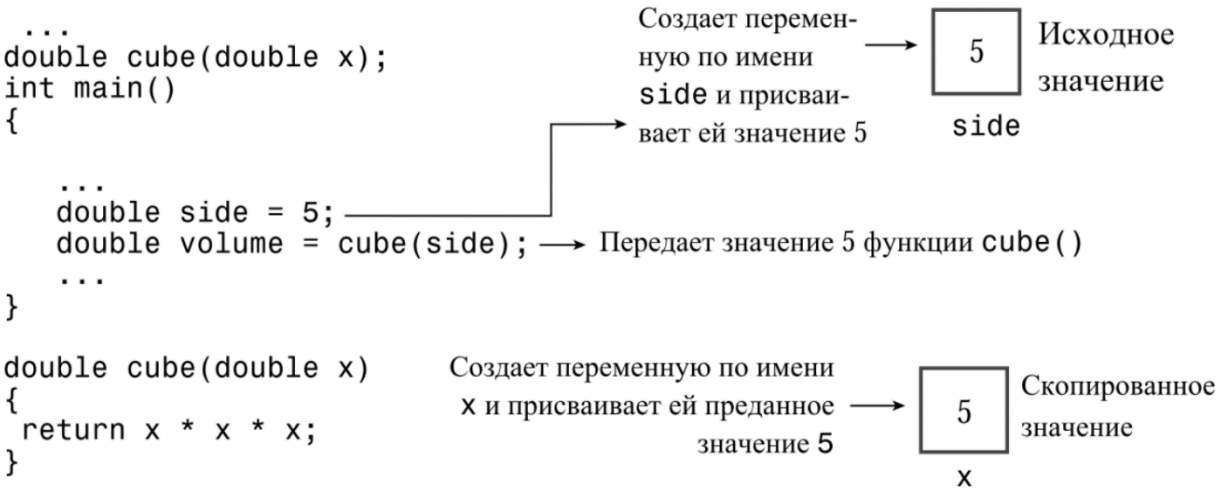

Рисунок 4.6 — Передача по значению

Переменные, включая параметры, объявленные в функции, являются приватными по отношению к этой функции. Когда функция вызывается, компьютер выделяет память, необходимую для этих переменных. Когда функция завершается, компьютер освобождает память, которая была использована этими переменными. Такие переменные называются локальными переменными, потому что они локализованы в пределах функции. Как уже упоминалось ранее, это помогает предохранить целостность данных и означает, что если вы объявили переменную *x* в *main* (), а другую переменную *x* в какой-то другой функции, то это будут две совершенно разные, никак не связанные друг с другом переменные (рисунок 4.7). Такие переменные также называются автоматическими переменными, поскольку размещаются и освобождаются автоматически во время выполнения программы.

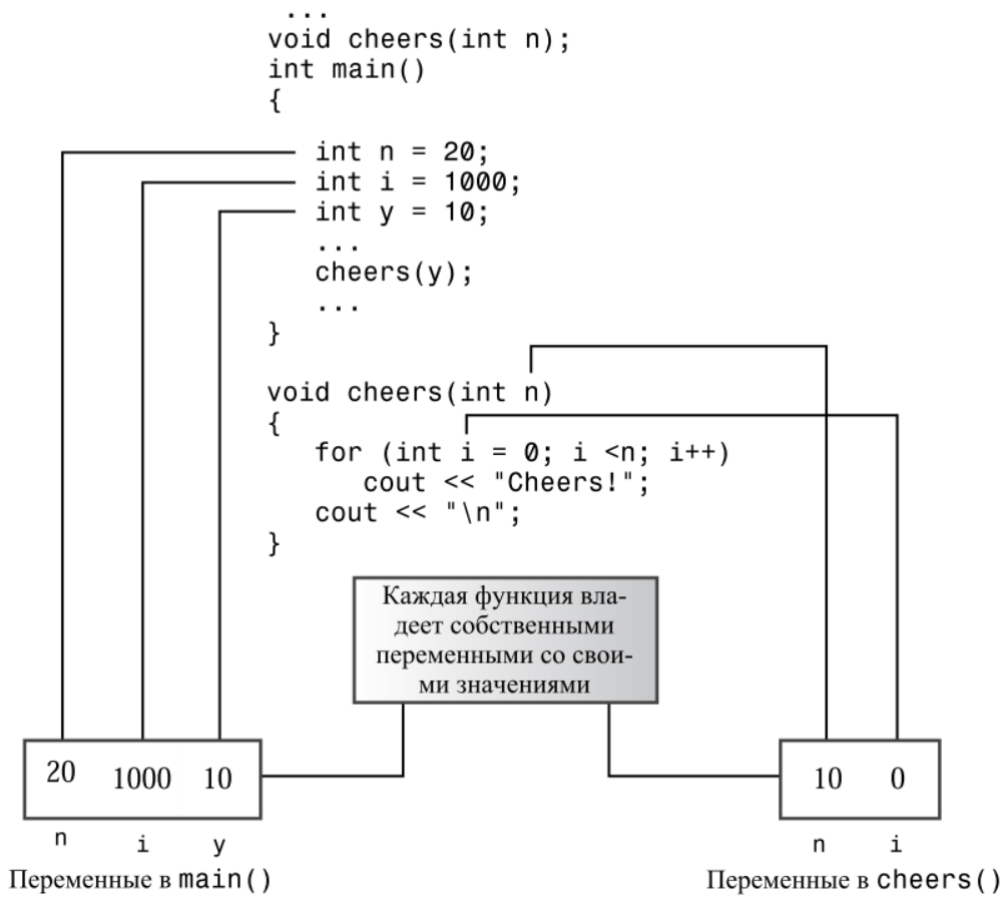

Рисунок 4.7 — Локальные переменные

Множественные аргументы. Функция может принимать более одного аргумента. При вызове функции такие аргументы просто отделяются друг от друга запятыми:

n_chars('*R*', 25) ;

Это передает два аргумента функции *n chars* (), определение которой будет приведено чуть позже.

Аналогично, при определении функции используется разделенный запятыми список параметров в ее заголовке:

void n_chars(*char c*, *int n*) // два параметра

Этот заголовок устанавливает, что функция *n chars* () принимает один параметр типа *char* и один — типа *int*.

Функции и массивы. Функции могут служить инструментами для обработки более сложных типов, таких как массивы и структуры.

int sum_arr(*int arr*[], *int n*) // *arr* = имя массива, *n* = размер

Квадратные скобки указывают на то, что *arr* — массив, а тот факт, что они пусты, говорит о том, что эту функцию можно применять с массивами любого размера. Остальную часть функции можно записать так, как если бы аргумент *arr* был массивом (рисунок 4.8).

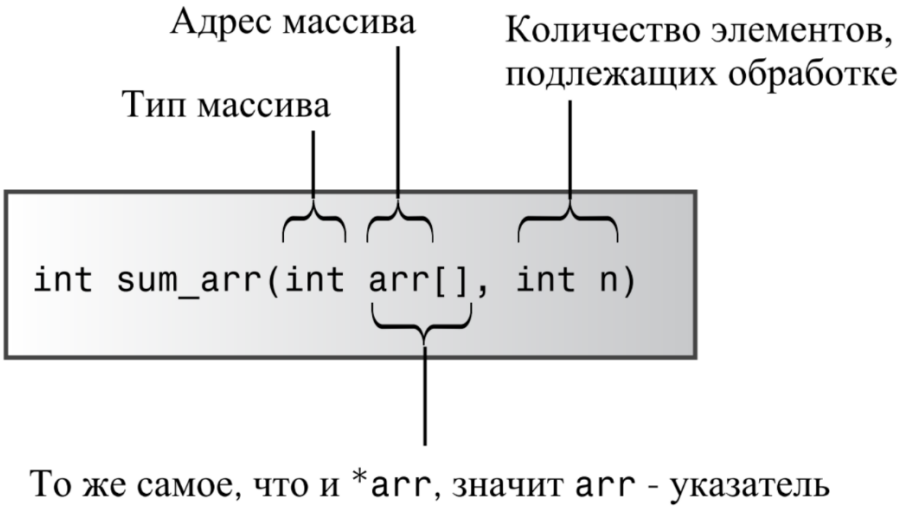

Рисунок 4.8 — Передача функции информации о массиве

Отображение массива и защита его посредством *const.* Построить функцию для отображения содержимого массива очень просто. Ей передается имя массива и количество заполненных элементов, а она использует цикл отображения каждого из них. Но есть еще одно обстоятельство, нужно гарантировать, что функция отображения не внесет в исходный массив никаких изменений. Если только назначение функции не предусматривает внесения изменений в переданные ей данные, вы должны каким-то образом предохранить ее от этого. Такая защита обеспечивается автоматически для обычных аргументов, потому что *C++* передает их по значению, и функция имеет дело с копиями. Но функция, работающая с массивом, обращается к оригиналу. В конце концов, именно поэтому предыдущая функция *fill_array* () в состоянии выполнять свою работу. Чтобы предотвратить случайное изменение содержимого массива-аргумента, при объявлении формального аргумента можно применить ключевое слово *const*:

void show_array(*const double ar*[], *int n*) ;

Это объявление устанавливает, что указатель *ar* указывает на константные данные. Это значит, что использовать *ar* для изменения данных нельзя. То есть обратиться к такому значению, как к *ar* [0], можно, но изменить его не получится.

Указатели и спецификатор *const.* Использование ключевого слова *const* с указателями характеризуется рядом тонких моментов, поэтому присмотримся к нему повнимательнее. Применять ключевое слово *const* с указателями можно двумя способами. Первый — заставить указатель указывать на константный объект, тем самым предотвращая модификацию объекта через указатель. Второй способ — сделать сам указатель константным, запретив переустанавливать его на что-нибудь другое. Теперь обратимся к деталям.

Сначала объявим *pt* как указатель на константу:

```
int age = 39;
const int * pt = &age;
```

Это объявление устанавливает, что *pt* указывает на *const int* (в данном случае — 39). Таким образом, не можем использовать *pt* для изменения этого значения. Другими словами, значение **pt* является константным и не может быть изменено:

```
*pt += 1; // неправильно, потому что pt указывает на const int;
cin>> *pt; // неправильно по той же причине.
```

Такое объявление *pt* не обязательно значит, что значение, на которое он указывает, действительно является константой; это значит лишь, что значение постоянно, только когда к нему обращаются через *pt* (рисунок 4.9). Например, *pt* указывает на *age*, а age — не константа. Можем изменить значение *age* непосредственно, используя переменную *age*, но не можем изменить это значение через указатель *pt*:

```
*pt = 20; // неправильно, потому что pt указывает на const int;
age = 20; // правильно, потому что age не объявлено как const.
```

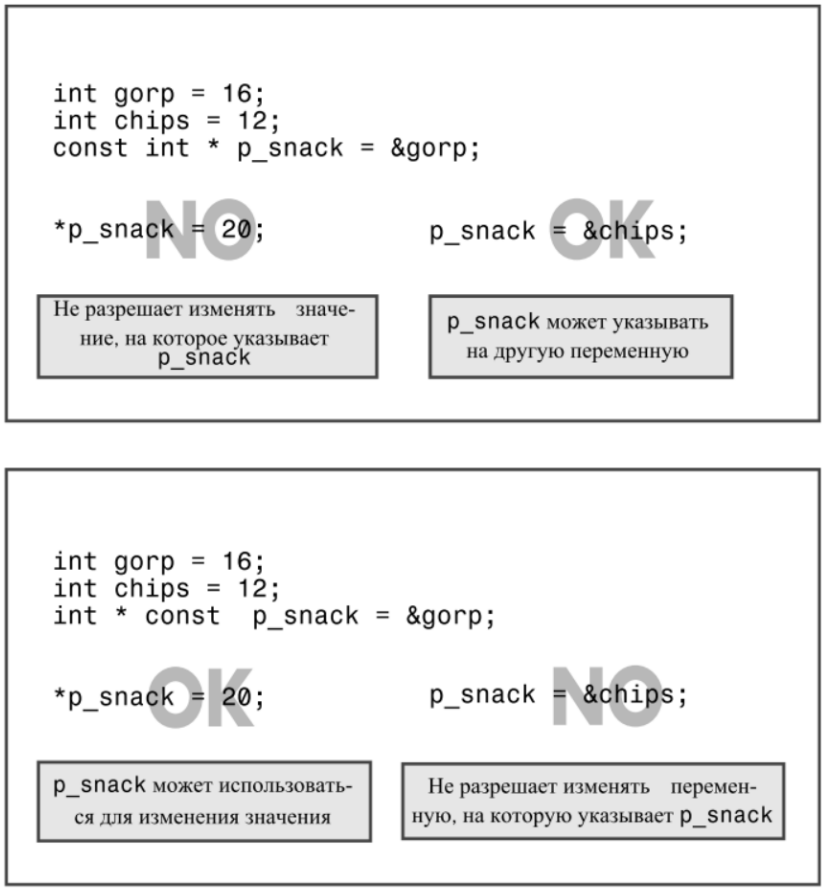

Рисунок 4.9 — Указатели па константы и константные указатели

Функции с аргументами-строками в стиле *C*

Предположим, что требуется передать строку функции в виде аргумента. Доступны три варианта представления строки:

– массив *char;*

– константная строка в двойных кавычках (также называемая строковым литералом);

– указатель на *char*, установлены в адрес начала строки.

Все три варианта являются типом указателя на *char*, поэтому все три можно использовать в качестве аргументов функций, обрабатывающих строки:

char ghost[15] = "*galloping*"; *char* * *str* = "*galumphing*";
int nl = *strlen* (*ghost*) ; // *ghost* – это *&ghost*[0]
*intn*2 = *strlen* (*str*); // указатель на *char*
int n3 = *strlen*("*gamboling*"); // адрес строки

Можно сказать, что передаем строку как аргумент, но на самом деле передаем адрес ее первого символа. Это подразумевает, что

прототип строковой функции должен использовать *char** как тип формального параметра, представляющего строку.

Функции, возвращающие строки в стиле *C*. Теперь предположим, что требуется написать функцию, возвращающую строку. Конечно, функция может это сделать. Но она может вернуть адрес строки, и это наиболее эффективно. Например, определяет функцию *buildstr* (), возвращающую указатель. Эта функция принимает два аргумента: символ и число. Используя *new*, она создает строку, длина которой равна переданному числу, и инициализирует каждый ее элемент значением переданного символа. Затем она возвращает указатель на эту новую строку.

Функции и структуры. Самый прямой способ использования структуры в программе — это трактовать их так же, как обычные базовые типы, т. е. передавать в виде аргументов, и если нужно, то использовать их в качестве возвращаемых значений. Однако существует один недостаток в передаче структур по значению. Если структура велика, то затраты, необходимые для создания копии структуры, могут значительно увеличить потребности в памяти и замедлить работу программы. По этой причине (и еще потому, что изначально язык *C* не позволял передавать структуры по значению) многие программисты на *C* предпочитают передавать адрес структуры и затем использовать указатель для доступа к ее содержимому. Рассмотрим два первых варианта, начиная с передачи и возврата целых структур.

Передача адресов структур. Если нужно сэкономить время и пространство памяти за счет передачи адресов структуры вместо самой структуры, то потребуется переписать функции так, чтобы они использовали в качестве аргументов указатели на структуры. Давайте посмотрим, как можно переписать функцию *show_polar* (). Для этого понадобится внести три изменения:

– при вызове функции передать ей адрес структуры (*&pplace*) вместо самой структуры (*pplace*);

– определить формальный параметр как указатель на структуру *polar*, т. е. *polar**. Поскольку функция не должна модифицировать структуру, дополнительно задать модификатор *const*;

– поскольку формальный параметр теперь будет указателем на структуру вместо самой структуры, использовать операцию вместо операции точки.

Функции и объекты класса *string.* Хотя строки в стиле *C* и класс *string* служат в основном одним и тем же целям, класс *string* больше похож на структуру, чем на массив. Например, структуру можно присвоить другой структуре, а объект — другому объекту. Структуру можно передавать как единую сущность в функцию, и точно так же можно передавать объект. Когда требуется несколько строк, можно объявить одномерный массив объектов *string* вместо двумерного массива *char*.

Функции и объекты *array.* Объекты классов в *C++* основаны на структурах, поэтому некоторые из соглашений, принятых для структур, применимы также и к классам. Например, функции можно передать объект по значению и тогда она будет действовать на копии исходного объекта. В качестве альтернативы можно передать указатель на объект, что позволит функции оперировать на исходном объекте.

Рекурсия. Функция *C++* обладает интересной характеристикой — она может вызывать сама себя. Эта возможность называется рекурсией. Рекурсия — важный инструмент в некоторых областях программирования, например таких, как искусственный интеллект.

Рекурсия с одиночным рекурсивным вызовом. Если рекурсивная функция вызывает саму себя, затем этот новый вызов снова вызывает себя и т. д., то получается бесконечная последовательность вызовов, если только код не включает в себе нечто, что позволит завершить эту цепочку вызовов. Обычный метод состоит в том, что рекурсивный вызов помещается внутрь оператора *if.* Например, рекурсивная функция типа *void* по имени *recurs* () может иметь следующую форму:

voidrecurs(список Аргументов)
{операторы1 *if* (проверка)
recurs (аргументы) операторы2}

5 УКАЗАТЕЛИ НА ФУНКЦИИ. ВСТРОЕННЫЕ ФУНКЦИИ. ССЫЛОЧНЫЕ ПЕРЕМЕННЫЕ

Указатели на функции

Функции, как и элементы данных, имеют адреса. Адрес функции — это адрес в памяти, где находится начало кода функции на машинном языке. Обычно пользователю ни к чему знать этот адрес, но может быть полезно для программы. Например, можно написать функцию, которая принимает адрес другой функции в качестве аргумента, что позволяет первой функции найти вторую и запустить ее. Такой подход сложнее, чем простой вызов второй функции из первой, но он открывает возможность передачи разных адресов функций в разные моменты времени. То есть первая функция может вызывать разные функции в разное время.

Получение адреса функции. Получить адрес функции очень просто: используйте имя функции без скобок. То есть, если имеется функция *think*(), то ее адрес записывается как *think*. Чтобы передать функцию в качестве аргумента, нужно просто передать ее имя. Удостоверьтесь в том, что понимаете разницу между адресом функции и передачей ее возвращаемого значения:

process(*think*); // передача адреса *think*() функции *process*();
thought(*think*()); // передача возвращаемого значения *think*() функции *thought*().

Вызов *process*() позволяет внутри этой функции вызвать функцию *think*(). Вызов *thought* () сначала вызывает функцию *think*() и затем передает возвращаемое ею значение функции *thought*().

Объявление указателя на функцию. Чтобы объявить указатель на тип данных, нужно явно задать тип, на который будет указывать этот указатель. Аналогично указатель на функцию должен определять, на функцию какого типа он будет указывать. Это значит, что объявление должно идентифицировать тип возврата функции и ее сигнатуру (список аргументов). То есть объявление должно предо-

ставлять ту же информацию о функции, которую предоставляет и ее прототип.

Например, предположим, что одна из функций для оценки затрат времени имеет следующий прототип:

double pam(*int*); // прототип.

Вот как должно выглядеть объявление соответствующего типа указателя:

double (**pf*)(*int*); // *pf* указывает на функцию, которая принимает один аргумент типа *int* и возвращает тип *double.*

Объявление требует скобок вокруг **pf*, чтобы обеспечить правильный приоритет операций. Скобки имеют более высокий приоритет, чем операция «*», поэтому

**pf (int)* означает, что *pf()* — функция, которая возвращает указатель, в то время как *(*pf)(int)* означает, что *pf* — указатель на функцию:

double (**pf*)(*int*); // *pf* указывает на функцию, возвращающую *double*;

double **pf*(*int*); // *pf*() — функция, возвращающая указатель на *double.*

После соответствующего объявления указателя *pf* ему можно присваивать адрес подходящей функции:

double pam(*int*); *double* (**pf*) (*int*);
pf = *pam*; // *pf* теперь указывает на функцию *pam*().

Использование указателя для вызова функции. Теперь обратимся к завершающей части этого подхода — использованию указателя для вызова указываемой им функции. Ключ к этому находится в объявлении указателя. Вспомним, что там (**pf*) играет ту же роль, что имя функции. Поэтому все, что потребуется сделать — использовать (**pf*), как если бы это было имя функции:

double pam(*int*); *double* (**pf*) (*int*);
pf = pam; // *pf* теперь указывает на функцию *pam*();
double x = *pam*(4); // вызвать *pam*(), используя ее имя;
double y = (**pf*)(5); // вызвать *pam*(), используя указатель *pf*.

В действительности *C++* позволяет использовать *pf*, как если бы это было имя функции:

double y = *pf* (5); // также вызывает *pam*(), используя указатель *pf*.

Встроенные функции *C++*

Встроенные функции являются усовершенствованием языка *C++*, предназначенным для ускорения работы программ. Основное различие между встроенными и обычными функциями связано не с написанием кода, а с тем, каким образом компилятор внедряет функцию в программу.

Встроенные функции *C++* предоставляют альтернативу. Скомпилированный код такой функции непосредственно встраивается в код программы. Иначе говоря, компилятор подставляет вместо вызова функции ее код. В результате программе не нужно выполнять переход к другому адресу и возвращаться назад. Таким образом, встраиваемые функции выполняются немного быстрее, чем обычные, однако за это нужно платить дополнительным расходом памяти. Если в десяти различных местах программа выполняет вызов одной и той же встроенной функции, ее код будет содержать десять копий этой функции (рисунок 5.1).

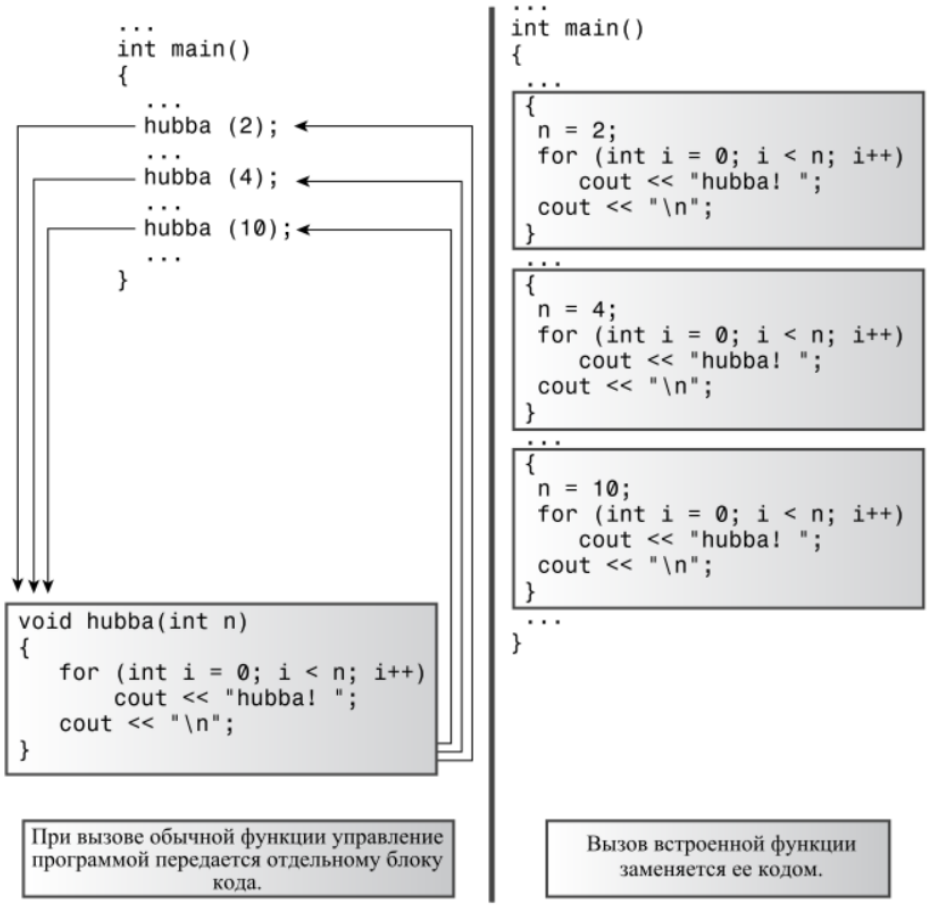

Рисунок 5.1 — Различия между встроенными и обычными функциями

Чтобы воспользоваться встроенной функцией, нужно предварить объявление функции ключевым словом *inline*.

Общепринято опускать прототип и помещать полное описание (заголовок и весь код функции) туда, где обычно находится прототип.

Ссылочные переменные

Язык *C++* вводит в практику новый составной тип данных — ссылочную переменную. Ссылка представляет собой имя, которое является псевдонимом, или альтернативным именем, для ранее объявленной переменной. Ссылки представляют собой удобную альтернативу указателям при обработке крупных структур посредством функций. Они играют важную роль при создании классов.

Создание ссылочных переменных. Как уже упоминалось, в языках *C* и *C++* символ «&» используется для обозначения адреса переменной. Язык *C++* придает символу «&» дополнительный смысл и задействует его для объявления ссылок. Например, чтобы *rodents* стало альтернативным именем для переменной *rats*, необходимо написать следующее:

```
int rats;
int & rodents = rats; // rodents становится псевдонимом имени rats.
```

В таком контексте символ «&» не является операцией взятия адреса. В этом случае «&» воспринимается как часть идентификатора типа данных. Подобно тому как выражение *char** в объявлении означает указатель на *char*, выражение *int*& представляет собой ссылку на *int*. Объявление ссылки позволяет взаимозаменяемо использовать идентификаторы *rats* и *rodents*. Они ссылаются на одно и то же значение, а также на один и тот же адрес памяти.

Ссылки как параметры функций. Чаще всего ссылки используются в качестве параметров функции, при этом имя переменной в функции становится псевдонимом переменной в вызывающей программе. Такой метод передачи аргументов называется передачей по ссылке. Передача параметров по ссылке позволяет вызываемой функции получить доступ к переменным в вызывающей функции. Реализация этого средства в *C++* представляет собой дальнейшее развитие

основных принципов языка *C*, где возможна только передача по значению. Передача по значению приводит к тому, что вызываемая функция оперирует копиями значений из вызывающей программы (рисунок 5.2).

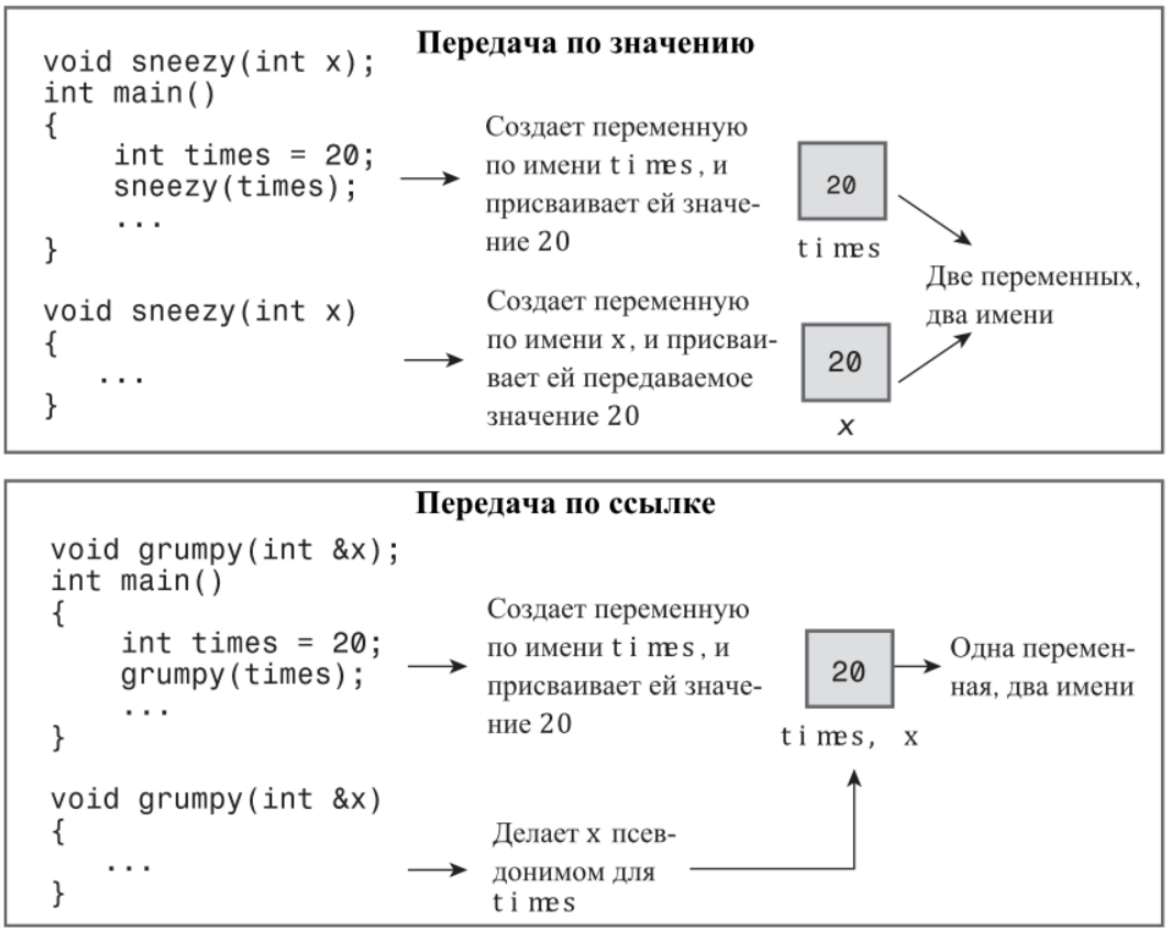

Рисунок 5.2 — Передача по значению и передача по ссылке

Временные переменные, ссылочные аргументы и квалификатор *const.* *C++* может генерировать временную переменную, если фактический аргумент не соответствует ссылочному аргументу. В настоящее время *C++* допускает это только в случае, когда аргумен-том является ссылка с квалификатором *const*, но не всегда.

Временная переменная создается при условии, что ссылочный параметр является *const*, компилятор генерирует временную переменную в двух ситуациях:

– тип фактического аргумента выбран правильно, но сам параметр не является *lvalue*;

– тип фактического параметра выбран неправильно, но может быть преобразован в правильный тип.

Что такое *lvalue*? Аргумент, являющийся *lvalue*, представляет собой объект данных, на который можно ссылаться по адресу. Например, переменная, элемент массива, член структуры, ссылка и разыменованный указатель — все они являются *lvalue.*

Использование ссылок при работе со структурами. Метод использования ссылки на структуру в качестве параметра функции ничем не отличается от метода применения ссылки на базовую переменную: при объявлении параметра структуры достаточно воспользоваться операцией ссылки «&». Например, предположим, что есть следующее определение структуры:

```
struct free_throws {
std::string name;
int made;
int attempts;
float percent;
};
```

Затем функция, использующая ссылку на этот тип, может иметь такой прототип:

void set_pc(*free_throws* & *ft*); // использование ссылки на структуру.

Если функция не должна изменять структуру, необходимо применить:

const: void display(*const free_throws* & *ft*); // не разрешать изменения структуры.

Аргументы по умолчанию. Аргумент по умолчанию представляет собой значение, которое используется автоматически, если соответствующий фактический параметр в вызове функции не указан. Например, если функция *void wow* (*int n*) определена так, что *n* по умолчанию имеет значение 1, то вызов функции *wow* () означает то же самое, что и *wow* (1). Это свойство позволяет использовать функции более гибким образом. Предположим, что функция *left* () возвращает первые *n*символов строки, при этом сама строка и число *n* являются аргументами. Точнее, функция возвращает указатель на новую

строку, представляющую собой выбранный фрагмент исходной строки.

Необходимо установить значение по умолчанию, для этого применяется прототип функции. Поскольку компилятор использует прототип, чтобы узнать, сколько аргументов имеет функция, прототип функции также должен сообщить программе о возможности наличия аргументов по умолчанию. Метод заключается в присваивании значения аргументу в самом прототипе.

Перегрузка функций. Полиморфизм функций — это удобное добавление *C++* к возможностям языка *C*. В то время как аргументы по умолчанию позволяют вызывать одну и ту же функцию с различным количеством аргументов, полиморфизм функций, также называемый перегрузкой функций, предоставляет возможность использовать несколько функций с одним и тем же именем. Слово полиморфизм означает способность иметь множество форм, следовательно, полиморфизм функций позволяет функции иметь множество форм. Подобным же образом выражение перегрузка функций означает возможность привязки более чем одной функции к одному и тому же имени таким образом перегружая имя. Оба выражения означают одно и то же, но мы будем пользоваться вариантом перегрузка функций, как более строгим. С применением перегрузки функций можно разработать семейство функций, которые выполняют в точности одно и то же, но с использованием различных списков аргументов.

6 КЛАССЫ ПАМЯТИ, ДИАПАЗОНЫ ДОСТУПА И СВЯЗЫВАНИЕ. ПРОСТРАНСТВА ИМЕН

Классы памяти

Категории хранения влияют на то, как информация может совместно использоваться разными файлами. В языке *C++* применяются три схемы хранения данных. Эти схемы отличаются продолжительностью нахождения данных в памяти:

– автоматическая продолжительность хранения. Переменные, объявленные внутри определения функции, включая параметры функции, имеют автоматическую продолжительность хранения. Они создаются, когда выполнение программы входит в функцию или блок, где эти переменные определены. После выхода из блока или функции используемая переменными память освобождается. В *C++* существуют два вида автоматических переменных;

– статическая продолжительность хранения. Переменные, объявленные за пределами определения функции либо с использованием ключевого слова *static*, имеют статическую продолжительность хранения. Они существуют в течение всего времени выполнения программы. В языке *C++* существуют три вида переменных со статической продолжительностью хранения;

– потоковая продолжительность хранения. В наши дни многоядерные процессоры распространены практически повсеместно. Такие процессоры способны поддерживать множество выполняющихся задач одновременно. Это позволяет программе разделить вычисления на отдельные потоки, которые могут быть обработаны параллельно. Переменные, объявленные с ключевым словом *thread_local*, хранятся на протяжении времени существования содержащего их потока. (Вопросы параллельного программирования в этой книге не рассматриваются.);

– динамическая продолжительность хранения. Память, выделяемая операцией *new*, сохраняется до тех пор, пока она не будет освобождена с помощью операции *delete* или до завершения программы, смотря какое из событий наступит раньше. Эта память

имеет динамическую продолжительность хранения и часто называется свободным хранилищем, или кучей.

Область видимости и связывание

Область видимости (или контекст) определяет доступность имени в пределах файла (единицы трансляции). Например, переменная, определенная в функции, может быть использована только в этой функции, но ни в какой-либо другой, в то время как переменная, определенная в файле до определений функций, может применяться во всех функциях. Связывание описывает, как имя может разделяться различными единицами трансляции. Имя с внешним связыванием может совместно использоваться разными файлами, а имя с внутренним связыванием — функциями внутри одного файла. Имена автоматических переменных не имеют никакого связывания, поскольку не являются разделяемыми.

Переменная *C++* может иметь одну из нескольких возможных областей видимости. Переменная с локальной областью видимости (которая также называется областью видимости блока) известна только внутри блока, где она определена. Вспомните, что блок — это последовательность операторов, заключенная в фигурные скобки. Например, тело функции является блоком, однако в него могут быть вложены и другие блоки. Переменная, имеющая глобальную область видимости (которая часто называется областью видимости файла), известна во всем файле, начиная с точки, где она определена. Автоматические переменные имеют локальную область видимости, а статические переменные могут иметь различную область видимости в зависимости от того, как они определены. Имена, используемые в области видимости прототипа функции, доступны только в пределах круглых скобок, которые содержат список аргументов. Элементы, объявленные в классе, имеют область видимости класса. Переменные, объявленные в пространстве имен, имеют область видимости пространства имен.

Функции *C++* могут иметь область видимости класса или область видимости пространства имен, включая глобальную область

видимости, но не могут иметь локальную область видимости. (Функция не может быть определена внутри блока, поскольку если бы она могла иметь локальную область видимости, то была бы известна только самой себе и, следовательно, не могла быть вызванной из другой функции. Такая функция вообще не могла бы считаться функцией.)

Различные варианты хранения в *C++* характеризуются продолжительностью существования, областью видимости и связыванием. Рассмотрим классы хранения *C++* в терминах их свойств. Начнем с исследования ситуации, имевшей место до ввода в язык пространств имен, и посмотрим, как они изменили общую картину.

Автоматическая продолжительность хранения. Параметры функции и переменные, объявленные внутри функции, по умолчанию имеют автоматическую продолжительность хранения. Они обладают локальной областью видимости и не имеют связывания. Другими словами, если объявить переменную по имени *texasBmain*(), а затем объявить еще одну переменную с тем же именем в функции *oil*(), будут созданы две независимые переменные, каждая из которых известна только в той функции, в которой объявлена. Любые операции с переменной *texas* в функции *oil*() не оказывают влияния на переменную *texas* в *main*() и наоборот. Кроме того, каждой переменной выделяется память, когда выполнение программы входит в самый вложенный блок, содержащий определение переменной, и каждая переменная прекращает существование, когда выполнение программы покидает этот блок.

Если определить переменную внутри блока, ее время существования и область видимости ограничиваются этим блоком. Предположим, что в начале *main*() определена переменная *teledeli*. Теперь пусть в *main*() создается новый блок, в котором определяется новая переменная по имени *websight*. В этом случае переменная *teledeli* является видимой как во внешнем, так и во внутреннем блоке, в то время как *websight* существует только во внутреннем блоке и находится в области видимости с точки своего определения до тех пор, пока выполнение программы не доберется до конца блока:

```
int main ()
{
int teledeli = 5;
{
// Переменной websight выделяется память cout << "Hello\n";
int websight = –2; // начинается область видимости websight
cout << websight << ' ' << teledeli << endl;
}
// websight прекращает существование
cout << teledeli << endl;
}
// Переменная teledeli прекращает существование
```

А что если переменной во внутреннем блоке назначить имя *teledeli* вместо *websight*, в результате чего получится две переменные с одним и тем же именем, одна из которых находится во внешнем блоке, а другая — во внутреннем. В этом случае программа интерпретирует имя *teledeli* как переменную, локальную по отношению к блоку, во время выполнения операторов этого блока. Принято говорить, что новое определение скрывает предыдущее. Новое определение попадает в область видимости, а предыдущее из нее временно удаляется. Когда выполнение программы покидает блок, исходное определение возвращается обратно в область видимости (рисунок 6.1).

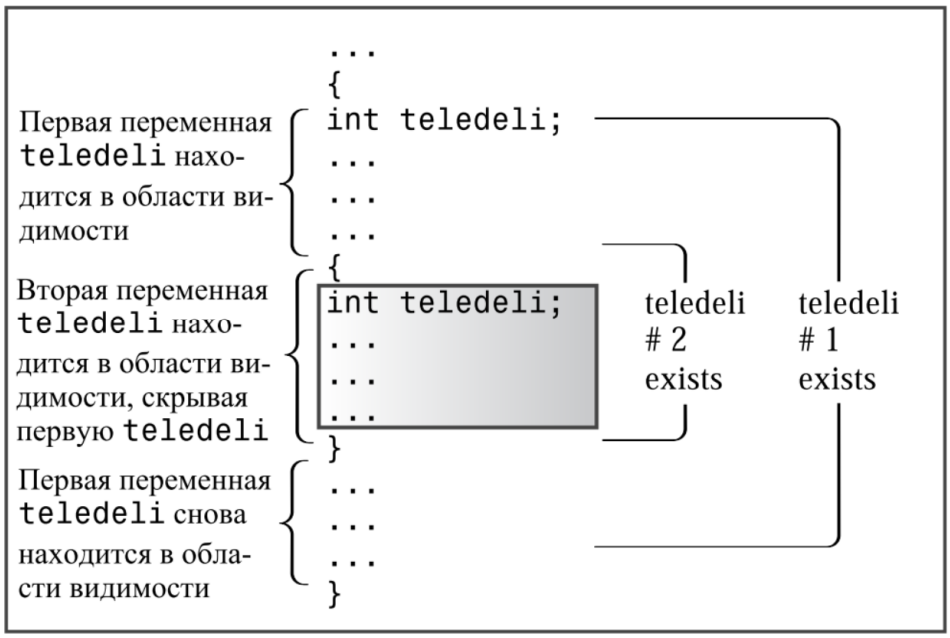

Рисунок 6.1 — Блоки и область видимости

Инициализация автоматических переменных

Автоматическую переменную можно инициализировать с помощью любого выражения, значение которого известно на момент объявления переменной. Ниже приведен пример инициализации переменных *x*, *big*, *y* и *z*:

```
int w; // значение w не определено;
int x = 5; // инициализация числовым литералом;
int big = INT_MAX – 1; // инициализация константным выражением;
int y = 2 * x; // использование ранее определенного значения x;
cin >> w;
int z = 3 * w; // использование нового значения w.
```

Автоматические переменные и стек. Чтобы получить более полное представление об автоматических переменных, рассмотрим их реализацию обычным компилятором *C++*. Поскольку количество автоматических переменных растет или сокращается по мере того, как функции начинают и завершают выполнение, программа должна управлять автоматическими переменными в процессе своей работы. Стандартная методика состоит в выделении области памяти, которая будет использоваться в качестве стека, управляющего движением переменных.

Термин стек применяется потому, что новые данные размещаются, образно говоря, поверх старых данных (т. е. в смежных, а не в тех же самых ячейках памяти), а затем удаляются из стека, после того как программа завершит работу с ними. По умолчанию размер стека зависит от реализации, однако обычно компилятор предоставляет опцию изменения размера стека.

Программа отслеживает состояние стека с помощью двух указателей. Один указывает на базу стека, с которой начинается выделенная область памяти, а другой — на вершину стека, которая представляет собой следующую ячейку свободной памяти. Когда происходит вызов функции, ее автоматические переменные добавляются в стек, а указатель вершины устанавливается на свободную ячейку памяти, следующую за только что размещенными переменными. После завершения функции указатель вершины снова принимает значение,

которое он имел до вызова функции. В результате эффективно освобождается память, которая использовалась для хранения новых переменных (рисунок 6.2).

Стек построен по принципу *LIFO* (*last-in*, *first-out* — последним пришел, первым обслужен). Это означает, что переменная, которая попала в стек последней, удаляется из него первой. Такой механизм упрощает передачу аргументов.

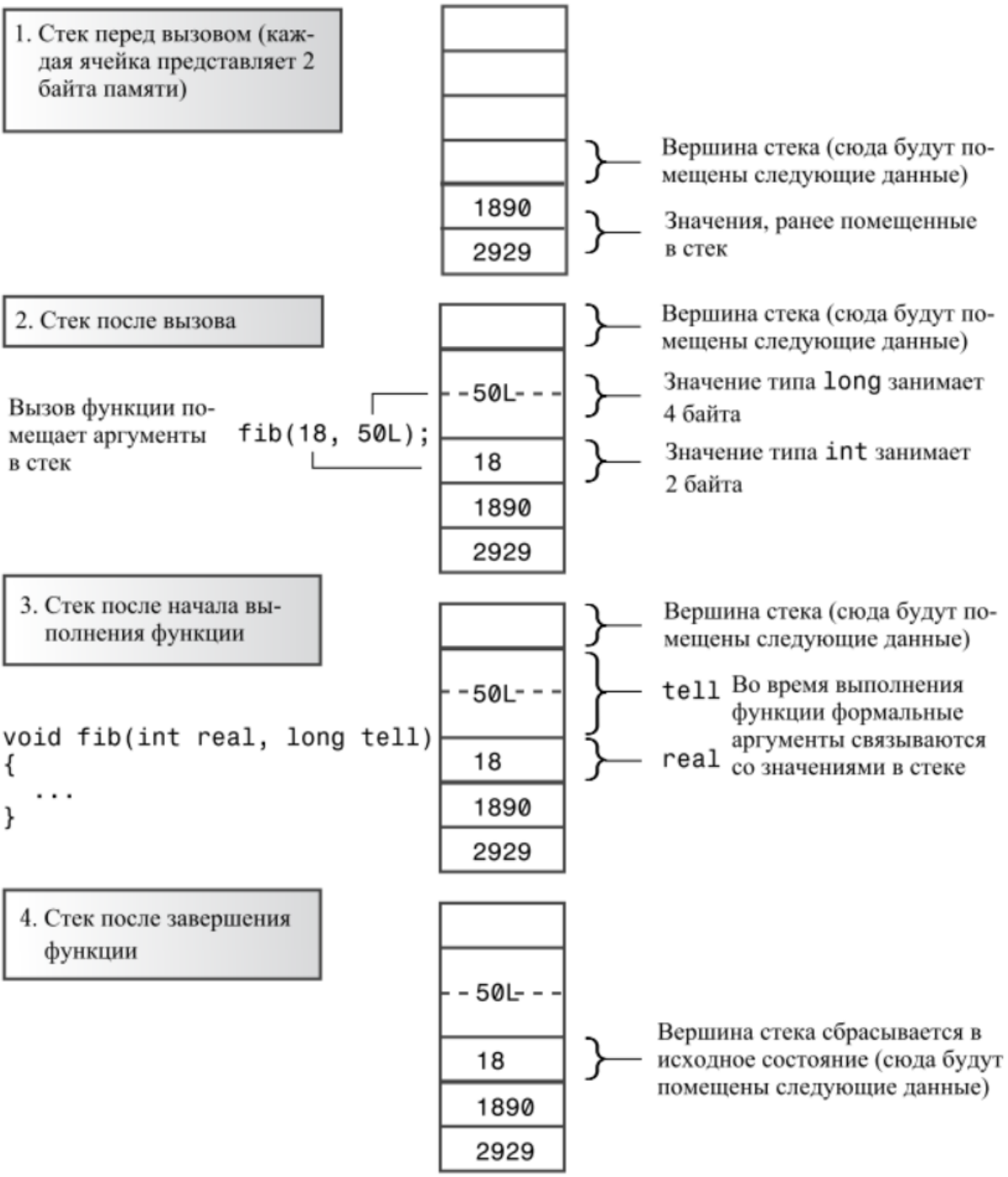

Рисунок 6.2 — Передача аргументов с использованием стека

Регистровые переменные. Ключевое слово *register* было первоначально введено в языке *C*, чтобы рекомендовать компилятору использовать для хранения автоматической переменной регистр центрального процессора:

register int count_fast; // запрос на создание регистровой переменной.

Идея заключалась в том, что это ускоряло доступ к переменной. До появления стандарта *C++* 11 это ключевое слово применялось в *C++* похожим образом, но с одним отличием. Поскольку оборудование и компиляторы стали более совершенными, эта рекомендация была обобщена и начала указывать на тот факт, что переменная интенсивно используется и, возможно, компилятор сумеет уделить ей особое внимание. В *C++* 11 эта рекомендация является устаревшей, и ключевое слово *register* остается просто способом идентифицировать переменную как автоматическую. Учитывая, что *register* может применяться только с переменными, которые будут автоматическими в любом случае, одна из причин использования этого ключевого слова — указать, что действительно нужна автоматическая переменная, возможно, с тем же самым именем, что и у внешней переменной. Точно таким же было первоначальное назначение *auto*. Однако более важная причина того, что ключевое слово *register* осталось, связана с желанием сохранить допустимым существующий код, в котором оно используется (рисунок 6.3).

Описание хранения	Продолжительность	Область видимости	Связывание	Способ объявления
Автоматическая	Автоматическая	Блок	Нет	В блоке
Регистровая	Автоматическая	Блок	Нет	В блоке, с использованием ключевого слова `register`
Статическая без связывания	Статическая	Блок	Нет	В блоке, с использованием ключевого слова `static`
Статическая с внешним связыванием	Статическая	Файл	Внешнее	Вне всех функций
Статическая с внутренним связыванием	Статическая	Файл	Внутреннее	Вне всех функций, с использованием ключевого слова `static`

Рисунок 6.3 — Пять видов хранения переменных

Инициализация статических переменных. Статические переменные могут быть инициализированными нулями, они могут быть подвергнуты инициализации константным выражением, и они могут быть подвергнуты динамической инициализации. Инициализация нулями означает установку переменной в значение ноль. Для скалярных типов ноль предусматривает приведение к соответствующему типу. Например, нулевой указатель, который представлен как 0 в коде *C++*, может иметь ненулевое внутреннее представление, поэтому переменная типа указателя будет инициализирована этим значением. Члены структуры являются инициализированными нулями, и любой заполняющий бит установлен в ноль.

Инициализация нулями и инициализация константным выражением вместе называются статической инициализацией. Это значит, что переменная инициализируется, когда компилятор обрабатывает файл (или единицу трансляции). Динамическая инициализация означает, что переменная инициализируется позже.

Переменные с внешним связыванием часто называются просто внешними переменными. Они обязательно имеют статическую продолжительность хранения и область видимости файла. Внешние переменные определяются вне всех функций и поэтому являются внешними по отношению к любой функции. Например, они могут быть объявлены до описания функции *main* () или в заголовочном файле. Внешнюю переменную можно использовать в любой функции, которая следует в файле после определения переменной. Поэтому внешние переменные также называются глобальными, в отличие от автоматических переменных, которые являются локальными.

7 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. ОБЪЕКТЫ И КЛАССЫ

Объектно-ориентированное программирование

Объектно-ориентированное, или **объектное**, **программирование** (ООП) — парадигма программирования, в которой основными концепциями являются понятия **объект** и **класс**.

Класс — это тип, описывающий устройство объектов. Понятие «класс» подразумевает некоторое поведение и способ представления. Понятие «объект» подразумевает нечто, что обладает определённым поведением и способом представления. Говорят, что объект — это экземпляр класса. Класс можно сравнить с чертежом, согласно которому создаются объекты.

Наиболее важными инструментальными средствами ООП являются:

– абстрагирование;

– инкапсуляция и сокрытие данных;

– полиморфизм;

– наследование;

– повторное использование программных кодов.

Класс представляет собой единственное наиболее важное усовершенствование языка *C++*, предназначенное для реализации этих свойств и связывания их в единое целое.

Абстрагирование и классы

Жизнь полна сложных вещей, и один из способов решения возникающих сложных проблем заключается в построении упрощающих абстракций (Абстракция, или абстракт (от лат. *abstractio* — «отвлечение».) — отвлечение в процессе познания от несущественных сторон, свойств, связей предмета или явления с целью выделения их существенных, закономерных признаков.) В теории вычислительных систем абстракция является решающим шагом в представлении информации и ее взаимодействии с пользователем. Иначе говоря, вы создаете абстрактное представление самых важных операционных

особенностей проблемы и формулируете решение этих проблем, используя одни и те же термины.

Что представляет собой тип? Давайте подумаем над тем, из чего состоит тип. Сначала вы склонны думать о типе данных, опираясь на какие-то внешние признаки, — как они хранятся в памяти. Тип данных *char*, например, занимает один байт памяти, а тип *double* довольно часто занимает восемь байтов памяти. Однако непродолжительные размышления приводят нас к заключению, что тип данных определяется в терминах операций, которые могут быть выполнены над этими данными. Например, тип *int* может быть использован всеми арифметическими операциями. Вы можете складывать, вычитать, умножать и делить целые числа. Вы также можете применять к целым числам операцию деления по модулю (*%*).

А теперь рассмотрим указатели. Указатель может затребовать не больше памяти, чем тип данных *int*. Он вполне может иметь внутреннее представление в виде целого числа. Однако указатель не допускает выполнения операций, возможных при работе с целыми числами, например взять два указателя и перемножить их друг с другом. Такое действие не имеет смысла, поэтому в *C++* оно не реализовано. Таким образом, когда вы объявляете переменную как тип *int* или как указатель на переменную типа *float*, вы не только выделяете ей память, но и устанавливаете, какие операции могут быть выполнены над этой переменной. Одним словом, спецификация основного типа позволяет определить:

– какой объем памяти необходим для размещения соответствующего объекта данных;

– какие операции или методы могут быть выполнены над объектом данных.

Для встроенных типов данных эта информация заложена в компилятор. Но когда в *C++* задается тип, определяемый пользователем, нужно самостоятельно определить эту информацию.

Класс. В языке *C++* класс представляет собой платформу для перевода абстракции в тип, определяемый пользователем. Он сочетает в себе представление данных с методами упорядочивания этих данных в компактные пакеты. Рассмотрим класс, который представ-

ляет пакет акций. Ограничим перечень операций, которые нам предстоит выполнять с акциями, следующим списком:

– получить пакет в компании;

– приобрести дополнительные акции того же пакета;

– продать пакет;

– корректировать среднюю стоимость одной акции пакета;

– отображать данные о пакете акций.

Для поддержки этого интерфейса нужно хранить некоторые виды информации, применим упрощенный подход — будем хранить такую информацию:

– наименование компании;

– число акций в пакете;

– цену каждой акции;

– общую стоимость всех акций пакета.

Дадим определения этого класса. В большинстве случаев спецификация класса состоит из двух частей:

– объявления класса, в котором описаны компоненты данных с помощью терминов элементов данных, и общедоступный интерфейс, описанный на языке функций-элементов;

– определений методов класса, которые описывают, как реализуются конкретные функции-элементы данного класса.

Проще говоря, определение класса является общим представлением класса, в то время как определения методов добавляют необходимые подробности. В программе, представленной в листинге 7.1, приводится объявление экспериментального класса *Stock*. (Чтобы легче можно было отличать классы от других объектов, мы будем следовать вполне обычному, но отнюдь не универсальному соглашению, предусматривающему написание имен классов с заглавной буквы.) Вы убедитесь в том, что оно во многом подобно объявлению структуры, в которую добавлены дополнительные элементы.

Листинг 7.1 Первая часть файла stocks.cpp

```
// начало файла stocks.cpp file
class Stock // объявление класса
{
private:
```

```
char company[30];
int shares;
double share_val;
double total_val;
void set_tot()
{ total_val = shares * share_val; };
public:
void acquire(const char * co, int n, double pr) ;
void buy(int num, double price);
void sell(int num, double price);
void update(double price);
void show() ;
}; // обратите внимание на точку с запятой в конце
```

Рассмотрим общие свойства класса. Ключевое слово *class* в языке *C++* идентифицирует код, определяющий конструкцию класса. Синтаксис отождествляет имя *Stock* с именем типа этого нового класса. Это объявление позволяет нам объявлять переменные, именуемые объектами, или экземплярами, типа *Stock*. Каждый отдельный объект представляет собой отдельный вклад. Например, объявления

```
Stock sally;
Stock solly;
```

создают два объекта типа *Stock* с именами *sally* и *solly*. Объект *sally*, например, мог представлять вклады *Sally* в конкретной компании.

Обратите внимание на то обстоятельство, что информация, которую мы решили сохранить, поступает в форме элементов данных класса, таких как *company* и *shares*. Элемент данных company объекта *sally*, например, содержит имя компании, элемент данных *share* — число акций, которыми владеет Салли, элемент *share_val* — стоимость каждой акции, а элемент *total_val* — значение общей стоимости всех акций.

То же самое можно сказать и о тех операциях, которые по нашему замыслу должны быть представлены в виде функций-элементов, таких как *sell*() и *update*(). Функции-элементы класса получили название *методов* класса. Вместо них может быть объявлена функция-элемент, такая как, например, *set_tot*(), либо она может быть представлена прототипом, как и остальные функции этого класса. Полное определение других функций-элементов будет дано далее,

однако, чтобы описать интерфейсы функций, достаточно и прототипов.

Связывание данных и методов в один функциональный модуль — самое замечательное свойство класса. Благодаря такой конструкции объект *Stock* автоматически устанавливает правила, регламентирующие использование этого объекта. Прототипы функций в объявлении класса *Stock* показывают, каким образом создаются функции-элементы.

Новыми являются также ключевые слова *private* и *public*. Эти метки описывают управление доступом к элементам классов. Любая программа, которая использует объект конкретного класса, может получить непосредственный доступ к общедоступной части класса (рисунок 7.1).

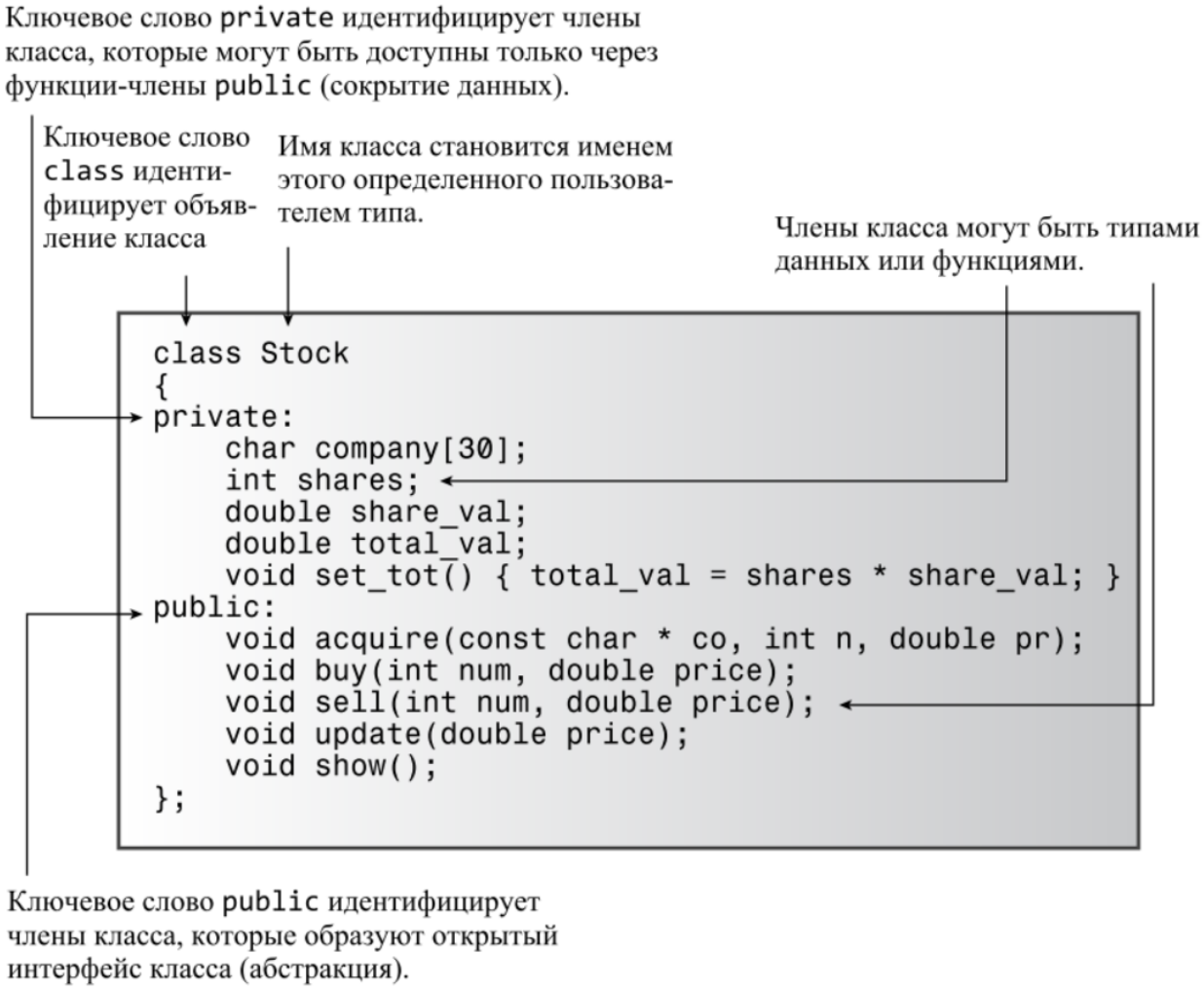

Рисунок 7.1 — Класс *Stock*

Программа может получить доступ к приватным элементам объекта только путем использования общедоступных функций-элементов (или путем использования дружественной функции). Например, единственный способ, позволяющий внести изменения в элемент *shares* класса *Stock*, заключается в использовании одной из функций-элементов класса *Stock*. Таким образом, общедоступные функции-элементы выступают как посредники между программой и приватными элементами объекта; они обеспечивают интерфейс между объектом и программой. Такая изоляция данных от непосредственного доступа, реализованная программой, называется *сокрытием данных*.

Конструкция класса сделана такой, чтобы отделить общедоступный интерфейс от специфики реализации. Общедоступный интерфейс представляет собой абстрактный компонент этой конструкции. Размещение деталей реализации в одном месте и отделение их от абстракции называется *инкапсуляцией*.

Сокрытие данных (размещение данных в приватном разделе класса) представляет собой пример инкапсуляции. Таким же примером является сокрытие деталей реализации в приватном разделе, как это делает класс *Stock* с функцией *set_tot*().

Другим примером инкапсуляции может служить обычная практика размещения определений функций класса в файле, отделенном от объявления классов. Обратите внимание на тот факт, что сокрытие данных позволяет не только предотвратить прямой доступ к данным, но и освобождает от необходимости следить за представлением данных.

Например, функция-элемент *show()* среди многих других данных отображает общую сумму вклада. Эта величина может храниться как часть соответствующего объекта, что имеет место в рассматриваемом примере, либо она может быть при необходимости вычислена. С точки зрения пользователя не имеет никакого значения, какой подход будет применен. Что нужно знать обязательно — так это то, какие действия выполняют функции-элементы. Иначе говоря, требуется знать, какие типы аргументов принимает конкретная функция-элемент и какой тип имеет возвращаемое ею значение, если таковое имеется. Принцип заключается в том, что детали реализации

отделяются от конструкции интерфейса. Если в дальнейшем найдется более подходящий путь для реализации представления данных или деталей функций-элементов, можно внести изменения в эти детали, не затрагивая программного интерфейса, благодаря чему эти программы станут более удобными в работе.

Общедоступный или приватный? Элементы класса можно объявить независимо от того, являются ли они элементами данных или функциями-элементами и находятся ли они в общедоступном или в приватных разделах класса. Однако одним из основных принципов ООП является сокрытие данных, поэтому элементы данных, как правило, размещаются в приватном разделе. Функции-элементы, которые образуют интерфейс класса, размещаются в общедоступном разделе; в противном случае вызвать эти функции из программы невозможно. Как показывает объявление класса *Stock*, можно также поместить функции-элементы в приватный раздел. Нельзя обратиться к этим функциям прямо из программы, однако их могут использовать общедоступные методы. Как правило, приватные функции-элементы используются для манипулирования деталями реализации, которые не являются составной частью публичного интерфейса.

Вовсе не требуется прибегать к помощи ключевого слова *private* в объявлениях классов, поскольку управление доступом к объектам класса, задаваемое этим ключевым словом, принято по умолчанию:

```
класс World
{ float mass; // приватная переменная, заданная по умолчанию
char name[20]; // приватный массив, заданный по умолчанию
public:
void tellall(void);
. . .
};
```

Тем не менее будет явно использоваться метка *private*, чтобы подчеркнуть важность понятия сокрытия данных.

Классы и структуры. Описания классов во многом совпадают с объявлениями структур, отличие состоит в том, что добавляются функции-элементы и метки *public* и *private*. Фактически *C++* распространяет на структуры те же свойства, какими обладают классы, но

типом доступа к структуре по умолчанию будет *public*, а типом доступа к классу — *private*.

Программисты, работающие в среде *C++,* обычно используют классы для реализации описаний класса, в то время как использование структур ограничивается представлением чистых объектов данных или, время от времени, классов без приватных компонентов.

Реализация классов и функций-элементов. Перед нами все еще стоит задача выполнения второй части спецификации класса: составить программные коды для тех функций-элементов, которые представлены прототипом в объявлении класса. Рассмотрим следующую проблему. Определения функций-элементов во многом аналогичны определениям обычных функций. Каждое такое определение состоит из заголовка функции и тела функции. Они могут иметь возвращаемые типы и аргументы. В то же время они обладают двумя специфическими характеристиками.

1. Когда вы определяете функцию-элемент, вы используете оператор определения диапазона доступа (*::*) для идентификации класса, которому эта функция принадлежит.

2. Методы класса могут осуществлять доступ к компонентам класса типа *private*.

Перейдем к обсуждению этих вопросов. Прежде всего, в заголовке функции-элемента используется оператор определения диапазона доступа (*::*) с целью указать, какому классу эта функция принадлежит.

Например, заголовок функции-элемента *update()* имеет вид:

void Stock::update(double price)

Такая форма записи означает, что мы определяем функцию *update()*, которая является элементом класса *Stock*. Но эта запись не только идентифицирует функцию *update()* как функцию-элемент, она также означает, что мы имеем возможность воспользоваться этим именем для обозначения функции-элемента другого класса. Например, функция *update()* класса *Buffoon* будет иметь такой заголовок:

void Buffoon::update()

Таким образом, оператор определения диапазона доступа идентифицирует класс, которому принадлежит определение метода. Мы говорим, что идентификатор *update()* обладает диапазоном доступа класса. Другие функции-элементы класса *Stock* могут при необходимости воспользоваться методом *update()* без употребления оператора определения диапазона доступа.

Это объясняется тем, что они принадлежат к одному и тому же классу, обеспечивая попадание функции *update()* в диапазон доступа. Одна из особенностей имен методов заключается в том, что полное имя метода конкретного класса содержит в себе имя этого класса. Мы говорим, что *Stock::update()* — это уточненное имя функции. Просто *update()*, с другой стороны, является аббревиатурой (неуточненным именем) полного имени, как раз это имя может быть использовано в диапазоне доступа класса.

Второй специальной характеристикой методов является тот факт, что метод может получить доступ к приватным элементам класса. Например, метод *show()* вполне может использовать следующий программный код:

```
cout<< "Company:" << company << "Shares:" << shares << '\n' << "SharePrice: $" << share_val << "TotalWorth: $" << total_val<< '\n';
```

В этом программном коде *company*, *shares* и т.п. — это приватные элементы данных класса *Stock*. Если вы попытаетесь воспользоваться функциями, не являющимися элементами класса *Stock*, для доступа к этим элементам данных, компилятор намертво заблокирует вам доступ. Имея в виду эти два соображения, мы сможем обеспечить выполнение методов класса, как показано в листинге 7.2. Определения этих методов могут находиться в отдельном файле или в одном файле с определением класса. Наилучший способ, который мы применим несколько позже, состоит в использовании заголовочного файла для хранения объявления класса и использования файла с исходным программным кодом для хранения определений функций-элементов этого класса.

Листинг 7.2 Программа stocks.cpp

```
// more stocks.cpp — реализация
// Функций-элементов класса
#include <iostream>
using namespace std;
#include <cstdlib> // или stdlib.h для exit()
#include <cstring> // или string.h для strncpy()
void Stock::acquire(const char * co, int n, double pr)
{
strncpy(company, co, 29); // при необходимости выполняется усечение строки co
company[29] = '\0';
shares = n ;
share_val = pr;
set_tot() ;
}
void Stock::buy(int num, double price)
{
shares += num;
share_val = price;
set_tot() ;
}
void Stock::sell(int num, double price)
{
if (num > shares)
{
cerr << "You can't sell more than you have!\n";
exit(1) ;
}
shares -= num;
share_val = price;
set_tot() ;
}
void Stock::update(double price)
{
share_val = price ;
set_tot() ;
}
void Stock:: show()
{
cout << "Company:" << company
```

```
<< "Shares:" <<shares <<'\n'
<<"Share Price: $" <<share_val
<<"Total Worth: $" <<total_val
<<'\n';
}
```

Функция *acquire()* выполняет обработку первоначального вклада конкретной компании, в то время как функции *buy()* и *sell()* осуществляют операции по добавлению или снятию сумм с существующего вклада. Если пользователь предпринимает попытку продать больше акций, чем у него есть, функция *sell()* обращается к функции *exit()*, которая прекращает выполнение программы. Четыре функции элемента устанавливают или переустанавливают значение элемента *total_val*. Вместо того чтобы выполнять эти вычисления четыре раза подряд, класс каждый раз обращается к функции *set_tot()*. Поскольку эта функция является просто средством реализации программного кода, а вовсе не частью общедоступного интерфейса, класс переводит функцию-элемент *set_tot()* в категорию приватных.

Встроенные методы

Любая функция с определением в объявлении класса автоматически становится встроенной. Таким образом, *Stock::set_tot()* является встроенной функцией. В объявлениях классах часто используются встроенные функции для небольших функций-элементов, а функция *set_tot()* соответствует этому требованию. Вы можете, если захотите, определить функцию-элемент вне объявления соответствующего класса и тем не менее сделать ее встроенной, для этого достаточно воспользоваться спецификатором *inline* во время определения функции в разделе реализации класса:

```
class Stock
{
. . .
private:
. . .
void set_tot(); // определение хранится отдельно
public:
. . .
```

```
};
inline void Stock::set_tot() // использование
// спецификатора inline в определении
{ total_val = shares * share_val; }
```

Поскольку встроенные функции обладают внутренним связыванием, они известны только файлу, в котором объявлены. Простейший способ сделать так, чтобы определения встроенных функций стали доступны для всех файлов в многофайловой программе, состоит во включении определения встроенной функции во все заголовочные файлы, в которых определен соответствующий класс. Между прочим, согласно правилу подстановки, определение метода в объявлении класса эквивалентно замене определения метода прототипом и последующей перезаписи этого определения в качестве встроенной функции непосредственно после объявления класса. Иначе говоря, первоначальное определение функции *set_tot()* эквивалентно определению, которое было рассмотрено выше.

Выбор объекта. Перейдем к одному из самых важных аспектов использования объектов: как применить метод классов к объекту. В программном коде

```
shares += num;
```

используется элемент *shares* для одного из объектов. Однако для какого объекта? Это очень интересный вопрос! Чтобы получить на него ответ, рассмотрим сначала, как создаются объекты. Простейший путь состоит в объявлении переменных класса:

```
Stock kate, joe;
```

Этот программный код создает два объекта класса *Stock*: один с именем *kate*, а другой с именем *joe*. Рассмотрим, как использовать функции-элементы для работы с одним из этих объектов. Ответ, как и в случае со структурами и элементами структур, следует искать в использовании оператора принадлежности:

```
kate.show(); // объект kate обращается к функции-элементу
joe.show(); // объект joe обращается к функции-элементу
```

В результате первого вызова начинает выполняться функция *show()* как элемент объекта *kate*. Это означает, что этот метод интер-

претирует *shares* как *kate.shares* и *share_val* – как *kate.share_val*. Аналогично вызов *joe.show()* заставляет метод *show()* интерпретировать *shares* и *share_val* соответственно как *joe.shares* и *joe.share_val* (рисунок 7.2).

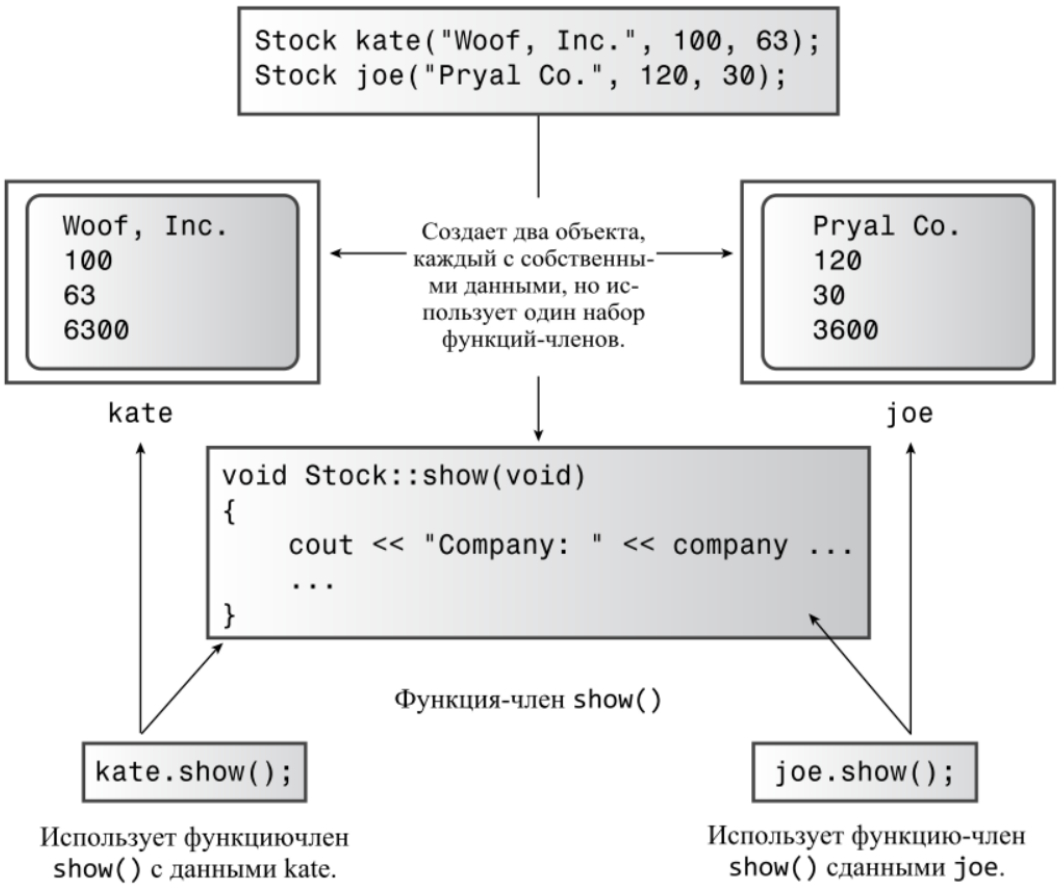

Рисунок 7.2 — Объекты, данные и функции

Аналогично вызов функции *kate.sell()* приводит к вызову функции *set_tot()*, как если бы это была функция *kate.set_tot()*, передавая этой функции нужные ей данные из объекта *kate*. Каждый новый создаваемый объект включает область памяти для хранения собственных внутренних переменных, элементов класса. Однако все объекты одного класса совместно используют одни и те же методы калсса, при этом существует одна копия каждого метода. Предположим, например, что *kate* и *joe* — объекты класса *Stock*. Далее, *kate.shares* занимает один фрагмент памяти, а *joe.shares* — другой фрагмент памяти. Но в то же время как *kate.show(),* так и *joe.show()* вызывают один и тот же метод, т.е. оба выполняют один и тот же блок программных кодов. Они всего лишь применяют этот код к различным данным.

Использование классов. Вы уже знаете, как определять класс и методы класса. Следующий шаг состоит в написании программы, которая создает и использует объекты соответствующего класса.

В задачу *C++* входит использование классов, которые максимально применяют встроенные типы, такие как *int* и *char*. Можно создать объект класса посредством объявления переменной класса или с помощью спецификатора *new* таким образом, чтобы построить объект типа класс. Можно передавать объекты как аргументы, возвращать их как возвращаемые значения функции и присваивать один объект другому. В *C++* имеются средства для инициализации объектов, средства обучения *cin* и *cout*, обеспечивающие распознавание объектов, и даже средства автоматического преобразования типов между объектами подобных классов. Вы еще не научились пользоваться всеми этими средствами, поэтому начнем с наиболее простых. В самом деле, вы уже видели, как объявить объект класса и вызвать функцию-элемент. Эти приемы в сочетании с объявлением класса и объявлениями функций-элементов составляют законченную программу, представленную в листинге 7.3. Она создает объект класса *Stock*, которому присвоено имя *stock1*. Программа довольно проста, однако она позволяет проверить все свойства, которыми мы наделили этот класс.

Листинг 7.3 Полный текст программы stocks.cpp

```
// stocks.cpp — завершенная программа
#include <iostream>
using namespace std;
#include<cstdlib> // или stdlib.h для функции exit()
#include <cstring> // или string.h для функции strncpy()
class Stock
{
private:
char company[30];
int shares;
double share_val;
double total_val;
void set_tot()
{ total_val = shares * share_val; }
public:
```

```
void acquire (const char *co, int n, double pr) ;
void buy(int num, double price);
void sell(int num, double price);
void update(double price);
void show() ;
};
void Stock::acquire(const char *co, int n, double pr)
{
strncpy(company, co, 29) ; // при необходимости выполняется усе-
чение строки co
company[29] = '\0';
shares = n;
share_val = pr ;
set_tot() ;
}
void Stock::buy(int num, double price)
{
shares += num;
share_val = price;
set_tot() ;
}
void Stock::sell(int num, double price)
{
if (num > shares)
{
cerr << "You can't sell more than you have!\n";
exit(1) ;
}
shares -= num;
share_val = price;
set_tot() ;
}
void Stock::update(double price)
{
share_val = price ;
set_tot() ;
}
void Stock:: show()
{
cout << "Company:" << company << "Shares:" <<shares <<'\n'
<<"Share Price: $" <<share_val
```

```
<<"Total Worth: $" <<total_val <<'\n';
}
int main ()
{
Stock stock1;
stock1.acquire("NanoSmart", 20, 12.50);
cout.precision(2); // формат #.##
cout.setf(ios_base::fixed); // формат #.##
cout.setf(ios_base::showpoint); // формат #.##
stock1.show () ;
stock1.buy(15, 18.25);
stock1.show ();
return 0;
}
```

Данная программа использует три команды форматирования. Результатом выполнения этих команд является отображение двух цифр справа от десятичной точки, включая нули в младших разрядах. Фактически в условиях сложившейся практики необходимы только первые две команды, а в более ранних реализациях достаточно было только первой и третьей команды. Использование всех трех команд форматирования дает один и тот же результат в обеих реализациях.

А тем временем предлагаем выходные данные программы:

```
Company: NanoSmart Shares: 20
Share Price: $12.50 Total Worth: $250.00
Company: NanoSmart Shares: 35
Share Price: $18.25 Total Worth: $638.75
```

Обратите внимание на то обстоятельство, что функция *main()* представляет собой всего лишь платформу для тестирования конструкции класса *Stock*. При условии, что класс функционирует так, как требуется, мы можем сейчас использовать класс *Stock* как тип, определенный пользователем, в других программах.

Деструкторы и конструкторы классов

Существуют специальные функции, получившие названия *конструкторов* и деструкторов, которые обычно резервируются за каждым классом. Посмотрим, для чего они нужны и как их создавать. Одна из целей *C++* заключается в том, чтобы сделать использование

объектов класса таким же простым, как и использование стандартных типов. Тем не менее, вы не можете инициализировать объект *Stock* тем же способом, что и обыкновенные типы *int* или *struct*:

```
int year = 2001; // правильно
struct thing { char * pn; int m; };
thing amabob = {"wodget", -23}; // правильно
Stock hot = { "Sukie's Autos, Inc.", 200, 50.25} ;
// неверно!
```

Причина, по которой вы не можете инициализировать объект *Stock* таким способом, заключается в том, что разделы данных имеют статус приватных, а это означает, что программа не может получить прямой доступ к элементам данных. Как вы уже могли убедиться, единственный способ, благодаря которому программа может получить доступ к элементам данных, — это использование функций-элементов. В связи с этим необходимо разработать соответствующую функцию-элемент, если вы намерены выполнить инициализацию объекта. (Вы можете инициализировать объект класса, как показано ранее, если определите элементы данных как общедоступные, а не как приватные, но определение общедоступных данных противоречит одной из основных целей применения класса — сокрытию данных.) В общем случае лучше всего, когда все объекты инициализируются во время их создания. Рассмотрим, например, следующий программный код:

```
Stock gift;
gift.buy(10, 24.75);
```

В рассматриваемой реализации класса *Stock* объект *gift* не имеет значения для элемента *company*. Конструкция класса предполагает, что вызовет функцию *acquire()* прежде чем обращаться к другим функциям-элементам, однако каких-либо специальных средств для реализации этого предположения не существует. Один из способов, позволяющих обойти эту трудность, заключается в том, чтобы объекты были инициализированы автоматически в момент их создания. Чтобы осуществить эту идею, предусмотрены специальные функции-элементы, называемые *конструкторами классов*, в задачу которых входит построение новых объектов и присвоение элементам данных

этих объектов начальных значений. Точнее, *C++* назначает имена этим функциям-элементам и предлагает синтаксис для их использования, а вы обеспечиваете метод их определения. Это имя совпадает с именем класса. Например, потенциальным конструктором класса *Stock* является функция-элемент с именем *Stock()*. Прототип конструктора и заголовок обладают весьма интересным свойством — несмотря на то, что у конструктора нет возвращаемого значения, он не объявляется с типом *void*. По существу, у конструктора нет объявленного типа.

Объявление и определение конструкторов. А теперь создадим конструктор *Stock*. Поскольку предусмотрено, что объект *Stock* получает три значения из внешнего мира, необходимо назначить этому конструктору три аргумента. (Четвертое значение, элемент *total_val*, вычисляется по значениям элементов *shares* и *share_val*, поэтому не требуется передавать его конструктору.) Вполне возможно, что вы намерены всего лишь задать значение для элемента *company*, а другим элементам присвоить нулевое значение. Это можно сделать с помощью аргументов, заданных по умолчанию. Следовательно, прототип будет иметь следующий вид:

```
// прототип конструктора с некоторыми
// аргументами, заданными по умолчанию
Stock (const char * co, int n = 0, double pr = 0.0);
```

Первый аргумент является указателем на строку, которая используется для инициализации элементов класса символьного массива *company*. Аргументы *n* и *pr* передают значения элементам *shares* и *share_val*. Обратите внимание на то, что возвращаемый тип отсутствует. Прототип находится в общедоступном разделе объявления класса. Ниже представлено возможное определение конструктора:

```
// определение конструктора
Stock::Stock(constchar *co, intn, doublepr)
{
strncpy(company, co, 29);
company[29] = ' \0 ' ;
shares = n ;
share_val = pr;
```

set_tot() ;
}

Это тот же программный код, который мы использовали в функции *acquire()*. Различие заключается в том, что программа автоматически вызывает конструктор в тот момент, когда она объявляет объект.

Использование конструктора. В *C++* предусмотрены два способа инициализации объекта с использованием конструктора. Первый из них — это явный вызов конструктора:

Stock food = Stock("World Cabbage", 250, 1.25);

С помощью этой команды элементу *company* объекта *food* присваивается строка *"World Cabbage"*, элементу *shares* — значение *250* и т. д. Второй способ предусматривает неявный вызов конструктора:

Stock food("World Cabbage", 250, 1.25);

C++ использует конструктор класса всякий раз, когда создается объект этого класса, даже если вы используете спецификатор *new* в целях динамического распределения памяти. Конструктор со спецификатором *new* используется следующим образом:

*Stock *pstock = new Stock("Electroshock Games", 18, 19.0);*

Использование конструкторов отличается от применения других методов класса. Обычно объект используется для вызова метода:

stock1.show(); // объект stock1 вызывает метод show()

Тем не менее вы не можете воспользоваться объектом, чтобы вызвать конструктор, так как до тех пор, пока конструктор не закончит работу по построению конкретного объекта, такого объекта не существует.

Конструктор, заданный по умолчанию. Конструктором, заданным по умолчанию, является конструктор, используемый для построения объекта, когда явные значения для инициализации отсутствуют. Другими словами, конструктор используется для объявлений такого рода:

Stock stock1;
// используется конструктор, заданный по умолчанию

Но позвольте, программа, представленная в листинге 7.3, уже делала это! Причина того, что этот оператор все-таки работает, заключается в том, что, если нет никаких конструкторов, *C++* автоматически использует конструкторы, заданные по умолчанию. Таким конструктором является версия по умолчанию конструктора, заданного по умолчанию, и она не выполняет никаких действий. Для класса *Stock* он будет выглядеть следую щим образом:

Stock:: Stock () { }

Окончательный результат состоит в том, что объект *stock1* создается без инициализации его элементов, точно так же как оператор *int x*; создает переменную *x*, не присваивая ей значения. То обстоятельство, что конструктор, заданный по умолчанию, не имеет аргументов, отражает тот факт, что в объявлении не появляются никакие значения. Интересно отметить, что компилятор предоставляет конструктор, заданный по умолчанию, только в том случае, если вы не определите никакого конструктора. После того как вы назначите конкретный конструктор конкретному классу, обязанность по предоставлению конструктора, заданного по умолчанию, переходит от компилятора к вам. Если вы воспользуетесь конструктором, который не используется по умолчанию, таким как, например,

*Stock(const char *co, int n, double pr);*

и не предложите своей собственной версии конструктора, заданного по умолчанию, то объявление вида

Stock stock1; // невозможно с текущим конструктором

вызовет ошибку. Причина такого поведения заключается в том, что у вас может возникнуть необходимость сделать невозможным создание неинициализированных объектов. С другой стороны, вам может понадобиться создавать объекты без явной инициализации. В таком случае придется определить собственный конструктор. Это должен быть конструктор, которому не нужны аргументы. Вы можете определить конструктор по умолчанию двумя способами. Один из них состоит в том, чтобы присвоить значения, заданные по умолчанию, всем аргументам существующего конструктора:

*Stock(const char * co = "Error", int n = 0, double pr = 0.0);*

Второй способ предусматривает использование перегрузки функции для определения второго конструктора, который при этом не имеет аргументов:

Stock () ;

На практике обычно требуется инициализировать объекты, чтобы быть уверенным, что все элементы начинаются с известных, корректно выбранных значений. Таким образом, конструктор, заданный по умолчанию, как правило, осуществляет неявную инициализацию значений всех элементов. В данном случае, например, можно определить конструктор для класса *Stock* следующим образом:

```
Stock: : Stock ()
{
strcpy(company, "no name");
shares = 0 ;
share_val = 0.0;
total_val = 0.0;
}
```

После того как вы воспользовались одним из методов (не указывая аргументов или значений, заданных по умолчанию для всех аргументов), чтобы создать конструктор по умолчанию, можно объявить переменные объекта, не выполняя их явной инициализации:

```
// неявно вызывает конструктор,
// заданный по умолчанию
Stock first;
// вызывает его явно
Stock first = Stock ();
// вызывает его неявно
Stock *prelief = new Stock;
```

Тем не менее не дайте ввести себя в заблуждение неявной формой конструктора, не используемого по умолчанию:

```
// вызывает конструктор
Stock first ("Concrete Conglomerate");
// объявляет функцию
Stock second () ;
// вызывает конструктор,
// заданный по умолчанию
Stock third;
```

Первое объявление вызывает конструктор, который не является конструктором, заданным по умолчанию, т.е. конструктор, который принимает аргументы. Второе объявление утверждает, что *second()* — функция, которая возвращает объект *Stock*. *При неявном вызове конструктора, заданного по умолчанию, не употребляйте круглые скобки*.

Деструкторы. Когда вы используете конструктор для построения объекта, программа берет на себя обязанность отслеживать этот объект до тех пор, пока он не выполнит возложенную на него задачу. В этот момент программа автоматически вызывает специальную функцию-элемент, которая называется *деструктор*. Деструктор должен уничтожить весь оставшийся «мусор», таким образом он служит конструктивным целям. Например, если ваш конструктор использует спецификатор *new* при распределении памяти, деструктор с помощью оператора *delete* освобождает память. Как и конструктор, деструктор имеет специальное имя: имя класса, которому предшествует тильда (~). Деструктор класса *Stock* имеет имя *~Stock()*. Итак, подобно конструктору, деструктор не имеет возвращаемого значения и объявленного типа. В отличие от конструктора, у деструктора не может быть аргументов. Следовательно, прототип деструктора класса *Stock* должен быть таким:

```
~Stock() ;
```

Поскольку у деструктора *Stock* нет особо важных обязанностей, мы можем закодировать его как функцию, которая не выполняет никаких действий.

```
Stock::~Stock()
{
}
```

Однако только для того, чтобы вы могли увидеть, когда производится обращение к деструктору, запишем его в таком виде:

```
Stock::~Stock() // деструктор класса
{
cout << "Bye, " << company << "! \n" ;
}
```

Когда следует обращаться к деструктору? Это решение принимает компилятор, *ваш программный код не должен содержать явных обращений к деструктору*. Если вы создаете объект класса статической памяти, то его деструктор вызывается автоматически в момент окончания выполнения программы. Если вы создаете объект класса автоматической памяти, что, собственно говоря, мы и делали раньше, то его деструктор вызывается автоматически, когда программа выходит из блока программного кода, в котором объект был определен. Если объект создается с использованием спецификатора *new*, он размещается в динамически распределяемой области памяти или в свободной памяти, а его деструктор вызывается автоматически, когда используется оператор *delete* для освобождения памяти. И наконец, программа может создавать временные объекты для того, чтобы выполнять определенные операции; в этом случае программа автоматически вызывает деструктор, что бы тот удалил объект, когда программа прекращает использование этого объекта. Поскольку деструктор вызывается автоматически, когда объект класса прекращает функционировать, деструктор должен быть наготове. Если вы не позаботились о своем деструкторе, компилятор предоставит вам деструктор, заданный по умолчанию, который не выполняет никаких действий.

Совершенствование класса Stock. Следующий шаг состоит во включении конструкторов и деструкторов в определения классов и методов. На этот раз мы будем придерживаться обычной практики, сложившейся в *C++*, и организуем программу в виде нескольких отдельных файлов. Поместим описание рассматриваемого класса в з*аголовочный файл* с именем *stock1.h*. Методы класса помещаются в файл с именем *stock1.cpp*. Заголовочный файл, содержащий объявление класса, и файл исходного программного кода, содержащий определения методов, должны иметь одно и то же базовое имя, чтобы можно было отслеживать, какие файлы принадлежат друг другу. Использование отдельных файлов для объявления класса и для функций-элементов отделяет абстрактное определение интерфейса (объявление класса) от деталей реализации (определения функций-

элементов). Например, можно реализовать объявление класса как заголовочный текстовый файл, а определения функций — как скомпилированные коды, используя эти ресурсы, поместить программу в третий файл, который называется *usestok1.cpp*.

Заголовочный файл. В листинге 7.4 показан заголовочный файл. Из него прототипы конструкторов и деструкторов помещаются в объявление исходного класса. Помимо этого, он не включает функцию *acquire()*, которая теперь, когда у класса есть конструкторы, больше не нужна.

Листинг 7.4 Программа stock1.h

```
// stockl.h
#ifndef _STOCK1_H_
#define _STOCK1_H_
class Stock
{
private:
char company[30];
int shares;
double share_val;
double total_val;
void set_tot()
{ total_val = shares * share_val; }
public:
Stock(); // конструктор по умолчанию
Stock(const char * co, int n = 0, double pr = 0.0) ;
~Stock(); // деструктор noisey
void buy(int num, double price);
void sell(int num, double price);
void update(double price);
void show() ;
};
#endif
```

Управление заголовочными файлами. Заголовочный файл следует включать в тот или иной файл только один раз, однако может случиться так, что в файле окажется сразу несколько заголовочных файлов. Например, вы можете использовать заголовочный файл, который сам содержит другой заголовочный файл. Имеется стандарт-

ный метод *C/C++*, который позволяет избежать нескольких включений заголовочных файлов. В его основу положена директива *#ifndef* (аббревиатура от слов *if not defined* — если не определен) препроцессора. Сегмент кода вида

```
#ifndef _STOCK1_H_
. . .
#endif
```

означает: выполнить операторы, заключенные между *#ifndef* и *#endif*, если только имя *_STOCK1_H_* не было определено раньше с помощью директивы *#define* препроцессора. Обычно оператор *#define* используется для построения символьных констант, как, например, в следующем случае:

```
#define MAXIMUM 4096
```

Однако простое использование оператора *#define* с именем достаточно только для того, чтобы установить, что имя определено:

```
#define _STOCK1_H_
```

Метод, который использует программа, представленная в листинге 7.4, предназначается для того, чтобы удалить содержимое этого файла в *#ifndef*:

```
#ifndef _STOCK1_H_
#define _STOCK1_H_
// поместить содержимое файла включения
// в этом месте
#endif
```

Первый раз, когда компилятор встречает этот файл, имя *_STOCK1_H_* не должно быть определенным. Мы выбираем имя, опираясь на имя файла включения с несколькими символами подчеркивания, вставленными таким образом, чтобы это имя не могло совпасть с именем, определенным где-нибудь в другом месте. Если такое случается, то компилятор просматривает, что содержится между *#ifndef* и *#endif*. В процессе просмотра компилятор читает строку, определяющую *_STOCK1_H_*. Если компилятор обнаруживает еще одно включение файла *stock1.h* в одном и том же файле, он отмечает, что имя *_STOCK1_H_* определено, и переходит к строке, которая

следует за *#endif*. Обратите внимание на то, что этот метод не препятствует компилятору включать файл дважды. Вместо этого он вынуждает компилятор игнорировать содержимое всех последующих включений, кроме первого. Большая часть заголовочных файлов в стандартном *C* и *C++* используют эту схему.

Файл реализации. В листинге 7.5 представлены определения методов. Здесь размещается файл *stock1.h*, реализующий условия для объявления класса. (Напомним, что заключение имени файла в двойные кавычки вместо скобок означает, что компилятор осуществляет его поиск в том месте, где были обнаружены исходные файлы.) Кроме того, в программе, представленной в этом листинге, используются системные файлы *iostream* и *cstring*, поскольку указанные выше методы используют *cin*, *cout* и *strncpy()*. Этот файл добавляет определения методов конструктора и деструктора в уже имеющиеся методы.

Листинг 7.5 Программа stock1.cpp

```
// stock1.cpp — методы класса Stock
#include <iostream>
#include <cstdlib> // или stdlib.h для exit()
#include <cstring> // или string.h для strncpy()
using namespace std;
#include "stock1.h"
// конструкторы
Stock::Stock()
{
// конструктор по умолчанию
strcpy(company, "no name");
shares = 0;
share_val = 0.0;
total_val = 0.0;
}
Stock:: Stock (const char *co, int n, double pr)
{
strncpy(company, co, 29);
company[29] = ' \0';
shares = n ;
share_val = pr;
set_tot() ;
```

```
}
// деструктор класса
Stock::~Stock() // деструктор класса verbose
{
cout <<"Bye, " <<company <<"!\n" ;
}
// другие методы
void Stock::buy(int num, double price)
{
shares += num;
share_val = price;
set_tot() ;
}
void Stock::sell(int num, double price)
{
if (mim > shares)
{
cerr <<"You can't sell more than you have! \n " ;
exit(1);
}
shares -= num;
share_val = price;
set_tot() ;
}
void Stock::update(double price)
{
share_val = price;
28
set_tot() ;
}
void Stock:: show ()
{
cout <<"Company: " <<company <<"Shares: " <<shares <<'\n'
<<"Share Price: $" <<share_val <<"Total Worth: $"
<<total~val <<' \n' ;
}
```

Клиентский файл. В программу, представленную в листинге 7.6, включен небольшой модуль для тестирования новых методов. Подобно программе *stock1.cpp*, она содержит файл *stock1.h*, посредством которого осуществляется объявление классов. Эта программа

демонстрирует конструкторы и деструкторы и использует те же команды форматирования, которые выполнялись в программе, представленной в листинге 7.5. Чтобы скомпилировать программу в полном составе, воспользуйтесь методом, ориентированным на многофайловые программы.

Листинг 7.6 Программа usestok1.cpp

```
// usestok1.cpp — использование класса Stock
#include <iostream>
using namespace std;
#include "stock1.h"
int main()
{
// использование конструкторов для
// построения новых объектов
Stock stock1("NanoSmart", 12, 20.0);
// синтаксис 1
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
// синтаксис 2
cout.precision(2); // формат #.##
cout.setf (ios_base::fixed, ios_base::floatfield); // формат #.##
cout.setf(ios_base::showpoint); //формат #.##
stock1. show() ;
stock2.show();
stock2 = stock1; // назначение объекта
// использование конструктора для переустановки объекта
stock1 = Stock("Nifty Foods", 10, 50.0);
// объект temp
coat << "After stock reshuffle:\n";
stock1. show () ;
stock2.show() ;
return 0;
29
}
```

Результаты выполнения программы:

```
Company: NanoSmart Shares: 12
Share Price: $20.00 Total Worth: $240.00
Company: Boffo Objects Shares: 2
Share Price: $2.00 Total Worth: $4.00
Bye, Nifty Foods!
```

After stock reshuffle:
Company: Nifty Foods Shares: 10 S
hare Price: $50.00 Total Worth: $500.00
Company: NanoSmart Shares: 12
Share Price: $20.00 Total Worth: $240.00
Bye, NanoSmart!
Bye, Nifty Foods!

Оператор

Stock stock1("NanoSmart", 12, 20.0);

создает объект класса *Stock* под именем *stock1* и инициализирует его элементы данных заданными значениями. Оператор

Stock stock2 = Stock("Boffo Objects" , 2, 2.0) ;

использует вторую разновидность синтаксиса для построения и инициализации объекта с именем *stock2*. Вы можете использовать конструктор не только для инициализации нового объекта. Например, в составе функции *main()* имеется такой оператор.

stock1 = Stock("Nifty Foods", 10, 50.0);

Объект *stock1* уже существует. Таким образом, вместо того чтобы инициализировать объект *stock1*, этот оператор присваивает новые значения данному объекту, вынуждая конструктор построить новый, временный объект и в дальнейшем осуществить копирование содержимого этого нового объекта в объект *stock1*. Оператор

stock2 = stock1; // присвоение объекта

показывает, что вы можете присвоить один объект другому объекту того же типа. Как и в случае присваивания структур, в процессе присваивания объекта типа класс по умолчанию копируются элементы одного объекта в элементы другого. В этом случае первоначальное содержимое объекта *stock2* затирается.

Обратите внимание на то обстоятельство, что в результате выполнения программы сначала отображается *Bye, Nifty Foods!*, а затем уже содержимое нового объекта *stock1*. После этого в самом конце программа говорит: *Bye, NanoSmart!* и *Bye, Nifty Foods!*. Откуда исходят эти трогательные прощальные слова? Напомним, что в деструкторе имеется оператор вывода, обеспечивающий этот эффект, чтобы вы могли видеть, когда вызван деструктор. (Это всего лишь средство

обучения, а не обычное средство проектирования!) Два заключительных прощальных приветствия имеют место, когда прекращается выполнение функции *main()*, при этом два локальных объекта (*stock1* и *stock2*), которые объявляет программа, не попадают в диапазон доступа. Поскольку подобного рода автоматические переменные поступают в стек, объект, созданный последним, удаляется первым, а объект, созданный первым, удаляется последним. (Обратите внимание на то, что строка *"NanoSmart"* первоначально находилась в *stock1*, однако позднее была переведена в *stock2*.) Когда программа использует конструктор для присвоения атрибутов *Nifty Food* объекту *stock1*, она сначала создает временный, безымянный объект, в котором сохраняются эти значения. Затем эти значения копируются в объект *stock1*. По завершению этого, когда все задачи, возложенные на временный объект, будут выполнены, программа вызывает деструктор, чтобы удалить его. Первый удаленный *Nifty Foods* — это временный объект, а второй удаленный *Nifty Foods* — это объект *stock1*. Этот незначительный эпизод показывает, что между двумя приводимыми ниже операторами существует принципиальное различие:

```
Stock stock2 = Stock("Boffo Objects",2 , 2.0);
// временный объект
stockl = Stock("Nifty Foods", 10, 50.0);
```

Первый из представленных операторов выполняет инициализацию, он создает объект с заданным значением. Второй оператор является оператором присваивания. Он создает временный объект и затем копирует его в существующий. Этот оператор менее эффективен, чем оператор инициализации. (Тем не менее компилятору предоставлена возможность реализации формы инициализации (рассматривалось ранее применительно к *stock2*) путем создания временного объекта с последующим копированием его содержания в *stock2*.)

Результаты подсчета, отображенные функцией *show()* при завершении выполнения программы, показывают, что обе операции — присваивание и восстановление объектов с помощью конструктора и последующего присваивания — выполняются.

Функции-элементы типа *const.* Рассмотрим следующие фрагменты программного кода:

```
const Stock land = Stock("Kludgehorn Properties");
land.show() ;
```

В рассматриваемой версии *C++* компилятор отвергает вторую строку. Почему? Да потому, что код функции *show()* не дает гарантии, что не будет модифицирован вызывающий объект, который, будучи объявленным как *const*, не должен подвергаться изменениям. Вы уже решали раньше подобного рода проблемы, объявляя аргументы функции ссылками *const* или указателями *const*. Но при этом перед нами встают синтаксические проблемы: метод *show()* вообще не имеет аргументов. Вместо этого объект, который он использует, предоставляется ему неявно путем вызова метода. Все, что необходимо в таких случаях, — это новое синтаксическое средство, которое гарантирует, что функция не внесет изменений в вызывающий ее объект. В *C++* эта проблема решается с помощью ключевого слова *const*, которое ставится после скобок функции. Другими словами, объявление *show()* принимает такой вид:

```
void show() const; // обещает не вносить
// изменений
// в вызывающий объект
```

Аналогично начальная часть определения функции принимает такой вид:

```
void stock::show() const // обещает не вносить изменений
// в вызывающий объект
```

Функции, объявленные и определенные таким способом, называются функциями-элементами типа *const*. Необходимо размещать методы класса в категории *const* всякий раз, когда нужно, чтобы они не изменяли вызывающий объект.

Обзор конструкторов и деструкторов. Конструктор — это функция-элемент класса специального назначения, которая вызывается всякий раз, когда создается некоторый объект этого класса. Конструктор какого-либо класса имеет то же имя, что и его класс. Однако благодаря перегрузке функций вы можете иметь сразу несколько конструкторов с одним и тем же именем при условии, что каждый из них

имеет собственную сигнатуру, иначе говоря, собственный список аргументов. Кроме того, конструктор не имеет объявленного типа. Как правило, конструктор используется для инициализации элементов объекта класса. Проводимая вами инициализация должна соответствовать списку аргументов конструктора. Например, предположим, что у класса *Bozo* имеется следующий прототип конструктора класса:

```
// прототип конструктора
Bozo (char * fname, char * lname) ;
```

Его нужно использовать для инициализации новых объектов следующим образом:

```
// первичная форма
Bozo bozetta = bozo("Bozetta" , "Biggens");
// укороченная форма
Bozo fufu("Fufu", "O'Dweeb");
// динамический объект
Bozo *pc = new Bozo("Popo", "Le Peu") ;
```

Если конструктор имеет только один аргумент, то этот конструктор вызывается, когда вы инициализируете объект значением, которое имеет тот же тип, что и аргумент конструктора. Например, предположим, что у вас имеется прототип этого конструктора:

```
Bozo(int age);
```

Вы можете использовать любую из представленных ниже форм инициализации объекта:

```
Bozo dribble = bozo(44); // первичная форма
Bozo roon(66); // вторичная форма
Bozo tubby =32; // специальная форма для конструкторов
// с одним аргументом
```

Фактически третий пример представляет новый способ инициализации, мы его раньше не рассматривали, однако сейчас, по-видимому, самый подходящий момент сообщить вам об этом.

Конструктор, который вы можете использовать с единственным аргументом, предоставляет возможность использовать синтаксис присваивания для инициализации объекта некоторым значением:

```
Classname object = value;
```

Конструктор, заданный по умолчанию, не имеет аргументов, он используется в тех случаях, когда вы создаете объект без явной его инициализации. Если вы не можете воспользоваться каким-либо из конструкторов, компилятор выделяет конструктор, заданный по умолчанию. В противном случае нужно воспользоваться своим собственным конструктором. Он может вообще не иметь аргументов, иначе должны быть заданы значения по умолчанию для всех аргументов:

```
Bozo () ; // прототип конструктора по умолчанию
// значение по умолчанию для класса
Bistro Bistro(const char * s = "Chez Zero");
```

Эта программа использует конструктор, заданный по умолчанию, для неинициализированных объектов:

```
Bozo bubi; // используется значение по умолчанию
Bozo *pb = new Bozo; // используется значение по умолчанию
```

Как и в процессе построения объекта, программа вызывает конструктор, а деструктор вызывается, когда нужно уничтожить объект. Каждый класс может иметь только один деструктор. Он не имеет возвращаемого типа, не может даже иметь тип *void;* у него нет аргументов, а его имя является именем соответствующего класса, которому предшествует тильда. Деструктор класса *Bozo*, например, имеет следующий прототип:

```
~Bozo ()
```

; // деструктор класса

Деструкторы класса становятся необходимыми, когда конструкторы класса используют оператор new.

8 НАСЛЕДОВАНИЕ КЛАССОВ

Наследование – концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

Наследование позволяет:

– добавлять новые возможности в существующий класс. Например, в существующий базовый класс массива можно добавить арифметические операции;

– добавлять данные, которые представляет класс. Например, взяв за основу базовый класс строки, можно породить класс, в котором добавлен член данных, представляющий цвет, и который будет использоваться при выводе строки на экран;

– изменять поведение методов класса. Например, от класса *Passenger*, который представляет услуги, предоставляемые пассажиру авиалинии, можно породить класс *First Class Passenger*с более высоким уровнем обслуживания.

Класс, который наследует данные, называется подклассом (*subclass*), производным классом (*derived class*), или дочерним классом (*child*). Класс, от которого наследуются данные или методы, называется суперклассом (*super class*), базовым классом (*base class*), или родительским классом (*parent*). Термины «родительский» и «дочерний» чрезвычайно полезны для понимания наследования. Как ребенок получает характеристики своих родителей, производный класс получает методы и переменные базового класса.

Наследование полезно, поскольку оно позволяет структурировать и повторно использовать код, что, в свою очередь, может значительно ускорить процесс разработки. Несмотря на это, наследование следует использовать с осторожностью, поскольку большинство изменений в суперклассе затронут все подклассы, что может привести к непредвиденным последствиям (рисунок 8.1).

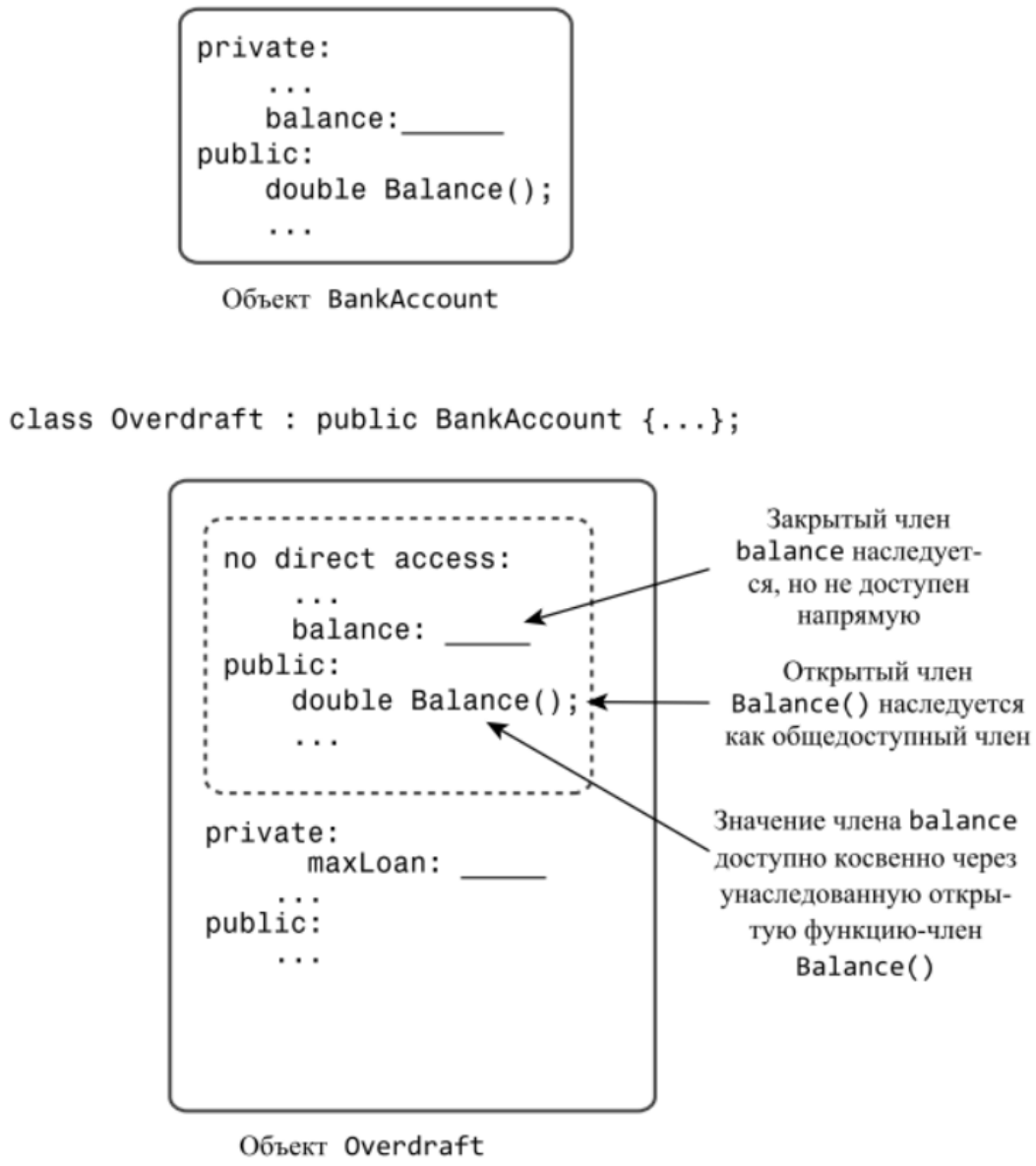

Рисунок 8.1 — Объекты базового и производного классов

Типы наследования. В *C ++* есть несколько типов наследования:

1) публичный (*public*) — публичные (*public*) и защищенные (*protected*) данные наследуются без изменения уровня доступа к ним;

2) защищенный (*protected*) — все унаследованные данные становятся защищенными;

3) приватный (*private*) — все унаследованные данные становятся приватными.

Для базового класса *Device*, уровень доступа к данным не изменяется, но поскольку производный класс *Computer* наследует данные как приватные, данные становятся приватными для класса *Computer*.

Конструкторы и деструкторы. В *C ++* конструкторы и деструкторы не наследуются. Однако они вызываются, когда дочерний класс инициализирует свой объект. Конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. Деструкторы вызываются в обратном порядке.

Множественное наследование. Если порожденный класс наследует элементы одного базового класса, то такое наследование называется одиночным. Однако возможно и множественное наследование. Множественное наследование позволяет порожденному классу наследовать элементы более чем от одного базового класса. Синтаксис заголовков классов расширяется так, чтобы разрешить создание списка базовых классов и обозначения их уровня доступа:

```
class X
{...};
class Y
{...};
class Z
{...};
class A : public X, public Y, public Z
{...};
```

Виртуальное наследование. Виртуальное наследование (*virtual inheritance*) предотвращает появление множественных объектов базового класса в иерархии наследования. Таким образом, конструктор базового класса *Device* будет вызван только единожды, а обращение к методу *turn_on*() без его переопределения в дочернем классе не будет вызывать ошибку при компиляции.

Абстрактный класс. В *C++* класс, в котором существует хотя бы один чистый виртуальный метод (*pure virtual*), принято считать абстрактным. Если виртуальный метод не переопределен в дочернем классе, код не скомпилируется. Также в *C++* создать объект абстрактного класса невозможно, попытка вызовет ошибку при компиляции.

Интерфейс. *C++*, в отличие от некоторых языков ООП, не предоставляет отдельного ключевого слова для обозначения интерфейса (*interface*). Тем не менее реализация интерфейса возможна путем создания чистого абстрактного класса (*pure abstract class*) — класса, в котором присутствуют только декларации методов. Такие классы часто называют абстрактными базовыми классами (*Abstract Base Class — ABC*).

Раннее и позднее связывание. Когда речь заходит об объектно-ориентированных языках программирования часто используются два термина: раннее и позднее связывание. По отношению к *C++* эти термины соответствуют событиям, которые возникают на этапе компиляции и на этапе исполнения программы соответственно.

В терминах объектно-ориентированного программирования раннее связывание означает, что объект и вызов функции связываются между собой на этапе компиляции, что вся необходимая информация, для того чтобы определить, какая именно функция будет вызвана, известна на этапе компиляции программы. В качестве примеров раннего связывания можно указать стандартные вызовы функций, вызовы перегруженных функций и перегруженных операторов. Принципиальным достоинством раннего связывания является его эффективность, оно более быстрое и обычно требует меньше памяти, чем позднее связывание. Его недосток — невысокая гибкость.

Позднее связывание означает, что объект связывается с вызовом функции только во время исполнения программы, а не раньше. Позднее связывание достигается в *C++* с помощью использования виртуальных функций и производных классов. Его достоинством является высокая гибкость. Оно может применяться для поддержки общего интерфейса, позволяя при этом различным объектам иметь свою собственную реализацию этого интерфейса. Более того, оно помогает создавать библиотеки классов, допускающие повторное использование и расширение.

Какое именно связывание должна использовать программа, зависит от предназначения программы. Фактически достаточно сложные программы используют оба вида связывания. Позднее

связывание является одним из самых мощных добавлений языка *C++* к возможностям языка *C*. Платой за такое увеличение мощи программы служит некоторое уменьшение ее скорости исполнения, поэтому использование позднего связывания оправдано, если оно улучшает структурированность и управляемость программы. Надо иметь в виду, что проигрыш в производительности невелик, и, когда ситуация требует позднего связывания, можно использовать его без всякого сомнения.

9 ГРАФИЧЕСКИЙ ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС. СРЕДА РАЗРАБОТКИ *Qt*

Графический пользовательский интерфейс (ГПИ) (*graphical user interface, GUI*) — разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений.

В отличие от интерфейса командной строки в ГПИ пользователь имеет произвольный доступ (с помощью устройств ввода — клавиатуры, мыши, джойстика и т. п.) ко всем видимым экранным объектам (элементам интерфейса) и осуществляет непосредственное манипулирование ими. Чаще всего элементы интерфейса в ГПИ реализованы на основе метафор и отображают их назначение и свойства, что облегчает понимание и освоение программ неподготовленными пользователями.

Графический интерфейс пользователя является частью пользовательского интерфейса и определяет взаимодействие с пользователем на уровне визуализированной информации.

Можно выделить следующие виды ГПИ:

– простой: типовые экранные формы и стандартные элементы интерфейса, обеспечиваемые самой подсистемой ГПИ;

– истинно графический, двумерный: нестандартные элементы интерфейса и оригинальные метафоры, реализованные собственными средствами приложения или сторонней библиотекой;

– трёхмерный: на данный момент слабо классифицирован.

Достоинства

Графический интерфейс является интуитивно понятным, «дружелюбным» для пользователей любого уровня; для работы с программами обработки графики — единственно возможный.

Недостатки

Графический интерфейс потребляет больше памяти по сравнению с текстовым интерфейсом; сравнительная сложность организации удаленной работы; невозможна автоматизация работы при

условии, если она не была заложена разработчиком программы; к графическому интерфейсу трудно привыкнуть пользователям, которые работали с интерфейсом командной строки.

Основные элементы графического интерфейса. Элемент интерфейса, элемент управления, виджет – примитив графического интерфейса пользователя, который имеет стандартный внешний вид и выполняет стандартные действия (рисунок 9.1).

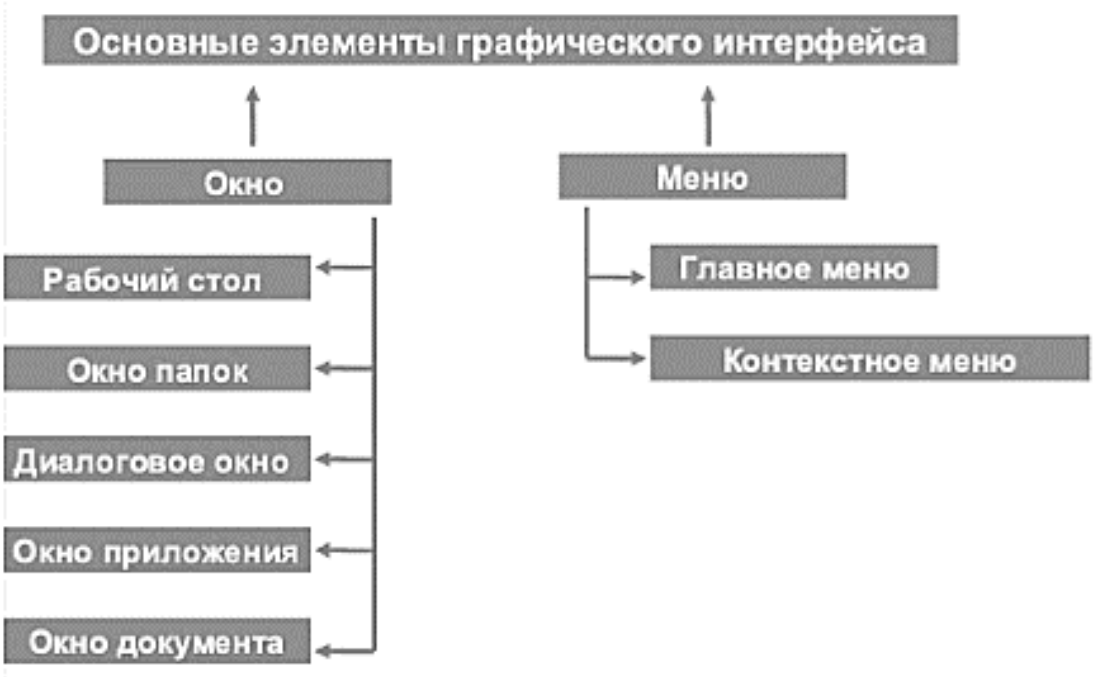

Рисунок 9.1 — Основные элементы графического интерфейса

***Qt Creator* —** это полностью интегрированная среда разработки (IDE), которая предоставляет инструменты проектирования и разработки сложных приложений для множества настольных и мобильных платформ.

Одним из главнейших достижений *Qt Creator* является то, что он позволяет команде разработчиков работать над проектом на различных платформах с использованием общих инструментов для разработки и отладки.

Чтобы быть в состоянии собирать и запускать приложения, *Qt Creator* нуждается в той же информации, которая требуется компилятору. Эта информация указана в настройках сборки и запуска проекта.

Создание проекта позволит:

– группировать файлы вместе;

– добавить собственные шаги сборки;

– включить формы и файлы ресурсов;

– указать настройки для запускаемых приложений.

Можно или создать проект с нуля, или импортировать существующий проект. *Qt Creator* генерирует все необходимые файлы в зависимости от типа создаваемого проекта. Например, если вы выберите создание приложения с графическим интерфейсом пользователя, *Qt Creator* создаст пустой *.ui* файл, который вы можете изменить в интегрированном *Qt Designer*.

Qt Creator интегрирован с кросс-платформенными системами автоматизации сборки: *qmake* и *CMake*. Также можно импортировать существующие проекты, которые не используют *qmake* или *CMake*, и указать *Qt Creator* просто проигнорировать вашу систему сборки.

Qt Creator поставляется с редактором кода и *Qt Designer* для проектирования и сборки графических интерфейсов пользователя из виджетов *Qt*.

Так как он является *IDE*, *Qt Creator* отличается от текстового редактора тем, что знает, как собирать и запускать приложения, понимает языки *C++* и *QML* как код, а не как простой текст. Это позволяет:

– писать хорошо форматированный код;

– угадывать, что вы хотите написать, и дополнять код;

– отображать сообщения об ошибках и предупреждения;

– перемещаться между классами, функциями и символами;

– предоставлять контекстно-зависимую справку по классам, функциям и символам;

– осмысленно переименовывать символы так, что другие символы с таким же именем, но принадлежащие другим областям действия не будут переименованы;

– показывать место в коде, где функция была описана или вызвана.

10 БИБЛИОТЕКА *Qt*

10.1 Виджеты. Компоновка виджетов

Виджеты

Виджеты — это исходные элементы для создания пользовательского интерфейса в *Qt.* Виджеты могут отображать данные и информацию о состоянии, получать ввод от пользователя и предоставлять контейнер для других виджетов, которые должны быть сгруппированы. Виджет, не встроенный в родительский виджет, называется окном. Родительский виджет содержит в себе различные дочерние виджеты.

Класс *QWidge*t предоставляет базовую возможность для отрисовки на экране и для обработки событий пользовательского ввода. Все элементы пользовательского интерфейса, предоставляемые *Qt*, являются подклассами *QWidget* или используются в сочетании с подклассом *QWidget*. Создание пользовательских виджетов выполняется наследованием от *QWidget* или подходящего подкласса и переопределения виртуальных обработчиков событий.

Компоновка виджетов. Создадим небольшое приложение «Приложение Age (возраст)», которое демонстрирует применение менеджеров компоновки для размещения виджетов в окне и использование сигналов и слотов для синхронизации работы двух виджетов (листинг 10.1). Приложение предлагает пользователю указать свой возраст, что можно сделать при помощи либо наборного счетчика (*spinbox*), либо ползунка (*slider*). Это приложение состоит из трех виджетов: *QSpinBox*, *QSlider* и *QWidget*. *QWidget* является главным окном приложения. Виджеты *QSpinBox* и *QSlider* помещены внутрь *QWidget*, и они являются дочернимивиджетами по отношению к *QWidget*. С другой стороны, можно сказать, что *QWidget* является родительским виджетом по отношению к *QSpinBox* и *QSlider*. Сам *QWidget* не имеет родителя, потому что используется в качестве окна самого верхнего уровня. Конструкторы *QWidget* и все его подклассы принимают параметр *QWidget**, задающий родительский виджет.

Листинг 10.1 — Приложение Age

```
01 #include <QApplication>
02 #include <QHBoxLayout>
03 #include <QSlider>
04 #include <QSpinbox>
05 int main (int argc, char *argv[])
06 {
07 QApplication app(argc, argv);
08 QWidget *window = new QWidget;
09 window->setWindowTitle("Enter Your Age");
10 QSpinBox *spinBox = new QSpinBox;
11 QSlider *slider = new QSlider(Qt::Horizontal);
12 spinBox->setRange(0, 130);
13 slider->setRange(0, 130);
14 QObject::connect(spinBox, SIGNAL(valueChanged(int)),
15 slider, SLOT(setValue(int)));
16 QObject::connect(slider, SIGNAL(valueChanged(int)),
17 spinBox, SLOT(setValue(int)));
18 spinBox->setValue(35);
19 QHBoxLayout *layout = new QHBoxLayout;
20 layout->addWidget(spinBox);
21 layout->addWidget(slider);
22 window->setLayout(layout);
23 window->show();
24 returnapp.exec();
25 }
```

Строки 8 и 9 создают и настраивают виджет *QWidget*, который является главным окном приложения. Вызывается функция *setWindowTitle*() для вывода текстовой строки в заголовке окна. Строки 10 и 11 создают виджеты *QSpinBox* и *QSlider*, а строки 12 и 13 устанавливают допустимый диапазон изменения их значений. Вполне можем допустить, что возраст человека не будет превышать 130 лет. Можно было бы передать *window* в конструкторах *QSpinBox* и *QSlider*, указывая на то, что *window* должен быть их родительским виджетом, но здесь это делать необязательно, поскольку система компоновки определит это самостоятельно и автоматически установит родительский виджет для наборного счетчика и ползунка. Два вызова функции *QObject::connect*(), выполненные в строках с 14-й по

17-ю, обеспечивают синхронизацию работы наборного счетчика и ползунка, заставляя их всегда показывать одинаковое значение. Если один из виджетов изменяет значение, то генерируется сигнал *valueChanged(int)* и вызывается слот *setValue(int)* другого виджета с новым значением возраста. В строке 18 наборный счетчик устанавливается в значение 35. В результате виджет *QSpinBox* генерирует сигнал *valueChanged(int)* с целочисленным аргументом 35. Этот аргумент передается слоту *setValue(int)* виджета *QSlider*, и в результате ползунок устанавливается в значение 35. Ползунок затем также генерирует сигнал *valueChanged(int)*, поскольку его значение изменилось, и вызывает слот *setValue(int)* наборного счетчика. Но на этот раз функция *setValue(int)* не будет генерировать сигнал, поскольку наборный счетчик уже имеет значение 35. Это не позволяет повторять эти действия бесконечно.

В строках с 19-й по 22-ю размещены виджеты наборного счетчика и ползунка, используя менеджер компоновки. Менеджер компоновки — это объект, который устанавливает размер и положение виджетов, расположенных в зоне его действия. *Qt* имеет три основных класса менеджеров компоновки:

– *QHBoxLayout* размещает виджеты по горизонтали слева направо (или справа налево, в зависимости от культурных традиций);

– *QVBoxLayout* размещает виджеты по вертикали сверху вниз;

– *QGridLayout* размещает виджеты в ячейках сетки.

Выполненный в строке 22 вызов *QWidget::setLayout()* устанавливает менеджер компоновки для окна. За кулисами создаются дочерние связи *QSpinBox* и *QSlider* с виджетом, для которого установлен менеджер компоновки, и по этой причине нам не требуется в явной форме задавать родительский виджет при конструировании виджета, размещаемого в зоне действия менеджера компоновки.

Несмотря на то, что не задавали в явной форме положение и размер ни одного из виджетов, *QSpinBox* и *QSlider* аккуратно расположились в ряд. Это объясняется тем, что *QHBoxLayout* автоматически определяет разумные размеры и положение виджетов, попадающих в зону его действия, в зависимости от потребностей этих виджетов. Менеджеры компоновки освобождают нас от нудного

кодирования размещения виджетов нашего приложения на экране и гарантируют плавное изменение размеров окон. Используемый средствами разработки *Qt* подход к построению графического пользовательского интерфейса легко понятен и очень гибок. Программисты добавляют виджеты к компоновщикам графических элементов, которые автоматически устанавливают для них нужные размер и положение. Управление работой графического интерфейса осуществляется через взаимодействие виджетов друг с другом посредством применения механизма сигналов и слотов *Qt*.

Типы виджетов:

– *QMainWindow* — в большинстве случаев, наиболее подходящий выбор. Наследуясь от данного класса мы получаем уже готовые средства для размещения меню, строки статуса и центрального поля, которое можно реализовать как в стиле *SDI* (*Single Document Interface*), так и в стиле *MDI* (*Multi Document Interface*);

– *QWidget* — этот класс является простейшим виджетом. В терминологии *Qt* это простейший элемент, с которым связана какая-то графическая область на экране. В качестве базового класса для главного окна используется, как правило, при создании простых однооформенных приложений и отлично подходит для начальных «ученических» целей по причине того, что не содержит ничего «лишнего»;

– *QDialog* — базовый класс для создания модальных диалоговых окон.

Прикрепляемые виджеты и панели инструментов. Прикрепляемыми являются виджеты, которые могут крепиться к определенным областям главного окна приложения *QMainWindow* или быть независимыми «плавающими» окнами. *QMainWindow* имеет четыре области крепления таких виджетов: одна сверху, одна снизу, одна слева и одна справа от центрального виджета. В таких приложениях, как *Microsoft Visual Studio* и *Qt Linguist* широко используются прикрепляемые окна для обеспечения очень гибкого интерфейса пользователя. В *Qt* прикрепляемые виджеты представляют собой экземпляры класса *QDockWidget.*

Каждый прикрепляемый виджет имеет свой собственный заголовок, даже когда он прикреплен. Пользователи могут перемещать прикрепляемые окна с одного места крепления на другое, передвигая полосу заголовка. Они могут также отсоединять прикрепляемое окно от области крепления и сделать его независимым плавающим окном, располагая прикрепляемое окно вне областей крепления. Свободные плавающие прикрепляемые окна всегда находятся «поверх» их главного окна. Пользователи могут закрыть *QDockWidget,* щелкая по кнопке закрытия, расположенной в заголовке виджета. Любые комбинации этих возможностей можно отключать с помощью вызова *QDockWidget::setFeatures()*.

В ранних версиях *Qt* панели инструментов рассматривались как прикрепляемые виджеты, использующие те же самые области крепления. Начиная с *Qt* 4, панели инструментов размещаются в собственных областях, расположенных по периметру центрального виджета, и они не могут открепляться (рисунок 10.1). Если требуется иметь плавающую панель инструментов, можно просто поместить ее внутрь *QDockWindow.*

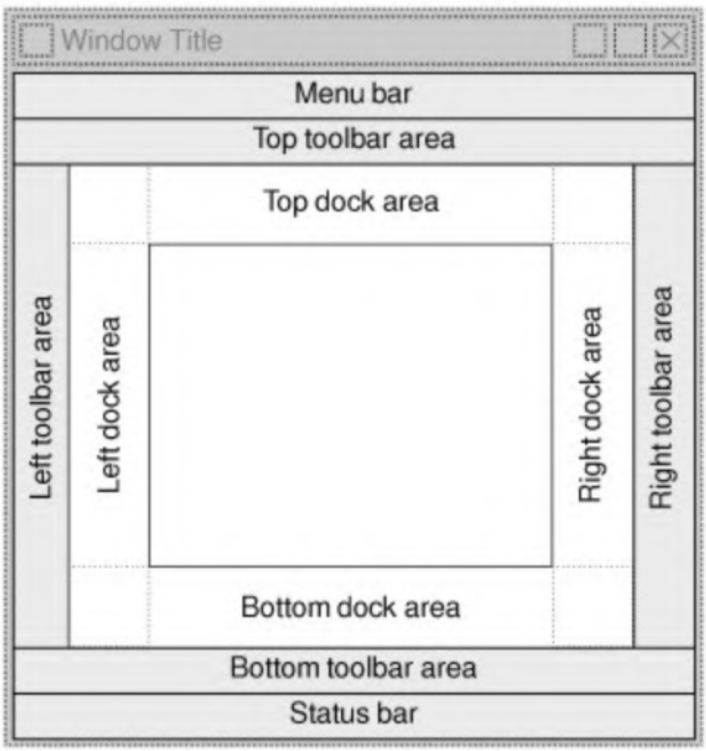

Рисунок 10.1 — Области крепления виджетов
и области панелей инструментов QMainWindow

Углы, обозначенные пунктирными линиями, могут принадлежать обеим соседним областям крепления. Например, мы могли бы верхний левый угол назначить левой области крепления с помощью вызова *QMainWindow::setCorner(Qt::TopLeftCorner, Qt::LeftDock WidgetArea)*.

Следующий фрагмент программного кода показывает, как для существующего виджета (в данном случае для *QTreeWidget)* можно оформить оболочку в виде *QDockWidget* и вставить ее в правую область крепления:

```
QDockWidget *shapesDockWidget = new QDockWidget(tr("Shapes"));
shapesDockWidget->setWidget(treeWidget);
shapesDockWidget->setAllowedAreas(Qt::LeftDockWidgetArea
| Qt::RightDockWidgetArea);
addDockWidget(Qt::RightDockWidgetArea, shapesDockWidget);
```

В вызове *setAllowedAreas()* задаются допустимые области крепления прикрепляемого окна. В нашем случае пользователю можно перетаскивать прикрепляемое окно только в левую или правую область крепления, где имеется достаточно пространства по вертикали для его нормального отображения. Если допустимые области не задаются явно, пользователь может перетаскивать прикрепляемое окно в любую из четырех областей.

Ниже приводится фрагмент из конструктора подкласса *QMainWindow*, который показывает, как можно создавать панель инструментов, содержащую *QComboBox, QSpinBox* и несколько кнопок *QToolButton:*

```
QToolBar *fontToolBar = newQToolBar(tr("Font"));
fontToolBar->addWidget(familyComboBox);
fontToolBar->addWidget(sizeSpinBox);
fontToolBar->addAction(boldAction);
fontToolBar->addAction(italicAction);
fontToolBar->addAction(underlineAction);
fontToolBar->setAllowedAreas(Qt::TopToolBarArea
| Qt::BottomToolBarArea);
addToolBar(fontToolBar);
```

Если хотим сохранять позиции всех прикрепляемых виджетов и панелей инструментов, чтобы иметь возможность их восстановления

при следующем запуске приложения, то можем написать почти такой же программный код, как для сохранения состояния разделителя *QSplitter,* используя функции класса *QMainWindow saveState()* и *restoreState():*

```
01 void MainWindow::writeSettings()
02 {
03 QSettings settings("Software Inc.", "Icon Editor");
04 settings.beginGroup("mainWindow");
05 settings.setValue("size", size());
06 settings.setValue("state", saveState());
07 settings.endGroup();
08 }
09 void MainWindow::readSettings()
10 {
11 QSettings settings("Software Inc.", "Icon Editor");
12 settings.beginGroup("mainWindow");
13 resize(settings.value("size").toSize());
14 restoreState(settings.value("state").toByteArray());
15 settings.endGroup();
16 }
```

Наконец, *QMainWindow* обеспечивает контекстное меню, в котором представлены все прикрепляемые окна и панели инструментов. Используя это меню, пользователь может закрывать и восстанавливать прикрепляемые окна и панели инструментов.

10.2 Взаимодействие виджетов.

Механизм сигналов и слотов

Описание технологии сигналов и слотов. Механизм сигналов и слотов играет решающую роль в разработке программ *Qt*. Он позволяет прикладному программисту связывать различные объекты, которые ничего не знают друг о друге. До этого уже соединяли некоторые сигналы и слоты, объявляли наши собственные сигналы и слоты, реализовывали наши собственные слоты и генерировали наши собственные сигналы. Давайте рассмотрим этот механизм более подробно.

Слоты почти совпадают с обычными функциями, которые объявляются внутри классов *C++* (функции-члены). Они могут быть виртуальными, перегруженными, открытыми (*public*), защищенными (*protected*) и закрытыми (*private*), они могут вызываться непосредственно, как и любые другие функции-члены *C++*, и их параметры могут быть любого типа. Однако слоты (в отличие от обычных функций-членов) могут подключаться к сигналам и в результате они будут вызываться при каждом генерировании соответствующего сигнала.

Оператор *connect()*

connect (отправитель, *SIGNAL*(сигнал), получатель, *SLOT*(слот));

где отправитель и получательявляются указателями на объекты *QObject* и где сигнал и слотявляются сигнатурами функций без имен параметров. Макросы *SIGNAL()* и *SLOT()* фактически преобразуют свои аргументы в строковые переменные. В приводимых ранее примерах мы всегда подключали разные слоты к разным сигналам. Существует несколько вариантов подключения слотов к сигналам.

К одному сигналу можно подключать много слотов:

connect(*slider, SIGNAL*(*valueChanged*(*int*)),
spinBox, SLOT(*setValue*(*int*)));
connect(*slider, SIGNAL*(*valueChanged*(*int*)),
this, SLOT(*updateStatusBarIndicator*(*int*)));

При генерировании сигнала последовательно вызываются все слоты, причем порядок их вызова не определен. Один слот можно подключать ко многим сигналам:

connect(*lcd, SIGNAL*(*overflow*()),
this, SLOT(*handleMathError*()));
connect(*calculator, SIGNAL*(*divisionByZero*()),
this, SLOT(*handleMathError*()));

Данный слот будет вызываться при генерировании любого сигнала. Один сигнал может соединяться с другим сигналом:

connect(*lineEdit, SIGNAL*(*textChanged*(*const QString &*)),
this, SIGNAL(*updateRecord*(*const QString &*)));

При генерировании первого сигнала будет также генерироваться второй сигнал. В остальном связь «сигнал-сигнал» не отличается от связи «сигнал-слот». Связь можно аннулировать:

disconnect(*lcd, SIGNAL*(*overflow*()),
this, SLOT(*handleMathError*()));

Это редко приходится делать, поскольку *Qt* автоматически убирает все связи при удалении объекта. При успешном соединении сигнала со слотом (или с другим сигналом) их параметры должны задаваться в одинаковом порядке и иметь одинаковый тип:

connect(*ftp, SIGNAL*(*rawCommandReply*(*int, const QString &*)),
this, SLOT(*processReply*(*int, const QString &*)));

Имеется одно исключение, а именно: если у сигнала больше параметров, чем у подключенного слота, то дополнительные параметры просто игнорируются:

connect (*ftp, SIGNAL*(*rawCommandReply*(*int, const QString &*), *this*, *SLOT*(*checkErrorCode*(*int*)));

Если параметры имеют несовместимые типы либо будет отсутствовать сигнал или слот, то *Qt* выдаст предупреждение во время выполнения программы, при условии, что сборка программы проводилась в отладочном режиме. Аналогично *Qt* выдаст предупреждение, если в сигнатуре сигнала или слота будут указаны имена параметров.

Метаобъектная система *Qt*. Одним из главных преимуществ средств разработки *Qt* является расширение языка *C++* механизмом создания независимых компонентов программного обеспечения, которые можно соединять вместе, несмотря на то что они могут ничего не знать друг о друге.

Этот механизм называется метаобъектной системой, и он обеспечивает две основные служебные функции: взаимодействие сигналов и слотов и анализ внутреннего состояния приложения (*introspection*). Анализ внутреннего состояния необходим для реализации сигналов и слотов и позволяет прикладным программистам получать «метаинформацию» о подклассах *QObject* во время выполнения программы, включая список поддерживаемых объектом сигналов и слотов и имена их классов. Этот механизм также поддерживает

свойства (для *Qt Designer)* и перевод текстовых значений (для интернационализации приложений), и создает основу для системы сценариев в *Qt* (*Qt Script for Applications — QSA*). В стандартном языке *C++* не предусмотрена динамическая поддержка метаданных, необходимых системе метаобъектов *Qt*. В *Qt* эта проблема решена за счет применения специального инструментального средства компилятора *moc*, который просматривает определения классов с макросом *Q_OBJECT* и делает соответствующую информацию доступной функциям *C++*. Поскольку все функциональные возможности *moc* обеспечиваются только с помощью «чистого» *C++*, метаобъектная система *Qt* будет работать с любым компилятором *C++*.

Этот механизм работает следующим образом:

– макрос *Q_OBJECT* объявляет некоторые функции, которые необходимы для анализа внутреннего состояния и которые должны быть реализованы в каждом подклассе *QObject:metaObject(), tr(), qt_metacall()*, и некоторые другие;

– компилятор *moc* генерирует реализации функций, объявленных макросом *Q_OBJECT,* и всех сигналов;

– такие функции-члены класса *QObject,* как *connect()* и *disconnect(),* во время своей работы используют функции анализа внутреннего состояния.

Все это выполняется автоматически при работе *qmake, moc* и при компиляции *QObject,* и поэтому крайне редко может возникнуть необходимость вспомнить об этом механизме. Однако если вам интересны детали реализации этого механизма, вы можете воспользоваться документацией по классу *QMetaObject* и просмотреть файлы исходного кода C++, сгенерированные компилятором *moc.* До сих пор мы использовали сигналы и слоты только при работе с виджетами. Но сам по себе этот механизм реализован в классе *QObject,* и его не обязательно применять только в пределах программирования графического пользовательского интерфейса. Этот механизм можно использовать в любом подклассе *QObject:*

```
01 classEmployee : public QObject
02 {
03 Q_OBJECT
```

```
04 public:
05 Employee() { mySalary = 0; }
06 int salary() const { return mySalary; }
07 public slots:
08 void setSalary(int newSalary);
09 signals:
10 void salaryChanged(int newSalary);
11 private:
12 int mySalary;
13 };
14 void Employee::setSalary(int newSalary)
15 {
16 if (newSalary != mySalary) {
17 mySalary = newSalary;
18 emit salaryChanged(mySalary);
19 }
20 }
```

10.3 Создание диалоговых и главных окон программ

Диалоговые окна предоставляют пользователю возможность задавать необходимые значения параметров и выбирать определенные режимы работы. Они называются диалоговыми окнами, или просто диалогами (*dialogs*), поскольку представляют собой средство, с помощью которого пользователи и приложения могут «переговариваться» друг с другом. Большинство приложений с графическим пользовательским интерфейсом имеют главное окно с панелью меню и инструментальной панелью, а также десятки диалоговых окон, естественно дополняющих главное окно. Можно также создать приложение из одного диалогового окна, которое будет непосредственно реагировать на выбор пользователя, выполняя соответствующие действия (например, таким приложением может быть калькулятор). Первое диалоговое окно мы создадим полностью вручную, чтобы было ясно, как выглядит исходный код такой программы. Затем мы покажем способы построения диалоговых окон в *Qt Designer,* который является средством визуального проектирования в *Qt.* Использование *Qt Designer* позволяет получать результат значительно быстрее, чем

при ручном кодировании, и полученные в нем различные варианты проектов легче тестировать и изменять в будущем.

Быстрое проектирование диалоговых окон. Создание диалогового окна как при ручном кодирования, так и при использовании *Qt Designer* предусматривает выполнение следующих шагов:

– создание и инициализация дочерних виджетов;

– размещение дочерних виджетов в менеджерах компоновки;

– определение последовательности переходов по клавише табуляции;

– установка соединений «сигнал-слот»;

– реализация пользовательских слотов диалогового окна.

Изменяющиеся диалоговые окна. Ранее были рассмотрены способы формирования диалоговых окон, которые всегда содержат одни и те же виджеты. В некоторых случаях требуется иметь диалоговые окна, форма которых может меняться.

Наиболее известны два типа изменяющихся диалоговых окон: расширяемые диалоговые окна (*area extension dialogs*) и многостраничные диалоговые окна (*multi-page dialogs*).

Оба типа диалоговых окон можно реализовать в *Qt* либо с помощью непосредственного кодирования, либо посредством применения *Qt Designer.* Расширяемые диалоговые окна, как правило, имеют обычное (нерасширенное) представление и содержат кнопку для переключения между обычным и расширенным представлениями этого диалогового окна. Расширяемые диалоговые окна обычно применяются в тех приложениях, которые предназначаются как для неопытных, так и для опытных пользователей и скрывают дополнительные опции до тех пор, пока пользователь явным образом не захочет ими воспользоваться.

Создавать в *Qt* другой распространенный тип изменяющихся диалоговых окон, многостраничные диалоговые окна даже еще проще как при ручном кодировании, так и при использовании *Qt Designer.*

Динамические диалоговые окна. Динамическими называются диалоговые окна, которые создаются на основе файлов *.ui,* сделанных в *Qt Designer,* во время выполнения приложения. Вместо

преобразования файла *.ui* компилятором *uic* в программу на *C++* можно загрузить этот файл на этапе выполнения, используя класс *QUiLoader:*

```
QUiLoader uiLoader;
QFile file("sortdialog.ui");
QWidget *sortDialog = uiLoader.load(&file);
if (sortDialog) {
}
```

Также можно осуществлять доступ к дочерним виджетам формы при помощи функции:

```
QObject::findChild<T>():
QComboBox *primaryColumnCombo =
sortDialog->findChild<QComboBox *>("primaryColumnCombo");
if (primaryColumnCombo) {
}
```

Функция *findChild<T>()* является шаблонной функцией-членом, которая возвращает дочерний объект по заданному имени и типу. Эта функция отсутствует для *MSVC* 6 из-за ограничений этого компилятора. Если необходимо использовать компилятор *MSVC* 6, вместо этой функции следует вызывать глобальную функцию *qFindChild<T>*(), которая работает точно так же.

Класс *QUiLoader* расположен в отдельной библиотеке. Для использования класса *QUiLoader* в приложении *Qt* должны добавить в файл *.pro* следующую строку:

```
CONFIG += uitools
```

Динамические диалоговые окна позволяют изменять компоновку элементов формы без повторной компиляции приложения. Они могут также использоваться для создания «тонких» клиентских приложений, когда в исполняемый модуль встраивается только основная форма пользовательского интерфейса, а все другие формы создаются по мере необходимости.

Встроенные классы виджетов и диалоговых окон. *Qt* содержит четыре вида кнопок: *QPushButton, QToolButton, QcheckBox* и *QRadioButton.* Кнопки *QPushButton* и *QToolButton* получили наибольшее распространение и используются для инициации какого-то дей-

ствия при их нажатии, но могут применяться и как переключатели (один щелчок нажимает кнопку, другой щелчок отпускает кнопку). Флажок *QCheckBox* может использоваться для включения и выключения независимых опций, в то время как переключатели (радиокнопки) *QRadioButton* обычно задают взаимоисключающие возможности.

Контейнеры *Qt* — это виджеты, которые содержат в себе другие виджеты. Фрейм *QFrame,* кроме того, может использоваться самостоятельно просто для вычерчивания линий, и он наследуется многими другими классами виджетов, включая *QToolBox* и *QLabel.*

QTabWidget и *QToolBox* являются многостраничными виджетами. Каждая страница является дочерним виджетом, и страницы нумеруются с нуля.

Виджеты для просмотра списков элементов оптимизированы для работы с большими наборами данных, и в них часто используются полосы прокрутки. Работа полосы прокрутки реализуется классом *QAbstractScrollArea,* который является базовым для просмотра списков элементов и для других виджетов, обеспечивающих скроллинг.

Qt имеет несколько виджетов, которые предназначены для простого отображения информации. Наиболее важным из них является текстовая метка *QLabel,* и она может также использоваться для форматированного отображения текста (используя простой синтаксис, подобный *HTML*) и вывода изображений.

Текстовый браузер *QTextBrowser* представляет собой подкласс поля редактирования, работающий только в режиме чтения и обеспечивающий основные возможности формата *HTML*, включая списки, таблицы, изображения и гипертекстовые ссылки. *Qt Assistant* использует браузер *QTextBrowser* для представления документации пользователю.

Qt содержит несколько виджетов для ввода данных. Строка редактирования *QLineEdit* может ограничивать ввод данных, применяя маску ввода или функцию проверки допустимости данных. Поле редактирования *QTextEdit* является подклассом *QAbstractScrollArea,* позволяющим редактировать тексты большого объема.

Qt содержит стандартный набор диалоговых окон для выбора пользователем цвета, шрифта, файла или для печати документа.

В системах *Windows* и *Mac Os X* по мере возможности используются «родные» диалоговые окна, а не их общие аналоги.

Qt содержит разнообразные диалоговые окна для передачи сообщений об ошибках и других сообщений, причем они обеспечивают обратную связь с пользователем. При выполнении продолжительных операций могут использоваться диалоговый индикатор состояния процесса *QProgressDialog* и показанный ранее индикатор состояния процесса без обратной связи *QProgressBar*. Очень удобно пользоваться диалоговым окном *QInputDialog*, когда пользователю требуется ввести одну строку или одно число.

Встроенные виджеты и стандартные диалоговые окна обладают большими функциональными возможностями, которыми можно пользоваться без дополнительного программирования. Многие специальные требования обеспечиваются установкой свойств виджетов или путем применения механизма сигналов и слотов и реализации пользовательских определений слотов. В некоторых случаях может возникнуть потребность в создании пользовательского виджета без помощи стандартных средств. В *Qt* это делается просто, и возможности пользовательских виджетов будут обладать таким же свойством независимости от платформы, как и возможности встроенных виджетов *Qt*. Пользовательские виджеты даже можно интегрировать в *Qt Designer,* и тогда они могут применяться так же, как и встроенные виджеты *Qt*.

10.4 Возможности разработки сетевых приложений

Программирование поддержки сети в *Qt*. Для того чтобы облегчить создание сетевых кросс-платформенных приложений, разработчики фреймворка *Qt* предусмотрели модуль работы с сетью *QtNetwork*. Модуль *QtNetwork* содержит как высокоуровневые классы, такие как *QHttp* или *QFtp*, так и классы *QAbstractSocket, QTcpServer, QUdpSocket*, с помощью которых можно работать с сетью на низком уровне.

Сокетное соединение. Сокет — это устройство пересылки данных с одного конца связи на другой. Другой конец может принадлежать процессу, работающему на локальном компьютере, а может располагаться и на удаленном компьютере, подключенному к Интернету и расположенному в другом полушарии Земли. Сокетное соединение — это соединение типа точка-точка (*point to point*), которое производится между двумя процессами.

Сокеты разделяют на дейтаграммные (*datagram*) и поточные. Дейтаграммные сокеты осуществляют обмен пакетами данных. Поточные сокеты устанавливают связь и производят потоковый обмен данными через установленную ими связь. На практике поточные сокеты используются гораздо чаще, чем дейтаграммные из-за того, что они предоставляют дополнительные механизмы, направленные против искажения и потери данных. Поточные сокеты работают в обоих направлениях, то есть то, что один из процессов записывает в поток, может быть считано процессом на другом конце связи, и наоборот. Для дейтаграммных сокетов *Qt* предоставляет класс *QUdpSocket*, а для поточных — класс *QTcpSocket*.

Класс *QTcpSocket* содержит набор методов для работы с *TCP* (*Transmition Control Protocol* — протокол управления передачей данных) — это сетевой протокол низкого уровня, являющийся одним из основных протоколов в Интернете. С его помощью можно реализовать поддержку для стандартных сетевых протоколов, таких как: *HTTP*, *FTP*, *POP3*, *SMTP*, и даже для своих собственных протоколов. Этот класс унаследован от класса *QAbstractSocket*, который, в свою

очередь, наследует класс *QIODevice.* А это значит, что для доступа (чтения и записи) к его объектам необходимо применять все методы класса *QIODevice* и использовать классы потоков *QDataStream* или *QTextStream*. Работа класса *QTcpSocket* асинхронна, что дает возможность избежать блокирования приложения в процессе его работы. Но если вам это не нужно, то можете воспользоваться серией методов, начинающихся со слова *waitFor*. Вызов этих методов приведет к ожиданию выполнения операции и заблокирует на определенное время исполнение вашей программы. Не рекомендуется вызывать эти методы в потоке графического интерфейса.

Модель «клиент-сервер». Сценарий модели «клиент-сервер» выглядит очень просто: сервер предлагает услуги, а клиент ими пользуется. Программа, использующая сокеты, может выполнять либо роль сервера, либо роль клиента.

Для того чтобы клиент мог взаимодействовать с сервером, ему нужно знать его *IP*-адрес и номер порта, через который клиент, желающий воспользоваться этими услугами сервера, должен сообщить о себе. Когда клиент устанавливает соединение с сервером, система назначает данному соединению отдельный сокет. После этого осуществляется связь между двумя этими сокетами, по которой высылаются данные запроса к серверу. А сервер высылает клиенту по этому соединению готовые, обработанные результаты согласно его запросам. Сервер не ограничен связью только с одним клиентом, на самом деле он может обслуживать многих клиентов. Каждому сокету соответствует уникальный номер порта. Некоторые номера зарезервированы для так называемых стандартных служб.

Реализация сервера с помощью класса *QtcpServer.* Для реализации сервера *Qt* предоставляет удобный класс *QTcpServer*, который предназначен для управления входящими *TCP*-соединениями. Программа, листинг которой приведен ниже, является реализацией простого сервера, который принимает и подтверждает получение запросов клиентов.

```
#include <QtGui>
#include "MyServer.h"
```

```
int main(int argc, char** argv)
{
QApplication app(argc, argv);
MyServer server(2323);

server.show();

return app.exec();
}
```

Создается объект сервера. Чтобы запустить сервер, нужно создать объект класса *MyServer*, передав в конструктор номер порта, по которому должен осуществляться нужный сервис. В данном случае передается номер порта, равный 232:

```
#ifndef _MyServer_h_
#define _MyServer_h_
#include <QWidget>
class QTcpServer;
class QTextEdit;
class QTcpSocket;
class MyServer : public QWidget {
Q_OBJECT
private:
QTcpServer* m_ptcpServer;
QTextEdit*  m_ptxt;
quint16     m_nNextBlockSize;
private:
void sendToClient(QTcpSocket* pSocket, const QString & str);
public:
MyServer(int nPort, QWidget* pwgt = 0);
public slots:
virtual void slotNewConnection();
void slotReadClient   ();
};
#endif //_MyServer_h_
```

В классе *MyServer* объявляем атрибут *m_ptcpServer*, который и является основной частью для управления нашим сервером. Атрибут *m_nNextBlockSize* служит для хранения длины следующего, полученного от сокета блока.

```
MyServer::MyServer(int nPort, QWidget* pwgt /*=0*/) :
QWidget(pwgt)
m_nNextBlockSize(0)
{
m_ptcpServer = new QTcpServer(this);
if (!m_ptcpServer->listen(QHostAddress::Any, nPort)) {
QMessageBox::critical(0,
"Server Error",
"Unable to start the server:"
+ m_ptcpServer->errorString()
);
m_ptcpServer->close();
return;
}
connect(m_ptcpServer, SIGNAL(newConnection()),
this, SLOT(slotNewConnection())
);
m_ptxt = new QTextEdit;
m_ptxt->setReadOnly(true);
//Layout setup
QVBoxLayout* pvbxLayout = new QVBoxLayout;
pvbxLayout->addWidget(new QLabel("<H1>Server</H1>"));
pvbxLayout->addWidget(m_ptxt);
setLayout(pvbxLayout);
}
```

Для установки сервера необходимо вызвать в конструкторе метод *listen()*. В этот метод необходимо передать номер порта, который мы получили в конструкторе. При возникновении ошибочных ситуаций, например невозможности захвата порта, этот метод возвратит значение *false*.

Далее производим соединение с сигналом *newConnection()*, который высылается при каждом присоединении нового клиента.

Для отображения информации создаем виджет многострочного текстового поля (указатель *m_ptxt*) и устанавливаем в нем вызовом метода *setReadOnly()* режим, делающий возможным только просмотр информации.

```
/*virtual*/ void MyServer::slotNewConnection()
{
```

```
QTcpSocket* pClientSocket = m_ptcpServer->nextPendingConnection();
connect(pClientSocket, SIGNAL(disconnected()),
pClientSocket, SLOT(deleteLater())
);
connect(pClientSocket, SIGNAL(readyRead()),
this, SLOT(slotReadClient())
 );
sendToClient(pClientSocket, "Server Response: Connected!");
}
```

Метод *slotNewConnection()* вызывается каждый раз при соединении с новым клиентом. Из этого метода выполняем соединения с сигналами *disconnected()* и *readyRead()*, которые предупреждают об отсоединении клиента и его готовности к передаче данных соответственно. В завершение вызываем метод *sendToServer()* для отсылки приветствия присоединенному клиенту. В этом методе вторым параметром мы передаем строку. Внутри самого метода будет сгенерирован временной штамп, который будет отослан клиенту вместе с переданной строкой.

Для подтверждения соединения с клиентом необходимо вызвать метод *nextPendingConnection()*, который возвратит сокет, посредством которого можно осуществлять дальнейшую связь с клиентом. Соединяем сигнал *disconnected()*, высылаемый сокетом при отсоединении клиента, со слотом *deleteLater()*, предназначенным для его последующего уничтожения. При поступлении запросов от клиентов высылается сигнал *readyToRead()*, который мы соединяем со слотом *slotReadClient()*.

```
void MyServer::slotReadClient()
{
QTcpSocket* pClientSocket = (QTcpSocket*)sender();
QDataStream in(pClientSocket);
in.setVersion(QDataStream::Qt_4_2);
for (;;) {
if (!m_nNextBlockSize) {
if (pClientSocket->bytesAvailable() < sizeof(quint16)) {
break;
}
```

```
in >> m_nNextBlockSize;
}
if (pClientSocket->bytesAvailable() < m_nNextBlockSize) {
break;
}
QTime time;
QString str;
in >> time >> str;
QString strMessage =
time.toString() + " " + "Client has sended - " + str;
m_ptxt->append(strMessage);
m_nNextBlockSize = 0;
sendToClient(pClientSocket,
"Server Response: Received \"" + str + "\""
);
}
}
```

Сначала производится преобразование указателя, возвращаемого методом *sender()*, к типу *QTcpSocket*. Цикл *for* нам нужен, так как не все высланные клиентом данные могут прийти одновременно. Поэтому сервер должен быть в состоянии получать как весь блок целиком, так и только часть блока, а также и все блоки сразу. Каждый переданный сокетом блок начинается полем, описывающим размер блока. Размер блока, который считывается при условии, что размер полученных данный не меньше двух байт и атрибут *m_nNextBlockSize* равен нулю (то есть размер блока неизвестен). Если доступных данных для чтения больше или равно размеру блока, тогда производится считывание из потока данных в переменные *time* и *str*. После этого значение переменной *time* преобразуется вызовом метода *toString()* в строку и используется вместе со строкой *str* для строки сообщения *strMessage*. Строка сообщения добавляется в виджет текстового поля вызовом метода *append()*. Анализ блока данных завершается присваиванием атрибуту *m_nNextBlockSize* значения 0, которое говорит о том, что размер очередного блока данных неизвестен. Вызовом метода *sendToClient()* сообщается клиенту о том, что успешно удалось прочитать высланные им данные.

```
void MyServer::sendToClient(QTcpSocket* pSocket, const QString &
str)
{
QByteArray  arrBlock;
QDataStream out(&arrBlock, QIODevice::WriteOnly);
out.setVersion(QDataStream::Qt_4_2);
out << quint16(0) << QTime::currentTime() << str;
out.device()->seek(0);
out << quint16(arrBlock.size() - sizeof(quint16));
pSocket->write(arrBlock);
}
```

В методе *sendToClient()* формируем данные, которые будут высланы клиенту. Но есть один нюанс, который заключается в том, что нам неизвестен размер блока и, следовательно, мы не можем записывать данные сразу в сокет, так как размер блока должен быть выслан в первую очередь. Поэтому прибегаем к следующим действиям. Для начала создаем объект *QByteArray*. В него записываются все данные блока, причем записывается размер, равный 0. После этого перемещается указатель на начало блока вызовом метода *seek()*, вычисляем и записываем размер блока в поток (*out*) (вычисляется как размер *arrBlock* с вычитанием из него *sizeof*(*quint16*)), затем созданный блок записывается в сокет вызовом метода *write()*.

Класс *QFtp.* Для облегчения работы *Qt* предоставляет специализированные классы *QFtp* и *QHttp*, базирующиеся на классе *QObject*. Работа этих классов асинхронна, поэтому не нужно бояться, что приложение будет заблокировано на время отправки сообщений или получения данных. Для того чтобы получать информацию о процессе выполнения команд, эти классы предоставляют сигналы, которые можно соединить, например для визуального отображения, со слотом *setProgress()* виджета индикатора прогресса. В конце выполнения операций высылается сигнал *done()*.

Передача файлов является часто выполняемой операцией практически во всех сетях. *FTP* (*File Transfer Protocol* — протокол передачи файлов) — наиболее известный из старых протоколов и являющийся одной из первых сетевых служб. Целью создания этого протокола было предоставление пользователям доступа к файлам на

удаленном компьютере. Кроме предоставляемого сервиса, протокол имеет целый ряд команд, с помощью которых можно управлять удаленным компьютером для передачи данных. Например:

– *get* — скопировать файл с удаленного сервера;

– *put* — копировать файл на удаленный сервер;

– *rmdir* — удалить каталог на удаленном сервере;

– *mkdir* — создать каталог на удаленном сервере;

– *cd* — открыть каталог на удаленном сервере;

– *rename* — переименовать файл или каталог на удаленном сервере;

– *close* — закрыть соединение с удаленным сервером.

Класс *QFtp* реализует сторону клиента *FTP*-протокола и содержит в себе методы для наиболее часто используемых операций с *FTP*. Названия этих методов соответствует названиям *FTP*-команд, например: *get()* соответствует команде *get*, а *put()* — команде *put* и т. д. Также класс *QFtp* предоставляет возможность исполнения любых *FTP*-команд. Для этого в метод *rawCommand()* нужно передать строку, содержащую нужную команду. Например:

ftp.rawCommand("MKDIR MyDir");

Каждый из методов возвращает идентификационный номер, который используется в сигналах класса *QFtp*. Этим можно воспользоваться для оповещения пользователя о проводимых операциях. Например, в начале исполнения одной из команд объект класса *QFtp* высылает сигнал *commandStarted*(*int*), а по завершению команды высылается сигнал *commandFinished*(*int*, *bool*). При этом идентификационный номер проводимой операции содержится в параметре типа *int*.

Класс *QFtp* содержит методы для осуществления соединения с *FTP*-сервером: *connectToHost()*, *login()*. Листинг демонстрирует считывание файла index.txt, размещенного на удаленном компьютере, посредством команд *FTP* и запись его в текущей директории под именем *index.txt*. Для этого сначала создается объект класса *QFile* и открывается файл для записи, а затем запускается серия из пяти *FTP*-команд. Нужно обратить внимание на второй аргумент метода *get()*, с помощью которого задается устройство ввода/вывода.

```
QFile file("index.txt");
QFtp ftp;
if (file.open(QIODevice::WriteOnly))
{
ftp.connectToHost("ftp.microsoft.com");
ftp.login();
ftp.cd("Softlib");
ftp.get("index.txt", &file);
ftp.close();
file.close();
}
```

Класс *QHttp.* *HTTP* (*Hyper Text Transfer Protocol* — протокол передачи гипертекста) является стандартным и самым известным протоколом для обмена данными в сетях. Его использование проще, чем рассмотренного *FTP*-протокола. В нем применяется только одно соединение, в то время как в *FTP* — два: одно — для отсылки команд, другое — для перекачивания данных.

Qt предоставляет класс *QHttp* для реализации стороны клиента *HTTP*-протокола. Использование этого класса очень похоже на использование класса *QFtp*. Пример демонстрирует запись растрового изображения, находящегося на *www.geocities.ru/mslerm/images*, в текущую директорию под именем *qtbook.gif.*

```
QFile file("book.gif");
QHttp http;
if (file.open(QIODevice::WriteOnly))
{
http.setHost("www.geocities.com");
http.get("/mslerm/images/qtbook.gif", &file);
http.close();
file.close();
}
```

Сокетные соединения — это стандартный механизм обмена данными через сеть в обоих направлениях. Каждому сокету соответствует пара значений: сетевой адрес и номер порта.

Сценарий «клиент-сервер» выглядит следующим образом: сервер занимает определенный порт, по которому он предоставляет свои услуги, после этого он начинает ожидать поступления запросов от

клиентов через этот порт. Чтобы подключиться к серверу, клиент должен знать его адрес и номер порта. Для соединения с сервером клиент должен создать сокет.

Для более простой работы с сетью *Qt* предоставляет специализированные классы *QFtp* и *QHttp*, представляющие собой реализации стороны клиента.

Работа с файлами, директориями и потоками ввода/вывода в *Qt.* Редко встречается приложение, которое не обращается к файлам. Работа с директориями (папками, в терминологии ОС *Windows*) и файлами — это та область, в которой не все операции являются платформонезависимыми, поэтому *Qt* предоставляет свою собственную поддержку этих операций, состоящую из следующих классов:

– *QDir* — для работы с директориями;

– *QFile* — для работы с файлами;

– *QFileInfo* — для получения файловой информации;

– *QIODevice* — абстрактный класс для ввода/вывода;

– *QBuffer* — для эмуляции файлов в памяти компьютера.

Ввод/вывод. Класс *QIODevice.* *QIODevice* — это абстрактный класс, обобщающий устройство ввода/вывода, который содержит виртуальные методы для открытия и закрытия устройства ввода/вывода, а также для чтения и записи блоков данных или отдельных символов.

Реализация конкретного устройства происходит в унаследованных классах.

В *Qt* есть четыре класса, наследующие класс *QIODevice:*

– *QFile* — класс для проведения операций с файлами;

– *QBuffer* — позволяет записывать и считывать данные в массив *QByteArray*, как будто бы это устройство или файл;

– *QAbstractSocket* — базовый класс для сетевой коммуникации посредством сокетов.

– *QProcess* —предоставляет возможность запуска процессов с дополнительными аргументами и позволяет обмениваться информацией с этими процессами посредством методов, определенных в *QIODevice.*

Для работы с устройством его необходимо открыть в одном из режимов, определенных в заголовочном файле класса *QIODevice:*

– *QIODevice::NotOpen* — устройство не открыто (это значение не имеет смысла передавать в метод *open());*

– *QIODevice::ReadOnly* — открытие устройства только для чтения данных;

– *QIODevice::writeOnly* — открытие устройства только для записи данных;

– *QIODevice::ReadWrite* — открытие устройства для чтения и записи данных (то же, что и *IO_ReadOnly | IO_WriteOnly*);

– *QIODevice::Append* — открытие устройства для добавления данных;

– *QIODevice::Unbuffered* — открытие для непосредственного доступа к данным в обход промежуточных буферов чтения и записи;

– *QIODevice::Text* — применяются преобразования символов переноса строки в зависимости от платформы. Для ОС *Windows*, например, *\r\n*, а для *MacOS* – /r;

– *QIODevice::Truncate* — все данные устройства, по возможности, должны быть удалены при открытии.

Для того чтобы в любой момент времени исполнения программы узнать, в каком из режимов было открыто устройство, нужно вызвать метод *openMode().*

Считывать и записывать данные можно с помощью методов *read()* и *write()* соответственно. Для чтения всех данных сразу определен метод *readAll(),* который возвращает их в объекте типа *QByteArray*. Строку или символ можно прочитать методами *readLine()* и *getChar()* соответственно.

В классе *QIODevice* определен метод для смены текущего положения *seek()*. Получить текущее положение можно вызовом метода *pos()*.Эти методы применимы только для прямого доступа к данным. При последовательном доступе, каким является сетевое соединение, они теряют смысл. Более того, в этом случае теряет смысл и метод *size()*, возвращающий размер данных устройства. Все эти операции применимы только для *QFile, QBuffer* и *QTemporaryFile.*

Для создания собственного класса устройства ввода/вывода, для которого *Qt* не предоставляет поддержки, необходимо унаследовать класс *QIODevice* и реализовать в нем методы *readData()* и *writeData()*. В большинстве случаев может потребоваться перезаписать методы *open(), close()* и *atEnd()*.

11 МАТЕМАТИЧЕСКИЕ ПАКЕТЫ Mathcad И Matlab

Вычислительная мощь компьютера позволяет использовать его как средство автоматизации научной работы. Для решения сложных расчетных задач используют специально написанные программы. В то же время в научной работе встречается широкий спектр задач ограниченной сложности, для решения которых можно использовать универсальные средства. К такого рода задачам относятся:

– подготовка научно-технических документов, содержащих текст и формулы, записанные в привычной для специалистов форме;

– вычисление результатов математических операций, в которых участвуют числовые константы, переменные и размерные физические величины;

– операции с векторами и матрицами;

– решение уравнений и систем уравнений (неравенств);

– статистические расчеты и анализ данных;

– построение двумерных и трехмерных графиков;

– тождественные преобразования выражений (в том числе упрощение), аналитическое решение уравнений и систем;

– дифференцирование и интегрирование, аналитическое и численное;

– решение дифференциальных уравнений;

– проведение серий расчетов с разными значениями начальных условий и других параметров.

К универсальным программам, пригодным для решения таких задач, относится программа ***Mathcad***. Это автоматизированная система, позволяющая динамически обрабатывать данные в числовом и аналитическом (формульном) виде.

Программа *Mathcad coneTdieT* содержит в себе возможности для проведения расчетов и подготовки форматированных научных и технических документов.

Научно-технические документы обычно содержат формулы, результаты расчетов в виде таблиц данных или графиков, текстовые

комментарии или описания, другие иллюстрации. В программе *Mathcad* им соответствуют два вида объектов: формулы и текстовые блоки. Формулы вычисляются с использованием числовых констант, переменных, функций (стандартных и определенных пользователем), а также общепринятых обозначений математических операций. Введенные в документ *Mathcad* формулы автоматически приводятся к стандартной научно-технической форме записи. Графики, которые автоматически строятся на основе результатов расчетов, также рассматриваются как формулы. Комментарии, описания и иллюстрации размещаются в текстовых блоках, которые игнорируются при проведении расчетов. Чтобы буквенные обозначения можно было использовать при расчетах по формулам, этим обозначениям должны быть сопоставлены числовые значения. В программе *Mathcad* буквенные обозначения рассматриваются как переменные, и их значения задаются при помощи оператора присваивания (вводится символом «:»). Таким же образом можно задать числовые последовательности, аналитически определенные функции, матрицы и векторы. Если все значения переменных известны, то для вычисления числового значения выражения (скалярного, векторного или матричного) надо подставить все числовые значения и произвести все заданные действия. В программе *Mathcad* для этого применяют оператор вычисления (вводится символом «=»). В ходе вычисления автоматически используются значения переменных и определения функций, заданные в документе ранее. Удобно задать значения известных параметров, провести вычисления с использованием аналитических формул, результат присвоить некоторой переменной, а затем использовать оператор вычисления для вывода значения этой переменной. Например:

$g:=9.8$
$M:=3$
$f:=Mg$
$f=29.4$

Изменение значения любой переменной, коррекция любой формулы означает, что все расчеты, зависящие от этой величины, необходимо проделать заново. Такая необходимость возникает при выборе подходящих значений параметров или условий, поиске

оптимального варианта, исследовании зависимости результата от начальных условий. Электронный документ, подготовленный в программе *MathCad*, готов к подобной ситуации. При изменении какой-либо формулы программа автоматически производит необходимые вычисления, обновляя изменившиеся значения и графики. Например, если документ содержит формулы $x := 4,\ \sqrt{x} = 2$, то при измени значения переменной *x* сразу же будет видно, что изменился и результат расчета: $x := 9,\ \sqrt{x} = 3$. При проведении расчетов с использованием реальных физических величин учитывают их единицу измерения. Чтобы расчет был корректен, все данные должны быть приведены в одну систему единиц — в этом случае результат расчетов получится в этой же системе. Здесь скрывается характерный источник ошибок при расчетах вручную. В программе *Mathcad* единицы измерения (в любой системе) присоединяют к значению величины с помощью знака умножения. Данные автоматически преобразуются в одну и ту же систему единиц (по умолчанию СИ) и обрабатываются в этом виде. Размерный результат выдается вместе с полученной единицей измерения. Например:

$U := 1000 khp \cdot t = 0.5 yr$ (*kph* — километры в час, *yr* — годы);

$S := V - t \cdot S = 4.383^{-10^8}\ m$ (результат получен в метрах).

При работе с матрицами приходится применять такие операции, как сложение матриц, умножение, транспонирование. Часто возникает необходимость в обращении матриц и в декомпозиции (разложении в произведение матриц специального вида). Для квадратных матриц представляет интерес поиск собственных значений и собственных векторов. Программа *Mathcad* позволяет выполнить все эти операции с помощью стандартных обозначений математических операторов (сложение, умножение) или встроенных функций. Например:

$$\begin{pmatrix}1 & 2\\ 1 & 1\end{pmatrix}^T = \begin{pmatrix}1 & 1\\ 2 & 1\end{pmatrix}$$

$$\left|\begin{pmatrix}1 & 2\\ 1 & 1\end{pmatrix}\right| = -1$$

$$\begin{pmatrix}1 & 2\\ 1 & 1\end{pmatrix}^{-1} = \begin{pmatrix}-1 & 2\\ 1 & -1\end{pmatrix}$$

$$\begin{pmatrix}1 & 2\\ 1 & 1\end{pmatrix}^T \cdot \begin{pmatrix}1 & 1\\ 2 & 1\end{pmatrix} = \begin{pmatrix}2 & 3\\ 3 & 5\end{pmatrix}$$

Уравнения и системы уравнений, возникающие в практических задачах, обычно можно решить только численно. Методы численного решения реализованы и в программе *Mathcad*. Блок уравнений и неравенств, требующих решения, записывается после ключевого слова *given* (дано). При записи уравнений используется знак логического равенства (комбинация клавиш CTRL+=). Значения переменных, удовлетворяющие системе уравнений и неравенств, находятся с помощью стандартной функции

find.

$x := 1 \quad y := 0$

given

$x - y = 2$

$\sin(x) = \sin(y)$

$$find(x, y) = \begin{pmatrix}2.571\\ 0.571\end{pmatrix}$$

При обработке результатов экспериментов часто встречаются задачи статистического анализа серий данных. Для такого рода задач программа *Mathcad* предоставляет средства интерполяции данных, предсказания дальнейшего поведения функции, а также построения функций заданного вида, наилучшим образом соответствующих имеющемуся набору данных. При статистическом анализе можно использовать стандартные функции распределения вероятности и генераторы случайных величин с заданным распределением. При аналитических вычислениях результат получают в нечисловой форме в

результате тождественных преобразований выражений. Более сложные преобразования позволяют находить аналитические решения некоторых уравнений и систем. Для такого рода вычислений в программе *Mathcad* используют оператор аналитического вычисления (клавиатурная комбинация CTRL+), а также команды меню *Symbolics* (аналитические вычисления). Переменные при аналитических вычислениях рассматриваются как неопределенные параметры.

Приемы работы с системой Mathcad. Документ программы *Mathcad msuBdieTCR* содержит объекты: формулы и текстовые блоки. В ходе расчетов формулы обрабатываются последовательно, слева направо и сверху вниз, а текстовые блоки игнорируются. Ввод информации осуществляется в месте расположения курсора. Программа *Mathcad* использует три вида курсоров. Если ни один объект не выбран, используется крестообразный курсор, определяющий место создания следующего объекта. При вводе формул используется уголковый курсору, указывающий текущий элемент выражения. При вводе данных в текстовый блок применяется текстовый курсор в виде вертикальной черты.

Ввод формул. Формулы — основные объекты рабочего листа. Новый объект по умолчанию является формулой. Чтобы начать ввод формулы, надо установить крестообразный курсор в нужное место и начать ввод букв, цифр, знаков операций. При этом создается область формулы, в которой появляется уголковый курсор, охватывающий текущий элемент формулы, например имя переменной (функции) или число. При вводе бинарного оператора по другую сторону знака операции автоматически появляется заполнитель в виде черного прямоугольника. В это место вводят очередной операнд. Для управления порядком операций используют скобки, которые можно вводить вручную. Уголковый курсор позволяет автоматизировать такие действия. Чтобы выделить элементы формулы, которые в рамках операции должны рассматриваться как единое целое, используют клавишу «ПРОБЕЛ». При каждом ее нажатии уголковый курсор «расширяется», охватывая элементы формулы, примыкающие к данному. После ввода знака операции элементы в пределах уголкового курсора

автоматически заключаются в скобки. Элементы формул можно вводить с клавиатуры или с помощью специальных панелей управления. Панели управления (рисунок 11.1) открывают с помощью меню *View* (Вид) или кнопками панели управления *Math* (Математика).

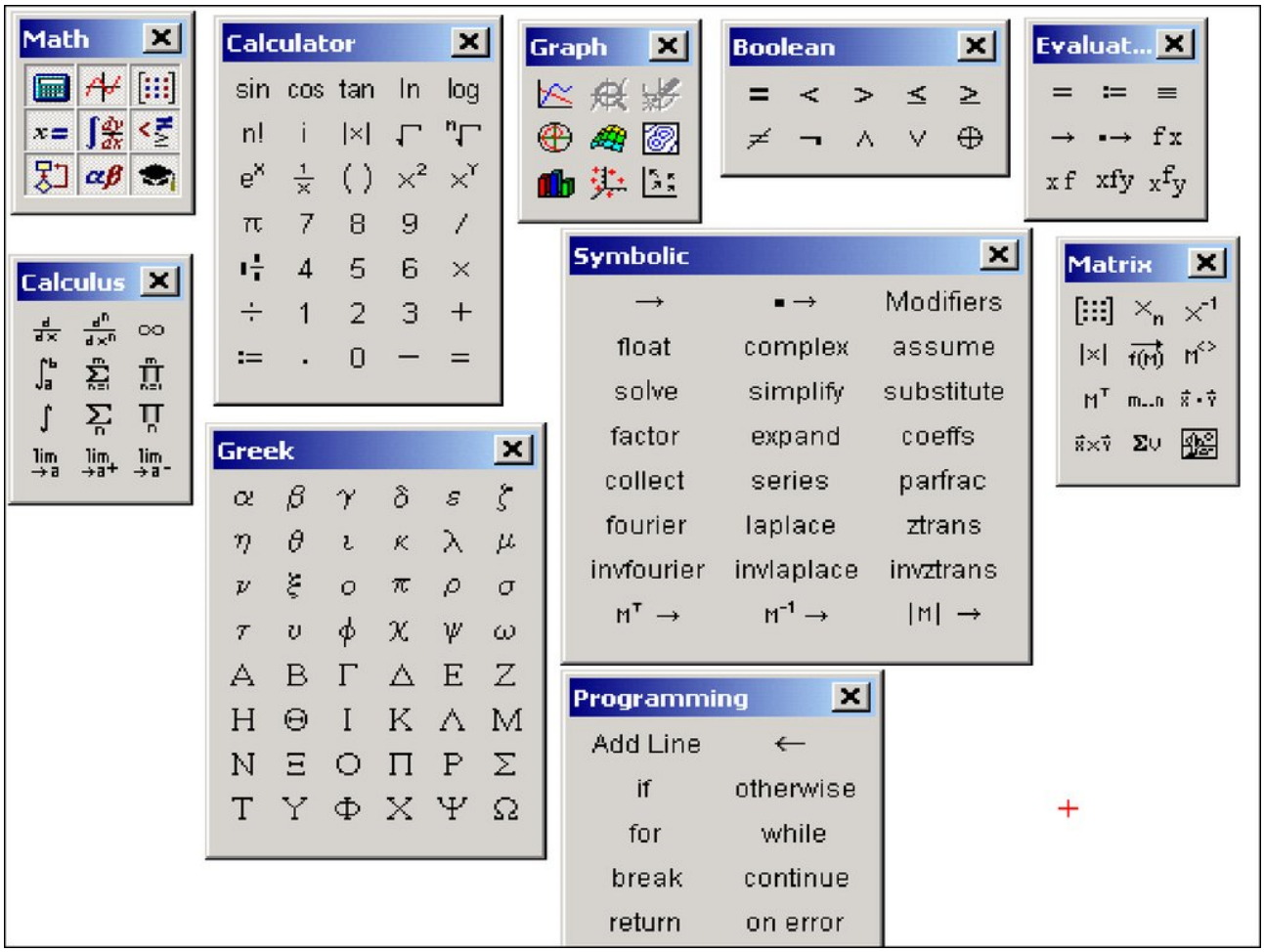

Рисунок 11.1 — Панели инструментов программы Mathcad для ввода формул

Для ввода элементов формул предназначены следующие панели управления:

– *Calculator* (Счет) для ввода чисел, знаков типичных математических операций и наиболее часто употребляемых стандартных функций;

– *Evaluation* (Вычисление) для ввода операторов вычисления;

– *Boolean* (Логика) для ввода знаков отношения и логических операций;

– *Graph* (График) для построения графиков;

– *Matrix* (Матрица) для ввода векторов и матриц, и задания матричных операций;

– *Calculus* (Исчисление) для задания операций, относящихся к математическому анализу;

– *Greek* (Греческий алфавит) для ввода греческих букв (их можно также вводить с клавиатуры, если сразу после ввода соответствующего латинского символа нажимать сочетание клавиш CTRL+G);

– *Symbolic* (Аналитические вычисления) для управления аналитическими преобразованиями. Введенное выражение обычно вычисляют или присваивают переменной.

Для вывода результата выражения используют знак вычисления, который выглядит как знак равенства и вводится при помощи кнопки *Evaluate Numerically* (Вычислить выражение) на панели инструментов *Evaluation* (Вычисление). Знак присваивания изображается как «:=», а вводится при помощи кнопки *Definition* (Определить) на панели инструментов *Evaluation* (Вычисление). Слева от знака присваивания указывают имя переменной. Оно может содержать латинские и греческие буквы, цифры, символы «'», «_», а также описательный индекс. Описательный индекс вводится с помощью символа «.» и изображается как нижний индекс, но является частью имени переменной. «Настоящие» индексы, определяющиеотдельный элемент вектора или матрицы, задаются по-другому. Переменную, которой присвоено значение, можно использовать далее в документе в вычисляемых выражениях. Чтобы узнать значение переменной, следует использовать оператор вычисления. В следующем примере вычислена площадь круга с радиусом 2 (использованы переменные *r* и *s*, значение постоянной *p* определено в программе *Mathcad* по умолчанию):

$r := 2$

$S := \pi r^2 = 12.566$

Ввод текста. Текст, помещенный в рабочий лист, содержит комментарии и описания и предназначен для ознакомления, а не для использования в расчетах. Программа *Mathcad* определяет назначение текущего блока автоматически при первом нажатии клавиши ПРОБЕЛ. Если введенный текст не может быть интерпретирован как

формула, блок преобразуется в текстовый и последующие данные рассматриваются как текст. Создать текстовый блок без использования автоматических средств позволяет команда «*Insert»,* «*Text Region»* («Вставка», «Текстовый блок»). Иногда требуется встроить формулу внутрь текстового блока. Для этого служит команда «*Insert»*, «*Math Region»* («Вставка», «Формула»).

Форматирование формул и текста. Для форматирования формул и текста в программе *Mathcad* используется панель инструментов *Formatting* (Форматирование). С ее помощью можно индивидуально отформатировать любую формулу или текстовый блок, задав гарнитуру и размер шрифта, а также полужирное, курсивное или подчеркнутое начертание символов. Кроме того, в текстовых блоках можно задавать тип выравнивания *VL*, маркированные и нумерованные списки. В качестве средств автоматизации используются стили оформления. Выбрать стиль оформления текстового блока или элемента формулы можно из списка *Style* (Стиль) на панели инструментов *Formatting* (Форматирование). Для формул и текстовых блоков применяются разные наборы стилей. Чтобы изменить стиль оформления формулы или создать новый стиль, используется команда «*Format»*, «*Equation»* («Формат», «Выражение»). Изменение стандартных стилей *Variable*s (Переменные) и *Constants* (Константы) влияет на отображение формул по всему документу. Стиль оформления имени переменной учитывается при ее определении. При оформлении текстовых блоков можно использовать более широкий набор стилей. Настройка стилей текстовых блоков производится при помощи команды *«Format»,* «*Style»* («Формат», «Стиль»).

Работа с матрицами. Векторы и матрицы рассматриваются в программе *Mathcad* как одномерные и двумерные массивы данных. Число строк и столбцов матрицы задается в диалоговом окне *Insert Matrix* (Вставка матрицы), которое открывают командой «*Insert»*, «*Matrix»* («Вставка», «Матрица»). Вектор задается как матрица, имеющая один столбец. После щелчка на кнопке *OK* в формулу вставляется матрица, содержащая вместо элементов заполнители. Вместо каждого заполнителя надо вставить число, переменную или выраже-

ние. Для матриц определены следующие операции: сложение, умножение на число, перемножение и прочие. Допустимо использование матриц вместо скалярных выражений: в этом случае предполагается, что указанные действия должны быть применены к каждому элементу матрицы, и результат также представляется в виде матрицы. Например, выражение $M + 3$, где M — матрица, означает, что к каждому элементу матрицы прибавляется число. Если требуется явно указать необходимость поэлементного применения операции к матрице, используют знак векторизации, для ввода которого служит кнопка «*Vectorize*» («Векторизация») на панели инструментов *Matrix* (Матрица). Например:

$\begin{pmatrix} 1 & 3 \\ -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -1 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 7 & 2 \\ 1 & 2 \end{pmatrix}$ — обычное произведение матриц;

$\begin{pmatrix} 1 & 3 \\ -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -1 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & -3 \\ -2 & 1 \end{pmatrix}$ — поэлементное произведение матриц с использованием векторизации.

Для работы с элементами матрицы используют индексы элементов. Нумерация строк и столбцов матрицы начинается с нуля. Индекс элемента задается числом, переменной или выражением и отображается как нижний индекс. Он вводится после щелчка на кнопке *Subscript* (Индекс) на панели инструментов *Matrix* (Матрица). Пара индексов, определяющих элемент матрицы, разделяется запятой, например при построении графиков требуется выделить вектор, представляющий собой столбец матрицы. Номер столбца матрицы отображается как верхний индекс, заключенный в угловые скобки, например $M^{\langle 0 \rangle}$. Для его ввода используется кнопка «*Matrix Column*» («Столбец») на панели инструментов *Matrix* (Матрица). Чтобы задать общую формулу элементов матрицы типа $M_{ij} := i + j$, используют диапазоны. Диапазон фактически представляет собой вектор, содержащий арифметическую прогрессию, определенную первым, вторым и последним элементами. Чтобы задать диапазон, следует указать значение первого элемента, через запятую — значение второго

и через точку с запятой — значение последнего элемента. Точка с запятой при задании диапазона отображается как две точки «..». Диапазон можно использовать как значение переменной, например $x := 0,0.01,\ldots,1$. Если разность прогрессии равна единице (то есть элементы просто нумеруются), значение второго элемента и соответствующую запятую опускают. Например, чтобы сформировать по приведенной выше формуле матрицу размером 6x6, перед этой формулой надо указать $i := 0\ldots5,\ j := 0\ldots5$. При формировании матрицы путем присвоения значения ее элементам размеры матрицы можно не задавать заранее. Всем неопределенным элементам автоматически присваиваются нулевые значения. Например, формула $M_{5,5} = 1$ создает матрицу *M* размером 6x6, у которой все элементы, кроме расположенного в правом нижнем углу, равны 0.

Стандартные и пользовательские функции. Произвольные зависимости между входными и выходными параметрами задаются при помощи функций. Функции принимают набор параметров и возвращают значение, скалярное или векторное (матричное). В формулах можно использовать стандартные встроенные функции, а также функции, определенные пользователем. Чтобы использовать функцию в выражении, надо определить значения входных параметров в скобках после имени функции. Имена простейших математических функций можно ввести с панели инструментов *Calculator* (Счет). Информацию о других функциях можно почерпнуть в справочной системе. Вставить в выражение стандартную функцию можно при помощи команды «*Insert*», «*Function*» («Вставка», «Функция»). В диалоговом окне *Insert Function* (Вставка функции) слева выбирается категория, к которой относится функция, а справа – конкретная функция. В нижней части окна выдается информация о выбранной функции. При вводе функции через это диалоговое окно автоматически добавляются скобки и заполнители для значений параметров. Пользовательские функции должны быть сначала определены. Определение задается при помощи оператора присваивания. В левой части указывается имя пользовательской функции и, в скобках, формальные параметры – переменные, от которых она зависит. Справа от знака

присваивания эти переменные должны использоваться в выражении. При использовании пользовательской функции в последующих формулах ее имя вводят вручную. В диалоговом окне *InsertFunction* (Вставка функции) оно не отображается.

Решение уравнений и систем. Для численного поиска корней уравнения в программе *Mathcad* используется функция *root*. Она служит для решения уравнений вида $f(x)=0$, где $f(x)$ — выражение, корни которого нужно найти, а x — неизвестное. Для поиска корней с помощью функции *root* надо присвоить искомой переменной начальное значение, а затем вычислить корень при помощи вызова функции:

$$root(f(x),x).$$

Здесь $f(x)$ — функция переменной x, используемой в качестве второго параметра. Функция *root* возвращает значение независимой переменной, обращающее функцию $f(x)$ в 0. Например:

$$x:=1$$

$$root(2-\sin(x)-x,x)=1.895$$

Если уравнение имеет несколько корней (как в данном примере), то результат, выдаваемый функцией *root*, зависит от выбранного начального приближения.

Если надо решить систему уравнений (неравенств), используют так называемый блок решения, который начинается с ключевого слова *gwen* (дано) и заканчивается вызовом функции *find* (найти). Между ними располагают «логические утверждения», задающие ограничения на значения искомых величин, иными словами, уравнения и неравенства. Всем переменным, используемым для обозначения неизвестных величин, должны быть заранее присвоены начальные значения. Чтобы записать уравнение, в котором утверждается, что левая и правая части равны, используется знак логического равенства — кнопка «*Equalto»* («Равно») на панели инструментов *Boolean* (Логика). Другие знаки логических условий также можно найти на этой панели. Заканчивается блок решения вызовом функции *find*, у которой в качестве аргументов должны быть перечислены искомые величины. Эта

функция возвращает вектор, содержащий вычисленные значения неизвестных. Например:

$$x := 0$$

$$y := 0$$

given

$$x + y = 1$$

$$x^2 + y^2 = 4$$

$$find(x, y) = \begin{pmatrix} 1.823 \\ 0.823 \end{pmatrix}$$

Построение графиков. Чтобы построить двумерный график в координатных осях *XY*, надо дать команду «*Insert*», «*Graph*», «*X-Y Plot*» («Вставка», «График», «Декартовы координаты»). В области размещения графика находятся заполнители для указания отображаемых выражений и диапазона изменения величин. Заполнитель у середины оси координат предназначен для переменной или выражения, отображаемого по этой оси. Обычно используют диапазон или вектор значений. Граничные значения по осям выбираются автоматически в соответствии с диапазоном изменения величины, но их можно задать и вручную. В одной графической области можно построить несколько графиков. Для этого надо у соответствующей оси перечислить несколько выражений через запятую. Разные кривые изображаются разным цветом, а для форматирования графика надо дважды щелкнуть на области графика. Для управления отображением построенных линий служит вкладка «*Traces*» («Линии») в открывшемся диалоговом окне. Текущий формат каждой линии приведен в списке, а под списком расположены элементы управления, позволяющие изменять формат. Поле «*Legend Label*» («Описание») задает описание линии, которое отображается только при сбросе флажка «*Hide Legend*» («Скрыть описание»). Список «*Symbol*» («Символ») позволяет выбрать маркеры для отдельных точек, список «*Line*» («Тип линии») задает тип линии, список «*Color*» («Цвет») – цвет. Список «*Type*» («Тип») определяет способ связи отдельных точек, а список «*Weight*» («Толщина») – толщину линии. Точно так же можно построить и отформатировать график в полярных координатах. Для его

построения надо дать команду «*Insert*», «*Graph*», «*Polar Plot*» («Вставка», «График», «Полярные координаты»).

Для построения простейшего трехмерного графика, необходимо задать матрицу значений. Отобразить эту матрицу можно в виде поверхности — «*Insert*», «*Graph*», «*Surface Plot*» («Вставка», «График», «Поверхность»), столбчатой диаграммы — «*Insert*», «*Graph*», «*3D Bar Plot*» («Вставка», «График», «Столбчатая диаграмма») или линий уровня – «*Insert*», «*Graph*», «*Contour Plot*» («Вставка», «График», «Линии уровня»). Для отображения векторного поля при помощи команды «*Insert*», «*Graph*», «*Vector Field Plot*» («Вставка», «График», «Поле векторов») значения матрицы должны быть комплексными. В этом случае в каждой точке графика отображается вектор с координатами, равными действительной и мнимой частям элемента матрицы. Во всех этих случаях после создания области графика необходимо указать вместо заполнителя имя матрицы, содержащей необходимые значения. Для построения параметрического точечного графика командой «*Insert*», «*Graph*», «*3D Scatter Plot*» («Вставка», «График», «Точки в пространстве») необходимо задать три вектора с одинаковым числом элементов, которые соответствуют *x, y* и двум координатам точек, отображаемых на графике. В области графика эти три *Beicropa* указываются внутри скобок через запятую. Аналогичным образом можно построить поверхность, заданную параметрически. Для этого надо задать три матрицы, содержащие, соответственно, *x,y* и *z* координаты точек поверхности. Теперь надо дать команду построения поверхности «*Insert*», «*Graph*», «*Surface Plot*» («Вставка», «График», «Поверхность») и указать в области графика эти три матрицы в скобках и через запятую. Таким образом можно построить практически любую криволинейную поверхность (например, представленную на рисунок 11.2), в том числе с самопересечениями. Диалоговое окно для форматирования трехмерных графиков также открывают двойным щелчком на области графика.

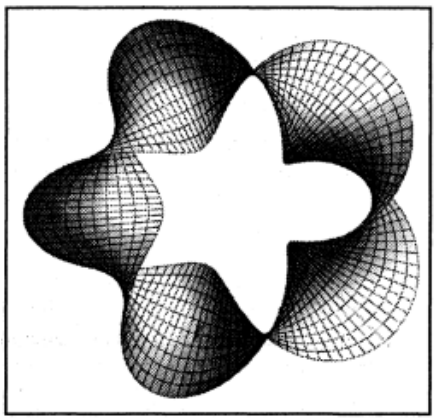

Рисунок 11.2 — Пятикратно перекрученная замкнутая лента, заданная параметрически

Аналитические вычисления. С помощью аналитических вычислений находят аналитические или полные решения уравнений и систем, проводят преобразования сложных выражений (например, упрощение). Иначе говоря, при таком подходе можно получить нечисловой результат. В программе *Mathcad* конкретные значения, присвоенные переменным, при этом игнорируются – переменные рассматриваются как неопределенные параметры. Команды для выполнения аналитических вычислений в основном сосредоточены в меню *Symbolics* (Аналитические вычисления).

Чтобы упростить выражение (или часть выражения), надо выбрать его при помощи уголкового курсора и дать команду «*Symbolics*», «*Simplify*» («Аналитические вычисления», «Упростить»). При этом выполняются арифметические действия, сокращаются общие множители и приводятся подобные члены, применяются тригонометрические тождества, упрощаются выражения с радикалами, а также выражения, содержащие прямую и обратную функции (типа e^x). Некоторые действия по раскрытию скобок и упрощению сложных тригонометрических выражений требуют применения команды «*Symbolics*», «*Expand*» («Аналитические вычисления», «Раскрыть»). Команду «*Symbolics*», «*Simplify*» («Аналитические вычисления», «Упростить») применяют в наиболее сложных случаях. Например, с ее помощью можно:

– вычислить предел числовой последовательности, заданной общим членом;

– найти общую формулу для суммы членов числовой последовательности, заданной общим членом;

– вычислить производную данной функции;

– найти первообразную данной функции или значение определенного интеграла.

Другие возможности меню «*Symbolics*» (Аналитические вычисления) состоят в выполнении аналитических операций, ориентированных на переменную, использованную в выражении. Для этого надо выделить в выражении переменную и выбрать команду из меню «*Symbolics»,* «*Variable*» («Аналитические вычисления», «Переменная»). Команда «*Solve»* («Решить») ищет корни функции, заданной данным выражением.

Другие возможности использования этого меню включают:

– аналитическое дифференцирование и интегрирование: «*Symbolics»*, «*Variable»*, «*Differentiate»* («Аналитические вычисления», «Переменная», «Дифференцировать») и «*Symbolics»*, «*Variable»*, «*Integrate»* («Аналитические вычисления», «Переменная», «Интегрировать»);

– замена переменной: «*Symbolics»*, «*Variable»*, «*Differentiate»* («Аналитические вычисления», «Переменная», «Дифференцировать») — вместо переменной подставляется содержимое буфера обмена;

– разложение в ряд Тейлора: «*Symbolics»*, «*Variable*», «*Expand to Series*» («*Аналитические вычисления*», «*Переменная*», «*Разложить в ряд*»);

– представление дробно-рациональной функции в виде суммы простых дробей с линейными и квадратичными знаменателями: «*Symbolics»*, «*Variable»*, «*Convert to Partial Fraction»* («Аналитические вычисления», «Переменная», «Преобразовать в простые дроби»).

Наконец, самым мощным инструментом аналитических вычислений является оператор аналитического вычисления, который вводится с помощью кнопки «*Evaluate Symbolically»* («Вычислить аналитически») на панели инструментов «*Evaluation»* («Вычисление»). Его

можно, например, использовать для аналитического решения системы уравнений и неравенств. Любое аналитическое вычисление можно применить с помощью ключевого слова. Для этого используют кнопку «*Symbolic Keyword Evaluation*» («Вычисление с ключевым словом») на панели инструментов «*Evaluation*» («Вычисление»). Ключевые слова вводятся через панель инструментов «*Symbolics*» («Аналитические вычисления»). Они полностью охватывают возможности, заключенные в меню «*Symbolics*» («Аналитические вычисления»), позволяя также задавать дополнительные параметры (рисунок 11.3).

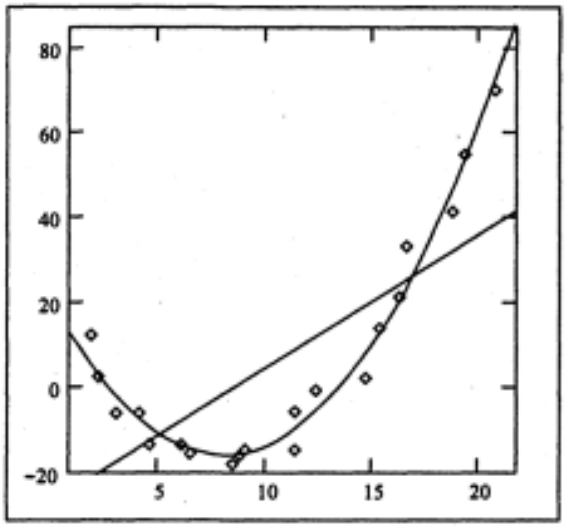

Рисунок 11.3 — Набор точек аппроксимирован
с помощью многочленов первого и второго порядка

Результат можно использовать для анализа решения при различных значениях этих переменных. При аналитическом решении уравнений и систем за одну операцию можно найти все существующие решения.

Дифференцирование и интегрирование заданных функций вручную – обычно несложная, но трудоемкая операция. В программе *Mathcad* для вычисления производной, а также неопределенных и определенных интегралов могут использоваться символические вычисления с помощью меню «*Symbolics*», «*Variable*» («Аналитические вычисления», «Переменная»). Если функция не задана аналитически или не позволяет получить первообразную в виде формулы,

имеется возможность численного дифференцирования и численного расчета определенных интегралов.

Численные методы используют и для решения дифференциальных уравнений. С помощью программы *Mathcad* можно решать уравнения и системы уравнений первого порядка с заданными начальными условиями. Уравнение более высокого порядка надо сначала преобразовать в систему уравнений первого порядка.

Matlab **—** матричная лаборатория, наиболее развитая система программирования для научно-технических расчетов, дополненная к настоящему времени несколькими десятками более частных приложений, относящихся к вычислительной математике, обработке информации, конструированию электронных приборов, экономике и ряду других разделов прикладной науки.

Matlab предназначен прежде всего для программирования численных алгоритмов. Он разрабатывается уже более 15 лет и возник на основе более ранних прикладных пакетов *LINPACK* и *EIGPACK*, созданных в 1970-е гг. в США, и в свою очередь повлиял на появление таких систем, как *Mathcad*, *MAPLE* и *Mathematica*. Совершенствование системы *Matlab* происходило как в связи с достижениями в вычислительной математике, так и в связи с изменениями в архитектуре персональных компьютеров и развитием общесистемных средств. Со временем *Matlab* был дополнен целым рядом приложений (*toolboxes*), далеко раздвинувших границы его применимости.

Matlab — система программирования высокого уровня, которая работает как интерпретатор и включает большой набор инструкций (команд) для выполнения самых разнообразных вычислений, задания структур данных и графического представления информации. Команды эти разбиты на тематические группы, расположенные в различных директориях системы. Команды с большим возможным объемом вычислений написаны на *C*, но много и таких команд, которые представлены в терминах этих первых. Поэтому система оказывается почти открытой для пользователя. Имеются большие возможности для вывода двумерной и трехмерной графики и средства управления ею. Пользователь может без особых затруднений добавлять свои команды и писать программы в терминах уже существующих команд. Можно

обмениваться данными с программами на этих языках, а из них обращаться к системе. Краткость и наглядность программирования и исключительные возможности визуализации результатов делают систему очень эффективной при поисках и апробации новых алгоритмов, при проведении разовых расчетов и в учебном процессе, поскольку ее можно осваивать без предварительного знакомства с основами программирования и выполнять такие сложные примеры, которые невозможно делать с использованием других систем.

Система *Matlab* состоит из пяти основных частей.

Язык *Matlab*. Это язык матриц и массивов высокого уровня с управлением потоками, функциями, структурами данных, вводом-выводом и особенностями объектно-ориентированного программирования.

Среда *Matlab*. Это набор инструментов и приспособлений, с которыми работает пользователь или программист *Matlab*. Она включает в себя средства для управления переменными в рабочем пространстве *Matlab*, вводом и выводом данных, а также создания, контроля и отладки *M*-файлов и приложений *Matlab*.

Управляемая графика. Это графическая система *Matlab*, которая включает в себя команды высокого уровня для визуализации двух- и трехмерных данных, обработки изображений, анимации и иллюстрированной графики. Она также включает в себя команды низкого уровня, позволяющие полностью редактировать внешний вид графики, так же как при создании графического пользовательского интерфейса (*GUI*) для *Matlab* приложений.

Библиотека математических функций. Это обширная коллекция вычислительных алгоритмов: от элементарных функций, таких как сумма, синус, косинус, комплексная арифметика, до более сложных, таких как обращение матриц, нахождение собственных значений, функции Бесселя, быстрое преобразование Фурье.

Программный интерфейс. Это библиотека, которая позволяет писать программы, взаимодействующие с *Matlab*. Она включает средства для вызова программ из *Matlab* (динамическая связь) как вычислительного инструмента и для чтения – записи *MAT*-файлов.

Matlab — это уникальная коллекция реализаций современных численных методов.

Matlab включает в себя большое количество уже готовых математических средств, функций и операций, которые решают множество практических задач, для чего ранее приходилось готовить достаточно сложные программы.

Пакет применяется в основном для работы с массивами данных, матрицами.

Предназначения среды *Matlab*:

– математические расчеты;

– разработка алгоритмов;

– обработка экспериментальных данных;

– визуализация данных;

– моделирование систем и процессов;

– разработка приложений;

– матричные, векторные, логические операторы;

– элементарные и специальные функции;

– полиномиальная арифметика;

– многомерные массивы; массивы записей; массивы ячеек;

– дифференциальные уравнения;

– решение систем линейных уравнений;

– поиск корней нелинейных алгебраических уравнений;

– оптимизация функций нескольких переменных;

– одномерная и многомерная интерполяция.

Начиная с версии 6, программа *Matlab* имеет интерфейс, который называется «Рабочий стол программы *Matlab»* (далее — «Рабочий стол»). В этот интерфейс входит окно «*Command Window»* («Командное окно»). По умолчанию «Рабочий стол» включает в себя четыре окна: окно «*Command Window»* («Командное окно») в правой части «Рабочего стола», окна «*Current Directory»* («Текущий каталог») и «*Workspace»* («Рабочая область») в верхней левой части и окно «*Command History»* («История команд») в нижней левой части. Обратите внимание, что для переключения между окнами «*Current Directory»* («Текущий каталог») и «*Workspace»* («Рабочая область») имеются вкладки, повторяющие название окна. Можно управлять

отображением окон с помощью меню «Рабочего стола» (в версии 6 меню «*View*» («Вид»)), расположенного в верхней части «Рабочего стола», кроме того, вы можете регулировать размеры окон путем перетаскивания границ окон с помощью мыши. Окно «*Command Window*» («Командное окно») представляет собой окно, в котором вы вводите команды и инструкции, заставляющие программу *Matlab* вычислять, рисовать и выполнять множество других впечатляющих вещей.

Рабочий стол включает в себя строку меню и панель инструментов. Панель инструментов содержит значки (ярлыки), предоставляющие доступ к некоторым элементам программы, которые вы можете выбрать через меню. Многие элементы меню имеют также клавиатурные комбинации, которые отображаются справа от пункта меню. Некоторые из этих клавиатурных комбинаций зависят от вашей операционной системы. Каждое окно на Рабочем столе содержит две маленькие кнопки в верхнем правом углу.

Хотя Рабочий стол предоставляет некоторые новые возможности и общий интерфейс для версий программы *Matlab* под управлением операционных систем *Windows* и *Unix*, тем не менее программа с открытым Рабочим столом может работать гораздо медленнее, чем базовый интерфейс окна «*Command Window*» («Командное окно»), особенно на старых компьютерах. Все переменные находятся в области памяти компьютера, называемой «Рабочей областью». Полный перечень заданных переменных отображается в одноименном окне «*Workspace*» («Рабочая область»). Отобразить это окно можно введя команду «*workspace*», или, при открытом Рабочем столе, щелкнув мышью на вкладке «*Workspace*» («Рабочая область») в нижней части окна «*Current Directory*» («Текущий каталог»). Окно «*Workspace*» («Рабочая область») содержит список текущих переменных и их размеры (но не значения переменных).

Основные команды главного меню *Matlab.* Открытая позиция строки меню содержит различные операции и команды. Выделенная команда или операция исполняется при нажатии клавиши *Enter* (Ввод). Выполнение команды можно также осуществить щелчком

мыши или нажатием на клавиатуре клавиши, соответствующей выделенному символу в названии команды.

Между командами и операциями нет особых отличий, и в литературе по информатике их часто путают. Будем считать командой действие, которое исполняется немедленно. А операцией — действие, которое требует определенной подготовки, например открытие окна для установки определенных параметров.

Параметр (*option*) — это значение определенной величины, действующее во время текущей сессии. Параметрами обычно являются указания на применяемые наборы шрифтов, размеры окна, цвет фона и т. д.

Перейдем к описанию основного меню системы *Matlab*:

– *File* — работа с файлами;

– *Edit* — редактирование сессии;

– *View* — вывод и скрытие панели инструментов;

– *Web* — доступ к Интернет-ресурсам;

– *Windows* — установка *Window*s-свойств окна;

– *Help* — доступ к справочным подсистемам.

Подменю *File* содержит ряд операций и команд для работы с файлами:

– *New* — открывает подменю с позициями:

– *M–file* — открытие окна редактора/отладчика *m*-файлов;

– *Figure* — открытие пустого окна графики;

– *Model* — открытие пустого окна для создания *Simulink*-модели;

– *GUI* — открытие окна разработки элементов графического интерфейса пользователя;

– *Open* — открывает окно загрузки файла;

– *Close Command Windows* — закрывает окно командного режима работы (оно при этом исчезает с экрана);

– *Import data* — открывает окно импорта файлов данных;

– *Save Workspace As* — открывает окно записи рабочей области в виде файла с заданным именем;

– *Set Path* — открывает окно установки путей доступа файловой системы;

– *Preferences* — открывает окно настройки элементов интерфейса;

– *Print* — открывает окно печати всего текущего документа;

– *Print Selection* — открывает окно печати выделенной части документа;

– *Exit* — завершает работу с системой.

Меню «*Edit*» содержит операции и команды редактирования, типичные для большинства приложений *Windows*:

– *Undo* (отменить) — отмена результата предшествующей операции;

– *Redo* (повторить) — отмена действия последней операции «*Undo*»;

– *Cut* (вырезать) — вырезание выделенного фрагмента и перенос его в буфер;

– *Copy* (копировать) — копирование выделенного фрагмента в буфер;

– *Paste* (вставить) — вставка фрагмента из буфера в текущую позицию курсора;

– *Clear* (очистить) — операция очистки выделенной области;

– *Select All* (выделить) — выделение всей сессии;

– *Delete* (стереть) — уничтожение выделенного объекта;

– *Clear Command Windows* (очистить командное окно) — очистка текста сессии (с сохранением созданных объектов);

– *Clear Command History* — очистка окна истории;

– *Clear Workspace* — очистка окна браузера рабочей области.

Элементарные математические выражения. Как и большинство других языков программирования, *Matlab* предоставляет возможность использования математических выражений, но в отличие от многих из них эти выражения в *Matlab* включают матрицы. Основные составляющие выражения:

– переменные;

– числа;

– операторы;

– функции.

Переменные. В *Matlab* нет необходимости в определении типа переменных или размерности. Когда *Matlab* встречает новое имя переменной, он автоматически создает переменную и выделяет соответствующий объем памяти. Если переменная уже существует, *Matlab* изменяет ее состав и, если это необходимо, выделяет дополнительную память. Например, *num_students* = 25 создает матрицу 1×1 с именем *num_students* и сохраняет значение 25 в ее единственном элементе.

Имена переменных состоят из букв, цифр или символов подчеркивания. *Matlab* использует только первые 31 символ имени переменной. *Matlab* чувствителен к регистрам, он различает заглавные и строчные буквы. Поэтому *A* и *a* — не одна и та же переменная. Чтобы увидеть матрицу, связанную с переменной, просто введите название переменной.

Числа. *Matlab* использует принятую десятичную систему счисления, с необязательной десятичной точкой и знаками плюс-минус для чисел. Научная система счисления использует букву *e* для определения множителя степени десяти. Мнимые числа используют *i* или *j* как суффикс. Некоторые примеры правильных чисел приведены ниже:

0.0001

9.6397238

$1.60210e^{-20}$

$6.02252e^{23}$

$-3.14159\,j$

$3e5i$

Числа с плавающей точкой обладают ограниченной точностью, приблизительно 16 значащих цифр, и ограниченным диапазоном, приблизительно от 10^{-308} до 10^{308}.

Операторы. Выражения используют обычные арифметические операции и правила старшинства:

«+» — сложение;

«–» — вычитание;

«*» — умножение;

«/» — деление;

«'» — комплексно сопряженное транспонирование;

«()» — определение порядка вычисления.

Специальные символы:

«[ ]» — квадратные скобки используют для создания матриц и векторов;

«,» — запятая применяется для разделения элементов матриц и операторов в строке ввода;

«;» — точка с запятой отделяет строки матриц, а точка с запятой в конце оператора (команды) отменяет вывод результата на экран;

«:» — двоеточие используется для указания диапазона (интервала изменения величины) и в качестве знака групповой операции над элементами матриц;

«%» — знак процента обозначает начало комментария;

«!» — отмечает начало команды «*DOS*»;

«’» — апостроф указывает на символьные строки.

Базовые функции:

– *ABS* — абсолютное значение;

– *ANGLE* — аргумент комплексного числа;

– *REAL, IMAG* — действительная и мнимая части комплексного числа;

– *CONJ* — операция комплексного сопряжения;

– *SIGN* — вычисление знака числа;

–*CEIL, FIX, FLOOR, ROUND* — функции округления;

– *REM* — функция остатка;

– *GCD* — наибольший общий делитель;

– *LCM* — наименьшее общее кратное;

– *RAT, RATS* — представление результата в виде рационального числа или цепной дроби.

Трансцендентные функции:

– *SQRT* — квадратный корень;

– *EXP* — экспоненциальная функция;

– *LOG* — функция натурального логарифма;

– *POW2* — экспонента по основанию 2;

– *NEXTPOW2* — ближайшая степень по основанию 2;

– *LOG2* — функции логарифма;

– *LOG10* — функции логарифма.

Тригонометрические функции:

– *SIN, SINH* — функции синуса;

– *ASIN, ASINH* — функции обратного синуса;

– *CSC, CSCH* — функции косеканса;

– *ACSC, ACSCH* — функции обратного косеканса;

– *COS, COSH* — функции косинуса;

– *ACOS, ACOSH* — функции обратного косинуса;

– *SEC, SECH* — функции секанса;

– *ASEC, ASECH* — функции обратного секанса;

– *TAN, TANH* — функции тангенса;

– *ATAN, ATAN2, ATANH* — функции обратного тангенса;

– *COT, COTH* — функции котангенса;

– *ACOT, ACOTH* — функции обратного котангенса.

Преобразования системы координат:

– *CART2POL* — преобразование декартовой системы координат в полярную и цилиндрическую;

– *CART2SPH* — преобразование декартовой системы координат в сферическую;

– *POL2CART* — преобразование полярной и цилиндрической систем координат в декартову;

– *SPH2CART* — преобразование сферической системы координат в декартову.

Специальные функции:

– *BESSEL* — функции Бесселя;

– *BETA, BETACORE, BETAINC, BETALN* — бета-функции;

– *ELLIPJ* — эллиптические функции Якоби;

– *ELLIPKE* — полные эллиптические интегралы;

– *ERF, ERFCORE, ERFC, ERFCX, ERFINV* — функции ошибок;

– *GAMMA, GAMMAINC, GAMMALN* — гамма-функции.

Пакеты расширений *Simulink*:

а) *Neural Networks Toolbox*;

б) *Fuzzy Logic Toolbox*;

в) *Symbolic Math Toolbox*;

г) пакеты математических вычислений:

1) *NAG Foundation Toolbox*;
2) *Spline Toolbox*;
3) *Statistics Toolbox*;
4) *Optimization Toolbox*;
5) *Partial Differential Equations Toolbox*;

д) пакеты анализа и синтеза систем управления:

1) *Control System Toolbox*;
2) *Nonlinear Control Design Toolbox*;
3) *Robust Control Toolbox*;
4) *Model Predictive Control Toolbox*;
5) *Analysis and Synthesis*;
6) *Stateflow*;
7) *Quantitative Feedback Theory Toolbox*;

е) *LMI Control Toolbox*;

ж) пакеты идентификации систем:

1) *System Identification Toolbox*;
2) *Frequency Domain System Identification Toolbox*;

з) дополнительные пакеты расширения Matlab:

1) *Communications Toolbox*;
2) *Digital Signal Processing (DSP) Blockset*;
3) *Fixed-Point Blockset*;

и) пакеты для обработки сигналов и изображений:

1) *Signal Processing Toolbox*;
2) *Higher-Order Spectral Analysis Toolbox*;
3) *Image Processing Toolbox*;
4) *Wavelet Toolbox*;

к) прочие пакеты прикладных программ:

1) *Financial Toolbox*;
2) *Mapping Toolbox*;
3) *Power System Blockset*;
4) *Data Acquisition Toolbox и Instrument Control Toolbox*;
5) *Database toolbox и Virtual Reality Toolbox*;
6) *Excel Link*;
7) *Matlab Compiler*.

Пакет расширения *Simulink* системы *Matlab* является ядром интерактивного программного комплекса, предназначенного для математического моделирования линейных и нелинейных динамических систем и устройств, представленных функциональной блок-схемой, именуемой *S*-моделью, или просто моделью.

Simulink можно запустить, нажав соответствующую пиктограмму в линейке меню или набрав команду «*simulink»* в командной строке главного окна *Matlab*, при этом открывается окно браузера библиотек.

При создании новой модели пользователь может выбрать пункт меню *File/New/Model* или обратиться к имеющимся примерам, вызвав команду *help simdemos* или открыв папку *Matlab\toolbox\simulink \simdemos*. Использование примеров особенно полезно на начальных этапах работы с *Simulink*.

Как программное средство *Simulink* — типичный представитель визуально-ориентированных языков программирования. Программа моделируемой схемы автоматически генерируется в процессе ввода выбранных блоков компонентов, их соединения и задания параметров блоков.

Построенная модель сохраняется в файле с расширением *.mdl* или *.slx.* Фактически спроектированная в *Simulink* модель (*S*-модель) является программой, которую можно просмотреть с помощью текстового редактора или редактора файлов системы *Matlab*. Отметим, что даже для довольно простых моделей файлы могут содержать тысячи строк программного кода. Такая возможность может быть полезной в первую очередь для опытных пользователей с целью модернизации *S*-модели.

Когда модель построена, перед выполнением моделирования необходимо предварительно задать ее параметры. Задание параметров расчета выполняется в панели управления меню «*Simulation», «Model Configuration Parameters»* или комбинацией клавиш Ctrl+E.

Установка параметров расчета модели выполняется с помощью элементов управления, размещенных на вкладке «*Solver»*. Эти элементы разделены на три группы:

– *Simulation time* (интервал моделирования, или, иными словами, время расчета);

– *Solver options* (параметры расчета);

– *Tasking and Sample time options* (параметры постановки задач и временных отсчетов).

Важные параметры этих групп.

Время расчета задается указанием начального (*Start time*) и конечного (*Stop time*) значений времени расчета.

При выборе параметров расчета необходимо указать способ моделирования «*Type»* и метод расчета нового состояния системы. Для параметра «*Type»* доступны два варианта:

– фиксированный шаг (*Fixed-step*) — интервальное моделирование;

– переменный (*Variable-step*) шаг — событийное моделирование.

Список методов расчета нового состояния системы содержит несколько вариантов. Первый вариант «*discrete*» используется для расчета дискретных систем. Остальные методы используются для расчета непрерывных систем. Эти методы различны для переменного «*Variable-step*» и для фиксированного «*Fixed-step*» шага времени, но, по сути, представляют собой процедуры решения систем дифференциальных уравнений. При выборе «*Fixed-step»* необходимо также задать режим расчета «*Mode*». Для параметра «*Mode»* доступны три варианта:

– *MultiTasking* (многозадачный) — необходимо использовать, если в модели присутствуют параллельно работающие подсистемы и результат работы модели зависит от временных параметров этих подсистем;

– *SingleTasking* (однозадачный) — используется для тех моделей, в которых недостаточно строгая синхронизация работы отдельных составляющих не влияет на конечный результат моделирования;

– *Auto* (автоматический выбор режима) — позволяет *Simulink* автоматически устанавливать режим расчета модели.

Подробное описание каждого из методов расчета состояний системы приведено во встроенной справочной системе *Matlab*.

Литература

1. Бланшет Ж., Саммерфилд М. *Qt* 4: программирование GUI на *C++*. М. : Кудиц-Пресс, 2008. 736 с.

2. Прат С. Язык программирования *C++*. Лекции и упражнения. Вильямс, 2012. 635 с.

Учебное издание

Семкин Артем Олегович
Перин Антон Сергееви

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ.
ЯЗЫКИ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Подписано в печать 14.10.2021. Формат 60×84/16.
Усл. печ. л. 10,46. Тираж 150. Заказ 257.

Томский государственный университет
систем управления и радиоэлектроники
634050, г. Томск, пр. Ленина, 40.
Тел. (3822) 533018.