И.Е. Красикова, И.В. Красиков

C++

ПРОСТО КАК 2×2

Эксмо
МОСКВА
2005

УДК 004.43
ББК 32.973.202
К 78

Редакция учебной и деловой литературы «Eksmo Education»

Выпускающий редактор В.В. Александров

Красикова И.Е., Красиков И.В.

К 78 С++. Просто как дважды два. — М.: Изд-во Эксмо, 2005. — 160 с., ил. — (Просто как дважды два).

ISBN 5-699-11691-5

Данная книга представляет собой карманный справочник по С++. Этот справочник предназначен в первую очередь для начинающих программистов на этом языке, а также программистов среднего уровня, и его основное предназначение — помочь вспомнить ту или иную особенность языка программирования. Кроме непосредственно синтаксиса языка программирования, в книге приводится ряд рекомендаций, позволяющих использовать этот синтаксис для написания эффективных, надежных и безопасных программ. В соответствии со своим предназначением книга рассчитана в первую очередь на начинающих программистов, однако в качестве настольной она будет полезна любому программисту-практику.

УДК 004.43
ББК 32.973.202

ISBN 5-699-11691-5

Содержание

Моему маленькому сынишке Антону
— И. Е. Красикова

Киевскому физико-математическому интернату —
выпускникам 1980 года и преподавателям.
— И. В. Красиков

Благодарности

В первую очередь я хочу поблагодарить моего годовалого сынишку Антона за то, что он позволил мне всерьез поработать над этой книгой. Я благодарна всем сотрудникам редакции, кто так или иначе приложил свой труд и умение к выпуску этой книги, а особенно В. В. Александрову, который сумел убедить меня взяться за эту работу.

И. Е. Красикова

Мне бы хотелось отойти от общепринятой практики и выразить свою благодарность не только помогавшим в работе над этой конкретной книгой, а всем, кто помог мне стать тем, кто я есть — автором этой книги. Это те, кто учил меня и помогал учиться. В первую очередь это мой самый первый учитель математики — моя мать Н. Н. Красикова. Это мои школьные учителя В. Ф. Криволапов и особенно И. Н. Яковлев, в кабинете которого я впервые познакомился с такой замечательной вещью, как микрокалькулятор, университетский преподаватель А. Т. Янишевский. Особая благодарность — Л. Л. Михайленко, которая одолжила мне свой учебник по Алголу на зимние каникулы на первом курсе, благодаря чему я начал изучение с приличного процедурного языка программирования. Не могу не поблагодарить В. В. Картузова, поверившего в мои способности программиста в тот момент, когда о них было очень трудно судить. Мне трудно упомянуть всех тех, кто бескорыстно помогал мне, консультируя, подавая идеи и снабжая документацией, но большинство из них — участники (хотя и частично уже бывшие) сети FidoNet. Огромное всем вам спасибо!

И. В. Красиков

Введение

Данная книга задумана как справочник, что в первую очередь предполагает определенные знания языка. Как было хорошо замечено в одном фантастическом рассказе — чтобы правильно задать вопрос, надо знать половину ответа. Здесь вы найдете ответы именно на такие вопросы — когда вы знаете часть материала, и вам надо либо быстро пополнить знания, либо освежить их в памяти. Небольшой объем книги не позволяет полностью раскрыть все аспекты излагаемых тем, поэтому просьба с пониманием отнестись к имеющимся в книге упрощениям (не влияющим на корректность изложения) и сокращениям.

Здесь нет привязки к конкретному компилятору языка, однако, если вы работаете с компилятором, который не в полном объеме поддерживает стандарт, вы можете столкнуться с тем, что он не в состоянии скомпилировать исходный текст, содержащий ту или иную описанную возможность языка. В первую очередь это относится к шаблонам.

Там, где полное изложение материала невозможно из-за огромного объема (например, при описании шаблонов), нам пришлось идти на достаточно существенное сокращение. Исходя из наличия большого количества компиляторов, которые не соответствуют последним версиям стандарта (когда в книге говорится о «старом» компиляторе, имеется в виду неполная поддержка современного стандарта C++, а не год его выпуска), сокращению в первую очередь подверглись именно новшества. Более того, в книге совершенно не упомянуты некоторые особенности языка, с которыми, как нам кажется, программистам приходится встречаться достаточно редко, а кроме того, при необходимости можно обратиться к соответствующей специализированной литературе.

Фрагменты исходных текстов носят в первую очередь дидактический характер, поэтому решаемые ими задачи зачастую весьма надуманны, а сами решения не являются оптимальными — ведь цель таких фрагментов только в том, чтобы показать применение того или иного механизма либо особенности языка. Форматирование исходных текстов также не везде последовательно — с тем, чтобы сами тексты занимали по возможности меньше места в книге.

Поскольку эта книга в первую очередь — краткий справочник именно по языку C++, а не способам его применения, здесь не рассматриваются вопросы проектирования, парадигмы объектно-ориентированного программирования, не описываются специально такие «киты», как инкапсуляция или полиморфизм, — эти темы затронуты только в рамках рассмотрения конкретных возможностей языка программирования. Тема

проектирования, на наш взгляд, не может быть изложена в книге такого объема, даже если она будет посвящена только одной этой теме. Тем не менее в книге имеется ряд практических рекомендаций и советов по написанию программ на C++.

Книга разделена на 9 глав в соответствии с общепринятым изложением основ языка программирования. Это главы, посвященные типам и объявлениям, выражениям и конструкциям языка, функциям, макросам и препроцессору, исходным файлам и программам, классам, исключениям, шаблонам и стандартной библиотеке. Однако это разделение весьма условно, поскольку все части языка очень тесно взаимосвязаны. Поэтому в книге имеется множество перекрестных ссылок, которые помогут вам найти дополнительный материал из других разделов.

В данной книге используются пиктограммы, акцентирующие ваше внимание на отдельных абзацах.

Такой пиктограммой выделены рекомендации по проектированию и кодированию, которые, хотя зачастую и не являются частью языка и не обязательны к выполнению, но стоят того, чтобы если и не применять их в своих программах, то по крайней мере подумать над ними и их возможным использованием.

Очень внимательно относитесь к сведениям, помеченным такой пиктограммой. Они предостерегут вас от ошибок, которые могут стать причиной как мелких, так и более серьезных проблем.

Глава 1. Типы и объявления

C++ относится к строго типизированным языкам программирования и имеет набор базовых типов, которые отражают характерные особенности организации памяти и способы хранения данных. Кроме базовых, в C++ могут быть определены пользовательские типы данных.

Логический тип

Логический тип данных в C++ — `bool`. Переменная типа `bool` может принимать только одно из двух значений — `true` (истина) и `false` (ложь). Логические значения являются результатом логических операций, например, сравнения чисел.

При преобразовании в целочисленные типы значение `true` преобразуется в `1`, а значение `false` — в `0`; при преобразовании целых типов в тип `bool` нулевое значение преобразуется в `false`, а все ненулевые значения — в `true`.

Символьные типы

Переменная типа `char` может хранить один из набора символов компьютера. Практически всегда для переменной такого типа отводится 8 бит, так что всего существует 256 различных символов, представимых этим типом.

Каждый символ имеет определенное числовое значение, однако вопрос соответствия символа его числовому значению выходит за рамки данной книги.

При преобразовании символьного типа в целочисленный мы получаем числовое значение символа. Существует определенная проблема, заключающаяся в том, что числовое значение `char` может быть интерпретировано как знаковое (от `-128` до `127`) или беззнаковое (от `0` до `255`). Язык позволяет явным образом определить указание диапазона: тип `signed char` означает знаковый символьный тип, `unsigned char` — беззнаковый. Как именно трактуется тип `char` — как знаковый или как беззнаковый, — зависит от реализации.

Символьный литерал представляет собой символ, заключенный в одинарные кавычки, например `'a'`. Кроме того, символьный литерал с

особым значением может состоять и из нескольких символов, при этом первым символом всегда идет обратная косая черта \.

Название	Литерал
Перевод строки	`\n`
Горизонтальная табуляция	`\t`
Вертикальная табуляция	`\v`
Забой	`\b`
Возврат каретки	`\r`
Перевод страницы	`\f`
Звуковой сигнал	`\a`
Обратная косая черта	`\\`
Вопросительный знак	`\?`
Апостроф	`\'`
Двойная кавычка	`\"`
Восьмеричное число	`\ooo`
Шестнадцатеричное число	`\xhh`

Последовательность `\ooo` представляет собой символ, числовое значение которого представлено как восьмеричное число `ooo` (которое может состоять из одной, двух или трех цифр). То же относится и к последовательности `\xhh`, где `hh` — шестнадцатеричное число из одной или двух шестнадцатеричных цифр.

Для повышения удобочитаемости текста и устранения возможных неоднозначностей при использовании восьмеричных чисел всегда используйте три цифры, а при шестнадцатеричных — две, добавляя при необходимости ведущие нули.

Целочисленные типы

Целочисленные типы представлены типом `int`, который может применяться со спецификатором знаковости — `signed` или `unsigned` и со спецификатором размера — `short` или `long`. Используя спецификатор, можно не указывать тип `int`, т.е. `long` означает то же, что и `long int`. В случае отсутствия спецификатора знаковости целочисленный тип всегда интерпретируется как знаковый.

Беззнаковые (**unsigned**) типы в основном предназначены для представления массивов битов; использование данных типов с целью увеличения диапазона представления положительных чисел или для гарантии того, что число будет всегда неотрицательным, как правило, лишено смысла.

Что касается размеров целых чисел, то, как правило, **short int** имеет размер 16 бит, **int** и **long int** — 32 бит. В ряде реализаций имеются также дополнительные целочисленные типы, такие как **__int64** или **long long**, имеющие размер 64 бит. Однако стандарт не предусматривает конкретного указания размеров целочисленных типов. Тем не менее в наиболее распространенных на сегодняшний день компиляторах выполняются следующие соотношения.

Тип	Размер, бит	Минимальное значение	Максимальное значение
signed char	8	-127	128
unsigned char	8	0	255
short int	16	-32768	32767
unsigned short int	16	0	65535
[long] int	32	-2147483648	2147483647
unsigned [long] int	32	0	4294967295
long long	64	0	18446744073709551615
unsigned long long	64	-9223372036854775808	9223372036854775807

В программе можно использовать непосредственные целочисленные значения, вводимые как литералы. Литералы могут быть десятичными, восьмеричными и шестнадцатеричными. Десятичные литералы представляют собой привычную запись числа в десятичной системе счисления. Литерал, начинающийся с нуля, за которым следуют цифры, представляет собой значение в восьмеричной системе счисления; литерал, начинающийся с **0x**, за которым следуют цифры и буквы **a**–**f** (независимо от регистра), представляет собой значение в шестнадцатеричной системе счисления. Например:

Десятичное	12	23	55	158	326
Восьмеричное	014	027	067	0236	0506
Шестнадцатеричное	0xC	0x17	0x37	0x9E	0x146

Восьмеричная и шестнадцатеричная формы записи чисел применяются в основном для представления массивов битов; при их использовании для записи «обычных» чисел могут возникнуть неприятные эффекты, связанные с внутренним представлением чисел в компьютере.

Компилятор должен вывести предупреждение, если значение, представленное литералом, слишком велико для данного типа.

Для явного указания беззнаковости числа можно использовать суффикс `U`, для указания типа `long` — суффикс `L`. Так, `5` — число типа `int`, `5U` — типа `unsigned int`, `5L` — типа `long int`.

Типы с плавающей точкой

Типы с плавающей точкой представлены тремя видами с разным размером — `float`, `double` и `long double`. Конкретный размер данных типов, диапазон и точность представления чисел зависят от конкретной реализации.

Как правило, следует использовать тип `double`; в настоящее время использование типа `float` не имеет того смысла, который был раньше, когда скорость работы программы существенно зависела от размера представления числа, а количество доступной памяти было строго ограничено.

Литералы с плавающей точкой по умолчанию имеют тип `double`. Компилятор должен вывести предупреждение, если значение, представленное литералом, слишком велико для данного типа. Однако это не обязательно в случае, если число попадает в диапазон, представимый данным типом, но его точное представление невозможно. Литералы представляют собой обычную десятичную запись числа с указанием умножения на степень 10 с использованием символа `e` — например, `1.234`, `78.5e-6` и т.д. Явное указание типа `float` возможно при помощи суффикса `f` — например `1.234f`.

Размеры типов

Конкретные размеры различных типов зависят от реализации. В C++ размеры выражаются в размерах `char`. Получить размер конкретного объекта или типа в C++ можно при помощи оператора `sizeof`. Таким образом, по определению, `sizeof(char)=1`.

По поводу размеров базовых типов гарантируются следующие соотношения.

```
1=sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
1<=sizeof(bool)<=sizeof(long)
sizeof(float)<=sizeof(double)<=sizeof(long double)
sizeof(N)=sizeof(signed N)<=sizeof(unsigned N)
```

Здесь ***N*** — **char**, **short int**, **int** или **long int**.

Перечисления

Перечисление (**enum**) определяет набор значений, задаваемый пользователем. Каждое перечисление является отдельным типом. Элементами перечисления являются именованные целые константы, например **enum {First,Second,Third};**. После определения элементы получают целые значения; по умолчанию элементам присваиваются значения, начиная с нуля, для каждого последующего элемента увеличивающиеся на 1. Таким образом, в приведенном примере значение **First** равно **0**, **Second** — **1**, и **Third** — **2**. Перечисление может быть именованным, например **enum Order {First,Second,Third};**. Перечисление можно инициализировать константным выражением интегрального типа, например **enum{First=3, Second=5,Third};**.

Размер перечисления (**sizeof**) является размером некоторого интегрального типа, который в состоянии содержать весь диапазон перечисления.

Перечисления при выполнении арифметических операций преобразуются в целые значения. Поскольку перечисления являются типами, определяемыми пользователем, для них можно вводить свои собственные операции, например **++** или **<<**.

Тип void

Тип **void** синтаксически является базовым, однако он может быть только частью других составных типов, так как объект типа **void** создать невозможно. В основном тип **void** применяется для указания на то, что функция не возвращает значения (**void func();**), и в качестве базового для указателей на объекты неизвестного типа (**void*ptr**). Указатель любого типа может преобразовываться в **void***.

Указатели и массивы

Тип `T*` для некоторого типа `T` является указателем на `T`, т.е. переменная типа `T*` содержит адрес объекта типа `T`. Основной операцией над указателями является разыменование, т.е. получение объекта, на который ссылается указатель (который находится по адресу, хранящемуся в указателе). Оператором разыменования является префиксный унарный оператор `*`.

Для данного типа `T` имеется соответствующий ему тип `T[size]`, который представляет собой массив из `size` элементов типа `T`. Элементы массива нумеруются от `0` до `size-1`. Количество элементов массива — константное выражение. Если в программе требуется массив с переменным размером, следует использовать `vector` (см. стр. 112). В C++ могут определяться и использоваться многомерные массивы, которые представляют собой массивы массивов, т.е. `int a[5][6]` — объявление переменной `a`, которая является массивом из 5 массивов по 6 целых чисел. Обращение к элементу (многомерного) массива выполняется при помощи указания его индекса, например: `a[2][3]`. Заметим, что в C++ запись `array[index]` эквивалентна записи `index[array]`, т.е. записи `a[2]` и `2[a]` тождественны.

Начальное значение массива можно указать при помощи списка значений, например: `int b[]={1,2,3};`. В этом случае массив объявляется без указания его размера, который вычисляется компилятором по количеству инициализирующих значений в списке, т.е. в приведенном примере массив `b` имеет размер `3`. Если размер задан явно, то количество значений в списке инициализации не должно превышать размер массива — такое превышение является ошибкой. Если же элементов в списке меньше, чем размер массива, всем остальным элементам присваивается значение `0`, например `int c[4]={1,2};` эквивалентно `int c[4]={1,2,0,0};`.

Частным случаем массива является строковый литерал, который представляет собой последовательность символов, заключенную в двойные кавычки — например, `"Строковый литерал"`. Строковый литерал содержит дополнительный завершающий нулевой символ (`'\0'`). Строковый литерал представляет собой массив с соответствующим количеством константных символов, т.е. тип приведенного литерала — `const char[18]`. С целью обратной совместимости с C разрешено присваивание строкового литерала переменной типа `char*`, однако при этом возникает опасность изменения строкового литерала посредством такого указателя — что, вообще говоря, является ошибкой. Память для строковых литералов выделяется статически, т.е. такой литерал может быть возвращаемым значением функции.

Рекомендуется вместо массивов использовать в своих программах `vector` или `string`.

Указатели и массивы тесно связаны: имя массива можно использовать в качестве указателя на его первый элемент (обратного преобразования указателя в массив не существует). Например, пусть дан массив `int a[4]`. Тогда инструкция `int*p=a;` делает `p` указателем на первый элемент массива `a` (эквивалентная запись — `int*p=&a[0];`). Эта возможность используется для передачи в функцию массива: если в качестве аргумента функции используется массив, то передается указатель на его первый элемент, т.е. аргумент типа `T[]` преобразуется в `T*` (подробнее см. стр. 48).

Обращение к элементам массива может выполняться как с использованием индексов, так и с использованием указателей. Оба способа эквивалентны.

Применение операторов `+`, `-`, `++` или `--` к указателю зависит от его типа. Если указатель `p` имеет тип `T*` и указывает на некоторый элемент массива типа `T`, то `p+1` указывает на следующий элемент массива (*а не на следующий байт!*), а `p-1` — на предыдущий, так что целое значение `p+1` больше целого значения `p` на `sizeof(T)`. Сложение указателей смысла не имеет и запрещено, а вычитание имеет смысл только в том случае, если оба указателя указывают на элементы одного массива (хотя в языке нет средства для проверки этого факта). При этом в результате вычитания получается количество элементов массива между элементами, на которые указывают эти указатели.

Константа `0` может быть значением указателя любого типа; гарантируется, что такой указатель не может указывать ни на один реальный объект. Указатель любого типа может преобразовываться в указатель `void*`; указатель на объект класса может преобразовываться в указатель на объект базового класса.

Константы

В C++ имеется спецификатор `const`, который указывает на то, что данное значение нельзя изменять непосредственно. Чтобы объявить объект константой, достаточно добавить указанный спецификатор в объявление. Поскольку присваивание константе не разрешено, при объявлении она долж-

на быть обязательно инициализирована, например, `const int dim=3;`. Последовательное использование данного спецификатора повышает надежность кода, так как в этом случае компилятор отслеживает все попытки изменения значения такой переменной.

Использование `const` с функциями-членами классов рассматривается в разделе «Константные функции-члены» (стр. 66).

Константы позволяют использовать их в качестве символьной замены различных «магических чисел» в коде, локализуя информацию. Так, если в программе интенсивно используется некоторый массив размером 64 элемента, и в ряде циклов индексная переменная пробегает значения от 0 до 63, то при изменении размера массива потребуется найти все вхождения размера в исходном тексте и заменить их другими значениями (заметим, что могут встречаться и связанные величины (например, половинный размер 32), найти которые окажется гораздо сложнее). Использование же символьной константы приводит к локализации этого значения и замене его только в одном месте исходного текста.

В операциях с указателями используются два объекта — сам указатель и объект, на который он указывает. Спецификатор `const` перед объявлением такого указателя делает константным объект, а не указатель. Чтобы сделать константным указатель, требуется объявить его не как `T*`, а как `T*const`. Например:

```
char s[] = "........";
const char * a = s;          // Указатель на константу
a[0] = 'x';                  // Ошибка
a = "xxxxxx";                // Корректно
char * const b = s;          // Константный указатель
b[0] = 'x';                  // Корректно
b = "xxxxxx";                // Ошибка
const char * const c = s;    // Константный указатель на
                             // константу
c[0] = 'x';                  // Ошибка
c = "xxxxxx";                // Ошибка
```

Ссылки

Ссылка представляет собой альтернативное имя объекта и чаще всего используется для указания аргументов и возвращаемых значений функций. Тип `T&` означает ссылку на `T`. Значение ссылки должно указываться при ее инициализации и после этого не может быть изменено. Все опера-

ции со ссылками выполняются с объектами, на которые они ссылаются. Вот пример объявления и работы со ссылкой:

```
int i = 0;
int& j = i;    // j - ссылка на i
j++;           // Значение i становится равным 1
```

Значение инициализации константной ссылки может не быть lvalue[1] и даже не иметь тип `T`, поскольку при этом выполняется приведение к типу `T` и создание временной переменной, на которую в дальнейшем и ссылается константная ссылка.

Структуры и объединения

Структуры представляют собой наборы элементов (почти) произвольных типов. Структурный тип объявляется при помощи ключевого слова **`struct`**, после которого следует имя типа и в фигурных скобках — перечисление через точку с запятой полей структуры. После перечисления полей следует закрывающая фигурная скобка и (обязательно) точка с запятой, например:

```
struct Person {
    char* Name;
    char* Address;
    int   Age;
};
```

Обращение к полям структуры выполняется при помощи оператора `.` (точка) с указанием имени объекта и через точку — поля. Если используется указатель на объект, то вместо точки используется оператор `->`, например:

```
Person P;
Person*Q = &P;
P.Age  = 25;
Q->Age = 40;
```

Обращение `p->m` эквивалентно `(*p).m`. Объекты структур можно присваивать друг другу, но другие стандартные операции, например сравнения, не определены.

Имя структурного типа может использоваться немедленно после его появления, но не для объявления других объектов этого типа:

[1] Использование указателей, массивов и других возможностей C++ позволяет выполнять присваивание не только переменным, но и выражениям, которые могут на первый взгляд выглядеть достаточно странно, например `*p[i+2]=0;`. *lvalue* (от left value) — это выражение, ссылающееся на объект, который, в свою очередь, представляет собой некоторую информацию в памяти.

```
struct Person {
    Person* Friend; // Корректно
    Person  Spouse; // Ошибка - рекурсивное определение
...
```

Для того чтобы два структурных типа могли ссылаться друг на друга, можно сначала объявить только имя типа:

```
struct Car;
struct Person {
    Car* automobile;
...
};
struct Car {
    Person* owner;
...
};
```

Для обеспечения обратной совместимости с C разрешено объявление в одной и той же области видимости структуры и не-структуры с одним и тем же именем, например:

```
struct pair { int a; int b; };
int pair;
struct pair P;
P.a = pair;
```

Две структуры являются различными типами, даже если они имеют одинаковую структуру:

```
struct S { int a; };
struct Q { int a; };
S x;
Q y = x;  // Ошибка! S и Q - разные типы
```

В структуре могут использоваться битовые поля, позволяющие обращаться к отдельным битам структуры. Такие поля должны относиться к интегральному или перечислимому типу. Они облегчают обращение к части слова, обычно — за счет увеличения генерируемого компилятором кода. Обращение к битовому полю выполняется так же, как и к обычному полю структуры, однако получить адрес битового поля невозможно. Приведем пример объявления битовых полей для 32-битового регистра **EFLAGS** процессора Intel Pentium:

```
struct EFLAGS {
    unsigned int CF      : 1; // Carry Flag
    unsigned int         : 1; // Не используется
    unsigned int PF      : 1; // Parity Flag
```

```
    unsigned int        : 1; // Не используется
    unsigned int AF     : 1; // Auxiliary Carry Flag
    unsigned int        : 1; // Не используется
    unsigned int ZF     : 1; // Zero Flag
    unsigned int SF     : 1; // Sign Flag
    unsigned int TF     : 1; // Trap Flag
    unsigned int IF     : 1; // Interrupt Enable Flag
    unsigned int DF     : 1; // Direction Flag
    unsigned int OF     : 1; // Overflow Flag
    unsigned int IOPL   : 2; // I/O Privilege Level
    unsigned int NT     : 1; // Nested Task
    unsigned int        : 1; // Не используется
    unsigned int RF     : 1; // Resume Flag
    unsigned int VM     : 1; // Virtual-8086 Mode
    unsigned int AC     : 1; // Alignment Check
    unsigned int VIF    : 1; // Virtual Interrupt Flag
    unsigned int VIP    : 1; // Virtual Interrupt
                             // Pending
    unsigned int ID     : 1; // ID Flag
    unsigned int reserved : 10;
};
```

Все поля имеют размер по 1 бит, за исключением **IOPL**, размер которого 2 бит, и **reserved** — 10 бит. Как видно из приведенного примера, могут существовать неименованные поля, обращение к которым невозможно, но которые необходимы для корректного именования отдельных битов слова.

Формально в C++ структура представляет собой класс (см. главу «Классы», стр. 63), все члены которого по умолчанию открыты. Однако в связи с тем, что в соответствии с принципом инкапсуляции данные — члены класса всегда следует делать закрытыми, структуры нужно использовать только для представления простейших агрегатов в стиле структур языка C, объединяющих в единое целое набор значений, не претендующих на инкапсуляцию и не обеспечивающих функциональность.

Объединение представляет собой структуру, члены которой располагаются по одному и тому же адресу, так что размер объединения равен размеру наибольшего его члена. В каждый момент времени объединение может хранить значение только одного из членов. Например,

```
union Value {
    int i;
    double d;
};
```

представляет собой объединение, которое хранит либо целое число, либо число с плавающей точкой — но только что-то одно. Обращение к полям объединения выполняется так же, как и к полям структуры, — при помощи точки или `->`.

Если объединение является частью структуры, при обращении появляется дополнительное поле. Допустим, что переменная **x** у нас объявлена следующим образом.

```
struct X {
    int j;
    Value v;
};
X x;
```

Тогда обращаться к полям объединения придется с указанием имени поля **v**, например, **x.v.i**. Если же сделать объединение в составе структуры безымянным, то указание дополнительного поля не надо:

```
struct X {
    int j;
    union {
        int i;
        double d;
    };
};
X x;
x.i = 5;  x.d = 5.0;
```

Категорически не рекомендуется использовать поля для «преобразования типов» – потому что такое преобразование как минимум не переносимо, а зачастую просто некорректно.

В состав объединений, как и классов и структур, могут входить и функции-члены, однако такое применение объединений весьма ограниченно, используется крайне редко, а потому в данной книге не рассматривается.

Преобразования типов

Определенные операции могут в зависимости от их операндов вызывать преобразование значения операнда от одного типа к другому. Фундаментальные типы в C++ могут преобразовываться один в другой огромным количеством способов. Значения **char** и **short int** могут использоваться там, где применяется **int** — во всех подобных случаях про-

исходит преобразование значения в `int`. Преобразование `short int` в `int` всегда включает знаковое расширение. Явно указанный тип `unsigned char` ограничивает изменения от 0 до максимального значения.

Преобразования между `float` и `double` выполняются настолько точно, насколько это позволяют представления данных типов. Преобразование чисел с плавающей точкой в целые является машинно-зависимым, в частности, может меняться направление усечения отрицательных чисел. Если целого числа не хватает для представления числа с плавающей точкой, то результат такого преобразования не определен.

При сочетании целого без знака и обычного целого обычное число преобразуется к беззнаковому виду. Значением преобразования является наименьшее целое без знака, равное целому со знаком по модулю 2 в степени размера в битах. При использовании дополнительного бинарного представления никаких изменений в двоичном представлении числа не происходит.

При преобразовании беззнакового целого в большее по размеру беззнаковое число результат численно совпадает с исходным значением, т.е. преобразование сводится к дополнению числа нулевыми битами слева.

Преобразования при арифметических операциях рассматриваются в разделе «Арифметические операции» на стр. 37.

Что касается указателей, то везде, где они присваиваются, инициализируются, сравниваются и тому подобное, могут выполняться следующие преобразования.

1. Константа `0` может преобразовываться в указатель, при этом гарантируется, что это значение порождает указатель, отличный от указателя на любой реальный объект.
2. Указатель любого типа может преобразовываться в `void*`.
3. Указатель на объект класса может преобразовываться в указатель на объект открытого базового класса для данного класса.
4. Имя массива может преобразовываться в указатель на его первый элемент.
5. Имя функции, не использующееся в вызове, преобразуется в указатель на функцию.

Везде, где инициализируются ссылки, ссылка на объект класса может преобразовываться в ссылку на объект открытого базового класса данного класса.

Явное преобразование типов выполняется либо унаследованным от C способом — при помощи конструкции `(type)value`, либо с помощью операторов преобразования типов.

Оператор `static_cast` осуществляет преобразование родственных типов, например, указателя на один тип в указатель на другой тип из той

же иерархии классов, перечислений в целые типы или типа с плавающей точкой в интегральный, например,

```
int*p = static_cast<int*>(malloc(100));
```

Оператор **reinterpret_cast** осуществляет преобразование между несвязанными типами, например, целых чисел в указатели или указателей в другие, несвязанные, указатели. Наличие этих двух видов операторов преобразования типов позволяет компилятору выполнять минимальную проверку типов при преобразовании **static_cast**, а программисту — легче обнаружить опасные преобразования, для выполнения которых требуется применение **reinterpret_cast**. Обычно при использовании этого вида преобразования типов получается значение нового типа, представленное той же битовой строкой, что и исходное, однако это не гарантируется. Если размер результата не ниже размера исходного значения, то обратное преобразование дает исходное значение.

Оператор **const_cast** используется для того, чтобы аннулировать действие модификатора **const**, т.е., например, привести указатель **const T*** к типу **T***. Однако использование результата такого преобразования безопасно только в том случае, если преобразуемый объект не был изначально объявлен как константный, либо если это преобразование используется для того, чтобы привести константный объект к неконстантному виду при вызове функции, аргумент которой объявлен как неконстантный, но фактически является таковым.

Преобразование типов **(type)value**, унаследованное от C, означает любое преобразование, которое может быть выражено при помощи комбинации операторов **static_cast**, **reinterpret_cast** и **const_cast** для получения значения типа **type** из выражения **value**. Использование такого преобразования типов намного опаснее, поскольку может выполнять как вполне корректные переносимые действия, так и, например, непереносимые преобразования между несвязанными типами.

Оператор **dynamic_cast** выполняет преобразование указателя или ссылки на базовый класс полиморфного типа в указатель или ссылку на производный класс. Все проверки при использовании данного преобразования выполняются во время выполнения программы. Ввиду ограниченного объема книги подробное рассмотрение преобразования **dynamic_cast** в ней отсутствует.

Информация о типах

В C++ имеется оператор **typeid**, который выдает объект, представляющий тип операнда, т.е. возвращает ссылку на тип стандартной библиотеки **type_info**, определенный в заголовочном файле **<typeinfo>**. Параметрами оператора могут быть как имя типа, так и выражение. Независимая от реализации часть **type_info** выглядит следующим образом:

```
class type_info {
public:
    // Обеспечивает сравнение типов
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    // Обеспечивает упорядочение типов
    bool before(const type_info& rhs) const;
    const char* name() const;    // Символьное имя типа
    private:                     // Копирование запрещено
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
```

Таким образом, оператор **typeid** позволяет сравнивать различные типы и даже упорядочивать их. Поскольку не гарантируется, что для каждого типа имеется только один объект **type_info**, для проверки равенства типов следует использовать оператор **==** с объектами **type_info**, но не с указателями на них.

В случае ошибки (например, передачи оператору нулевого указателя) оператор **typeid** генерирует исключение **bad_typeid**.

Объявления и инициализация

Для того чтобы программа на C++ могла использовать некоторое имя (идентификатор), его следует объявить, т.е. указать тип, соответствующий данному имени. Объявление состоит из (необязательных) спецификаторов, типа, объявляющей части и, возможно, инициализатора. За исключением объявлений функций и пространств имен, объявление завершается точкой с запятой.

В качестве спецификаторов могут выступать ключевые слова **virtual** (о котором будет рассказано в разделах, посвященных классам), **inline** (применим к функциям, указывает, что вместо вызова компилятор должен по возможности встраивать код тела функции в код программы), **extern** (которое указывает, что данное объявление является только объявлением, но не определением имени), **static** (его смысл поясняется позже; см. стр. 31 и 46), а также **auto** и **register**.

Избегайте применения спецификаторов `auto`, `register` и `inline`, поскольку первый в конечном счете не несет никакой семантической нагрузки, по сути, представляя разновидность комментария, а применение остальных в современных оптимизирующих компиляторах практически лишено смысла (см. [8]).

Объявляющая часть состоит из имени и, возможно, операторов объявления, к которым относятся `*` (указатель), `*const` (константный указатель), `&` (ссылка), `[]` (массив) и `()` (функция), причем последние два оператора являются суффиксами, в то время как остальные — префиксами. Приоритет суффиксных операторов выше, так что `int*a[];` — объявление массива указателей на `int`, а не указатель на массив. Спецификатор `volatile` указывает, что данная переменная может быть изменена в процессе работы внеязыковыми средствами, так что к ней не следует применять агрессивную оптимизацию.

В объявлении могут использоваться несколько имен, разделенных запятыми.

Операторы объявления относятся к ближайшему имени и не относятся к соседним именам. Следовательно, объявление `int*a,b;` объявляет указатель на `int` с именем `a` и переменную `b` типа `int`. В связи с этим рекомендуется избегать таких разнородных объявлений нескольких переменных, которые к тому же ухудшают читаемость программы.

Имя идентификатора в C++ состоит из последовательности букв и цифр, причем первым символом должна быть буква. К буквам относится также символ подчеркивания (_). C++ различает регистр символов, так что `A` и `a` — разные имена. В качестве пользовательских имен нельзя применять зарезервированные ключевые слова C++ (см. таблицу).

После имени объекта в объявлении может идти инициализатор. Если он отсутствует, то глобальным объектам, объектам из пространства имен и локальным статическим объектам (стр. 46) присваивается нулевое значение соответствующего типа. Локальные переменные и динамические объекты не инициализируются. Конкретные инициализаторы приводятся в подразделах, посвященных соответствующим типам переменных.

Инициализация может выполняться с использованием синтаксиса присваивания (например, `int i = 5`) или с использованием синтаксиса вызова (`int i(5)`). Предпочтителен второй способ, который, в частности, позволяет применять для объектов классов несколько аргументов инициализации.

Зарезервированные ключевые слова C++

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

Очевидно, что типы, получающиеся в результате комбинирования базовых типов, могут оказаться весьма сложными, так что зачастую имеет смысл упростить их описания при помощи объявлений **typedef**, которые объявляют синонимы типов. Например, объявление **typedef char*char_ptr;** объявляет **char_ptr** синонимом указателя на **char**, который затем может использоваться в качестве типа в объявлениях. Так, **char_ptr s[];** объявляет массив указателей на **char**. Следует подчеркнуть, что **typedef** создает не новые типы, а всего лишь синонимы, так что исходные типы можно использовать совместно с их синонимами.

Объявление **typedef** позволяет сделать текст существенно более понятным и удобочитаемым. Например, сравните объявление функции **set_new_handler()** без использования **typedef** и с ним:

```
void (*set_new_handler(void(*)()))();

typedef void (*new_handler)();
new_handler set_new_handler(new_handler);
```

Область видимости

Объявление вводит имя в область видимости, которая представляет собой часть исходного текста программы, которая может использовать данное имя.

Глобальные имена объявляются вне функций, классов или пространств имен, и их область видимости начинается с места объявления и заканчивается концом файла. Область видимости локальных имен, которые объявляются в теле функции, начинается с места объявления функции и заканчивается в конце блока, в котором они объявлены (*блоком* называется фрагмент исходного текста, помещенный в фигурные скобки).

Глобальные и совместно используемые переменные повышают связность различных частей программы, поэтому желательно их избегать. Кроме того, при использовании таких переменных следует тщательно следить за тем, чтобы инициализация одного объекта не зависела от другого, в особенности если эти объекты находятся в разных единицах трансляции.

Локальное объявление в блоке может скрывать объявление аналогичного имени в охватывающем блоке или глобальное имя, например:

```
int x;              // Глобальная переменная
void f()
{
    double x;       // Локальная переменная
    x = 1;          // Присваивание переменной типа double
    {
        void*x; // Локальная переменная
        x = 0;  // Присваивание переменной типа void*
    }
}
int * p = &x;       // Адрес глобальной переменной
```

Сокрытие имен переменных достаточно сложно обнаруживается, поэтому лучше использовать для своих переменных «значащие» имена, а не простые односимвольные `i`, `j`, `x`.

Обратиться к скрытому глобальному имени можно при помощи оператора разрешения области видимости `::`, например:

```
int x;          // Глобальная переменная
void f()
{
    double x; // Локальная переменная
    x = 1;    // Присваивание переменной типа double
    ::x = 1;  // Присваивание глобальной переменной
              // типа int
```

Способа непосредственного обращения к сокрытой локальной переменной не существует. Имена аргументов функции считаются объявленными в самом внешнем блоке функции.

Пространства имен

Если ряд объявлений можно логически объединить с использованием какого-либо критерия, то их можно объединить в некотором пространстве имен. Члены пространства имен объявляются следующим образом:

```
namespace Имя_пространства {
    Объявления и определения
}
```

Объявить новый член пространства имен вне его определения с использованием явного квалификатора нельзя; определить же его можно позже — либо в определении пространства имен, либо с явным использованием имени пространства и оператора разрешения области видимости (`::`). Пространства имен открыты, т.е. к ним можно добавлять имена в нескольких объявлениях, в том числе в различных заголовочных файлах.

Рекомендуется при определении предварительно объявленного члена пространства имен пользоваться синтаксисом с указанием явного квалификатора, а не с повторным открытием того же пространства имен. Это позволяет компилятору обнаруживать ошибки определения необъявленных функций, в то время как в случае повторного открытия пространства имен такая ошибка приведет к внесению нового, ошибочного члена в пространство имен.

```
namespace Test {
    void f();
}
void Test::ff()   // Ошибка: ff() не объ-
                  // явлена в Test
{ ... }
namespace Test {
    void ff()     // В Test добавляется
                  // функция ff()
    { ... }
}
```

Пространство имен является областью видимости, поэтому обычные правила областей видимости применимы и к пространствам имен. Если некоторое имя объявлено в пространстве имен или в охватывающей области, далее его можно использовать без проблем. Имена из других пространств имен можно использовать при помощи явного указания этого пространства в качестве квалификатора.

Если имя часто используется вне своего пространства имен, избежать постоянного указания его квалификатора можно при помощи объявления, которое говорит компилятору о том, что некоторое имя находится в указанном пространстве имен:

```
using Space::Name;
```

Здесь **Space** — имя пространства имен, а **Name** — имя из этого пространства. Такое **using**-объявление, по сути, вводит локальный синоним в текущую область видимости.

Можно сделать доступными все имена полностью из некоторого пространства имен, как если бы они были объявлены вне этого пространства. Для этого предназначена директива

```
using namespace Space;
```

Здесь **Space** — имя соответствующего пространства имен.

В случае, если размещение имени в пространстве имен преследует цель избежать возможного конфликта имен, и важно сохранить локальность кода, а не предоставить интерфейс пользователям, можно использовать неименованные пространства имен, т.е. пространства имен, объявленные без имени:

```
namespace {
    void f();
    void g();
```

Доступ к членам такого неименованного пространства имен осуществляется так же, как если бы оно имело некоторое уникальное имя, а после него была использована директива **using namespace** для этого уникального в пределах области видимости имени. В связи с этим очевидно, что доступ из одной единицы трансляции к члену неименованного пространства имен из другой единицы трансляции невозможен. (Ту же роль выполняет объявление глобального имени со спецификатором **static**, однако эта возможность унаследована от C и к применению не рекомендуется.)

Пространства имен могут иметь псевдонимы (например, для того, чтобы избежать использования в коде программы длинных имен). Псевдоним задается при помощи инструкции

```
namespace Alias = Original_Name_Of_Namespace;
```

Например:

```
namespace Special_Scientific_Library {
    double SpecialFunc();
    double Calculations();
}
namespace Sci = Special_Scientific_Library;
double x = Sci::SpecialFunc();
```

При поиске имен, если имя функции не найдено в контексте ее использования, поиск выполняется в пространствах имен ее аргументов. Это позволяет избежать излишнего использования **using**-директив и сократить программу. Само пространство имен при этом должно быть в области видимости, а функция должна быть объявлена до ее использования. Если аргументы функции — из разных пространств имен, то поиск осуществляется сначала в области видимости вызова, затем — в пространствах имен каждого аргумента, включая класс каждого аргумента и его базовые классы, а затем к найденным именам применяются обычные правила разрешения перегрузки. При вызове функции членом класса другие члены того же класса и его родительских классов имеют приоритет над функциями, найденными на основании информации о типах аргументов.

Механизм перегрузки работает сквозь границы пространств имен, т.е., если не указан явный квалификатор, просматриваются все подходящие имена функций, например:

```
namespace A { void f(int);  }
namespace B { void f(char); }
using namespace A;
using namespace B;

f(1);        // Вызов A::f(int)
f('a');      // Вызов B::f(char)
```

При необходимости создания интерфейса из набора существующих можно объединить пространства имен, используя объявления нового пространства и **using**-директивы, например:

```
namespace Lib1 {
    void f();
    void g();
}
namespace Lib2 {
    void h();
    void i();
}

namespace Lib {
    using namespace Lib1;
    using namespace Lib2;
    void j();
}
```

После такого объединения можно рассматривать пространство имен **Lib** как содержащее все члены пространства имен **Lib1** и **Lib2**, и использовать имя **Lib** как в качестве явного квалификатора, так и в **using**-директивах.

Можно, напротив, ограничить использование имен из пространства:

```
namespace Lib1 {
    void f();
    void g();
    void h();
    void i();
}

namespace Lib {
    using Lib1::f;
    using Lib1::g;
}
```

Теперь в пространстве имен **Lib** находятся только функции **f()** и **g()** (и все версии перегруженных функций **f()** и **g()**, если таковые имеются).

Комбинирование этих двух механизмов — объединения пространств имен и отбора — обеспечивает высокую гибкость, достаточную для решения большинства задач. Эти механизмы позволяют также разрешать возможные конфликты, например:

```
namespace Lib1 {
    void f();
```

```
    void g();
    class A {};
    class B {};
}
namespace Lib2 {
    void h();
    void i();
    class A {};
    class B {};
}

namespace Lib {
    using namespace Lib1;
    using namespace Lib2;
    using Lib1::A;  // Разрешение конфликта в пользу
                    // Lib1
    using Lib2::B;  // Разрешение конфликта в пользу
                    // Lib2
    void j();
```

Глава 2. Выражения и конструкции языка

В предыдущей главе были рассмотрены, если можно так сказать, «существительные» языка программирования C++. В этой главе мы познакомимся с его «глаголами» — т.е. выражениями и конструкциями, которые и выполняют действия с информацией, хранящейся в переменных различных типов. Правда, это знакомство мы начнем с исключения — конструкции, которая как раз не выполняет никаких действий, а именно — с комментария.

Комментарии

В C++ комментарием является все, расположенное после символов `//`, не являющихся частью строкового литерала, до конца строки. Унаследованы также комментарии в стиле C, когда комментарием является все, что начинается с пары символов `/*` и заканчивается `*/`.

Операторы

В приведенной далее таблице перечислены все операторы C++ в порядке снижения их приоритетов.

Разрешение области видимости	*class-name* `::`*member*
Разрешение области видимости	*namespace-name* `::`*member*
Глобально	`::`*name*
Глобально	`::`*qualified-name*
Выбор члена	*object*`.`*member*
Выбор члена	*pointer*`->`*member*
Доступ по индексу	*pointer*`[`*expr*`]`
Вызов функции	*expr*`(`*expr-list*`)`
Конструирование значения	*type*`(`*expr-list*`)`
Постфиксный инкремент	*lvalue*`++`
Постфиксный декремент	*lvalue*`--`
Идентификация типа	`typeid(`*type*`)`
Идентификация типа во время выполнения	`typeid(`*expr*`)`

Динамическое преобразование с проверкой	`dynamic_cast<type>(expr)`
Статическое преобразование с проверкой	`static_cast<type>(expr)`
Преобразование без проверки	`reinterpret_cast <type>(expr)`
Константное преобразование	`const_cast<type>(expr)`
Размер объекта	`sizeof expr`
Размер типа	`sizeof(type)`
Префиксный инкремент	`++lvalue`
Префиксный декремент	`--lvalue`
Дополнение	`~lvalue`
Отрицание	`!expr`
Унарный минус	`-expr`
Унарный плюс	`+expr`
Адрес	`&lvalue`
Разыменование	`*expr`
Создание (выделение памяти)	`new type`
Создание (выделение памяти и инициализация)	`new type(expr-list)`
Создание (размещение в памяти)	`new(expr-list)type`
Создание (размещение в памяти и инициализация)	`new(expr-list)type (expr-list)`
Уничтожение (освобождение памяти)	`delete pointer`
Уничтожение массива	`delete[] pointer`
Преобразование типа	`(type) expr`
Выбор члена	`object.*pointer-to-member`
Выбор члена	`pointer->pointer-to-member`
Умножение	`expr*expr`
Деление	`expr/expr`
Остаток от деления	`expr%expr`
Сложение	`expr+expr`
Вычитание	`expr-expr`
Сдвиг влево	`expr<<expr`
Сдвиг вправо	`expr>>expr`
Меньше	`expr<expr`
Меньше или равно	`expr<=expr`
Больше	`expr>expr`
Больше или равно	`expr>=expr`

Равно	`expr==expr`
Не равно	`expr!=expr`
Побитовое И	`expr&expr`
Побитовое исключающее ИЛИ	`expr^expr`
Побитовое ИЛИ	`expr|expr`
Логическое И	`expr&&expr`
Логическое ИЛИ	`expr||expr`
Условное выражение	`expr?expr:expr`
Простое присваивание	`lvalue=expr`
Умножение и присваивание	`lvalue*=expr`
Деление и присваивание	`lvalue/=expr`
Остаток и присваивание	`lvalue%=expr`
Сложение и присваивание	`lvalue+=expr`
Вычитание и присваивание	`lvalue-=expr`
Сдвиг влево и присваивание	`lvalue<<=expr`
Сдвиг вправо и присваивание	`lvalue>>=expr`
И и присваивание	`lvalue&=expr`
ИЛИ и присваивание	`lvalue|=expr`
Исключающее ИЛИ и присваивание	`lvalue^=expr`
Генерация исключения	`throw expr`
Запятая (последовательность)	`expr, expr`

В каждом блоке, разделенном горизонтальными линиями, расположены операторы с одинаковым приоритетом. Операторы в блоке, расположенном выше, имеют более высокий приоритет. Унарные операторы и операторы присваивания правоассоциативны, все остальные операторы левоассоциативны.

Если у вас есть хотя бы минимальные сомнения в порядке выполнения операторов, не стесняйтесь использовать скобки (которые к тому же существенно повышают удобочитаемость текста).

Арифметические операции

Основной вопрос при выполнении арифметических операций (`+`, `-`, `*`, `/`, `%`) — это тип полученного результата, так как в такой операции могут участвовать операнды разного типа. Общее правило — происходит пре-

образование значений к «максимальному» типу, и результат вычисления имеет этот тип. Общая схема такова.

Если один из операндов имеет тип `long double`, то и второй также преобразуется в `long double`.

Иначе, если один из операндов имеет тип `double`, то и второй также преобразуется в `double`.

Иначе, если один из операндов имеет тип `float`, то и второй также преобразуется в `float`.

Иначе, если один из операндов имеет тип `unsigned long`, то и второй также преобразуется в `unsigned long`.

Иначе, если один из операндов имеет тип `long int`, а другой — `unsigned int`, то, если `long int` достаточно для представления всех значений `unsigned int`, `unsigned int` преобразуется в `long int`; в противном случае оба операнда преобразуются в `unsigned long int`.

Иначе, если один из операндов имеет тип `long`, то и второй также преобразуется в `long`.

Иначе, если один из операндов имеет тип `unsigned`, то и второй также преобразуется в `unsigned`.

В противном случае оба операнда преобразуются в `int`.

Таким образом, результат деления в случае, если один из операндов имеет тип с плавающей точкой, будет также с плавающей точкой, но если оба типа целочисленны, то и деление будет целочисленным, с отбрасыванием дробной части (если результат больше `0`). Если результат отрицателен, то округление и знак остатка зависят от реализации, но в любом случае гарантируется выполнение тождества `(a/b)*b+a%b==a`.

Логические операторы

Логические операторы `&&` и `||` левоассоциативны и возвращают значение типа `bool`. Оператор `&&` возвращает значение `true`, если оба его операнда равны `true`, и `false` в противном случае. Оператор `||` возвращает значение `false`, если оба его операнда равны `false`, и `true` в противном случае. Гарантируется вычисление операндов слева направо, причем используются сокращенные вычисления — т.е., если левый операнд `&&` равен `false`, второй операнд не вычисляется; аналогично, если левый операнд `||` равен `true`, второй операнд не вычисляется.

Это правило неприменимо при использовании переопределенных (см. стр. 74) операторов `&&` и `||`, поэтому лучше эти операторы не перегружать. Не следует также перегружать оператор `,` (запятая), так как при этом для него перестает гарантироваться вычисление слева направо.

Побитовые логические операторы

Побитовые логические операторы `&`, `|`, `^`, `~`, `<<` и `>>` применяются к целым типам, т.е. к `bool`, `char`, `short`, `int`, `long` — возможно, с модификатором `unsigned`. Результатом данных операций также являются целочисленные значения. Представляя целочисленные значения как множество битов, данные операторы выполняют соответствующие действия над каждым битом множества (унарные) или парой соответствующих битов двух множеств (бинарные), за исключением операторов сдвига, которые работают со всей совокупностью битов.

В качестве наиболее распространенного применения этих операторов можно упомянуть установку и сброс определенных битовых флагов, например

```
unsigned char flag;
unsigned char bit0 = 0x01;
unsigned char bit2 = 0x04;
flag |=  bit0;    // Установка бита 0 в flag (равен 1)
flag &= ~bit2;    // Сброс бита 2 в flag (равен 0)
if (flag&(bit0|bit2)) {... // Проверка установки хотя бы
                           // одного из битов 0 и 2
```

Операторы отношений

Операторы отношений (`==`, `!=`, `>`, `<`, `>=`, `<=`) интуитивно понятны, и здесь следует лишь напомнить, что операторы равенства и неравенства дают точный ответ для целочисленных типов, но не для типов с плавающей точкой, которые представляют числа с определенной погрешностью, так что не следует использовать данные операторы для сравнения чисел с плавающей точкой.

Операторы присваивания

Оператор присваивания `=` назначает lvalue (см. примечание на стр. 19), находящемуся слева от символа оператора, значение выражения справа от символа оператора. Однако в связи с тем, что часто приходится встречаться с операциями вида `a = a @ b`, где `@` — некоторый бинарный оператор, для таких выражений введена сокращенная запись `a @= b`. Так, например, выражение `a += 5;` эквивалентно выражению `a = a + 5;`.

Операторы инкремента и декремента

Поскольку очень часто встречаются операции `a += 1` и `a -= 1`, для них введены специальные операторы инкремента `++` и декремента `--`, ко-

торые имеют два вида — префиксный и постфиксный. При применении данных операторов к переменной ее значение увеличивается (уменьшается) на 1, но результат выражения для префиксного и постфиксного операторов различен. Выражение с префиксным инкрементом (декрементом) равно значению переменной после увеличения (уменьшения), а с постфиксным — старому значению. Например:

```
int x = 5;
printf("%d\n",x++);   // Вывод: 5 (старое значение)
printf("%d\n",++x);   // Вывод: 7 (новое значение)
```

Для указателей рассматриваемые операторы работают в единицах размеров элементов, на которые они указывают, так что **p++** приводит к тому, что **p** указывает на следующий элемент массива.

В особенности часто операторы инкремента и декремента приводят к вопросу о последовательности вычислений — например, чему будет равно значение **i** после выполнения выражения наподобие **i = ++i + --i - i++;**.

Дело в том, что в C++ порядок вычисления подвыражений внутри выражений не определен, и, например, код **int i=1; v[i] = i++;** может вычисляться как **v[1]=1;**, **v[2]=1;** либо каким-то иным способом — в зависимости от конкретного компилятора. Поэтому никогда не используйте выражений, результат которых зависит от порядка вычисления!

Операторы new и delete

Оператор **new** служит для создания объекта в динамической памяти, а **delete** — для его уничтожения. Оператор **new** возвращает указатель на созданный объект, а в случае, если памяти не хватает, — генерирует исключение **bad_alloc** (об исключениях более подробно рассказывается в главе «Исключения», стр. 87). В ряде старых компиляторов при невозможности выделения памяти оператор **new** возвращает нулевой указатель.

```
int*i = new int;
*i = 5;
delete i;
```

Можно применять оператор **new** и для создания массивов, но в этом случае для освобождения памяти следует использовать оператор **delete[]**:

```
int*i = new int[5];
int[0] = 0;
int[4] = 4;
delete[]i;
```

Функция **set_new_handler()** позволяет указать функцию-обработчик, которая будет вызвана при нехватке памяти:

```
void nomem()
{
    cerr << "нет памяти\n";
    throw bad_alloc;
}

int main()
{
    set_new_handler(nomem);
...
```

Оператор **delete** сам проверяет неравенство переданного ему указателя нулевому, так что передача ему нулевого указателя не является некорректной.

Дополнительную информацию об операторах **new** и **delete** можно найти в разделе «Размещение объектов в памяти» на стр. 72.

Инструкции выбора

В C++ значение может проверяться в конструкциях **if** и **switch**, а также в тернарном операторе **?:**. Вот как выглядит синтаксис конструкций **if** и **switch**:

```
if (условие) инструкция
if (условие) инструкция else инструкция
switch (условие) инструкция
```

Если *условие* в конструкции **if** истинно (не равно нулю), выполняется первая (возможно, единственная инструкция, могущая быть составной); в противном случае — вторая, если таковая имеется. Таким образом, *условие* может быть не только логическим, но и, например, целым числом. Если **i** — целое, то **if(i)...** эквивалентно **if(i!=0)...**. Соответственно, для указателя **p** проверка **if(p)**, по сути, является определением того, представляет ли он собой указатель на допустимый объект.

С учетом сокращенных вычислений логических операторов (см. стр. 38) проверка **if(p&&...)** будет выполнять вторую часть условия (указанную троеточием) только в том случае, если указатель **p** — ненулевой. Таким образом, в такой конструкции можно смело использовать разыменование указателя, поскольку оно будет реально выполняться только в том случае, если указатель **p** — ненулевой, например:

```
if (p && (*p == 0)) ...
```

Часть инструкций **if** можно заменить более кратким тернарным оператором **?:**. Так,

```
if (a) m = b; else m = c;
```

можно записать проще — **m = a?b:c**. Более того, пользуясь тем фактом, что имя функции и указатель на нее взаимозаменяемы (см. стр. 50), можно использовать данный оператор, например, таким кажущимся необычным образом:

```
y = ((x>0) ? sin : cos)(x);
```

Конструкцию выбора **switch** удобно применять тогда, когда некоторая целочисленная величина может принимать некоторое множество значений, в ответ на которые предпринимаются различные действия. Если использовать инструкцию **if**, то потребуется сложная конструкция наподобие:

```
if (val == 1) f1();
else if (val == 2) f2();
else if (val == 3) f3();
else ff();
```

Воспользовавшись возможностью **switch**, этот исходный текст можно записать следующим образом:

```
switch(val)
{
    case 1: f1(); break;
    case 2: f2(); break;
    case 3: f3(); break;
    default: ff();
}
```

Каждая ветвь **case** завершается оператором **break**, который приводит к выходу из конструкции **switch**. Если оператор **break** опущен, продолжится выполнение кода из следующей ветви. Такое свойство данной конструкции в ряде задач может оказаться весьма полезным, но при использовании этой возможности следует тщательно документировать код, чтобы была ясна преднамеренность ваших действий.

Циклы

В C++ существует три вида циклов — **for**, **while** и **do**:

```
while(условие) инструкция
do инструкция while (выражение);
for(инициализация; условие; выражение) инструкция
```

Курсивом выделены необязательные части конструкций. Цикл **while** работает следующим образом: сперва проверяется *условие*, если оно истинно (не равно нулю) — выполняется *инструкция* (которая может быть составной). После *инструкции* цикл повторяется, т.е. идет проверка *условия*. Таким образом, если изначально *условие* ложно (равно нулю), то тело цикла не выполняется ни разу.

Цикл **do** сначала выполняет инструкцию, а затем переходит к проверке *условия*. Если условие не выполняется (равно нулю), происходит выход из цикла; в противном случае цикл повторяется. Тело цикла **do** всегда выполняется по крайней мере один раз.

Цикл **for** начинается с *инициализации*. Затем циклически проверяется *условие*, и, если оно истинно (не равно нулю) — выполняется *инструкция* (если *условие* ложно (равно нулю), цикл завершает работу). После *инструкции* выполняется *выражение* цикла и переход к новой итерации, начинающейся с проверки *условия*. Все части цикла могут быть пустыми; пустое условие рассматривается как всегда истинное. Таким образом, цикл **for(;;)** представляет собой «вечный» цикл.

В части *инициализации* цикла **for** можно объявлять переменные, область видимости которых в этом случае ограничена *инструкцией* цикла (в старых компиляторах область видимости простирается за пределы цикла **for** до завершения блока, содержащего цикл, — как если бы переменные были объявлены непосредственно перед циклом **for**).

Помимо штатного выхода из цикла (по достижении соответствующего условия), выйти из любого цикла можно при помощи оператора **break**. При использовании вложенных циклов оператор **break** относится к ближайшему охватывающему циклу. Второй оператор — **continue** — прерывает выполнение тела цикла (*инструкции* в приведенных схемах) и заставляет перейти к выполнению очередной итерации (проверке *условия*).

Инструкция goto

В C++ имеется инструкция безусловного перехода **goto**, но ее применение крайне ограничено, как правило, выходом из вложенных циклов или конструкции **switch**, поскольку оператор **break** действует только на ближайший охватывающий цикл или **swith**-конструкцию. Синтаксис **goto** прост и похож на синтаксис в других языках программирования:

```
goto Label;

Label: ...
```

Глава 3. Функции

Функции представляют собой описание способа выполнения той или иной составной операции, которая может быть вызвана из разных мест программы, а ее выполнение — зависеть от передаваемых функции аргументов.

Объявления функций

Объявление функции указывает ее имя, тип возвращаемого значения (если таковое имеется), количество и типы аргументов, которые должны быть переданы функции при ее вызове, например,

```
char* strcpy(char*to, const char*from);
```

Семантика передачи аргументов функции аналогична инициализации. При вызове проводится проверка типов аргументов, и при необходимости выполняется неявное преобразование типов.

Объявление функции может содержать имена аргументов, что иногда повышает удобочитаемость программы, однако компилятор их игнорирует.

Если в качестве типа возвращаемого значения указан **void**, это означает, что функция не возвращает значения.

Каждая функция, вызываемая в программе, должна быть где-то определена, причем только один раз. Определение функции — это ее объявление, в котором присутствует тело функции. Например,

```
int min(int,int);          // Объявление функции

int min(int x, int y)      // Определение функции
{
    return (x < y) ? x : y;
}
```

Типы параметров и возвращаемых значений в объявлениях и определении должны совпадать; имена аргументов частью типа не являются и могут не совпадать.

Возможны ситуации, когда функция не зависит от одного из своих аргументов (обычно это происходит при упрощении кода или при запланированном в дальнейшем расширении возможностей функции). Такой аргумент может не иметь имени в определении функции.

Функция может быть объявлена с использованием спецификатора **inline**, что делает ее встраиваемой — т.е. компилятор может оптимизировать код и вместо вызова функции вставить ее код непосредственно в точку вызова. Для этого тело функции должно находиться в данной обла-

сти видимости, так что обычно определения таких функций располагают в заголовочных файлах. Наличие спецификатора `inline` не гарантирует, что код функции будет в самом деле встраиваться вместо вызова, и никак не влияет на получение адреса функции. Следует также отметить, что в современных компиляторах применение спецификатора `inline` практически лишено смысла (см. стр. 26, а также [8]).

Локальные переменные

Используемые в функции локальные переменные инициализируются в момент выполнения строк, содержащих их определения. По умолчанию это происходит при каждом вызове функции, и каждый раз создается новая переменная, которая уничтожается при выходе из области видимости. Если локальная переменная объявлена со спецификатором `static`, то такая переменная инициализируется только один раз, при первом выполнении строки, содержащей ее определение, и не уничтожается до завершения работы программы, сохраняя свое значение между вызовами функции. Такое поведение статических локальных переменных позволяет использовать их вместо глобальных, избегая тем самым проблем с областями видимости глобальных переменных.

Рекомендуется объявлять локальные переменные как можно локальнее, чтобы свести время их жизни к минимуму. Наиболее распространенное исключение из этого правила — вынесение переменных за пределы цикла.

Передача аргументов

При вызове функции выделяется память под ее формальные аргументы, и каждому формальному аргументу присваивается значение соответствующего фактического аргумента. Семантика передачи аргументов идентична семантике инициализации. В частности, осуществляется проверка соответствия типов формальных и фактических аргументов, и при необходимости выполняются либо стандартные, либо пользовательские преобразования типов. Существуют специальные правила для передачи массивов в качестве аргументов, для передачи неуказанных аргументов, а также возможность указания аргументов по умолчанию.

Передача аргументов возможна как по значению, так и по ссылке. Фактические аргументы, передаваемые по значению, копируются, поэто-

му функция не может их модифицировать; при передаче по ссылке такое изменение возможно. Например, при выполнении функции `g()` из приведенного ниже кода

```
void f(int i, int&j)
{
    ++i;
    ++j;
}
void g()
{
    int i = 1, j = 1;
    f(i,j);
    cout << i << "  " << j << endl;
}
```

будут выведены значения 1 и 2, так как функция `f()` изменяет копию аргумента `i` и сам аргумент `j`, поскольку он передан по ссылке.

Чтобы указать, что функция не может изменять передаваемые по ссылке объекты, используется спецификатор **`const`**: **`void f(const Object&o);`**. Спецификатор **`const`** используется также для того, чтобы указать, что функция не может изменять объект, указатель на который передается в функцию. Так, при вызове функции **`void func(const char*s)`** в теле функции может изменяться сам указатель `s`, но не содержимое строки, на которую он указывает. Следует также отметить, что как **`const`**-ссылки могут передаваться литералы, константы и аргументы, требующие преобразования типов; передача их в качестве неконстантных ссылок запрещена.

При передаче в функцию объектов, которые не должны изменяться, предпочтительнее использовать передачу константных ссылок, а не значений. Однако, если в функции будет выполняться копирование переданного объекта, следует подумать о передаче его по значению и использовании создаваемой при этом копии. Например, в фрагменте

```
T& operator=(const T& t)
{
    T tmp(t);
    swap(tmp);
    return *this;
};
```

можно избежать явного копирования, передавая параметр по значению:

```
T& operator=(T t)
{
    swap(t);
    return *this;
};
```

Если в качестве аргумента функции используется массив, то передается указатель на его первый элемент, т.е. аргумент типа **T[]** преобразуется в **T***. Отсюда следует, что присваивание элементу массива, являющегося аргументом, приводит к изменению значения элемента исходного массива, т.е. массив не может быть передан по значению. В качестве возможного решения можно предложить использование С-строк с завершающим нулевым символом, что позволяет вычислить их размер, передачу размера в качестве дополнительного параметра или применение шаблона **vector**.

В функции могут использоваться значения аргументов по умолчанию. Эти значения указываются в объявлении функции при помощи конструкции **= value**. Например, объявление функции

```
void xyz(int a, int b = 5, int c = 10);
```

позволяет вызывать функцию с различным числом аргументов:

```
xyz(1,2,3);     // Вызов со всеми аргументами
xyz(2,3);       // Эквивалентно вызову xyz(2,3,10)
xyz(3);         // Эквивалентно вызову xyz(3,5,10)
```

На аргументы по умолчанию налагаются определенные ограничения. Во-первых, они должны быть последними в списке аргументов функции, т.е., например, нельзя объявить функцию **void xyz(int,int=2,int);**. Во-вторых, нельзя опустить один из аргументов по умолчанию, не опустив все последующие, т.е. нельзя вызвать функцию **xyz(1,,3)**. В-третьих, аргумент по умолчанию не может быть повторен или изменен в последующих объявлениях в той же области видимости.

C++, как и C, позволяет иметь функции с переменным количеством аргументов. Список аргументов в таких функциях завершается троеточием (**...**), что означает — некоторые аргументы, для которых заранее не известно ни их количество, ни их тип. Для работы с такими аргументами в заголовочном файле **<cstdarg>** (**<stdarg.h>** в старых компиляторах) имеются все необходимые макросы и определения. В теле функции с переменным числом аргументов следует объявить переменную типа **va_list**, для которой вызывать **va_start(x,y)**, где **x** — переменная типа **va_list**, а **y** — последний аргумент перед троеточием в объявлении функции. Затем все последующие аргументы вы можете получить при помощи

вызовов **va_arg(x,type)**, где **x** — переменная типа **va_list**, а **type** — тип очередного аргумента. По завершении работы обязательно надо вызвать **va_end(ap)**; этот вызов приведет стек в нормальное состояние, которое обеспечит возможность нормального вызова из функции. Повторные вызовы **va_*** в пределах одной функции не разрешены. Очевидно, что при этом не выполняется никакая проверка типов аргументов или их реального количества, и программист должен сам обеспечить корректную работу функции. В качестве примера опишем функцию, которая определяет минимальное из нескольких целых чисел. Поскольку узнать количество реально передаваемых аргументов невозможно, будем передавать его функции как первый аргумент.

```
int minx(int n, ...)
{
    va_list ap;
    va_start(ap,n);
    int m = va_arg(ap,int);
    while(--n)
    {
        int x = va_arg(ap,int);
        if (m > x) m = x;
    }
    va_end(ap);
    return m;
}
```

Использовать такие функции с переменным числом параметров не рекомендуется по многим причинам; одна из них — невозможность контроля типов компилятором.

Возвращаемые значения

Функция должна возвращать значение, если она не объявлена как **void**-функция, и наоборот, функция, объявленная как **void**, не должна возвращать никаких значений. Возвращаемое значение задается инструкцией **return**. Данная инструкция, кроме того, прекращает выполнение функции. В теле функции может быть несколько инструкций **return**; в теле функции, не возвращающей значения, инструкции **return** могут

отсутствовать — в этом случае возврат из функции выполняется по достижении окончания ее тела.

Семантика возврата значения из функции идентична семантике инициализации, т.е. инструкцию **return** можно рассматривать как инициализацию неименованной переменной возвращаемого типа. При этом осуществляется сравнение типа выражения в инструкции **return** с типом возвращаемого значения, и при необходимости выполняется стандартное либо пользовательское преобразование типа.

Поскольку, как уже упоминалось на стр. 46, локальные переменные (не объявленные как **static**) при выходе из функции уничтожаются, нельзя возвращать указатели или ссылки на них, несмотря на то, что формально язык это не запрещает.

Функция **void** не может возвращать значение, однако в качестве выражения **return** в такой функции может быть использован вызов другой **void**-функции, например,

```
void a() {};
void b() { return a(); }
```

Такая форма инструкции **return** (не поддерживаемая рядом старых компиляторов) важна при написании шаблонов функций, когда тип возвращаемого значения является параметром шаблона (о шаблонах функций см. стр. 93).

Указатели на функции

С функцией могут выполняться только два действия — ее вызов или получение ее адреса. Указатель, полученный взятием адреса функции, может затем использоваться для вызова функции, например:

```
void out(const char * s) {/*...*/};    // Функция
void (*func)(const char *);            // Указатель

void f()
{
    func = &out;          // Получение адреса
    out("test");          // Вызов через указатель
}
```

Компилятор распознает, что **func** является указателем на функцию, и вызывает функцию, на которую он указывает, т.е. операция разыменова-

ния (*) является не обязательной. Точно так же необязательным является оператор получения адреса (&), т.е. функция f() из примера выше может быть записана и так:

```
void f()
{
    func = out;          // Получение адреса
    (*out)("test");      // Вызов через указатель
}
```

Аргументы указателей на функцию объявляются точно так же, как и аргументы самих функций. При присваивании типы функций должны *в точности совпадать*. Правила передачи аргументов при вызове функций через указатели те же, что и при непосредственном вызове функций (см. раздел «Передача аргументов» на стр. 46).

В связи со сложностью синтаксиса объявлений функций зачастую имена типов указателей на функции определяются при помощи конструкции **typedef** — например, сравните объявления функции **set_new_handler** с использованием конструкции **typedef** и без нее, приведенные на стр. 27.

Основное применение указателей на функции — реализация простой формы полиморфных процедур, которые можно вызывать для объектов разных типов. Примером может служить библиотечная функция **qsort**, которая позволяет сортировать массивы различных объектов с использованием разных функций сравнения объектов. Функция **qsort** получает указатель на функцию сравнения в качестве одного из своих аргументов.

Перегрузка функций

Одно из важных отличий языка C++ от C состоит в том, что C++ позволяет иметь функции с одинаковыми именами, но различными типами аргументов. С точки зрения компилятора имеет значение не имя функции, а ее *сигнатура*, т.е. совокупность информации об имени функции и типах ее аргументов (тип возвращаемого значения в сигнатуру не входит). Таким образом, при наличии функций с одинаковым именем компилятор самостоятельно определяет по типу передаваемых аргументов, какая именно функция будет вызвана, например:

```
void out(long);
void out(double);
void out(const char*);
void g()
{
    out(1L);       // Вызов out(long)
```

```
    out("Hi!");     // Вызов out(const char*)
    out(1.0);       // Вызов out(double)
    out(1);         // Неоднозначность: out(double(1))
}                   // или вызов out(long(1)) ?
```

При разрешении перегрузки выполняется поиск функции путем следующей многоступенчатой проверки соответствия типов аргументов для всех функций с одинаковыми именами.

1. Если имеется функция с точным соответствием типов, т.е. с полным соответствием или с соответствием, достигаемым тривиальными преобразованиями (имя массива в указатель, имя функции в указатель на функцию или типа `T` в `const T`), то выбирается эта функция, иначе поиск продолжается.
2. Проверяется соответствие, достигаемое продвижением интегральных типов (`bool` в `int`, `char` в `int`, `short` в `int`, а также их `unsigned`-аналогов), а также `float` в `double` и `double` в `long double`. Если соответствующая функция не найдена, поиск продолжается.
3. Проверяется соответствие путем стандартных преобразований (например, `int` в `double`, `double` в `int`) указателей на производные типы в указатели на базовые типы, указатели на произвольные типы в указатели `void*`, `int` в `unsigned int`. Если соответствующая функция не найдена, поиск продолжается.
4. Проверяется соответствие, достигаемое применением пользовательских преобразований. Если соответствующая функция не найдена, поиск продолжается.
5. Проверяется соответствие, полученное за счет троеточия (`...`) в объявлении функции.

Если в результате функция не найдена, вызов отвергается за отсутствием соответствующей функции. Если же на одном уровне проверки находится несколько соответствующих функций, то вызов отвергается из-за неоднозначности. Разрешение перегрузки не зависит от порядка объявления функций.

Возвращаемые типы в разрешении перегрузки не участвуют. Не участвуют в разрешении перегрузки и функции, объявленные в разных областях видимости, например:

```
void out(int);
void g()
{
    void out(double);
    out(1);             // Вызывается out(double)
}
```

При разрешении перегрузки функций, имеющих несколько аргументов, для каждого аргумента находится наилучшее соответствие согласно приведенным выше правилам. Из множества допустимых функций выбирается та, у которой соответствие типа одного из аргументов типу формального параметра оказывается наилучшим, а соответствие типов всех остальных аргументов – не хуже, чем у прочих функций этого множества. Если такая функция не найдена, вызов считается неоднозначным.

Наличие неоднозначности, как правило, говорит о недостатках проектирования. Избежать неоднозначности можно двумя способами – добавлением функции, которая разрешает неоднозначность:

```
void out(long);
void out(double);
inline void out(int i) { out(long(i)); }
void g()
{
    out(1); // Вызов добавленной функции out(int)
}
```

либо явным приведением аргумента к требуемому типу:

```
void out(long);
void out(double);
void g()
{
    out(static_cast<long>(1)); // Вызов функции
                               // out(long)
}
```

Глава 4. Препроцессор и макросы

Препроцессор — это достаточно простой обработчик макросов в исходном тексте, работающий на уровне лексем до начала синтаксического разбора исходного текста. Директивы препроцессора начинаются с символа **#**, который должен быть первым символом строки, не являющимся пробельным.

Наиболее часто используемая директива препроцессора — **#include**, которая заставляет заменить ее текстом указанного в ней файла:

```
#include "myfile"
```

При включении заголовочных файлов стандартной библиотеки вместо кавычек используются угловые скобки, что заставляет искать указанный файл в каталоге стандартной библиотеки, а не в текущем. Следует обратить внимание на то, что наличие пробелов внутри **<>** или **<<>>** существенно, и препроцессор может не найти включаемый файл, если директива имеет вид **#include < iostream >**, а не **#include <iostream>**.

Макросы достались C++ в наследство от C, где они играли очень большую роль. Однако в C++ их применения следует избегать. Как правило, применение макросов говорит о плохом дизайне программы. Практически для всего, что вы можете достичь при помощи макросов, имеются специальные средства C++ — такие, как константы, шаблоны, встраиваемые функции и т.п. Макросы — это очень опасная игрушка!

Простой макрос определяется следующим образом:

```
#define MACRO macro definition
```

После такого определения все лексемы **MACRO** в исходном тексте будут заменены на текст **macro definition**, например, фрагмент

```
var = MACRO
```

превратится в

```
var = macro definition
```

Макрос может быть определен с аргументами, например, так.

```
#define MACRO(x,y) x*y*y
```

После чего можно использовать этот макрос с различными аргументами, которые будут буквально заменены при макроподстановке, например

```
x = MACRO(5,z+2)
```

превратится в

```
x = 5*z+2*z+2
```

Этот маленький пример помимо демонстрации применения макроса с аргументами показывает, как легко, воспользовавшись макросом, получить совсем не тот результат, к которому стремишься.

Кроме того, макросы понимают в достаточной мере только круглые и квадратные скобки. В C++, однако, определена конструкция с угловыми скобками, используемая в шаблонах. Макросы не в состоянии корректно обработать эту ситуацию, так что в вызове

```
MACRO(Foo<int, double>)
```

макрос полагает, что ему переданы два аргумента, а именно **Foo<int** и **double>**, в то время как в действительности эта конструкция представляет собой единый объект C++.

Имена макросов не могут быть перегружены (см. раздел «Перегрузка функций», стр. 51), и не должны использоваться рекурсивные макросы. Макросы работают со строками символов и практически ничего «не знают» о синтаксисе C++, и совсем ничего — о типах и областях видимости имен. Компилятор получает текст только после выполнения макроподстановок, так что ошибка, вызванная макросом, будет замечена им только в месте подстановки, а не определения макроса.

Добавим, что для конкатенации строк можно использовать в макроопределениях оператор **##**. Например,

```
#define MACRO(x,y) x##y
var = MACRO(boo,foo)
```

превращается в

```
var = boofoo
```

Унарный оператор # позволяет получить соответствующий строковый литерал. При этом

```
#define STR(s) #s
var = STR(foo)
```

превращается в

```
var = "foo"
```

Директива **#undef MACRO** гарантирует, что далее не существует макроса с указанным именем **MACRO**. Директива **#ifdef ARG** (или **#if defined(ARG)**) заставляет удалить весь текст от нее до соответству-

ющей директивы **#endif**, если макрос **ARG** не был определен (обратное действие — удаление текста, если макрос определен — имеет директива **#ifndef** (**#if !defined()**)). Можно использовать также интуитивно понятные директивы **#elif** и **#else** для составления сложных директив:

```
#if defined(VAR1)
...
#elif defined(VAR2)
...
#else
...
#endif
```

Эти директивы, в частности, позволяют избежать повторного включения заголовочных файлов, путем использования следующей структуры файла:

```
#ifndef FILEID  // Некоторое уникальное макроимя
#define FILEID

... Содержимое файла ...

#endif
```

При первом чтении файла макроимя **FILEID** не определено, поэтому текст файла считывается полностью, и при этом макрос **FILEID** становится определенным при помощи директивы **#define**, так что при втором чтении содержимое этого файла будет пропущено.

Директива **#error Text** заставляет компилятор сообщить об ошибке, выведя на экран указанный текст, и прервать компиляцию.

В C++ имеются некоторые предопределенные макросы.

Макрос	Описание
__LINE__	Десятичное число — номер текущей строки файла
__FILE__	Строковый литерал — имя текущего файла
__DATE__	Дата трансляции в формате «Mmm dd yyyy» (строковый литерал)
__TIME__	Время трансляции в формате «hh:mm:ss» (строковый литерал)
__cplusplus	Определен, если компиляция выполняется компилятором C++

В конкретных компиляторах могут существовать свои предопределенные макросы, указывающие тип и версию компилятора, различную другую информацию. Так, в ряде компиляторов (например, Open Watcom) макрос **__func__** указывает текущую функцию.

Глава 5. Исходные файлы и программы

Традиционной единицей компиляции является файл. Как правило, хранить законченную программу в одном файле оказывается невозможно. В частности, код стандартных библиотек и операционной системы не предоставляется в виде исходных текстов и не является частью текста пользовательской программы. Даже просто хранение всего исходного текста программы реального размера в одном файле непрактично и неудобно. Разбиение программы на отдельные файлы позволяет подчеркнуть ее логическую структуру, облегчает понимание и сопровождение. Кроме того, при внесении изменений в программу, состоящую из одного файла, требуется перекомпиляция всей программы.

Предоставленный программистом компилятору исходный файл обрабатывается препроцессором с выполнением всех макроподстановок (см. главу «Препроцессор и макросы», стр. 55) и включений файлов **#include**, в результате чего получается *единица трансляции*, с которой и работает компилятор.

Для того чтобы обеспечить возможность раздельной компиляции, программист должен предоставить компилятору все необходимые объявления, которые позволяют выполнить компиляцию единицы трансляции отдельно от остальной части программы. Объявления в программе, состоящей из нескольких раздельно компилируемых частей, должны быть согласованы в той же степени, что и в программе, состоящей из отдельного файла. Любой объект может быть объявлен в программе много раз (важно лишь, чтобы типы в объявлениях совпадали), но определение должно быть только одно. Вот пара примеров:

```
// file1.cpp
    int x = 1;       // Определение (есть инициализация)
    int f() { /* ... */ } // Определение (есть тело)

// file2.cpp
    extern int x;    // Объявление (наличие extern)
    int f();              // Объявление
    void g() { x = f(); } // Определение
```

Здесь все корректно, ни одно правило не нарушено. Ключевое слово **extern** говорит о том, что данное объявление глобальной переменной является только объявлением, а не определением. Если бы при этом переменная **x** была инициализирована, ключевое слово **extern** было бы про-

игнорировано, поскольку объявление с инициализацией всегда является определением.

```
// file1.cpp
    int x = 1;
    int b = 1;
    extern int c;

// file2.cpp
    int x;              // То же, что int x = 0;
    extern double b;
    extern int c;
```

Здесь имеется ряд ошибок. Переменная **x** определена дважды (поскольку глобальные переменные инициализируются по умолчанию). Переменная **b** объявлена с различными типами, а переменная **c** объявлена дважды, но не определена. Ошибки такого типа не обнаруживаются компилятором, который в каждый момент времени работает только с одной единицей трансляции, но большинство из них выявляется на стадии компоновки.

Встраиваемые функции (см. стр. 45) должны иметь *идентичные определения* в каждой единице трансляции. Данная ошибка — разные определения встраиваемой функции — крайне трудно обнаруживается компиляторами (если вообще обнаруживается).

По умолчанию **const** и **typedef** подразумевают внутреннюю компоновку, т.е. допустим следующий пример:

```
// file1.cpp
    typedef int T;
    const int x = 7;

// file2.cpp
    typedef double T;
    const int x = 8;
```

Однако такая практика является потенциальным источником ошибок. Для обеспечения согласованности лучше размещать глобальные **const** и **inline** в заголовочных файлах. Внешнюю компоновку константы можно обеспечить путем ее явного объявления:

```
// file1.cpp
    extern const int x = 7;

// file2.cpp
    extern const int x;
    void f() { cout << x; } // Выводится 7
```

Согласованность объявлений можно обеспечить простым способом, размещая их в заголовочных файлах, которые затем включаются директивой **#include** в исходные файлы, содержащие исполняемый код и/или определения объектов (см. стр. 55).

Обычно при разработке заголовочных файлов используется неписаное правило (не являющееся требованием языка), что заголовочный файл может содержать такие элементы.

Именованные пространства имен	`namespace N { /*...*/ }`
Определения типов	`struct Point { int x, y; };`
Объявления шаблонов	`template<class T> class Z;`
Определения шаблонов	`template<class T> class V { /* ... */ };`
Объявления функций	`extern int strlen(const char*);`
Определения встраиваемых функций	`inline char get(char*p) { return *p++; }`
Объявления данных	`extern int a;`
Определения констант	`const double pi = 3.1415926;`
Перечисления	`enum Color { red, yellow, green };`
Объявления имен	`class Matrix;`
Директивы включения	`#include <iostream>`
Макроопределения	`#define VERSION 12`
Директивы условной компиляции	`#ifdef __cplusplus`
Комментарии	`/* Вычисление длины файла */`

Заголовочный файл не должен содержать следующего.

Определения обычных функций	`char get(char*p) { return *p++; }`
Определения данных	`int a;`
Определения агрегатов	`int tbl[] = {1,2,3};`
Неименованные пространства имен	`namespace { /* ... */ }`
Экспортируемые определения шаблонов	`export template<class T> f(T t){/*...*/}`

Основное правило — правило одного определения — гласит, что каждый конкретный класс, перечисление, шаблон и тому подобное должны быть определены в программе только один раз. Более строгое правило допускает два определения класса, шаблона или встраиваемой функции в качестве определения одной и той же сущности тогда и только тогда, когда они находятся в различных единицах трансляции, идентичны лексема

за лексемой, и значения лексем одинаковы в обеих единицах трансляции. Например, в двух `.cpp`-файлах программы может быть определение, например `struct X {int x,y;};`, и, согласно правилу одного определения, это будет описание одной и той же структуры. Но такое описание чревато тем, что в конечном итоге при внесении изменений надо строго синхронизировать описания в обоих (а то и большем количестве) файлах. Поэтому гораздо разумнее выносить такие описания в заголовочный файл.

Не включайте в программу излишние заголовочные файлы и не используйте директиву `#include` там, где можно обойтись предварительным объявлением.

Глава 6. Классы

По сути, классы и есть то главное, что отличает C++ от C и делает его объектно-ориентированным языком программирования.

Классы в C++ — это пользовательские типы, представляющие собой набор объектов различных типов и функций для работы с ними, а также правила доступа к этим объектам и функциям со стороны других объектов и функций.

Объявление класса сходно с объявлением структуры (см. стр. 19) (структура в C++ формально является частным случаем класса, все члены которого открыты). Для этого используется ключевое слово **class**, после которого указывается имя класса, через двоеточие — базовый класс (базовые классы в случае множественного наследования), а затем в фигурных скобках перечисляются члены-данные и функции-члены класса, например:

```
class Test
{
public:
    Test(int value = 0);
    void inc();
    void dec();
    int  get() const;
private:
    int value_;
};
```

Функции-члены

Функции, объявленные внутри класса (функции-члены), могут вызываться только для переменной соответствующего класса с использованием стандартного синтаксиса доступа к членам структуры, например:

```
Test t;
Test * q = &t;
int x = t.get();
t.inc();
int y = q->get();
```

Функции-члены могут быть определены в объявлении класса (в этом случае они считаются встраиваемыми), например:

```
class Test
{
public:
    void inc() { ++value_; }
    void dec();
```

При объявлении вне класса с помощью оператора `::` должно быть явно указано имя класса, к которому относится данная функция, например:

```
void Test::dec() { --value_; }
```

В теле функции-члена имена членов этого же класса используются без явного указания объекта. Так, в приведенном выше примере **value_** является членом объекта, для которого вызывается данная функция.

Объявление класса может быть повторено в разных исходных файлах, если во всех них оно идентично — например, при помощи механизма включения заголовочных файлов. Определения же функций вне класса должны подчиняться правилу одного определения.

Указатель на член класса (как функцию-член, так и данные-члены) можно получить, применив оператор получения адреса **&** к полностью квалифицированному имени члена класса, например **&Test::dec**. Переменная типа «указатель на член класса **X**» объявляется с использованием формы **X::***. Поскольку обычно такое объявление неудобочитаемо, используются синонимы типов, определяемые с помощью оператора **typedef**.

Указатель на член **m** можно использовать в комбинации с объектом. Операторы **->*** и **.*** позволяют выразить такую комбинацию, например, **p->*m** связывает **m** с объектом, на который указывает **p**, а **obj.*m** связывает **m** с объектом **obj**. Полученный результат можно использовать в соответствии с типом **m**. Сохранение результата операций **->*** и **.*** для дальнейшего использования невозможно.

Разумеется, функции-члены, вызываемые через указатели, могут быть виртуальными.

Статические члены классов (см. стр. 71) не связаны с конкретными объектами класса, так что указатель на статический член работает так же, как и обыкновенный указатель.

```
class Test {
public:
    virtual void test()  { cout << "Test::test\n";  }
    static  void guest() { cout << "Test::guest\n"; }
};
class Best : public Test {
public:
```

```
    virtual void test() { cout << "Best::test\n"; }
};

typedef void (Test::*testfunc)(); // Указатель на
                                  // функцию-член

int main()
{
    testfunc f = &Test::test;
    Test t;.
    Best * b = new Best;

    (t.*f)();       // Test::test
    (b->*f)();      // Best::test (виртуальная
                    // функция-член)

    void (*tfunc)() = &Test::guest;
    tfunc();
    void (*bfunc)() = &Best::guest;
    bfunc();
    f = &Test::guest; // Ошибка присваивания указателя
                      // указателю на член
    return 0;
}
```

Ключевое слово this

Функции-члену неявно передается указатель на объект, для которого она вызвана. Для его использования в теле функции-члена используется ключевое слово **this**. Таким образом, определение рассмотренной выше функции-члена может выглядеть следующим образом:

```
void Test::dec() { --this->value_; }
```

В нестатической функции-члене класса **X** тип указателя **this** — **X***, для константной функции-члена этот тип — **const X***.

Однако **this** не является обычной переменной — невозможно получить ее адрес или присвоить ей какое-либо значение. Несмотря на то что некоторые старые компиляторы позволяют такие действия, это является ошибкой.

Одно из применений указателя **this** — возврат ссылки на объект, для которого вызывается функция (такое возвращаемое значение позволяет исполь-

зовать запись цепочки функций-членов, например: `x.f1().f2().f3()`. Для этого функции должны быть объявлены, как возвращающие ссылку на объект класса, а в теле функций использоваться возврат `return *this;`.

Константные функции-члены

Если функция-член не изменяет внутреннее состояние (данные-члены) объекта класса, то такая функция объявляется как константная, с использованием суффикса `const`, как это было сделано у функции `get()` в рассматриваемом нами классе. При определении такой функции также необходимо указывать суффикс `const`, т.е. он является частью типа функции:

```
int Test::get() const { return value_; }
```

Константная функция-член может быть вызвана как для константного, так и для неконстантного объекта класса, в то время как неконстантная функция-член может быть вызвана только для объекта класса, не являющегося константой.

Если некоторый член класса объявлен с использованием ключевого слова `mutable`, то такой член может изменяться константной функцией. Примером использования этой технологии может служить, например, некоторое предвычисление — пусть имеется некоторая функция, выполняющая сложные математические вычисления над объектом и возвращающая их результат (скажем, функция статистической обработки данных объекта). Поскольку такая функция не изменяет состояние объекта, эта функция должна быть константной. Однако, если она вызывается неоднократно, имеет смысл сохранить вычисленные значения, чтобы при следующем вызове не повторять все вычисления заново. Однако такое сохранение заставляет сделать функцию, по сути, не изменяющую объект, неконстантной. В этом случае лучше описать член для хранения вычисленных данных как `mutable`, оставив функцию константной.

Конструкторы и деструктор

Особое место среди функций-членов занимают конструкторы и деструктор. Конструктор — это функция-член с именем, совпадающим с именем класса. Конструктор — специальная функция, которая вызывается при создании объекта класса. Класс может иметь несколько конструкторов с разным набором аргументов. Если конструктору требуются аргументы, то они должны быть предоставлены при инициализации объекта. До тех пор, пока не будет завершен вызов конструктора, объект не существует.

Конструктор не может возвращать никакое значение, поэтому единственный способ сообщить об ошибке, происшедшей в процессе работы конструктора, — это генерировать исключение.

Конструктор вида `ClassName()` — конструктор по умолчанию. При отсутствии он генерируется компилятором автоматически, созданный таким образом конструктор неявно вызывает конструкторы по умолчанию для всех членов класса и конструкторы базовых классов. Генерация конструктора компилятором невозможна, если класс содержит члены, являющиеся константами или ссылками.

Конструктор специального вида `ClassName([const] ClassName&)` — это копирующий конструктор, который вызывается при инициализации создаваемого объекта другим объектом данного типа. При его отсутствии генерируется копирующий конструктор по умолчанию, почленно копирующий все члены-данные исходного объекта.

Крайне близок к копирующему конструктору оператор присваивания, который обычно объявляется как `ClassName& operator=(const ClassName&)`, и который отличается от копирующего конструктора тем, что копирующий конструктор должен инициализировать «чистую» (неинициализированную) память, в то время как оператор присваивания имеет дело с уже созданным объектом.

Рекомендуется следующая реализация оператора присваивания при помощи копирующего конструктора и функции обмена внутреннего содержимого двух классов (пусть, например, это функция-член `swap(T&)`):

```
T& operator=(const T& t)
{
    T tmp(t);
    swap(tmp);
    return *this;
};
```

Если в качестве членов класса выступают объекты других классов, то аргументы их конструкторов перечисляются в списке инициализации членов, который указывается в определении конструктора класса после его имени, будучи отделен двоеточием, после которого перечисляются члены с аргументами их инициализации, например

```
class A
{
    B b;
    C c;
...
public:
    A(int x):b(x+1),c(0) { ... }
};
```

Конструкторы членов вызываются в порядке объявления членов в классе, а не в порядке списка инициализации. Во избежание путаницы рекомендуется всегда составлять список инициализации в порядке, соответствующем объявлению класса. То же относится и к списку базовых классов (см. раздел «Наследование», стр. 78).

Член, не нуждающийся в аргументах конструктора, может не указываться в списке инициализации — при этом для него будет вызван конструктор по умолчанию.

Константные члены и члены-ссылки могут инициализироваться только посредством их указания в списке инициализации и никак иначе.

При уничтожении объекта такого класса сначала вызывается деструктор класса, а затем — деструкторы членов в порядке, обратном порядку вызова их конструкторов.

Деструктор — функция, которая автоматически вызывается при уничтожении объекта (выходе его из области видимости), и которая имеет имя `~ClassName`, где `ClassName` — имя класса. Эта функция не имеет аргументов и не возвращает значений. Деструктор, в отличие от конструктора, может быть виртуальным (см. стр. 84), и он всегда уникален для данного класса. Обычное назначение деструктора — освобождение захваченных ресурсов.

Правило Большой Тройки: если классу требуется пользовательский копирующий конструктор, оператор копирующего присваивания или деструктор, то, скорее всего, они ему нужны все одновременно.

В качестве примера использования конструкторов и деструкторов рассмотрим следующий исходный текст.

```
class Test
{
public:
   Test(int value = 0) { value_ = value;
       cout << "Test(" << value << ")" << endl; }
   Test(const Test& T) { value_ = T.value_;
       cout << "Test(Test&(" << value_ << "))" << endl; }
   ~Test() { cout << "~Test("<< value_ <<")" << endl; }
   void inc() {++value_; }
```

```
private:
    int value_;
};

int main()
{
    Test t(5);
    Test q(t);
    t.inc();
}
```

Вывод таков.

```
Test(5)
Test(Test&(5))
~Test(5)
~Test(6)
```

Конструктор локального объекта вызывается каждый раз при выполнении инструкции, содержащей объявление этой локальной переменной, а деструктор — при выходе из блока, содержащего это объявление. Деструкторы объектов всегда вызываются в порядке, обратном порядку вызова конструкторов.

Конструктор статического локального объекта вызывается один раз, при первом выполнении инструкции, содержащей определение объекта. Деструкторы локальных статических объектов вызываются в порядке, обратном вызову их конструкторов, при завершении выполнения программы. Точный момент вызова деструкторов локальных статических объектов не определен.

Нелокальная переменная, определенная вне любой функции, т.е. глобальная переменная, объявленная в пространстве имен или статически в классе, конструируется до вызова `main()`. Деструкторы таких переменных вызываются после выхода из функции `main()`. При использовании динамически компонуемых библиотек инициализация глобальных объектов в них будет выполняться при компоновке их кода в исполнимую программу.

Конструкторы нелокальных переменных в единице трансляции выполняются в порядке их определения; порядок объявления на порядок вызова конструкторов не влияет. Деструкторы нелокальных объектов вызываются в порядке, обратном вызову их конструкторов.

Для нелокальных объектов из разных единиц трансляции нет никаких гарантий по поводу порядка их конструирования и уничтожения.

Управление доступом

В объявлении класса могут использоваться модификаторы **public**, **private** и **protected** (по умолчанию все члены класса считаются объявленными как **private**). Эти модификаторы определяют, кто именно имеет право доступа к данному члену класса. Члены класса, объявленные как **public**, открыты для доступа любому коду — как в функциях-членах, так и вне их. Закрытые (**private**) члены класса доступны только для функций-членов класса, а защищенные (**protected**) — только для членов класса и членов классов, производных от данного. Строгое ограничение прав доступа к членам класса обеспечивает защиту от непреднамеренных ошибок (само собой, от злонамеренного доступа к закрытым членам класса этот механизм не спасает).

Данные-члены класса всегда следует делать закрытыми (**private**). Для широко используемых классов можно воспользоваться идиомой указателя на реализацию (Pimpl), когда для хранения закрытых членов (как переменных, так и функций) используется непрозрачный указатель (указатель на объявленный, но не определенный класс), определение и реализация которого находится в отдельном модуле.

Кроме того, в классе возможно объявление некоторой внешней по отношению к классу функции (или функций) с модификатором **friend** — такие функции имеют доступ ко всем членам класса, включая **private**:

```
class Test
{
    friend void mul(Test&t,int factor);
public:
...
};

void mul(Test&t,int factor) { t.value_ *= factor; };
```

Объявление функции-друга может размещаться как в закрытой, так и в открытой части объявления класса — значения это не имеет.

Другом класса может быть функция-член другого класса, например

```
class Test {
    friend void SomeClass::someFunc();
```

Если возникает ситуация, когда все функции одного класса являются друзьями другого класса, можно воспользоваться более короткой формой записи

```
class Test {
    friend class SomeClass;
```

Класс-друг должен быть объявлен в охватывающей области видимости или определен в области видимости, непосредственно охватывающей класс, объявивший его другом. Функцию-друга можно объявить так же, как и класс-друг, либо ее поиск осуществляется по ее аргументам, как будто она была объявлена вне класса в области видимости, непосредственно охватывающей класс, т.е. она должна быть либо явно объявлена в охватывающей области видимости, либо иметь аргументы этого класса.

Статические члены

Члены класса, объявленные как **static**, являются статическими, и являются частью класса, т.е. существует ровно одна копия статического члена, в то время как для обычных членов каждый объект класса имеет собственные, независимые члены. Статическими могут быть как члены-данные, так и функции-члены.

Обращение к статическим членам может выполняться как посредством объектов с использованием стандартного синтаксиса доступа к членам структуры, как к любым другим членам, так и без использования объектов, путем указания имени соответствующего класса и оператора **::**, например:

```
class Test {
...
    static int  count() { return Count; }
private:
    static int Count;     // Объявление статического
члена
...
};
int Test::Count = 0;      // Определение статического
члена
Test t;
cout << Test::count() << " " << t.count() << endl;
```

Статические члены-данные должны быть определены отдельно, вне класса, причем это определение должно подчиняться правилу одного определения, т.е. такое определение в программе должно быть единственным.

Указатели на статические члены ничем не отличаются от обычных указателей на функции или данные (см. стр. 64).

Массивы

При наличии у класса конструктора по умолчанию (или его генерации компилятором) разрешено определение массива элементов такого класса, например,

```
X x[10];
```

В результате будет создан массив из 10 объектов типа **X**, причем каждый будет проинициализирован при помощи вызова конструктора **X::X()**. Деструктор для каждого элемента вызывается при уничтожении массива. Для массивов, память для которых не выделяется оператором **new**, это выполняется неявно; массивы, созданные при помощи оператора **new**, удаляются оператором **delete**. Поскольку в C++ не различаются указатели на отдельный объект и на первый элемент массива, программист должен сам указать, удаляется ли отдельный объект или массив, например:

```
X* x1 = new X;
X* x2 = new X[10];
X* x3 = new X;
X* x4 = new X[10];
...
delete x1;          // Правильно
delete[] x2;        // Правильно
delete[] x3;        // Неправильно
delete x4;          // Неправильно
```

Чтобы избежать этих сложностей, лучше использовать вместо массивов **vector**:

```
vector<X>* x1 = new vector<X>(10);
X* x2 = new X;
...
delete x1;
delete x2;
```

Размещение объектов в памяти

По умолчанию оператор **new** создает объекты в свободной памяти. Однако в заголовочном файле **<new>** определен также оператор **new** с дополнительным параметром **void***, а именно —

```
void* operator new(size_t, void*where) { return where; }
```

Этот оператор позволяет размещать объекты в конкретном месте в памяти. Так, в следующем примере объект типа **X** размещается в буфере **buf**:

```
char buf[sizeof(X)];
X* x = new(buf)X;
```

Такой оператор **new** называется *размещающим* оператором **new**. При его использовании освобождение памяти объекта не должно выполняться при помощи стандартного оператора **delete**. Вместо этого для уничтожения такого объекта требуется явный вызов деструктора (**x->~X()**). Это достаточно опасная практика, так что без особой необходимости прибегать к этой методике не следует.

Можно определить свой собственный оператор **new** с дополнительным параметром, например, для размещения в различных областях памяти, указываемых данным параметром. В любом случае первым параметром оператора **new** является параметр типа **size_t**, указывающий объем требующейся памяти. Допустим, у нас есть класс-менеджер памяти **Mem**, с функциями **alloc** и **free** для выделения и удаления памяти соответственно. Тогда мы можем определить оператор **new**, работающий с помощью данного менеджера:

```
class Mem {
...
    virtual void * alloc(size_t);
    virtual void free(void*);
...
};
operator new(size_t sz, Mem&m)
{
    return m.alloc(sz);
}
...
Mem Storage;                // Менеджер памяти
X* x = new(Storage)X(p);    // Размещение в памяти,
                            // выделяемой Storage
```

Удаление такого объекта, как уже упоминалось, должно выполняться посредством явного вызова деструктора, например, при помощи такой функции:

```
void destroy(X*p, Mem&m)
{
    p->~X();
    m.free(p);
}
```

Все, сказанное для одного объекта, применимо также и для массивов объектов, с той разницей, что для них используется оператор **new[]**.

При перегрузке оператора **new** следует предоставлять и соответствующий оператор **delete**.

Перегрузка операторов

C++ поддерживает набор операторов для встроенных типов, однако большинство концепций, для которых обычно используются операторы, не являются встроенными типами C++ и представимы пользовательскими типами. C++ позволяет определять операторы для пользовательских классов, тем самым реализуя более привычную и удобную форму записи для работы с объектами, чем использование функций.

Программист может объявить функции, определяющие смысл следующих операторов:

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

Программистом не могут быть определены операторы разрешения области видимости (`::`), выбор члена (`.`), выбор члена через указатель на член (`.*`) и тернарный оператор `?:`. Невозможно также определить новую лексему оператора. Если существующего набора операторов недостаточно, то всегда можно воспользоваться функцией.

Имя операторной функции начинается с ключевого слова **operator**, за которым идет сам оператор, например, **operator[]**. Операторная функция определяется и может быть вызвана, как и любая другая функция. Использование оператора вместо вызова функции — всего лишь сокращенная форма записи. Пусть, например, определен **operator+(complex, complex)**. Тогда можно использовать две формы записи одного и того же выражения:

```
complex a, b;
complex c = a + b;              // Сокращенная форма
complex d = operator+(a,b);     // Явный вызов
```

Бинарный оператор можно определить либо в виде нестатической функции-члена с одним аргументом, либо в виде функции-нечлена с

двумя аргументами. Для любого бинарного оператора `@` выражение `a@b` интерпретируется либо как `a.operator@(b)`, либо как `operator@(a,b)`. Если определены обе функции, то для выяснения того, какая именно будет вызвана, используется механизм разрешения перегрузки (см. раздел «Перегрузка функций» на стр. 51).

Унарный оператор (как префиксный, так и постфиксный) можно определить либо в виде нестатической функции-члена без аргументов, либо в виде функции-нечлена с одним аргументом. Для любого унарного оператора `@` выражение `@a` интерпретируется либо как `a.operator@()`, либо как `operator@(a)`. Если определены обе функции, то для выяснения того, какая именно будет вызвана, используется механизм разрешения перегрузки (см. раздел «Перегрузка функций» на стр. 51). Для постфиксного унарного оператора `@` выражение `a@` интерпретируется либо как `a.operator@(int)`, либо как `operator@(a,int)`. Если определены обе функции, то для выяснения того, какая именно будет вызвана, используется механизм разрешения перегрузки.

Любой оператор может быть объявлен только с синтаксисом, существующим для него в грамматике языка; например, нельзя объявить оператор `%` как унарный или `+` с тремя аргументами.

Общее правило заключается в том, что при переопределении операторов следует сохранять их естественную семантику и максимальную согласованность. В частности, рекомендуется реализовывать постфиксный оператор при помощи префиксного, например:[2]

```
T T::operator++(int)
{
    T old(*this);
    ++*this;
    return old;
}
```

Точно так же, если у вас переопределен некий бинарный оператор `@`, желательно, чтобы был переопределен и оператор `@=`, причем обычно оператор `@` реализуется при помощи оператора `@=`:

```
T operator@(const T& lhs, const T& rhs )
{
    T temp( lhs );
    return temp @= rhs;
}
```

[2] Ряд компиляторов используют перегруженный префиксный оператор инкремента (декремента) вместо постфиксного, если последний отсутствует.

Что касается оператора присваивания, то его следует разрабатывать таким образом, чтобы для корректной работы он не должен был полагаться на проверку присваивания самому себе (соответствующий способ приведен на стр. 67).

Следует также по возможности использовать меньшее количество операторов, являющихся членами класса. Помимо прочего, область применения таких операторов оказывается *уже*, чем операторов, не являющихся членом класса. Например, пусть у нас есть класс строки **str** с оператором конкатенации строк **+**:

```
class str {
    public:
    str(const char *);
    str operator+(const str&);
...

str a("Hello");
str s = a + "World"; // a.operator+(str("World"))
str s = "World" + a; // Ошибка
```

Дело в том, что если оператор является членом класса, то первым операндом должен быть объект этого класса. Если же оператор не является членом класса, т.е. объявлен вне класса как

```
str operator+(const str&, const str&);,
```

то инструкция

```
str s = "World" + a;
```

трактуется как

```
s = operator+(str("World"),a);
```

и не вызывает никаких проблем.

С этим вопросом тесно связан вопрос поиска имен операторов, который осуществляется по следующим правилам. Если существует выражение **x@y**, где **x** имеет тип **X**, а **y** имеет тип **Y**, то, если **X** является классом, проверяется, определяет ли этот класс (или его базовый класс) **operator@** в качестве члена. Затем выполняется поиск объявления **@** в контексте окружения **x@y**. Затем, если **X** определен в пространстве имен **N**, выполняется поиск объявления **@** в **N**, и, если **Y** определен в пространстве имен **M**, выполняется поиск объявления **@** в **M**. Если найдено несколько подходящих операторов, используется механизм разрешения перегрузки.

Стандарт требует, чтобы операторы **=**, **[]**, **()** и **->** были членами, а операторы **new**, **new[]**, **delete**, **delete[]** — статическими членами. Для всех остальных функций можно применить следующее правило.

Если функция-оператор представляет собой **operator>>** или **operator<<** для потокового ввода-вывода, или если она требует преобразования типа левого аргумента, или если она может быть реализована с использованием только открытого интерфейса класса — ее следует сделать нечленом класса (в первых двух случаях — другом класса). Если требуется виртуальное поведение данной функции, то добавьте виртуальную функцию-член и реализуйте ее посредством соответствующей функции-нечлена; в противном случае сделайте ее членом класса.

Операторы преобразования типов

Еще один вид операторов — операторы преобразования типа, которые используются для преобразования объекта класса в объект некоторого другого типа. Такой оператор имеет вид **operator type()**, где **type** — тип, в который происходит преобразование. Именно этот оператор вызывается при явных и неявных преобразованиях типов. Рассматривая использованный выше класс **Test**, мы можем определить оператор преобразования в **int**.

```
class Test
{
...
    operator int() const { return value_; }
...

cout << 2 + t << endl;          // Неявное преобразование
                                // типа
cout << static_cast<int>(t)     // Явное преобразование
     << endl;                   // типа
```

Еще одно преобразование типов связано с наличием у типа, в который выполняется преобразование, конструктора с одним аргументом типа, из которого выполняется преобразование. При преобразовании типа **const char*** в тип **str** данную роль играл конструктор **str(const char*)**. Там, где такое поведение нежелательно, конструктор должен быть объявлен с использованием ключевого слова **explicit** — в этом случае конструктор может быть вызван только явно, но не для приведения типов.

Неявное преобразование типов — источник многих недоразумений и ошибок, и его следует избегать. Описывайте конструкторы с использованием ключевого слова **explicit**, а вместо операторов преобразования типов используйте именованные функции.

Наследование

Одной из важнейших концепций, связанных с классами C++, является наследование. Наследование означает, что один (производный) класс может быть наследником другого (базового) класса. При этом производный класс включает в себя все члены базового класса. Права доступа к членам базового класса определяются при объявлении производного класса. Указание базового класса в производном выполняется при его объявлении следующим образом:

```
class Derive: public Base {
...
```

Здесь **public Base** указывает, что **Base** является открытым базовым классом для **Derive**. Кроме ключевого слова **public** могут использоваться ключевые слова **protected** и **private**; об их применении и значении см. стр. 85.

При таком объявлении производного класса его объекты *являются* одновременно объектами базового класса, т.е. могут использоваться везде, где используется объект базового класса; обратное неверно. С точки зрения правил языка это означает, что при обращении к объекту производного класса посредством указателей и ссылок с ним можно обращаться так же, как и с объектом базового класса. Отношение *является* — ключевое при определении того, следует ли делать класс базовым для другого или лучше включить его в качестве члена (отношение *содержит*). Производный класс, в свою очередь, может выступать базовым классом для других классов. Отношение *является* транзитивно, т.е. все объекты производных классов, включая объекты классов, производных от производных, *являются* объектами базового класса.

Член производного класса может пользоваться открытыми и защищенными членами базового класса так, как если бы они были объявлены в самом производном классе; закрытые члены базового класса производный класс использовать не может (в противном случае сама концепция закрытых членов теряет смысл). Члены производного класса могут иметь те же имена, что и в базовом классе; в этом случае для явного указания, к какому именно члену выполняется обращение, используется оператор разрешения области видимости **::** с именем базового класса, например:

```
class Base {
    public:
    int a;
    void f();
};
```

```
class Derive: public Base {
    public:
    int a;
    void f();
    void inc() {
        ++a;            // Инкремент a производного
                        // класса
        ++Base::a;      // Инкремент a базового класса
    }
    void g() {
        f();            // Вызов f() производного класса
        Base::f();      // Вызов f() базового класса
    }
};
```

Не используйте наследование там, где можно обойтись включением.

Конструкторы и деструкторы

При создании объекта производного класса сначала создается базовый класс (вызывается его конструктор), затем члены производного класса, а затем — сам производный класс. Конструкторы классов вызываются в указанном порядке. Члены и подобъекты базовых классов конструируются в порядке их объявления в классе. Уничтожение объектов происходит в порядке, обратном созданию. *Порядок вызова деструкторов всегда обратен порядку вызова конструкторов.*

Конструктор базового класса (если таковой имеется) должен быть указан в списке инициализации производного класса (см. стр. 67) с соответствующими аргументами (если таковые имеются). В этом отношении конструктор базового класса ведет себя так же, как и конструкторы членов. Конструктор производного класса может указать инициализатор только для базового класса в целом, и не может указывать инициализаторы для членов базового класса.

```
class Base {
public:
    Base(int v):value_(v) {};
```

```
private:
    int value_;
};

class Derive: public Base {
public:
    Derive(int v)
      : Base(v)       // Вызов конструктора базового
                      // класса
      , value_(v)     // Инициализация члена базового
    {}                // класса невозможна
};
```

Копирующий конструктор и оператор присваивания определяет процесс копирования объектов класса.

```
class Base {
    ...
    Base& operator=(const Base&);
    Base(const Base&);
};

void f(const Derived&d)
{
    Base b = d; // Создание b из Base-части d
    b = d;      // Присваивание переменной b Base-части d
}
```

Поскольку копирующий конструктор **Base** и оператор присваивания **Base** ничего «не знают» о производном классе **Derived**, копируется только **Base**-часть объекта **Derived**. Этот эффект называется *срезкой*; к сожалению, он чреват ошибками и неприятными сюрпризами. Кстати говоря, именно эффект срезки зачастую является причиной того, что в функции передаются указатели и ссылки на объекты классов.

Заметим, что, если оператор копирующего присваивания не определен, он будет сгенерирован компилятором (т.е. операторы присваивания не наследуются).

Виртуальные функции

Поскольку наследование позволяет использовать вместо объекта базового класса объект производного класса, встает вопрос о том, как обеспечить необходимую функциональность, например, в такой ситуации:

```
void f(Base&b)
{
    b.out();
}
```

Здесь в функции `f()` требуется выполнить вызов функции-члена `out()`. В такой ситуации, объект какого бы производного класса не был передан в функцию `f()`, для него будет вызвана функция-член `Base::out()`, несмотря на то, что для производного класса она, возможно, переопределена (перекрыта). По сути, проблема состоит в том, чтобы определить, какого типа объект на самом деле передан функции.

Данная проблема решается при помощи виртуальных функций, объявленных в классе при помощи ключевого слова `virtual`. По сути, это ключевое слово означает, что вызов функции с этим именем через указатель или ссылку на объект данного класса может привести к вызову функции производного класса, если указатель или ссылка указывают на объект производного класса.

Для корректной работы механизма виртуальных функций требуется, чтобы типы аргументов функции с данным именем в производном классе были в точности те же, что и в базовом классе. (Единственное допускаемое отклонение — если в базовом классе виртуальная функция возвращает указатель или ссылку на базовый класс, то в производном классе данная функция может возвращать указатель или ссылку на производный класс. В остальном объявления функций должны быть идентичны.)

Никогда не изменяйте аргумент по умолчанию при перекрытии виртуальных функций. Он не является частью сигнатуры функции, и клиентский код будет невольно передавать различные аргументы в функцию, в зависимости от того, какой узел иерархии обращается к ней. Рассмотрим следующий пример.

```
class Base {
    virtual void Foo(int x = 0);
};
class Derived : public Base {
    virtual void Foo(int x = 1); // Разрешено,
};                               // но плохо...
Derived *pD = new Derived;
pD->Foo();                       // Вызов pD->Foo(1)
Base *pB = pD;
pB->Foo();                       // Вызов pB->Foo(0)
```

Не забывайте о том, что перекрытие может скрывать перегруженные функции из базового класса, например:

```
class Base{                        // ...
    virtual void Foo( int );
    virtual void Foo( int, int );
    void Foo( int, int, int );
};
class Derived : public Base { // ...
    virtual void Foo( int ); // Перекрывает
                             // Base::Foo(int),
                             // скрывая остальные
                             // функции Foo
};
Derived d;
d.Foo( 1 );          // Все в порядке
d.Foo( 1, 2 );       // Ошибка
d.Foo( 1, 2, 3 );    // Ошибка
```

Этот эффект связан с правилом поиска имен, согласно которому, если искомое имя найдено в некотором пространстве имен (в данном случае — текущего класса), то поиск прекращается, так что имена из базового класса не просматриваются.

Виртуальная функция обязана быть определена для класса, в котором она впервые объявлена (если только это не чисто виртуальная функция — см. стр. 84). Виртуальная функция может использоваться как обычная функция, даже если у класса нет производных классов. Производный класс, который не нуждается в собственной версии виртуальной функции, не обязан ее реализовывать — в этом случае будет использоваться функция базового класса. При объявлении функции в производном классе наличие ключевого слова **virtual** не обязательно — достаточно объявить функцию как виртуальную при первом объявлении в базовом классе.

Однако для удобочитаемости и последовательности рекомендуется указывать ключевое слово **virtual** при объявлении виртуальных функций во всех классах.

Для вызова виртуальной функции из функции производного класса, как и для обычной функции, используется вызов с указанием имени базового класса и оператора разрешения области видимости (**::**).

В качестве примера использования виртуальных функций приведем следующий код:

```
class Num {
public:
    virtual void out(ostream&o) { o << "Number"; }
};
class Double : public Num {
    double d;
public:
    Double(double g):d(g) {}
    // Слово virtual использовать не обязательно:
    void out(ostream&o) { o << d; }
};

class Complex : public Double {
    double im;
public:
    Complex(double r, double i = 0):Double(r),im(i) {}
    virtual void out(ostream&o) {
        o << "(";
        Double::out(o);     // Вызов функции базового
                            // класса
        o << "," << im << ")";
    }
};

void print(Num&n)
{
    n.out(cout);
    cout << endl;
}

int main()
{
    Complex c(2.0,3.0);
    Double  d(5.0);
    print(c);
    print(d);
}
```

Функция **print** корректно выводит в поток **cout** все виды чисел, используя для этого вызов соответствующей виртуальной функции **out()**. Обратите внимание на то, что функция **print()** без внесения каких-либо изменений будет корректно работать даже для тех унаследованных от **Num**

классов, которых пока что не существует в природе — лишь бы они предоставляли собственную версию виртуальной функции `out()`.

При рассмотрении приведенного примера становится понятно, что вряд ли мы будем когда-либо использовать объекты класса `Num` сами по себе, так как они не несут никакой информации, и цель этого класса — обеспечить наличие виртуальной функции `out()`. Более того, желательно, чтобы программист просто не мог создать объект типа `Num`. Этого можно достичь, сделав `Num` *абстрактным* классом, т.е. классом, выражающим некоторую абстрактную концепцию, которая должна реализовываться в *конкретных* классах. Для этого следует сделать одну или несколько виртуальных функций класса *чисто виртуальными*, что достигается при помощи инициализатора `= 0`:

```
class Num {
public:
    virtual void out(ostream&o) = 0;
};

Num n;      // Ошибка: объект абстрактного класса
```

Абстрактный класс можно использовать только как интерфейс и в качестве базы для других классов — как абстрактных, так и конкретных. Если чисто виртуальная функция не определена в производном классе, то она остается чисто виртуальной, а класс — абстрактным.

Чисто виртуальная функция может быть не только объявлена, но и определена, и использоваться в производных классах, как и обычные виртуальные функции базовых классов; класс при этом остается абстрактным, и объект такого класса создан быть не может.

```
class Num {
public:
    virtual void out(ostream&o) = 0
    { o << "Number"; }
};
```

Имеется еще одно ограничение на виртуальные функции — конструктор не может быть виртуальным. Это требование становится совершенно очевидным, если вспомнить, что до завершения работы конструктора объект не существует.

Деструктор может быть виртуальным; более того — он должен быть именно таковым, если возможно удаление объекта посредством указателя на базовый класс:

```
void DoAndDestroy(Base*b)
{
    b->doSomething();
```

```
    delete b;
}
```

Деструкторы базовых классов рекомендуется делать либо виртуальными и открытыми (**public**), либо невиртуальными и защищенными (**protected**).

Если в качестве параметра в данную функцию передается указатель на объект класса **Base**, при его удалении вызывается деструктор **~Base()**. Но если в функцию передается указатель на производный класс **Derived**, то при удалении при помощи инструкции **delete b** происходит следующее. Если деструктор **Base** не виртуальный, то будет вызван именно он, что, очевидно, неверно. Однако если деструктор **Base** – виртуальный, то будет вызван корректный деструктор **~Derived()**.

Права доступа

Как уже говорилось, при наследовании базовый класс может быть указан как **public**, **protected** или **private**. Рассмотрим, в чем состоит отличие этих объявлений. Пусть имеется следующий набор классов:

```
class X { /*...*/ };

class Y1: public    X {};
class Y2: protected X {};
class Y3: private   X {};
```

Поскольку **X** является открытым базовым классом для **Y1**, любая функция может неявно преобразовывать **Y1*** в **X*** там, где это нужно, и обращаться к открытым членам класса **X**.

Так как **X** – защищенный базовый класс для **Y2**, преобразовывать **Y2*** в **X*** и обращаться к открытым и защищенным членам класса **X** могут только члены и друзья **Y2**, а также члены и друзья производных от **Y2** классов.

В случае закрытого наследования **Y3** от **X** преобразовывать **Y3*** в **X*** и обращаться к открытым и защищенным членам класса **X** могут только члены и друзья **Y3**.

Множественное наследование

C++ допускает наследование не только от одного, но и от нескольких базовых классов. В этом случае объект производного класса будет содержать в себе подобъекты каждого из базовых классов:

```
class A {};
class B {};
class C: public A, public B {};
```

Одна из проблем, возникающая при множественном наследовании, — возможность ситуации, когда объект производного класса будет содержать несколько объектов базового класса, например:

```
class A {};
class B: public A {};
class C: public A {};
class D: public B, public C {};
```

В этом случае объект класса `D` содержит два подобъекта класса `A`. Если это не то, что вам надо, — следует использовать механизм виртуального наследования. Добавление ключевого слова `virtual` при наследовании гарантирует наличие не более одного подобъекта базового класса у всех производных классов, т.е. в нашем случае проблема решается следующим образом:

```
class A {};
class B: virtual public A {};
class C: virtual public A {};
class D: public B, public C {};
```

Конструктор базового класса, наследуемого виртуально, вызывается первым, после чего вызываются конструкторы базовых классов. Деструкторы, как обычно, вызываются в порядке, обратном порядку вызова конструкторов.

Глава 7. Исключения

Обычная методика извещения об ошибке в C — это возврат соответствующего значения функцией или присваивание некоторой переменной-флагу значения, свидетельствующего о происшедшей ошибке. Такое значение может включать информацию о том, какая именно ошибка произошла.

В C++ для обработки ошибок имеется механизм исключений. Код, в котором может произойти ошибка, заключается в блок **try{}**, за которым следует один (или несколько) блоков перехвата и обработки сгенерированного исключения **catch{}**. О происшедшей ошибке код может сообщить при помощи инструкции генерации исключения **throw**. Данная инструкция позволяет сгенерировать объект исключения, который может содержать полную информацию об ошибке. Этот объект передается в соответствующий его типу **catch**-блок, где и происходит обработка исключения. Например:

```
class ZeroDiv {};
void div(int i, int j, int&r)
{
    try {
        if (j==0) throw(ZeroDiv());
        r = i/j;
    } catch (ZeroDiv& x)
    {
        cout << "Деление на 0";
    }
}
```

Если код в **try**-блоке не генерирует исключения (в приведенном фрагменте **j!=0**), то код в **catch**-блоках не выполняется. Если же исключение генерируется, то управление передается **catch**-блоку с параметром соответствующего типа. При этом дальнейшее выполнение кода **try**-блока прекращается, и происходит уничтожение всех объектов, область видимости которых была ограничена данным блоком, т.е., в частности, вызываются деструкторы всех локальных объектов блока (так называемая *свертка* стека).

Обработчиков исключений может быть несколько. При генерации исключения просматриваются последовательно все имеющиеся обработчики на соответствие типу сгенерированного исключения (с учетом того, что наследование представляет собой отношение «является», блок **catch(Base&)** перехватит исключение **Derived**, если **Derived** — производный класс от **Base**). Если перехват исключения выполняется не по

ссылке, а по значению, то, как и при вызове функции, сгенерированное исключение «срезается» до перехваченного.

При необходимости отдельного перехвата исключений **Derived** и **Base** следует первым указывать блок **catch** для производного класса, иначе исключение **Derived** будет перехвачено обработчиком исключения **Base**:

```
try {
    throw Derived();
} catch (Derived&) {
    // Перехват будет выполнен здесь
} catch (Base&) {
}

try {
    throw Derived();
} catch (Base&) {
    // Перехват будет выполнен здесь,
    // так как Derived является Base
} catch (Derived&) {
}
```

Рекомендуется генерировать исключения по значению (не указатели), а перехватывать их как ссылки (обычно константные). Не рекомендуется позволять исключениям пересекать границы модулей. Для исключений рекомендуется использовать типы, определяемые пользователем, а не встроенные.

Как только находится обработчик сгенерированного исключения, дальнейший просмотр **catch**-блоков прекращается. В качестве типа перехватываемого исключения может быть указано троеточие, которое означает, что данный **catch**-блок перехватывает исключение любого типа. Таким образом, важное значение приобретает последовательность **catch**-блоков — сначала должны идти наиболее «специализированные» **catch**-блоки.

Если исключение оказывается неперехваченным, оно передается на обработку **catch**-блокам охватывающего **try**-блока. Если в конце концов обработчик данного исключения так и не будет найден, вызывается функция **terminate()**, которая по умолчанию завершает выполнение программы. Заменить эту функцию другой можно при помощи функции **set_terminate()**.

Исключение считается обработанным в момент его поступления в обработчик, что позволяет генерировать исключения в обработчике. Такое исключение будет обработано охватывающим `try`-блоком.

Возможна также повторная генерация обрабатываемого исключения при помощи оператора `throw` без параметров. Таким образом обработчик, который не в состоянии справиться с возникшим исключением, но перехвативший его, может передать его дальше, во внешний блок.

Поскольку при генерации исключения происходит вызов всех деструкторов, сами деструкторы не должны генерировать никаких исключений. Тому имеется много причин, но достаточно и того, что генерация исключения в момент, когда остается не перехваченным другое исключение, приводит к вызову функции `terminate()` и завершению работы программы.

При генерации исключения в конструкторе объект остается не созданным, и деструктор для него не вызывается. Поэтому конструктор должен сам позаботиться, например, об освобождении выделенных в процессе конструирования ресурсов.

Исключение — единственный способ указания на ошибку в конструкторе.

Со сверткой стека тесно связана идиома «выделение ресурса есть инициализация», при которой ресурс выделяется в конструкторе объекта, а освобождается в деструкторе. Это существенно еще и потому, что область видимости `try`-блока не распространяется на `catch`-блоки. Представьте ситуацию, когда в `try`-блоке открыт файл, после чего сгенерировано исключение. Следует закрыть файл в каждом из блоков, причем переменная, хранящая файл, должна располагаться до `try`-блока. В то же время, создав объект, который открывает файл в конструкторе, а закрывает в деструкторе, нам не надо заботиться о закрытии файла при исключении — оно выполнится автоматически. Это очень полезная и широко распространенная идиома, рекомендуемая к использованию в ваших программах.[3]

[3] В англоязычной литературе данная идиома носит название «resource acquisition is initialization» — RAII.

Спецификации исключений

При объявлении функции можно явно указать, какие исключения она может генерировать, добавив спецификацию исключений, представляющую собой ключевое слово **throw**, после которого в скобках перечисляются типы допустимых исключений, например

```
void f() throw(x1, x2);
```

Такое объявление означает, что при выполнении функции могут быть сгенерированы исключения **x1** и **x2** и никакие другие. В случае если будет сгенерировано иное исключение, отличное от **x1** или **x2**, будет вызвана функция **unexpected()**, которая по умолчанию вызывает функцию **terminate()**, завершающую работу программы (указать функцию, выполняющуюся вместо **unexpected()**, можно при помощи функции **set_unexpected()**). Однако, поскольку спецификации исключений не гарантируют невозможность генерации неперечисленных исключений, практическая польза от них мала, более того, в связи с необходимостью отслеживать реальные типы сгенерированных исключений они могут отрицательно влиять на производительность программы. Таким образом, спецификаций исключений лучше избегать (см., например, [8]).

В заключение следует сказать, что исключения могут использоваться не только для обработки ошибок, но и при штатной работе программы, например, для того же выхода из вложенных циклов вместо оператора **goto**. Еще одно замечание касается стиля программирования — не следует смешивать в одной программе разные способы обработки ошибок; как правило, это плохое решение, говорящее о неудачном проектировании.

Стандартные исключения

Библиотечные исключения являются частью иерархии классов, производной от класса **exception**. Стандартные исключения приведены в следующей таблице.

Исключение	Чем генерируется	Заголовочный файл
bad_alloc	new	<new>
bad_cast	dynamic_cast	<typeinfo>
bad_typeid	typeid	<typeinfo>
bad_exception	спецификации исключений	<exception>
out_of_range	at(), bitset::operator[]	<stdexcept>

Исключение	Чем генерируется	Заголовочный файл
`invalid_argument`	конструктор `bitset`	`<stdexcept>`
`overflow_error`	`bitset<>::to_ulong()`	`<stdexcept>`
`ios_base::failure`	`ios_base::clear()`	`<ios>`

Глава 8. Шаблоны

Зачастую код работы с объектами разных типов выглядит, по сути, одинаково — например, при реализации алгоритма сортировки или обработки связанных списков код практически не зависит от типа используемых объектов. Таким образом, программисту приходится либо реализовывать один и тот же алгоритм для разного типа данных, либо писать код для обобщенного базового типа (наподобие `void*` или некоторого базового класса `Object`), либо пользоваться препроцессором. Недостатки всех описанных способов решения проблемы очевидны. Шаблоны обеспечивают решение данной проблемы, лишенное недостатков, присущих описанным способам. Шаблон представляет собой функцию или класс, реализованные для одного или нескольких типов данных, которые в момент написания кода неизвестны (известны только предъявляемые к ним требования). При использовании шаблона ему явно или неявно передаются конкретные типы данных. Поскольку шаблоны являются средством языка C++, для них обеспечивается полная поддержка проверки типов и областей видимости.

При определении шаблона осуществляется проверка отсутствия некоторых синтаксических и, возможно, других ошибок, которые можно обнаружить вне зависимости от конкретного набора аргументов шаблона, например, использование неизвестных идентификаторов, несоответствие скобок, опущенные точки с запятой и т.п. Полная проверка кода выполняется только при его *инстанцировании*, т.е. генерации кода из определенного шаблона путем подстановки конкретных параметров. До тех пор, пока в исходном тексте программы нет ссылки на определенную специализацию шаблона, не выполняется и его инстанцирование, а следовательно, и полная проверка его корректности.

Шаблоны функций

Шаблон функции описывает, каким образом компилятор должен сгенерировать функцию по соответствующему набору аргументов шаблона. Это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов; другими словами, шаблон функции представляет целое семейство функций. Шаблон функции имеет следующий синтаксис — объявление шаблона при помощи ключевого слова `template`, за которым в угловых скобках идет перечисление формальных параметров шаблона при помощи ключевого слова `class` или `typename`,

после чего идет описание функции с использованием перечисленных параметров, например:

```
template<class T, class U>
bool less(T i, U j) {
    return i < j;
}
```

Следует отличать параметры шаблона, которые объявляются в угловых скобках перед именем шаблона функции, и параметры вызова, которые объявляются в круглых скобках после имени шаблона функции. Количество задаваемых параметров неограниченно, но, в отличие от шаблонов классов, в шаблонах функций нельзя использовать аргументы шаблона по умолчанию. При вызове функции компилятор выводит фактические параметры шаблона и генерирует соответствующую функцию, например

```
bool b = less(1, 2.0);
```

(В приведенном примере `1` имеет тип `int`, `2.0` — тип `double`, так что при инстанцировании тип `T` соответствует `int`, тип `U` — `double`.)

Не специализируйте шаблоны функций.

Специализация шаблонов функций возможна, но обычно не имеет смысла, поскольку вместо нее проще и надежней воспользоваться перегрузкой функций, тем более что обычной нешаблонной функции отдается предпочтение при выборе вызываемой функции. Шаблон функции не может быть частично специализирован (стр. 97), но может быть перегружен. Попытки частичной специализации шаблонов функций приводят к созданию новых первичных шаблонов функций. Вот как производится выбор вызываемой функции при наличии нескольких возможных вариантов.

1. Функции, не являющиеся шаблонами, считаются «привилегированными». При разрешении перегрузки обычная нешаблонная функция с типами параметров, подходящими для аргументов, имеет преимущество перед всеми соответствующими в той же степени шаблонными функциями.
2. Если такой «привилегированной» функции нет, в качестве претендентов рассматриваются первичные шаблоны функций. Какой именно первичный шаблон будет выбран, зависит от того, какой из них наиболее близко соответствует аргументам функции и является «наиболее специализированным» (использование термина «специализированный» в данном контексте не имеет никакого отношения к специализации шаблонов). Очевидно, что если имеется

только один «наиболее специализированный» первичный шаблон функции, то используется именно он. Если первичный шаблон специализирован для используемых типов, то будет применяться именно эта специализация; в противном случае будет выполнено инстанцирование первичного шаблона с корректными типами.

3. В противном случае, если имеется несколько таких «наиболее специализированных» первичных шаблонов функций, то вызов неоднозначен, так как компилятор не в состоянии выбрать наилучший из них. Программист при этом должен сделать выбор сам, и предпринять некоторые уточняющие действия, которые поясняют компилятору, какой выбор должен быть сделан.
4. Если ни один первичный шаблон функции не подходит, вызов считается некорректным, и программист должен исправить данный код.

Специализации шаблона функции не участвуют в разрешении перегрузки. Специализация используется только тогда, когда первичный шаблон функции уже выбран, причем на этот выбор не влияет наличие или отсутствие специализации шаблона.

Шаблоны классов

Так же, как и функции, классы могут быть параметризованы одним или несколькими типами. Типичным примером могут служить контейнеры, в которых содержатся объекты определенного типа (см. главу 9, Стандартная библиотека», стр. 101). Синтаксис шаблонов классов, по сути, такой же, как и шаблонов функций, например:

```
template<class T>
class BiArray {
public:
    BiArray(T t1, T t2);
    T first()  const { return elem[0]; }
    T second() const;
private:
    T elem[2];
};

template<class T>
```

```
BiArray<T>::BiArray(T t1, T t2)
{ elem[0] = t1; elem[1] = t2; }

template<class T>
T BiArray<T>::second() const
{ return elem[1]; }
```

Как видно из приведенного исходного текста, функции-члены могут определяться как в объявлении класса, так и вне его. Как и для шаблонов функций, в угловых скобках могут использоваться как ключевое слово **class**, так и ключевое слово **typename**. Внутри шаблона класса идентификатор **T** может использоваться в объявлениях членов и функций-членов, как и любой другой тип. Сам класс в данном случае имеет тип **BiArray<T>**, и именно его следует использовать в объявлениях. Так, объявление копирующего конструктора будет выглядеть в классе как

```
...
public:
    BiArray(BiArray<T> const&);
...
```

Для определения функции-члена класса вне класса следует указать, что это шаблон функции, и использовать полное имя шаблона класса, как и сделано в приведенном выше исходном тексте.

При использовании шаблона следует явно указывать аргументы шаблона:

```
int main() {
    BiArray<int> B(5,8);
    cout << B.first() << "  " << B.second() << endl;
    return 0;
}
```

Специализации шаблонов класса

Шаблон может быть специализирован для конкретных аргументов шаблона, т.е. позволяет изменить стандартное поведение для конкретных типов. При специализации шаблона класса следует определять все его функции-члены (специализация отдельной функции-члена перекрывает возможность дальнейшей специализации всего класса). Для специализации шаблона класса следует объявить класс, предварив его конструкцией **template<>** (в некоторых старых компиляторах, например, Open Watcom, данная конструкция опускается), и указать типы, для которых выполняется специализация, например:

```
template<>
class BiArray <char> {
public:
    BiArray(char t1, char t2);
    char first() const;
...
```

Для таких специализаций любая функция-член должна определяться как «обычная» функция-член с заменой включений параметров шаблонов специализированными типами.

Частичная специализация представляет собой специализацию, при которой некоторые параметры шаблона остаются задаваемыми пользователем. Например, для шаблона

```
template<class T1, class T2>
class Test {
...
};
```

возможны различные частичные специализации:

```
template<class T > // Оба параметра одного типа
class Test<T,T> {
...
};

template<class T > // Тип второго параметра - int
class Test<T,int> {
...
};

template<class T1, class T2 > // Оба параметра -
                              // указатели
class Test<T1*,T2*> {
...
};
```

Можно сравнивать различные степени специализации. Одна из специализаций более специализирована, чем другая, если каждый список аргументов, соответствующий первой специализации, соответствует и второй, но не наоборот. При прочих равных условиях предпочтение отдается более специализированной версии.

Аргументы по умолчанию

Для параметров шаблона можно определить значения по умолчанию, например:

```
template<class T = int>
class BiArray {
public:
    BiArray(T t1, T t2);
```

и использовать в программе тип **BiArray<>** без указания параметра шаблона. При наличии нескольких параметров шаблона использование значений по умолчанию выглядит следующим образом:

```
template<class T, class U = SomeType>
class Test {
...
};

Test<int,double> t1;
Test<char> t2;   // То же, что Test<char,SomeType> t2;
```

Параметры, не являющиеся типами

Как шаблоны классов, так и шаблоны функций могут иметь параметры, не являющиеся типами, т.е. представляющие собой обычные величины, например:

```
template<class T, int SIZE>
class BiArray {
public:
    BiArray(T t1, T t2);
    T first()  const { return elem[0]; }
    T second() const;
private:
    T elem[SIZE];
};
```

Для таких параметров также можно задавать значения по умолчанию. На параметры шаблонов, не являющиеся типами, налагается ряд ограничений. В общем случае такими параметрами могут быть только целочисленные константы (включая перечисления) или указатели на объекты с внешним связыванием. Использование чисел с плавающей точкой и объектов с типом класса в качестве параметров шаблона не допускается.

Шаблоны классов могут наследовать как обычные классы, так и шаблоны. Если данные или операции в базовом классе зависят от параметров

шаблона базового класса, то и сам базовый класс должен быть параметризован. Использование одного и того же параметра шаблона для базового класса и производного класса является наиболее распространенным случаем, однако это не обязательное требование. Возможна даже передача производного класса в качестве параметра в базовый класс:

```
template <class C> class base {...};

template <class T> class derive : public base<derive<T> >
{...};
```

(Обратите внимание на пробел между символами > в конструкции `base<derive<T> >`. Он связан с тем, что при его отсутствии лексический анализатор будет рассматривать два подряд идущих символа >> как соответствующий оператор, что приведет к ошибке.)

При наследовании следует учитывать, что два класса, сгенерированные из одного шаблона, не связаны друг с другом никакими отношениями, даже если использованы аргументы, один из которых является производным от другого.

Следует также упомянуть, что желательно указывать полностью любое имя, объявленное в базовом классе, который каким-либо образом зависит от параметра шаблона, для чего можно использовать конструкции `this->` или `base<T>`. Это позволит избежать некоторых возможных неоднозначностей.

Члены классов также могут быть шаблонами, однако эти и другие вопросы использования шаблонов выходят за рамки данной книги. Поэтому, если вы хотите познакомиться поближе с шаблонами C++, рекомендуем обратиться к книге [2], в которой все эти вопросы изложены подробно и на весьма высоком уровне.

Глава 9. Стандартная библиотека

С шаблонами тесно связана стандартная библиотека шаблонов (Standard Template Library, STL), которая представляет собой набор профессионально разработанных шаблонов, позволяющих писать лаконичные и эффективные программы на C++.

Имена стандартной библиотеки принадлежат пространству имен `std`, поэтому использующие возможности STL программы должны предварять такие имена префиксом `std::` или применять соответствующие `using`-объявления. В используемых далее фрагментах кода префиксы `std::` для простоты изложения опущены.

Стандартную библиотеку можно условно разделить на три основные части — контейнеры и итераторы, алгоритмы и потоки ввода-вывода. В силу ограниченности объема данной книги здесь будет приведено лишь краткое описание основных (не всех) шаблонов библиотеки, без детального пояснения принципов работы и особенностей реализаций. Кроме того, по той же причине опущены вопросы использования распределителей памяти.

Контейнеры и итераторы

Контейнер представляет собой объект, содержащий другие объекты; примерами контейнеров могут служить векторы, связанные списки, стеки. Каждый вид контейнера обладает собственным представлением в памяти и своими временными характеристиками выполнения тех или иных операций.

Обычное использование большинства контейнеров состоит в итеративном переборе всего содержимого контейнера элемент за элементом. Поскольку при выполнении таких действий способ хранения объектов в контейнере не должен играть роли, для доступа к элементам контейнера используются так называемые итераторы, которые обеспечивают операцию «получение следующего элемента», реализуемую для всех видов контейнеров.

Основные временные характеристики стандартных контейнеров можно представить следующей таблицей.

	`[]`	Операции со списками	Операции с началом	Стековые операции с концом	Итераторы
`vector`	const	O(n)+		const+	R
`list`		const	const	const	B
`deque`	const	O(n)	const	const	R
`stack`				const+	
`queue`			const	const+	
`map`	O(log(n))	O(log(n))+			B
`set`	O(log(n))	O(log(n))+			B

В столбце «Итераторы» R означает итератор с произвольным доступом, а B — двунаправленный итератор (операции для двунаправленных итераторов представляют собой подмножество операций для итераторов с произвольным доступом). Запись `const` означает, что время выполнения данной операции не зависит от количества элементов в данном контейнере, т.е., по сути, эквивалентна записи $O(1)$. Суффикс «+» показывает, что иногда операция может оказаться несколько более продолжительной (т.е. указывает амортизированное время выполнения операции). Подробнее с различными информационными структурами и алгоритмами работы с ними можно познакомиться, например, в [4, 5].

По умолчанию рекомендуется использовать контейнер `vector`; любой другой тип контейнера следует использовать, только если для вашей конкретной задачи этот тип контейнера подходит лучше (например, с точки зрения эффективности).

Поскольку элементы в контейнере представляют собой копии вставленных объектов, чтобы стать элементом контейнера, объект должен быть такого типа, который позволяет копирование. Альтернативным способом может быть хранение в контейнерах указателей вместо реальных объектов, что к тому же сохраняет полиморфное поведение объектов.

Ассоциативные контейнеры требуют, чтобы их элементы могли быть упорядочены (то же самое относится и ко многим операциям с контейнерами, например, `sort()`). По умолчанию для этого используется оператор `<`, но если он не подходит, программист может обеспечить альтернативу. Пользовательский критерий должен определять строгое слабое упорядо-

чение, т.е. для критерия упорядочения `cmp(x,y)` должны выполняться следующие соотношения.

1. `cmp(x,x) == false`.
2. Из `cmp(x,y)` и `cmp(y,z)` следует `cmp(x,z)`.
3. `eq(x,y)` определяется как `!(cmp(x,y)||cmp(y,x))`. Из `eq(x,y)` и `eq(y,z)` следует `eq(x,z)`.

Рекомендуется хранить в контейнерах только значения или интеллектуальные указатели.

Такое упорядочение для типа `T` можно определить как функцию `bool cmp(const T&,const T&)` или как оператор `bool operator()` в некотором классе. Вот пример использования пользовательского упорядочения строк не в лексикографическом порядке, а по длине строк (обратите внимание на разницу в передаче третьего аргумента функции `sort()`):

```
class Clen {
public:
    bool operator()(const string&s, const string&t)
    {
        return s.length() < t.length();
    }
};

bool Vlen (const string&s, const string&t)
{
    return s.length() > t.length();
}

int main()
{
    vector<string> v;
    v.push_back("Z");
    v.push_back("VVVVV");
    v.push_back("YY");
    v.push_back("WWWW");
    v.push_back("XXX");
```

```
    sort(v.begin(),v.end(), Vlen);
    for(vector<string>::const_iterator i = v.begin();
        i != v.end(); i++) cout << *i << endl;

    sort(v.begin(),v.end(), Clen());
    for(vector<string>::const_iterator i = v.begin();
        i != v.end(); i++) cout << *i << endl;
```

Вывод данного фрагмента кода будет следующим:

```
VVVVV
WWWW
XXX
YY
Z
Z
YY
XXX
WWWW
VVVVV
```

Итераторы

Итераторы позволяют стандартной библиотеке сделать алгоритмы независимыми от структур данных. Они представляют собой некоторую абстракцию указателей, обеспечивая операции, которые предоставляют доступ к элементам контейнера аналогично тому, как указатели предоставляют доступ к элементам массива. Итератор — чистая абстракция, так что итератором является все, что ведет себя так, как положено итератору.

Стандартные алгоритмы написаны в предположении, что итераторы отвечают требованиям, которые библиотека разбивает на категории. Каждый библиотечный алгоритм, использующий итераторы определенной категории, может работать с любым библиотечным или пользовательским классом, который предоставляет итератор, попадающий в данную категорию.

Выходной итератор, итератор для записи. Этот итератор можно использовать для поэлементного перемещения по контейнеру и для однократного вывода каждого элемента, опрашиваемого только один раз. Примером такого итератора служит `ostream_iterator`. Например, алгоритм `copy` требует, чтобы его третий аргумент обладал свойствами выходных итераторов.

Входной итератор, итератор для чтения. Данный итератор можно использовать для поэлементного перемещения по контейнеру и для многократного при необходимости считывания каждого элемента перед пере-

ходом к следующему. Примером такого итератора служит **istream_iterator**. Например, алгоритм **copy** требует, чтобы его первый и третий аргументы обладали свойствами входных итераторов.

Однонаправленный итератор. Этот итератор можно использовать для поэлементного перемещения по контейнеру, для возврата к элементам, на которые указывают ранее сохраненные итераторы, и для считывания либо вывода каждого элемента с нужной частотой. Например, алгоритм **replace** требует, чтобы его два первых аргумента обладали свойствами однонаправленных итераторов.

Двунаправленный итератор. Данный итератор можно использовать для поэлементного перемещения по контейнеру как в прямом, так и в обратном направлении. Например, контейнер **list** предоставляет двунаправленные итераторы, а алгоритм **reverse** требует, чтобы оба его аргумента обладали свойствами двунаправленных итераторов.

Итератор произвольного доступа. Этот итератор можно применять для перемещения по контейнеру, используя все операции, поддерживаемые указателями. Например, класс **vector** поддерживает итераторы произвольного доступа; алгоритму **sort** для работы необходимы итераторы произвольного доступа.

Все категории итераторов поддерживают проверку на равенство (неравенство). Итераторы произвольного доступа поддерживают все операции отношений.

Категории итераторов образуют вложенную структуру: каждый однонаправленный итератор одновременно является входным, каждый двунаправленный — однонаправленным, а итератор произвольного доступа — двунаправленным. Таким образом, любой алгоритм, работающий, например, с однонаправленными итераторами, будет успешно работать и с итераторами произвольного доступа.

В связи с тем, что постфиксная форма инкремента обычно использует префиксную форму (как правило, это выглядит следующим образом:

```
operator++(int) {
        tmp=*this;
        ++*this;
        return tmp;}
```

не рекомендуется использовать постфиксную форму инкремента, если только вам не нужно значение итератора до инкремента. Таким образом, в обычных циклах следует использовать **for(...;...;++p)**, а не **for(...;...;p++)**.

Все итераторы поддерживают следующие операции.

`++p`	Перемещает `p` к следующей позиции в контейнере.
`p++`	Префиксная операция `++p` возвращает lvalue `p` после перемещения, а `p++` возвращает копию предыдущего значения `p`.
`*p`	Элемент, на который указывает `p`. Для выходных итераторов операцию `*p` можно использовать только в качестве левого операнда операции присваивания (`=`), и каждое отдельное значение `p` можно применять таким способом только однократно. Для входных итераторов операцию `*p` можно использовать только для считывания данных, а инкремент `p` делает недействительными все копии, которые могли быть сделаны при использовании предыдущего значения `p`. Для всех прочих итераторных типов операция `*p` генерирует ссылку на соответствующее значение, хранящееся в элементе контейнера, и значение `p` остается действительным до тех пор, пока существует этот элемент.
`p == p2`	Возвращает значение `true`, если `p` равно `p2` (указывают на один и тот же элемент), и `false` в противном случае.
`p != p2`	Возвращает значение `false`, если `p` равно `p2` (указывают на один и тот же элемент), и `true` в противном случае.

Итераторы, отличные от выходных, поддерживают такую операцию.

`p->x`	Эквивалент операции `(*p).x`.

Двунаправленные итераторы и итераторы произвольного доступа поддерживают также операцию декремента (к которой применимо то же примечание, что и к операции инкремента).

`p--`	Перемещает `p` к предыдущей позиции в контейнере.
`--p`	Префиксная операция `--p` возвращает lvalue `p` после перемещения, а `p--` возвращает копию предыдущего значения `p`.

Итераторы произвольного доступа поддерживают все операции, применимые к обычным указателям, включая следующие.

`n+p`	Эквивалент операции `p+n`.
`p-n`	Эквивалент операции `p+(-n)`.
`p[n]`	Эквивалент операции `*(p+n)`.

`p+n`	Если `n>=0`, то результат представляет собой итератор, который указывает на позицию, расположенную через `n` позиций после элемента, на который указывает `p`. Результат не определен, если за элементом, на который указывает итератор `p`, находится меньше `n-1` элементов. Если `n<0`, то результат представляет собой итератор, который указывает на позицию, расположенную за `n` позиций до элемента, на который указывает `p`. Операция не определена, если элемент, расположенный в результирующей позиции, не попадает в диапазон контейнера.
`p2-p`	Операция определена только в том случае, если `p` и `p2` указывают на позиции одного и того же контейнера. Если `p2>=p`, результатом является количество элементов в диапазоне `[p,p2)`. В противном случае возвращается количество элементов в диапазоне `[p2,p)` со знаком минус. Тип результата — `ptrdiff_t`.
`p<p2`	Возвращает значение `true`, если `p` указывает на позицию, расположенную ближе к началу контейнера, чем позиция, на которую указывает `p2`. Операция не определена, если `p` и `p2` указывают на позиции в разных контейнерах.
`p<=p2`	Эквивалент операции `(p<p2)||(p==p2)`.
`p>p2`	Эквивалент операции `p2<p`.
`p>=p2`	Эквивалент операции `p2<=p`.

Резюмируя, доступные операции для разных типов контейнеров можно для простоты и понятности собрать в одну таблицу:

	Выходной	Входной	Однонаправленный	Двунаправленный	Произвольного доступа
Чтение:		`=*p`	`=*p`	`=*p`	`=*p`
Доступ:		`->`	`->`	`->`	`->` `[]`
Запись:	`*p=`		`*p=`	`*p=`	`*p=`
Итерация:	`++`	`++`	`++`	`++` `--`	`++` `--` `+` `-` `+=` `-=`
Сравнение:		`==` `!=`	`==` `!=`	`==` `!=`	`==` `!=` `<` `>` `>=` `<=`

Общие операции с контейнерами

Все контейнеры, а также класс `string`, предоставляют пользователю следующий интерфейс (здесь `Container` — условное обозначение типа контейнера, а `c` — объект типа `Container<T>`).

`Container<T>::iterator` `Container<T>::const_iterator`	Типы итераторов, связанные с контейнером `Container<T>`. Объекты данного типа можно использовать для чтения значений элементов контейнера; объекты типа `Container<T>::iterator` могут также применяться для модификации элементов контейнера.
`Container<T>::reverse_iterator` `Container<T>::const_reverse_iterator`	Типы итераторов, аналогичные предыдущим, но получающие доступ к элементам контейнера в обратном порядке.
`Container<T>::reference` `Container<T>::const_reference`	Синонимы для `T&` и `const T&` соответственно.
`Container<T>::size_type`	Целый беззнаковый тип, обеспечивающий возможность хранения максимального размера контейнера.
`Container<T>::value_type`	Тип элементов контейнера (синоним типа `T`).
`Container<T> c;`	Определяет `c` как пустой контейнер, для которого `c.size()==0`.
`Container<T> c1(c);`	Определяет `c1` как контейнер, для которого `c1.size()==c.size()`, а каждый элемент контейнера `c1` представляет собой копию соответствующего элемента контейнера `c`.
`c = c1`	Заменяет все элементы контейнера `c` элементами контейнера `c1`. Возвращает `c` как lvalue.
`c.begin()` `c.end()`	Итераторы, которые указывают на первый элемент (если таковой имеется) и на следующий за последним элементом контейнера соответственно. Функции генерируют значения типа `iterator` (или `const_iterator` — в зависимости от того, является ли контейнер `const`-объектом).

`c.rbegin()` `c.rend()`	Итераторы, аналогичные предыдущим, но обеспечивающие доступ к элементам контейнера в обратном порядке. Функции генерируют значения типа **`reverse_iterator`** (или **`const_reverse_iterator`** — в зависимости от того, является ли контейнер **`const`**-объектом).
`c.size()`	Количество элементов в контейнере **`c`**.
`c.empty()`	Если контейнер **`c`** пуст, возвращает значение **`true`**, в противном случае — значение **`false`**.
`c.clear()`	Очищает контейнер **`c`** (эквивалентна операции **`c.erase(c.begin(),c.end())`**). После выполнения этой функции **`c.size()==0`**. Не возвращает значения.

Последовательные контейнеры

В дополнение к общим контейнерным операциям класс **`string`** и последовательные контейнеры (**`vector`**, **`list`**, **`deque`**) поддерживают следующие операции.

`Container<T> c(n,t);`	Определяет контейнер **`c`**, содержащий **`n`** элементов, каждый из которых является копией объекта **`t`**.
`Container<T> c(b,e);`	Определяет контейнер **`c`** и инициализирует его копиями элементов из последовательности, определяемой итераторами **`b`** и **`e`**.

`c.insert(it,t)`
`c.insert(it,n,t)`
`c.insert(it,b,e)`
	Эти функции вставляют элементы в контейнер **`c`** непосредственно перед позицией, на которую указывает итератор **`it`**. Поскольку операция вставки может привести к перераспределению памяти, все итераторы в данном контейнере могут стать недействительными. Первая форма функции предназначена для вставки копии объекта **`t`** и возвращает итератор, который указывает на только что вставленный объект. Вторая форма позволяет вставить сразу **`n`** копий объекта **`t`** и не возвращает ничего. Третья функция также не возвращает ничего и вставляет в контейнер элементы последовательности, определенной входными итераторами **`b`** и **`e`**, которые не должны указывать на элементы самого контейнера **`c`**.

`c.erase(it)` `c.erase(b,e)`	Удаляет из контейнера либо отдельный элемент, на который указывает итератор `it`, либо последовательность элементов из диапазона `[b,e)`. В силу возможного переноса памяти итераторы, указывающие на элементы контейнера, могут стать недействительными. Функции возвращают итератор, указывающий на позицию непосредственно за областью удаления.
`c.assign(b;e)`	Заменяет элементы контейнера `c` элементами последовательности, определяемой входными итераторами `b` и `e`.
`c.front()`	Возвращает ссылку на первый элемент контейнера `c`. Если контейнер пуст, результат вызова функции не определен.
`c.back()`	Возвращает ссылку на последний элемент контейнера `c`. Если контейнер пуст, результат вызова функции не определен.
`c.push_back(t)`	Добавляет в конец контейнера копию объекта `t`, увеличивая при этом размер контейнера на единицу. Не возвращает значения.
`c.pop_back()`	Удаляет последний элемент из контейнера `c`. Не возвращает значения. Результат вызова функции не определен, если контейнер `c` пуст.
`inserter(c,it)`	Возвращает выходной итератор, который вставляет значения в контейнер `c`, начиная с позиции, непосредственно следующей за позицией, на которую указывает итератор `it`. Объявление содержится в заголовочном файле `<iterator>`.
`back_inserter(c)`	Возвращает выходной итератор, который позволяет добавлять новые объекты в конец контейнера. Объявление содержится в заголовочном файле `<iterator>`.
`v.max_size()`	Возвращает размер максимально возможного контейнера.
`v.swap(w)`	Эффективно обменивает содержимое двух контейнеров одного типа.

Функцию `c.push_back(t)` рекомендуется использовать для добавления элементов в контейнер, если позиция элемента не имеет значения: она имеет амортизированное время выполнения O(1). При заполнении контейнера исключительно с помощью вызовов `push_back()` каждый элемент в среднем копируется только один раз.

Дополнительные последовательные операции

Некоторые операции поддерживаются только теми контейнерами, для которых они могут эффективно выполняться.

`c[n]` `c.at(n)`	Ссылка на `n`-й элемент вектора, дека и строки. Начальный элемент расположен в позиции 0. Ссылка константна, если контейнер `c` является константой. Если `n` находится вне допустимого диапазона, для оператора `[]` ссылка не определена, а при использовании функции `at()` генерируется исключение `out_of_range`.	`vector` `deque`
`c.push_front(t)`	Вставляет копию объекта `t` в начало контейнера `c`, увеличивая размер контейнера на единицу. Не возвращает значения.	`list` `deque`
`c.pop_front()`	Удаляет первый элемент из контейнера `c`. Не возвращает значения. Если контейнер пуст, результат не определен.	`list` `deque`
`front_inserter(c)`	Возвращает выходной итератор, который позволяет вставлять новые значения в начало контейнера при помощи функции `push_front()`. Объявление содержится в заголовочном файле `<iterator>`.	`list` `deque`

Класс `vector`

Этот класс предоставляет динамически размещаемые в памяти массивы и поддерживает итераторы произвольного доступа. Кроме общих операций класс **`vector`** поддерживает следующие операции.

`v.reserve(n)`	Размещает в памяти объект `v` так, чтобы он мог увеличиваться и разместить по меньшей мере `n` элементов без дополнительного перераспределения памяти. Функция не инициализирует новые элементы; если `n` меньше текущего количества выделенной памяти, функция не выполняет никаких действий.
`v.resize(n, t = T())`	Размещает в памяти объект `v` так, чтобы он мог содержать `n` элементов. Если новый размер меньше старого, лишние элементы уничтожаются. Если размер больше, новые элементы инициализируются значением `t`.
`v.capacity()`	Возвращает размер выделенной памяти (количество элементов).

Стандарт гарантирует, что элементы **`vector`** в памяти всегда располагаются в виде последовательного массива, т.е., по сути, что `&v[j]+1 == &v[j+1]`.

Класс `deque`

Этот класс представляет двустороннюю очередь, т.е. последовательность элементов, оптимизированную таким образом, чтобы операции с обоими концами были столь же эффективны, как и для списков, а обращение по индексу приближалось по эффективности к таковому у вектора.

Класс `stack`

Данный класс может быть реализован на основе любого класса (по умолчанию — **`deque`**), поддерживающего операции **`back()`**, **`push_back()`** и **`pop_back()`**. Класс **`stack`** удаляет из интерфейса нестековые операции и дает перечисленным выше операциям общепринятые имена **`top()`**, **`push()`** и **`pop()`**. Инициализировать стек можно другим контейнером, а также указать используемый для хранения элементов класс, например:

```
vector<int> v;
stack<int, vector<int> > s(v);
```

Следует обратить внимание на то, что функция **`pop()`**, снимающая элементы со стека, не возвращает их значения, что связано с вопросами

безопасности исключений (см., например, [6]). Обратиться к элементу на вершине стека можно при помощи функции `top()`.

Класс `queue`

Класс очереди `queue` представляет собой интерфейс для контейнера, позволяющего вставлять элементы в конец и извлекать их из начала (по умолчанию используется `deque`). Может использоваться любой контейнер, предоставляющий операции `front()`, `back()`, `push_back()` и `pop_front()`. Вектор в качестве базового контейнера для очереди использоваться не может, поскольку он не предоставляет операцию `pop_front()`.

Класс `list`

Этот класс предоставляет динамически размещаемые в памяти двусвязные списки и поддерживает двунаправленные итераторы (в отличие от `deque` и `vector`, которые поддерживают итераторы произвольного доступа). В дополнение к общим операциям класс `list` поддерживает следующие операции.

`l.splice(it,l2)`	Вставляет все элементы списка `l2` в список `l` непосредственно перед позицией, на которую указывает итератор `it`, и удаляет эти элементы из списка `l2`. Все итераторы и ссылки в `l2` становятся недействительными. Размер списка `l` в результате операции равен сумме размеров исходных списков, а размер списка `l2` — равен 0. Функция не возвращает ничего.
`l.splice (it,l2,it2)` `l.splice(it,l2,b,e)`	Вставляет элемент, на который указывает итератор `it2`, либо последовательность элементов из диапазона `[b,e)` в список `l` непосредственно перед позицией, на которую указывает итератор `it`, и удаляет эти элементы из списка `l2`. Переносимые элементы должны принадлежать списку `l2`. Делает недействительными итераторы и ссылки на вставленные элементы в `l2`. Функция не возвращает ничего.
`l.remove(t)` `l.remove_if(p)`	Удаляет из списка `l` все элементы, значения которых равны `t` или для которых истинен предикат `p` (см. стр. 121). Функция не возвращает ничего.
`l.sort(cmp)` `l.sort()`	Сортирует список `l`, используя для сравнения оператор `<` или предикат `cmp` (см. стр. 121), если таковой указан. Функция не возвращает ничего.

`l.merge(l2)` `l.merge(l2,cmp)`	Добавляет все элементы списка `l2` в список `l`, сортируя их в соответствии с предикатом `cmp` (см. стр. 121). Если в списках встречаются одинаковые элементы, то элемент из списка `l` будет предшествовать элементу из списка `l2`. По завершении работы список `l2` опустошается. Функция не возвращает ничего.
`l.uniqie()` `l.unique(pred)`	Удаляет все (кроме первого) элементы из последовательных групп, в которых все элементы равны (т.е. если `*i==*(i-1)`, где `i` — итератор, или для которых истинен предикат `pred(*i,*(i-1))` (см. стр. 121)). Функция не возвращает ничего.
`l.reverse()`	Обращает порядок элементов в списке. Функция не возвращает ничего.

Класс `string`

Этот класс нельзя непосредственно отнести к контейнерам, однако объекты `string` поддерживают описанные выше контейнерные операции. Объекты `string` предназначены для хранения символьных строк переменной длины и поддерживают итераторы произвольного доступа к символам этих строк. Класс `string`, в частности, поддерживает следующие операции (всего их более 100).

`string s(cp);`	Определяет `s` как объект типа `string`, инициализированный копией символов, заданных параметром `cp`.
`os << s`	Выводит символы объекта `s` в поток `os`; возвращает ссылку на потоковый объект `os`.
`is >> s`	Считывает слово из потока `is` в объект `s`, удаляя его предыдущее содержимое; возвращает ссылку на потоковый объект `is`. Слова разделяются пробельными символами.
`getline(is,s)`	Считывает данные из входного потока `is` вплоть до следующего символа новой строки (включая его) и сохраняет считанные символы (без символа новой строки) в объекте `s`, удаляя его предыдущее содержимое; возвращает ссылку на потоковый объект `is`.
`s += s2`	Добавляет объект `s2` к объекту `s`; возвращает ссылку на объект `s`.

`s + s2`	Возвращает результат конкатенации объектов `s` и `s2`.
`s` *`op`* `s2`	Возвращает значение типа `bool`, означающее истинность выполнения данной операции отношения *`op`*, которая может быть одной из следующих: `<`, `<=`, `>`, `>=`, `==` и `!=`. Два строковых объекта равны, если равны все их соответствующие элементы. Если один объект является префиксом другого, то более короткий объект считается меньшим, чем более длинный. В противном случае результат сравнения определяется сравнением первой пары соответствующих символов, в которых строки отличаются одна от другой.
`s substr(n,n2)`	Возвращает новую строку, которая содержит `n2` символов, скопированных из объекта `s`, начиная с позиции `n`. Результат не определен, если `n>s.size()`. Если `n+n2>s.size()`, то копируются символы до конца строки.
`s.c_str()`	Генерирует `const`-указатель на символьный массив с завершающим нулевым символом, который содержит копию символов объекта `s`. Массив сохраняется только до тех пор, пока для объекта `s` не будет вызвана функция `c_str()`, `data()` или любая неконстантная функция-член.
`s.data()`	То же, что и функция `c_str()`, но массив не завершается нулевым символом.
`s.copy(cp,n)`	Копирует первые `n` символов (без завершения нулевым символом) из объекта `s` в пользовательский символьный массив `cp`. Массив должен содержать по меньшей мере `n` символов.

Ассоциативные контейнеры

Ассоциативные контейнеры оптимизированы для быстрого поиска по ключу.

После того как ключ вставлен в ассоциативный контейнер, он никогда не должен изменять свое относительное положение в этом контейнере (лучше всего для этого никоим образом не изменять значение ключа).

Кроме общих контейнерных операций ассоциативные контейнеры поддерживают следующие операции.

`Container<T>::key_type`	Тип ключа контейнера. Ассоциативный контейнер с ключами типа `K` и элементами типа `V` содержит значения типа `value_type` (не `V`!), который представляет собой класс `pair<const K,V>`.
`Container<T> c(cmp);`	Определяет контейнер как пустой ассоциативный контейнер с использованием для упорядочения предиката `cmp` (см. стр. 121).
`Container<T> c(b,e,cmp);`	Определяет контейнер как ассоциативный контейнер с использованием для упорядочения предиката `cmp` (см. стр. 121), заполненный копиями значений последовательности, определяемой входными итераторами `b` и `e`.
`c.insert(b,e)`	Вставка в контейнер `c` элементов из последовательности, определяемой входными итераторами `b` и `e`. Контейнер `map` при этом вставляет только те элементы, ключи которых отсутствуют в контейнере `c`.
`c.erase(it)`	Удаление из контейнера `c` элемента, на который указывает итератор `it`. Не возвращает значения.
`c.erase(b,e)`	Удаление из контейнера `c` элементов, определяемых диапазоном итераторов `[b,e)`. Не возвращает значения.
`c.erase(k)`	Удаление из контейнера `c` элементов с ключом `k`. Возвращает количество удаленных элементов.
`c.find(k)`	Возвращает итератор, который указывает на элемент с ключом, равным значению `k`. Если такой элемент в контейнере отсутствует, возвращается значение `c.end()`.

Класс `pair`

Этот класс является вспомогательным для работы с ассоциативным контейнером `map`, и представляет собой абстракцию `pair<K,V>` для пары значений `K` и `V`. Над объектами данного класса можно выполнять следующие операции.

`x.first`	Первый элемент объекта `x` типа `pair`.
`x.second`	Второй элемент объекта `x` типа `pair`.
`pair<K,V> x(k,v);`	Определяет `x` как новый объект типа `pair`, состоящий из элементов, которые имеют типы `K` и `V` и значения `k` и `v` (тип члена `x.first` — `K`, а тип `x.second` — `V`). Для объявления объекта требуется знать типы его членов.
`make_pair(k,v)`	Генерирует новый объект типа `pair<K,V>` со значениями элементов `k` и `v`. Для создания объекта знать типы его членов не обязательно.

Классы `map`, `multimap`

Класс `map` предоставляет динамически размещаемые в памяти, независимые от типа ассоциативные массивы. Для хранения пар значений (ключ, значение), которые являются элементами объекта `map`, используется вспомогательный класс `pair`. Класс `map` поддерживает двунаправленные итераторы. Каждый объект `map` хранит значения типа `V` связанными с ключами типа `const K`, т.е. значения элементов объекта `map` могут изменяться, в то время как ключи — нет. Класс `multimap` отличается от класса `map` тем, что в нем могут содержаться несколько элементов с одинаковыми ключами, так что в нем отсутствует оператор индексации `[]`, а функция `insert()` возвращает не `pair<map<K,V>::iterator,bool>`, а итератор.

Кроме рассмотренных выше общих операций (стр. 108 и 116) класс `map` поддерживает следующие операции.

`map<K,V,P> m(cmp);`	Определяет `m` как новый пустой объект типа `map`, который содержит значения типа `V`, связанные с ключами типа `const K`, и использует предикат `cmp` (см. стр. 121) типа `P` для сравнения элементов при их вставке в данный контейнер.
`m[k]`	Генерирует ссылку на значение в объекте `m` в позиции, связанной с ключом `k`. Если такой элемент не найден, то в контейнер вставляется связанный с ключом `k` элемент типа `V`, инициализированный значением `V` по умолчанию. Вычисление `m[k]` может изменить объект `m`, поэтому для использования данной операции объект `m` не должен быть константным.

`m.insert (make_pair(k,v))`	Вставляет значение `v` в объект `m` в позицию, определяемую ключом `k`. Если в объекте `m` имеется элемент, соответствующий данному ключу, то его значение не изменится. Функция возвращает объект типа `pair<map<K,V>::iterator,bool>`, первый компонент которого указывает на элемент с заданным ключом, а второй показывает, был ли вставлен новый элемент или нет.
`m.find(k)`	Возвращает итератор, который указывает на элемент (если таковой имеется), связанный с ключом `k`. Если такого элемента нет, возвращается значение `m.end()`.
`*it`	Разыменование итератора. Генерирует объект типа `pair<const K,V>`, который содержит ключ и связанное с ним значение в позиции, на которую указывает итератор `it; it->first` имеет тип `const K` и представляет ключ, а `it->second` имеет тип `V` и представляет значение, соответствующее этому ключу.
`m.lower_bound(k)`	Возвращает итератор, указывающий на первый элемент с ключом `k`.
`m.upper_bound(k)`	Возвращает итератор, указывающий на первый элемент с ключом, превышающим `k`.
`m.equal_range(k)`	Возвращает пару итераторов `pair<iterator,iterator>`, которая определяет диапазон элементов `m`, значение ключа у которых равно `k`.
`m.count(k)`	Возвращает количество элементов `m` с ключом `k`.

Классы `set`, `multiset`

Множества можно рассматривать как ассоциативные массивы, в которых значение элементов игнорируется, и учитываются только их ключи. Это ведет к некоторым, достаточно небольшим, изменениям в интерфейсе класса. Так, в нем отсутствуют операторы индексации `[]`, а тип `value_type` представляет собой тип ключа. Класс `multiset` отличается от класса `set` тем, что в нем могут содержаться одинаковые ключи. Соответственно, функция `insert()`, как и в классе `multimap`, возвращает не пару, а итератор.

Алгоритмы и объекты-функции

Сам по себе любой контейнер практически мало полезен; он служит всего лишь средством для хранения объектов, с которыми следует выполнять некоторые действия. Стандартная библиотека содержит массу различных алгоритмов для выполнения большинства распространенных операций, которые могут потребоваться пользователю контейнера. Алгоритмы представляют собой шаблоны функций, так что они могут работать с разными типами контейнеров, содержащих различные элементы. Объекты-функции обеспечивают механизм приспособления стандартных алгоритмов к нуждам пользователя. Они предоставляют алгоритму всю необходимую для работы с данными пользователя информацию.

Алгоритмы не выполняют проверки диапазонов, так что предотвращение соответствующих ошибок вы должны обеспечить самостоятельно другими средствами.

Алгоритмы стандартной библиотеки реализуют большинство распространенных универсальных операций с контейнерами, например, просмотр, сортировку, поиск, вставку или удаление элементов. Все стандартные алгоритмы находятся в пространстве имен **std**, а их объявления — в заголовочном файле **<algorithm>**. В том же пространстве имен находятся и стандартные объекты-функции, а их объявления помещены в заголовочный файл **<functional>**.

Объекты-функции

Чтобы лучше понять, что такое объект-функция, рассмотрим простейший пример их применения.

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

template<class T, class Act>
T Action(const T& t1, const T& t2, Act A)
{
    return A(t1, t2);
};

template<class T>
class Sum {
public:
    T operator()(const T& t1, const T& t2)
{
    return t1+t2;
};
```

```
template<class T>
T Mul(const T& t1, const T& t2)
{
    return t1*t2;
}

int main()
{
    cout << "Action<int>(2,3,Sum<int>()) = " <<
            Action<int>(2,3,Sum<int>()) << endl;
    cout << "Action<int>(2,3,Mul<int>) = " <<
            Action<int>(2,3,Mul<int>) << endl;
}
```

Как видите, здесь в качестве одного из параметров шаблона функции **Action**, а именно параметра **Act**, может быть передан экземпляр класса с определенным оператором **operator()** или функция — словом, нечто, поддерживающее синтаксис вызова функции с указанным числом аргументов. Функция **Action** использует этот аргумент для выполнения некоторых действий, в данном случае — **Sum<int>()** (обратите внимание на круглые скобки — мы передаем в качестве аргумента объект типа **Sum<int>**, а не сам тип) для суммирования двух целых чисел, и **Mul<int>** (в этом случае круглых скобок нет, так как **Mul<int>** — имя функции (преобразуемое в указатель на функцию)) для их умножения.

Именно объекты-функции такого вида (экземпляры классов или функции) и используются во многих стандартных алгоритмах для выполнения конкретных действий, задаваемых пользователем алгоритма.

Стандартная библиотека предусматривает для создания объектов-функций базовые классы, назначение которых — дать стандартные имена типам аргументов и возвращаемых значений, и позволяет использовать вспомогательные объекты — такие, например, как связыватели. Вот эти базовые классы:

```
template<class Arg,
    class Result>
    struct unary_function
    {
        typedef Arg    argument_type;
        typedef Result result_type;
    };

template<class Arg1,
    class Arg2,
    class Result>
```

```
    struct binary_function
    {
        typedef Arg1   first_argument_type;
        typedef Arg2   second_argument_type;
        typedef Result result_type;
    };
```

Предикат — это объект-функция (или просто функция), которая возвращает значение типа `bool`. Например, в заголовочном файле `<functional>` в Visual C++ Toolkit 2003 имеется следующий предикат.

```
template<class Ty>
struct less: public binary_function<Ty, Ty, bool>
{
    bool operator()(const Ty& Left,
                    const Ty& Right) const
    {
        return (Left < Right);
    }
};
```

Он предназначен для сравнения двух объектов при помощи оператора `<` (заметим, что если тип `Ty` представляет собой тип класса, то оператор `<` может быть переопределен и иметь иную семантику).

Вот основные предикаты и арифметические объекты-функции, предоставляемые стандартной библиотекой C++ и объявленные в заголовочном файле `<functional>`.

Предикаты	Семантика
`equal_to`	`arg1 == arg2`
`not_equal_to`	`arg1 != arg2`
`greater`	`arg1 > arg2`
`less`	`arg1 < arg2`
`greater_equal`	`arg1 >= arg2`
`less_equal`	`arg1 <= arg2`
`logical_and`	`arg1 && arg2`
`logical_or`	`arg1 || arg2`
`logical_not`	`!arg`

Арифметические операции	Семантика
`plus`	`arg1 + arg2`
`minus`	`arg1 - arg2`
`multiplies`	`arg1 * arg2`
`divides`	`arg1 / arg2`
`modulus`	`arg1 % arg2`
`negate`	`-arg`

Бывают ситуации, когда можно решить задачу написанием пользовательского объекта-функции, который очень мало отличается от стандартного. В такой ситуации можно воспользоваться специальными классами — связывателями, адаптерами и отрицателями. Связыватель позволяет использовать объект-функцию с двумя аргументами как функцию с одним аргументом путем связывания одного аргумента со значением. Адаптер функций-членов позволяет использовать функции-члены, а адаптер указателя на функцию — соответственно указатели на функции в качестве аргументов алгоритмов. Отрицатель позволяет получить обратный предикат. Ниже перечислены связыватели, адаптеры и отрицатели стандартной библиотеки.

Функция	Тип объекта	Действие (`operator()()`)
`bind2nd(y)`	`bind2nd`	Вызывает бинарную функцию с `y` в качестве второго аргумента
`bind1st(x)`	`bind1st`	Вызывает бинарную функцию с `x` в качестве первого аргумента
`mem_fun()`	`mem_fun_t`	Вызывает 0-аргументную функцию-член через указатель
`mem_fun()`	`const_mem_fun_t`	Вызывает 0-аргументную константную функцию-член через указатель
`mem_fun()`	`mem_fun1_t`	Вызывает унарную функцию-член через указатель
`mem_fun_ref()`	`mem_fun_ref_t`	Вызывает 0-аргументную функцию-член через ссылку
`mem_fun_ref()`	`const_mem_fun_ref_t`	Вызывает 0-аргументную константную функцию-член через ссылку
`mem_fun_ref()`	`mem_fun1_ref_t`	Вызывает унарную функцию-член через ссылку

Функция	Тип объекта	Действие (operator()())
mem_fun_ref()	const_mem_fun1_ref_t	Вызывает унарную константную функцию-член через ссылку
mem_fun_ref()	const_mem_fun1_t	Вызывает унарную константную функцию-член через указатель
ptr_fun()	pointer_to_unary_function	Вызывает указатель на унарную функцию
ptr_fun()	pointer_to_binary_function	Вызывает указатель на бинарную функцию
not1()	unary_negate	Заменяет унарный предикат на его отрицание
not2()	binary_negate	Заменяет бинарный предикат на его отрицание

К сожалению, объем данной книги не позволяет полностью рассмотреть вопросы использования приведенных классов, поэтому ограничимся только парой примеров кода с комментариями.

```
void f(list<string> ls)
{
    typedef list<string>::iterator LSI;
    // Поиск с использованием для объекта функции-члена
    // без аргументов
    LSI p = find_if(ls.begin(),ls.end(),
                    mem_fun_ref(&string::empty));
}

void rotate_all(list<Shape*>&ls,double angle)
{
    // Использование функции-члена с одним аргументом
    // через указатель на объект
    for_each(ls.begin(),ls.end(),
             bind2nd(mem_fun(&Shape::rotate),angle));
}
```

Алгоритмы

Алгоритм for_each

Алгоритм `for_each(b,e,f)` выполняет операторный аргумент `f` для всех объектов последовательности, заданной входными итераторами `[a,b)`. Алгоритм не модифицирующий, так как он явно не изменяет последовательность. Однако, будучи применен к неконстантной последовательности, может изменять ее элементы.

Алгоритмы find, find_if, find_first_of, adjacent_find

Алгоритмы `find(b,e,t)` и `find_if(b,e,p)` осуществляют поиск элемента последовательности, заданной входными итераторами `[b,e)`, значение которого равно `t` (алгоритм `find`), или для которого выполняется предикат `p` (`find_if`). Оба алгоритма, обнаружив такой элемент, возвращают указывающий на него итератор.

Алгоритм `find_first_of(b,e,b2,d2[,p])` ищет в последовательности `[b,e)` первый элемент, который имеется в последовательности `[b2,d2)`, или для которого и элемента из второй последовательности выполняется предикат `p`.

Алгоритм `adjacent_find(b,e[,p])` ищет первый из двух последовательных совпадающих элементов (или тех, для которых выполняется предикат `p`).

Алгоритмы count, count_if

Алгоритмы `count(b,e,t)` и `count_if(b,e,p)` подсчитывают количество элементов последовательности `[b,e)`, равных `t` (или тех, для которых выполняется предикат `p`).

Алгоритмы equal, mismatch

Алгоритмы `equal(b,e,b2[,p])` и `mismatch(b,e,b2[,p])` сравнивают две последовательности — `[b,e)` и последовательность, начинающуюся с элемента, на который указывает итератор `b2` (конечный итератор второй последовательности не указывается; считается, что во второй последовательности количество элементов по крайней мере такое же, как и в первой). Алгоритм `equal` просто проверяет, одинаковы ли указанные последовательности (проверяя равенство пар элементов или выполнения для них предиката `p`), и возвращает соответствующее значение типа `bool`. Алгоритм `mismatch` находит первую пару несовпадающих элементов и возвращает итераторы на эти элементы в виде `pair<I1,I2>`.

Алгоритмы search, find_end, search_n

Алгоритм **search(b,e,b2,e2[,p])** возвращает однонаправленный итератор, указывающий на первое вхождение подпоследовательности **[b2,e2)** в последовательность **[b,e)**. Для проверки используется предикат **p** или оператор **==**, если предикат не задан.

Алгоритм **find_end(b,e,b2,e2[,p])** возвращает итератор, указывающий на последнее вхождение подпоследовательности **[b2,e2)** в последовательность **[b,e)**. Для проверки используется предикат **p** или оператор **==**, если предикат не задан.

Алгоритм **search_n(b,e,n,t[,p])** ищет первое вхождение в последовательность **[b,e)** как минимум **n** последовательных элементов **t**. Для проверки используется предикат **p** или оператор **==**, если предикат не задан.

Алгоритмы copy, copy_backward

Алгоритмы **copy(b,e,d)** и **copy_backward(b,e,d)** копируют элементы последовательности, заданной входными итераторами **b** и **e** в область-приемник, определенную выходным итератором **d**. При выполнении данного алгоритма предполагается, что приемник обладает достаточным местом для размещения копируемых значений. Возвращает значение, указывающее на позицию за последним элементом в приемнике. Заметим, что копировать не обязательно в контейнер — например, можно использовать копирование в поток. Алгоритм **copy_backward** копирует элементы в обратном порядке, начиная с последних элементов последовательности, что позволяет избежать неприятностей, если последовательности перекрываются, т.е. начало приемника находится в середине исходной последовательности. Соответственно, в связи с обратным направлением при работе этого алгоритма требуется наличие двунаправленных итераторов.

Алгоритм **copy_if** в стандартной библиотеке отсутствует, но при необходимости реализовать его очень легко:

```
template<class In, class Out, class Pred>
Out copy_if(In first, In last, Out res, Pred p)
{
    while(first != last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

Алгоритм `transform`

Алгоритм **`transform`**, несмотря на свое название, не обязательно трансформирует входную последовательность. Его задача — создать выход, который является преобразованием входа, основанным на операции, определенной пользователем.

Имеется два варианта алгоритма: **`transform(b,e,d,f)`** и **`transform(b,e,b2,d,f)`**. Если итератор **`b2`** не задан, функция **`f`** должна принимать один аргумент, и функция **`transform`** вызывает ее для всех элементов последовательности, определенной итераторами **`b`** и **`e`**. Если же итератор **`b2`** задан, то функция **`f`** должна принимать два аргумента, которые выбираются попарно из последовательности, заданной итераторами **`b`** и **`e`**, и последовательности такой же длины, начало которой задается входным итератором **`b2`**. Оба варианта функции **`transform`** помещают полученную в результате последовательность в приемную область, определенную выходным итератором **`d`**, и возвращают значение итератора, указывающего на элемент, расположенный за последним элементом приемника. Предполагается, что приемник имеет достаточно большой размер, чтобы вместить все сгенерированные элементы. Обратите внимание на то, что итератор **`d`** может быть равен итератору **`b`** (или **`b2`**, если таковой задан), и в этом случае результат замещает исходную последовательность.

Алгоритмы `unique`, `unique_copy`

При сборе информации часто происходит дублирование данных. Алгоритмы **`unique`** и **`unique_copy`** позволяют избавиться от соседних одинаковых элементов.

Алгоритм **`unique(b,e[,p])`** удаляет в последовательности, определяемой итераторами **`b`** и **`e`**, соседние одинаковые элементы (для проверки их одинаковости используется предикат **`p`** или, если таковой не задан, оператор **`==`**).

Как и прочие стандартные алгоритмы, алгоритм **`unique`** работает с итераторами (конкретно — с однонаправленными) и не знает тип контейнера, на который указывает итератор, так что он не может модифицировать контейнер. Он может только изменять значения элементов. Поэтому алгоритм **`unique`** не удаляет одинаковые элементы из входной последовательности, как можно было бы предположить, а просто преобразует последовательность, помещая уникальные элементы в ее начало, и возвращает итератор, указывающий на конец последовательности уникальных элементов. Элементы, находящиеся за этой «уникальной» последовательностью, не изменяются.

Алгоритм **`unique_copy(b,e,d[,p])`** выполняет копирование последовательности, заданной входными итераторами **`b`** и **`e`** в последовательность, начало которой задано выходным итератором **`d`**, по ходу копиро-

вания удаляя смежные дубликаты. Возвращает значение d после его увеличения на количество скопированных элементов. Предполагается, что принимающая последовательность достаточно велика, чтобы принять все копируемые элементы. Для тестирования используется предикат p (или оператор ==, если предикат не задан).

Еще раз обращаем ваше внимание на то, что данные алгоритмы удаляют только смежные одинаковые элементы, так что перед их выполнением последовательности следует отсортировать. Необходимо также убедиться, что используемый при сортировке критерий равенства совпадений тот же, что и при удалении одинаковых элементов.

Алгоритм replace

Имеется четыре разновидности алгоритма:

`replace(b,e,t1,t2)` заменяет все элементы последовательности, определяемой однонаправленными итераторами b и e, значения которых равны t1, значениями t2;

`replace_if(b,e,p,t2)` заменяет все элементы последовательности, определяемой однонаправленными итераторами b и e, для которых справедлив предикат p, значениями t2;

`replace_copy(b,e,d,t1,t2)` копирует все элементы последовательности, определяемой входными итераторами b и e, в последовательность, начало которой задано выходным итератором d, по ходу копирования заменяя элементы, значения которых равны t1, значениями t2;

`replace_copy_if(b,e,d,p,t2)` копирует все элементы последовательности, определяемой входными итераторами b и e, в последовательность, начало которой задано выходным итератором d, по ходу копирования заменяя элементы, для которых справедлив предикат p, значениями t2.

Алгоритм remove

Алгоритмы `remove(b,e,t)` и `remove_if(b,e,p)` переставляют элементы в последовательности, заданной однонаправленными итераторами b и e, чтобы элементы, значения которых не совпадают со значением t, или для которых предикат p, если таковой задан, возвращает значение `false`, группировались в начале указанной последовательности. Возвращают итератор, который указывает на элемент, расположенный за неудаленными элементами.

Алгоритмы `replace_copy(b,e,d,t)` и `replace_copy_if(b,e,d,p)` аналогичны алгоритмам `remove` и `remove_if`, но помещают копии элементов, значения которых не совпадают со значением t, или для которых предикат p, если таковой задан, возвращает значение `false`, в приемную область, заданную выходным итератором d. Возвращается значение итератора, указывающего на элемент, расположенный за последним эле-

ментом приемника. Предполагается, что приемная область достаточно велика, чтобы в ней разместились все копируемые значения. Элементы последовательности, заданной итераторами `b` и `e`, не изменяются, так что `b` и `e` должны быть только входными итераторами.

Алгоритмы `fill`, `generate`

Алгоритмы `fill` и `generate` предназначены для присвоения значений последовательностям.

Алгоритм `fill(b,e,t)` присваивает всем элементам последовательности, определяемой однонаправленными итераторами `b` и `e`, значения `t`. Алгоритм `fill_n(d,n,t)` присваивает значение `t` первым `n` элементам последовательности, определяемой выходным итератором `d`. Предполагается, что приемная область достаточно велика, чтобы в ней разместились все `n` значений.

Алгоритмы `generate(b,e,g)` и `generate_n(d,n,g)` работают так же, как и алгоритмы `fill`, но вместо одного значения используются последовательные вызовы функции-генератора `g`.

Алгоритмы `reverse`, `rotate`, `random_shuffle`

Алгоритмы `reverse(b,e)` и `reverse_copy(b,e,d)` изменяют порядок следования элементов последовательности, определяемой двунаправленными итераторами `b` и `e`, на обратный, т.е. первый элемент становится последним и т.д. Алгоритм `reverse_copy` выполняет копирование в обратном порядке элементов входной последовательности в последовательность, определяемую выходным итератором `d`. Предполагается, что приемная область достаточно велика, чтобы в ней разместились все копируемые значения.

Алгоритмы `rotate(b,m,e)` и `rotate_copy(b,m,e,d)` рассматривают последовательность, определяемую однонаправленными итераторами `b` и `e`, в качестве кольца и циклически сдвигают ее элементы, так что бывший средний элемент, определяемый однонаправленным итератором `m`, оказывается на месте первого, т.е. элемент из позиции `b+i` перемещается в позицию `b+(i+(e-m))%(e-b)`. Алгоритм `rotate_copy` выводит элементы входной последовательности в указанном порядке в последовательность, определяемую выходным итератором `d`. Предполагается, что приемная область достаточно велика, чтобы в ней разместились все копируемые значения.

Алгоритм `random_shuffle(b,e[,g])` «тасует» последовательность, определенную итераторами произвольного доступа `b` и `e`, при помощи генератора случайных чисел `g` (если таковой не определен, используется генератор случайных равномерно распределенных чисел стандартной библиотеки, так что все перестановки равновероятны). Генератор должен

представлять собой объект-функцию или просто функцию, возвращающую случайное число в диапазоне от 0 до значения, которое на 1 меньше переданного ей аргумента.

Алгоритм `swap`

Для обмена элементов контейнера наиболее эффективным способом является использование стандартных алгоритмов **`swap`**, специализированных для наиболее важных типов.

Алгоритм **`swap(a,b)`** меняет местами содержимое **`a`** и **`b`**, причем наиболее эффективным способом.

Алгоритм **`iter_swap(x,y)`** меняет местами элементы, на которые указывают его аргументы, представляющие собой однонаправленные итераторы.

Алгоритм **`swap_ranges(b,e,d)`** меняет местами элементы из диапазона, определяемого однонаправленными итераторами **`b`** и **`e`**, и диапазона той же длины, на начальный элемент которого указывает однонаправленный итератор **`d`**.

Алгоритм `sort`

Для работы алгоритмам сортировки необходимы итераторы с произвольным доступом (если далее в подразделе явно не указано иное), т.е. лучше всего они работают с контейнером **`vector`** и аналогичными ему. Контейнер **`list`**, например, итераторы произвольного доступа не предоставляют, так что сортировка такого контейнера должна выполняться при помощи специфической операции класса (см. стр. 114). Операцией сравнения по умолчанию является предикат **`less`**, который, в свою очередь, по умолчанию использует оператор **`<`**.

Алгоритмы **`sort(b,e[,cmp])`** и **`stable_sort(b,e[,cmp])`** сортируют последовательности элементов, определенные итераторами **`b`** и **`e`**, с использованием предиката **`cmp`** (если таковой определен, или **`less`** в противном случае) для проверки выполнения отношения «меньше».

Среднее время работы алгоритма **`sort`** — $O(n*\log(n))$; в худшем случае оно составляет $O(n*n)$. Для гарантированного поведения алгоритма для любых входных данных или если требуется сохранение исходного относительного расположения одинаковых элементов, можно использовать алгоритм **`stable_sort`**, время работы которого гарантированно составляет $O(n*\log(n)*\log(n))$.

Иногда в задаче не требуется полностью сортировать последовательность, так как в ней представляют интерес только первые элементы отсортированной последовательности. В таком случае можно использовать алгоритмы **`partial_sort(b,m,e[,cmp])`** и **`partial_sort_copy(b,e,b2,e2[,cmp])`**. Алгоритм **`partial_sort`** сортирует элементы

последовательности [b,e) до тех пор, пока последовательность [b,m) не примет тот же вид, что и в полностью отсортированной последовательности. Алгоритм partial_sort_copy сортирует последовательность, определенную входными итераторами b и e, до тех пор, пока в выходную последовательность [b2,e2) не будет выведено количество элементов, равное длине меньшей из двух последовательностей. Сортировка производится с использованием предиката cmp (если таковой определен, или less в противном случае) для проверки выполнения отношения «меньше».

Существует также алгоритм nth_element(b,n,e[cmp]), который сортирует элементы последовательности [b,e) до тех пор, пока промежуточный элемент, на который указывает итератор n, не примет то же значение, что и в полностью отсортированной последовательности. Сортировка производится с использованием предиката cmp (если таковой определен, или less в противном случае) для проверки выполнения отношения «меньше».

Алгоритм binary_search

Последовательный поиск типа find очень неэффективен, но это лучшее, чего можно добиться без сортировки и хэширования. Однако при работе с отсортированной последовательностью можно воспользоваться алгоритмом binary_search(b,e,t[,cmp]), который возвращает значение типа bool, указывающее, имеется ли в отсортированной последовательности, определенной однонаправленными итераторами b и e, значение t. Для проверки отношения «меньше» между двумя элементами используется предикат cmp, если таковой определен, и less в противном случае.

Может, однако, оказаться, что в последовательности много искомых элементов, поэтому в стандартной библиотеке имеются алгоритмы для поиска ряда одинаковых элементов и нижней и верхней границ этого ряда — соответственно equal_range(b,e,t[cmp]), который возвращает пару pair<l,u> итераторов, указывающих нижнюю и верхнюю границы ряда, lower_bound(b,e,t[cmp]) и upper_bound(b,e,t[cmp]). Если указанные алгоритмы не могут найти искомый элемент, они возвращают итераторы, указывающие на первый элемент, который больше указанного, или итератор e, если такого элемента в последовательности нет. Это свойство позволяет определить, в какое место последовательности надо вставить новый элемент, чтобы не нарушалось условие сортировки.

Алгоритм merge

Две отсортированные последовательности можно объединить в новую отсортированную последовательность, используя алгоритм merge,

или объединить две части одной последовательности при помощи **inplace_merge**.

Алгоритм **merge(b,e,b2,e2,d[,cmp])** объединяет две отсортированные последовательности, определенные парами входных итераторов **b** и **e** и **b2** и **e2**, и вносит их элементы в отсортированном порядке в последовательность, определенную выходным итератором **d** (предполагается, что приемник имеет достаточно большой размер, чтобы вместить все сгенерированные элементы). Сортировка производится с использованием предиката **cmp** (если таковой определен, или **less** в противном случае) для проверки выполнения отношения «меньше».

Алгоритм **inplace_merge(b,m,e[,cmp])** используется в основном тогда, когда имеется последовательность, которую можно отсортировать по разным критериям. Вот небольшой пример:

```
// Сортировка символов без учета регистра
bool ucmp(const char& t1, const char& t2)
{ return toupper(t1) < toupper(t2); }

int main()
{
    char p[] = "ABCDEFGHIJKLMNOPQabcdefghijklmnopq";
    inplace_merge(p,p+17,p+34,ucmp);
    cout << p << endl;
}
```

Здесь происходит слияние двух отсортированных подпоследовательностей, в результате чего получаем на выходе строку «**AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQq**».

Отметим, что эти алгоритмы слияния отличаются от слияния списков **list** (см. стр. 114) тем, что не удаляют элементы из входной последовательности — вместо этого элементы копируются. При равенстве элементы из первого диапазона всегда предшествуют элементам из второго диапазона.

Алгоритм partition

Алгоритмы **partition(b,e,p)** и **stable_partition(b,e,p)** делят последовательность, заданную двунаправленными итераторами **b** и **e**, так, чтобы элементы, для которых предикат **p** истинен, размещались в начале контейнера. Возвращает итератор, указывающий на первый элемент, для которого этот предикат дает значение **false**, или значение **e**, если предикат истинен для всех элементов последовательности. Функция **stable_partition** сохраняет относительный порядок элементов в каждой части последовательности.

Алгоритмы min, max, min_element, max_element, lexicographical_compare

Алгоритмы `min(t1,t2[,cmp])` и `max(t1,t2[,cmp])` выбирают меньшее и большее значения одинакового типа, основываясь на сравнении. Естественным обобщением этих функций для применения к последовательностям являются алгоритмы `min_element(b,e[,cmp])` и `max_element(b,e[,cmp])`, возвращающие итераторы, указывающие на наименьший (наибольший) элемент последовательности, заданной однонаправленными итераторами `b` и `e`.

Алгоритм `lexicographical_compare(b,e,b2,e2[,p])` возвращает значение типа `bool`, означающее, меньше ли последовательность элементов в диапазоне `[b,e)` последовательности элементов в диапазоне `[b2,e2)`. Для сравнения элементов используется предикат `p` или оператор `<`, если предикат не задан. Если одна последовательность является префиксом другой, то меньшей считается более короткая последовательность. В противном случае результат определяется сравнением первой пары соответствующих элементов, в которой обнаружено различие между последовательностями. Итераторы `b`, `e`, `b2` и `e2` — входные.

Алгоритм permutation

Алгоритмы `next_permutation(b,e[,cmp])` и `prev_permutation(b,e[,cmp])` предназначены для получения всех перестановок элементов последовательности в лексикографическом порядке. Функции возвращают значение типа `bool`, указывающее, имеется ли следующая перестановка. Если возвращается значение `false`, то последовательность представляет собой перестановку, в которой элементы располагаются в обратном лексикографическом порядке.

Алгоритм accumulate

Алгоритм `accumulate(b,e,t[,op])` объявлен в заголовочном файле `<numeric>` и суммирует все элементы последовательности, определенной входными итераторами `b` и `e`. Если определен бинарный оператор `op`, то для каждого итератора выполняется вычисление `t = op(t,*it)`, а результатом работы функции является копия `t`. Если оператор `op` не определен, то вычисляется `t = t+*it`. Заметим, что оператор `+` может быть перегружен, так что, например, алгоритм `accumulate` может использоваться для конкатенации всех строк последовательности.

Алгоритм inner_product

Алгоритм `inner_product(b,e,b2,t[,op,op2])` предназначен для накопления из двух последовательностей и объявлен в заголовочном файле `<numeric>`. Если не определены операторы `op` и `op2`, в `t` накапливается скалярное произведение двух последовательностей, заданных входными итераторами `b` и `e` и `b2` (для второй последовательности задается только ее начало). Если операторы не определены, то для каждой пары итераторов из двух последовательностей вычисляется `t = t+*it**it2`, а результатом работы функции является копия `t`. Если операторы определены, то вычисление выполняется по схеме `t = op(t,op2(*it,*it2))`.

Алгоритмы partial_sum, adjacent_difference

Взаимообратные алгоритмы `adjacent_difference(b,e,d[,op])` и `partial_sum(b,e,d[,op])` объявлены в заголовочном файле `<numeric>`. Алгоритм `adjacent_difference` вычисляет разности (или результат выполнения оператора) соседних элементов последовательности, заданной входными итераторами `b` и `e`, и помещает его в последовательность, задаваемую выходным итератором `d`. Например, если у нас была последовательность значений *a,b,c,d...*, то мы получим последовательность *a,b-a,c-b, d-c* и так далее. Алгоритм `partial_sum` выполняет обратное действие, т.е. накапливает частичные суммы, превращая последовательность значений *a,b,c,d...* в *a,a+b,a+b+c,a+b+c+d* и так далее. Обратите внимание на то, что алгоритмы реализованы таким образом, что выходная последовательность может совпадать с входной.

Потоки ввода-вывода

Для ввода-вывода информации в C++ используются так называемые *потоки*, которые представляют собой механизм преобразования значений различного типа в последовательность символов и наоборот. Здесь мы рассмотрим только потоки `istream`, `ostream`, `ifstream` и `ofstream`, причем весьма кратко. Более детально эти и остальные потоки (например, строковые) описываются в соответствующей литературе.

Стандартными в C++ являются потоки: `cin`, связанный со стандартным вводом, `cout`, связанный со стандартным выводом, и `cerr`, связанный со стандартным потоком ошибок (этот поток по умолчанию не буферизуется).

Считывание и запись данных выполняются при помощи следующих операций.

`is >> t`	Считывает значение из объекта входного потока `is` в объект `t`, опуская пробелы. Входные данные должны быть в форме, пригодной для преобразования в значение типа объекта `t`. Неправильные данные приводят к отказу, оставляя объект `is` в некорректном состоянии до тех пор, пока не будет вызвана функция `is.clear()`. Стандартная библиотека определяет оператор ввода для встроенных типов и `string`.
`os << t`	Посылает значение объекта `t` в выходной объект `os` в формате, соответствующем типу объекта. Стандартная библиотека определяет оператор вывода для встроенных типов и `string`.
`is.get(c)` `c = is.get()`	Считывает очередной символ из входного потока (даже если это пробельный символ) в объект `c`. Функция `get(c)` возвращает ссылку на входной поток.
`c = is.peek()`	Просмотр следующего символа в потоке без его выборки из потока.
`is.unget()`	Восстанавливает предыдущее состояние потока `is`, возвращаясь на один символ назад. Эта функция обычно используется, когда требуется считывание данных до тех пор, пока не встретится конкретный символ, который должен остаться в потоке для последующей обработки. Гарантируется запоминание только одного символа для восстановления потока.
`is.putback(c)`	Вносит в буфер символ `c`, который будет считан при следующей операции считывания.
`is.get(s,n[,c])`	Считывание не более `n-1` символов из потока `is` в буфер `s` размером `n` символов до заполнения буфера или до символа `c` или символа новой строки, если `c` не определен. Буфер завершается нулевым символом. Если встречен завершающий символ, он остается первым несчитанным символом в потоке (его следует оттуда удалить перед очередным вызовом `get()`). Функция возвращает ссылку на входной поток.

`is.getline(s,n)`	Считывание не более `n-1` символов из потока `is` в буфер `s` размером `n` символов до заполнения буфера или до символа новой строки. Завершающий символ из потока удаляется. Функция возвращает ссылку на входной поток.
`is.read(s,n)`	Считывание не более `n` символов в буфер `s`. Данная функция не полагается на завершающий символ и не добавляет после считанных символов нулевой — т.е. выполняется обычное чтение из файла в буфер без какой-либо обработки. Считываемое количество символов может оказаться меньше `n` при достижении конца файла. Функция возвращает ссылку на входной поток.
`is.ignore(n = 1)`	Считывание не более `n` символов в «никуда» — т.е. считывание без сохранения. По умолчанию количество считываемых символов равно 1, так что вызов `is.ignore()` просто означает «выбросить следующий символ из потока». Функция возвращает ссылку на входной поток.
`is.gcount()`	Возврат количества символов, считанных последней операцией неформатированного ввода из потока.

Форматирование вывода

Поток вывода имеет возможность форматировать вывод. Это делается при помощи набора флагов в классе `ios_base`.

`ios_base::skipws`	Пропуск пробельных символов на входе.
`ios_base::left`	Выравнивание — отступ после значения.
`ios_base::right`	Выравнивание — отступ перед значением.
`ios_base::internal`	Отступ между знаком и значением.
`ios_base::boolalpha`	Вывод символьного представления `true` и `false`.
`ios_base::dec`	Десятичная система счисления для целых чисел.
`ios_base::hex`	Шестнадцатеричная система счисления для целых чисел.
`ios_base::oct`	Восьмеричная система счисления для целых чисел.

`ios_base::scientific`	Число с плавающей точкой в научном формате.
`ios_base::fixed`	Число с плавающей точкой в обычном формате.
`ios_base::showbase`	Префикс `0` перед числом в восьмеричной системе счисления и `0x` перед числом в шестнадцатеричной системе счисления.
`ios_base::showpoint`	Вывод незначащих нулей.
`ios_base::showpos`	Явный '+' перед положительными целыми числами.
`ios_base::uppercase`	'`E`', '`X`' вместо '`e`', '`x`'.
`ios_base::adjustfield`	Используется в функции `setf` для выравнивания полей.
`ios_base::basefield`	Используется в функции `setf` для указания системы счисления.
`ios_base::floatfield`	Используется в функции `setf` для указания формата вывода чисел с плавающей точкой.

Для указания новых флагов и получения старых используются приведенные далее функции; здесь же показаны функции, определяющие некоторые параметры формата вывода.

`os.flags()`	Возвращает текущие флаги форматирования.
`os.flags(f)`	Устанавливает флаги форматирования `f` и возвращает старые флаги.
`os.setf(f[,o])`	Добавляет флаги `f`, т.е. эквивалентно `os.flags(os.flags()|f)`. При использовании сложных флагов, не выражающихся одним битом, аргумент `o` указывает, какое именно форматирование меняется (например, `setf(ios_base::dec,ios_base::basefield)` (установка восьмеричной системы счисления для вывода) или `setf(0,ios_base::floatfield)` (установка универсального формата вывода чисел с плавающей точкой).
`os.unsetf(f)`	Сбрасывает указанный флаг.

`os.width([n])`	Определяет минимальное число символов, которые выведутся *следующей* операцией вывода числа или строки. Можно указать символ-заполнитель при помощи вызова `os.fill(c)`. Еще раз обратите внимание на то, что действие этой функции распространяется только на одну операцию вывода. Функция без параметра возвращает значение, ранее связанное с данным потоком.
`os.precision([n])`	Устанавливает точность для вывода чисел с плавающей точкой (по умолчанию — 6). Действует на все операции вывода до следующего обращения к функции `precision()`. Функция без параметра возвращает значение, ранее связанное с данным потоком.

Указание формата можно выполнять с помощью *манипуляторов* — объектов специального типа, которые применяют функции обратного вызова и при использовании синтаксиса вывода в поток (`<<`) позволяют получить те же результаты, что и при вызове функций, например:

```
cout << left << setw(5) << setfill('#') << 15 << endl;
```

Ниже перечислены основные манипуляторы.

`boolalpha`	Установка флага `ios_base::boolalpha`.
`noboolalpha`	Сброс флага `ios_base::boolalpha`.
`showbase`	Установка флага `ios_base::showbase`.
`noshowbase`	Сброс флага `ios_base::showbase`.
`showpoint`	Установка флага `ios_base::showpoint`.
`noshowpoint`	Сброс флага `ios_base::showpoint`.
`showpos`	Установка флага `ios_base::showpos`.
`noshowpos`	Сброс флага `ios_base::showpos`.
`skipws`	Установка флага `ios_base::skipws`.
`noskipws`	Сброс флага `ios_base::skipws`.
`uppercase`	Установка флага `ios_base::uppercase`.
`nouppercase`	Сброс флага `ios_base::uppercase`.
`internal`	Отступ между знаком и значением.
`left`	Выравнивание — отступ после значения.
`right`	Выравнивание — отступ перед значением.
`dec`	Десятичная система счисления для целых чисел.

`hex`	Шестнадцатеричная система счисления для целых чисел.
`oct`	Восьмеричная система счисления для целых чисел.
`fixed`	Число с плавающей точкой в обычном формате.
`scientific`	Число с плавающей точкой в научном формате.
`endl`	Вывод `'\n'` и сброс буфера.
`ends`	Вывод `'\0'` и сброс буфера.
`flush`	Сброс буфера.
`ws`	Считывание и игнорирование пробельных символов.
`resetiosflags(f)`	Сброс флагов.
`setiosflags(f)`	Установка флагов.
`setbase(b)`	Установка системы счисления для вывода целых чисел.
`setfill(c)`	Указание символа-заполнителя.
`setprecision(n)`	Указание точности для чисел с плавающей точкой.
`setw(n)`	Указание ширины для следующего вывода.

Каждый поток имеет связанное с ним состояние, при помощи проверки которого вылавливаются ошибки и нестандартные условия.

`bool st.good()`	Следующая операция может выполняться потоком `st`.
`bool st.eof()`	Достигнут конец потока `st`.
`bool st.fail()`	Следующая операция потока `st` не выполняется (но никакие символы потока не утрачены).
`bool st.bad()`	Поток `st` в испорченном состоянии.
`void st.clear()`	Сброс флагов состояния потока `st`.
`operator void*()`	Не 0, если `!fail()`. Позволяет организовать цикл наподобие `while(is){...}`.
`bool operator!()`	То же, что и `fail()`.

Другие потоковые объекты

Итераторы входных и выходных потоков объявлены в заголовочном файле <iterator>.

`istream_iterator<T> in(is)`	Определяет объект `in` как итератор входного потока для считывания значений типа `T` из потока `is`.
`ostream_iterator<T> out(os, const char * sep = "")`	Определяет объект `out` как итератор выходного потока для записи значений типа `T` в поток `os`, используя `sep` в качестве разделителя после каждого выводимого элемента (по умолчанию — пустая строка).

Файловые потоки представляют собой потоки, связанные с физическими дисковыми файлами, и объявлены в заголовочном файле <fstream>.

`ifstream is(name, mode = in)`	Определяет потоковый объект `is` и связывает его с файлом с именем `name`, открывая его в режиме `mode` (см. ниже). Класс `ifstream` является наследником `istream`.
`ofstream os(name, mode = out)`	Определяет потоковый объект `os` и связывает его с файлом с именем `name`, открывая его в режиме `mode` (см. ниже). Класс `ofstream` является наследником `ostream`.
`st.open(name, mode)`	Явно открывает файл `name`.
`bool st.is_open()`	Определяет, открыт ли данный файл.
`st.close()`	Закрывает файл. Это неявно делается деструктором потока, так что явный вызов `close()` требуется только для явного закрытия файла до конца области видимости.

Режимы открытия файлов перечислены в классе `ios_base`:

`ios_base::app`	Добавление в конец.
`ios_base::ate`	Открытие и переход к концу файла.
`ios_base::binary`	Ввод-вывод в двоичном виде.
`ios_base::in`	Открытие для чтения.
`ios_base::out`	Открытие для записи.
`ios_base::trunc`	Обрезка файла до нулевой длины.

Список литературы

1. А. Александреску. *Современное проектирование на C++. Серия C++ In-Depth*, т. 3. — М.: Издательский дом «Вильямс», 2002.
2. Д. Вандевурд, Н. М. Джосаттис. *Шаблоны C++: Справочник разработчика*. — М.: Издательский дом «Вильямс», 2003.
3. Э. Кёниг, Б. Э. Му. *Эффективное программирование на C++. Серия C++ In-Depth*, т. 2. — М.: Издательский дом «Вильямс», 2002.
4. Д. Э. Кнут. *Искусство программирования, том 1. Основные алгоритмы, 3-е изд.* — М.: Издательский дом «Вильямс», 2000.
5. Т. Кормен, Ч. Лейзерсон, Р. Ривест. *Алгоритмы: построение и анализ*. М.: МЦНМО, 1999.
6. Г. Саттер. *Решение сложных задач на C++. Серия C++ In-Depth*, т. 4. — М.: Издательский дом «Вильямс», 2002.
7. Б. Страуструп. *Язык программирования C++, 3-е изд.* — СПб.; М.: «Невский диалект» — «Издательство БИНОМ», 1999.
8. H. Sutter. *Exceptional C++ Style*. Addison-Wesley, 2005.
9. H. Sutter, A. Alexandrescu. *C++ Coding Standards*. Addison-Wesley, 2005.
10. M. Wilson. *Imperfect C++. Practical Solutions for Real-Life Programming*. Addison-Wesley, 2005.

Предметный указатель

T

U

V

W

A

Б

Г

Д

Е

З

И

К

Научно-популярное издание

Красикова Ирина Евгеньевна

Красиков Игорь Владимирович

C++

ПРОСТО КАК ДВАЖДЫ ДВА

Ответственный редактор *И.Е. Федосова*
Литературный редактор *А.А. Макиевская*
Художественный редактор *Ю.В. Кузьменко*
Верстка *Р. А. Марчишин*
Корректор *З.В. Александрова*

ООО «Издательство «Эксмо»
127299, Москва, ул. Клары Цеткин, д. 18/5. Тел.: 411-68-86, 956-39-21.
Home page: www.eksmo.ru E-mail: info@eksmo.ru

По вопросам размещения рекламы в книгах издательства «Эксмо» обращаться в рекламный отдел. Тел. 411-68-74.

Оптовая торговля книгами «Эксмо» и товарами «Эксмо-канц»:
ООО «ТД «Эксмо». 142700, Московская обл., Ленинский р-н, г. Видное, Белокаменное ш., д.1. Тел./факс: (095) 378-84-74, 378-82-61, 745-89-16, многоканальный тел. 411-50-74.
E-mail: reception@eksmo-sale.ru

Мелкооптовая торговля книгами «Эксмо» и товарами «Эксмо-канц»:
117192, Москва, Мичуринский пр-т, д. 12/1. Тел./факс: (095) 411-50-76.
127254, Москва, ул. Добролюбова, д. 2. Тел.: (095) 745-89-15, 780-58-34.
www.eksmo-kanc.ru e-mail: kanc@eksmo-sale.ru

Полный ассортимент продукции издательства «Эксмо» в Москве в сети магазинов «Новый книжный»:
Центральный магазин — Москва, Сухаревская пл., 12
(м. «Сухаревская»,ТЦ «Садовая галерея»). Тел. 937-85-81.
Москва, ул. Ярцевская, 25 (м. «Молодежная», ТЦ «Трамплин»). Тел. 710-72-32.
Москва, ул. Декабристов, 12 (м. «Отрадное», ТЦ «Золотой Вавилон»). Тел. 745-85-94.
Москва, ул. Профсоюзная, 61 (м. «Калужская», ТЦ «Калужский»). Тел. 727-43-16.
Информация о других магазинах «Новый книжный» по тел. 780-58-81.

В Санкт-Петербурге в сети магазинов «Буквоед»:
«Книжный супермаркет» на Загородном, д. 35. Тел. (812) 312-67-34
и «Магазин на Невском», д. 13. Тел. (812) 310-22-44.

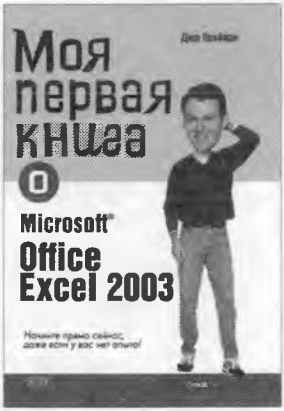
Полный ассортимент книг издательства «Эксмо»:
В Санкт-Петербурге: ООО СЗКО, пр-т Обуховской Обороны, д. 84Е.
Тел. отдела реализации (812) 265-44-80/81/82/83.
В Нижнем Новгороде: ООО ТД «Эксмо НН», ул. Маршала Воронова, д. 3.
Тел. (8312) 72-36-70.
В Казани: ООО «НКП Казань», ул. Фрезерная, д. 5. Тел. (8432) 70-40-45/46.
В Киеве: ООО ДЦ «Эксмо-Украина», ул. Луговая, д. 9.
Тел. (044) 531-42-54, факс 419-97-49; e-mail: **sale@eksmo.com.ua**

Подписано в печать с готовых монтажей 23.09.2005. Формат 84x108 $^{1}/_{32}$.
Печать офсетная. Бумага тип. Усл. печ. л. 8,4.
Доп. тираж 3 000 экз. Заказ № 1377.

Отпечатано в полном соответствии с качеством
предоставленных диапозитивов в ОАО "Тульская типография".
300600, г. Тула, пр. Ленина,109 .

Джо Крайнак

Моя первая книга о Microsoft® Office Excel 2003



Уникальный курс освоения Excel 2003 для начинающих!

Эта книга — ваш первый шаг к созданию информативных и наглядных таблиц в Microsoft Office Excel 2003. В ней изложены основы построения электронных таблиц, а также методика редактирования шаблонов, на основе которых эти таблицы создаются. Помимо этого, вы узнаете, как акцентировать внимание на определенных данных с помощью оригинальных шрифтов, колоритных затенений и обширной коллекции картинок, поставляемых с приложениями Microsoft Office. В данной книге шаг за шагом обсуждаются методы решений наиболее распространенных задач, а изложение материала сопровождается многочисленными рисунками, наглядно поясняющими специфику выполняемых действий. Прочитайте эту книгу, и вы подымитесь на следующую ступеньку — от начинающего до квалифицированного пользователя Microsoft Excel.

Приступайте к работе с Excel 2003, не теряя ни минуты! Эта книга поможет вам:

- Освоить способы создания формул рабочего листа
- Научиться вводить даты, текстовые и числовые значения
- Иллюстрировать данные с помощью картинок и других графических объектов
- Управлять списками и другими базами данных
- Создавать диаграммы
- Печатать электронные таблицы без каких-либо затруднений

ЭКСМО
www.eksmo.ru

Майкл Миллер

Моя первая книга о модернизации и ремонте компьютера

Уникальный курс освоения модернизации и ремонта компьютера для начинающих!

Эта книга призвана помочь в различных ситуациях. Кому-то она поможет разобраться в принципах работы компьютера, кому-то — модернизировать один или все его компоненты. И, наконец, многим эта книга окажется полезной, если что-то в работе пойдет не так — здесь показано, как диагностировать и исправить различные проблемы, возникающие в работе персонального компьютера. Все темы излагаются языком, не перегруженным техническими терминами и понятны даже новичку. Не обязательно быть техническим специалистом, чтобы уметь обслуживать компьютер. В книге приводятся практические советы, а также пошаговые инструкции для проведения модернизации и ремонта компьютера.

Приступайте к модернизации компьютера, не теряя ни минуты! Эта книга поможет вам:

- Разобраться в принципах работы компьютера
- Диагностировать и исправить различные проблемы
- Модернизировать один или все компоненты компьютера
- Узнать основы обслуживания системы
- Создать небольшие проводные и беспроводные сети
- Установить операционную систему Windows XP

ЭКСМО
www.eksmo.ru

Шелли О'Хара

Моя первая книга о Microsoft® Windows® XP

Уникальный курс освоения Windows XP для начинающих!

Если вы еще не знакомы с Windows XP или вообще с операционной системой Windows, эта книга определенно для вас. Она написана простым, понятным языком и содержит пошаговые объяснения всех ключевых моментов использования Windows XP. Вы научитесь запускать программы, отправлять сообщения электронной почты, воспроизводить музыку, печатать документы и многое другое. Книга содержит массу полезных советов и подробных инструкций. Если вы хотите научиться работать с Windows XP, но не знаете, с чего начать, эта книга — именно то, что вам нужно.

Приступайте к работе с Windows XP, не теряя ни минуты! Эта книга поможет вам:

- Установить Windows XP и ознакомиться с основной терминологией, используемой для обозначения распространенных действий
- Научиться запускать программы, создавать и сохранять документы
- Подключиться к Интернет
- Научиться отправлять и получать сообщения электронной почты с помощью Outlook Express и бесплатной почтовой системы Mail.Ru
- Обмениваться мгновенными сообщениями с помощью программы @Mail.Ru Agent (М-Агент)
- Воспроизводить музыку и смотреть фильмы с помощью проигрывателя Windows Media, а также работать с цифровыми фотографиями

ЭКСМО
www.eksmo.ru