ОБЪЕКТНО-
ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ
НА ЯЗЫКЕ
C++

2-е издание
А.Л. Фрицман

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C++

2-е издание

МОСКВА
ГОРЯЧАЯ ЛИНИЯ - ТЕЛЕКОМ
2001
Фридман А.Л.


В систематизированном виде излагаются основы объективно-ориентированного программирования, даются основные понятия и описываются возможности языка Си++. При этом основное внимание уделяется объяснению того, как теми или иными возможностями пользоваться. В конце книги помещен краткий справочник по языку Си++, в котором перечисляются все основные конструкции языка.

Для программистов начинающих изучать объективно-ориентированное программирование и язык Си++.

ББК 32.973
1. Введение

1.1. История языка C++

Разработчик языка C++ Бьярн Страуструп работал над его созда-
нием почти 20 лет. Первые работы по языку C с классами — предшест-
веннику C++, появились в 1979 году. Основные работы велись в иссле-
dовательском центре компании Bell Labs. Там же был разработан первый
компилятор языка — Cfront. В работе над языком принимали участие ве-
дущие практики и теоретики программирования. Первоначально язык су-
ществовал как академический, экспериментальный, но постепенно эволю-
ционировал в полномасштабный развитый язык промышленного
программирования. В 1997 году был принят международный стандарт
C++, который фактически подвел итоги 20-летнего развития.

Ниже приведена краткая хронология важнейших этапов развития
языка [13].

1979 г. Первая реализация компилятора языка C с классами — раннего
варианта C++.

1982 г. Первая публикация по языку C с классами.

1983 г. Появление названия C++.
Первая реализация компилятора C++.

1984 г. Первый справочник по языку C++.

1985 г. Компилятор Cfront фирмы AT&T версия 1.0 (первая коммерче-
ская версия)
Выход книги "The C++ Programming Language" Бьярна Страу-
струпа.

1986 г. Первая конференция OOPSLA, ставшей главной конференцией
по объектно-ориентированному программированию в мире.
Первый коммерческий перенос компилятора Cfront на персо-
нальный компьютер.

1987 г. Компилятор Cfront версия 1.2.
Первая конференция USENIX, специально посвященная языку
C++.
Появление компилятора GNU C++ (версия 1.13).

1988 г. Выпуск компилятора C++ фирмой Oregon Software.
Выпуск компилятора C++ фирмой Zortech.
Конференция USENIX по реализации языка C++.

1989 г. Версия 2.0 компилятора Cfront
Организационное собрание комитета ANSI X3J16 по стандарти-
зации языка C++ — начало процесса стандартизации.
1990 г. Выпуск компилятора Си++ фирмой Borland.
Выход книги "The Annotated C++ Reference Manual" Маргарет Эллис и Бьёрна Страуструпа — до появления официального стандарта фактический стандарт языка. Часто ее называли просто "Книга" (The Book) или ARM.
К проекту стандарта добавлены шаблоны и исключительные ситуации.

Версия 3.0 компилятора Cfront, в которой реализованы шаблоны.

1992 г. Выпуск компилятора Си++ фирмой DEC.
Выпуск компилятора Си++ фирмой Microsoft.
Выпуск компилятора Си++ фирмой IBM.

1993 г. Зарегистрирован проект стандарта ANSI/ISO.

1994 г. Стандартная библиотеки шаблонов STL включена в проект стандарта языка.

Выход третьего издания книги "The C++ Programming Language".

1.2. Назначение Си++ и области его использования

Язык Си++ — это универсальный язык программирования. Двумя его основными предшественниками были языки Си и Симула. Важнейшие свойства объектно-ориентированного программирования в языке Си++ являются развитием идей, заложенных в языке Симула. Язык Си был создан Деннисом Ричи прежде всего для операционной системы Unix. Постепенно Си пробрел широкое признание, появилась его реализация для практически всех операционных систем, и он стал стандартным языком системного и во многом прикладного программирования. От Си язык Си++ унаследовал эффективность, необходимую для системного программирования, начиная от написания драйверов внешних устройств до разработки компиляторов и систем управления базами данных.

Совместимость с Си обусловлила еще одну важную область применения Си++ — программирование сложных графических пользовательских интерфейсов. Интерфейс прикладных программ с такими графическими средами, как X Window и Microsoft Windows были разработаны прежде всего для Си. С развитием и распространением Си++ разработка графических пользовательских интерфейсов перешла к нему.

К настоящему времени ситуация с использованием различных языков продолжает меняться. Средства быстрой разработки графических интерфейсов типа Visual Basic, Powerbuilder и т.п. удовлетворили существ-
венную часть потребности в разработке простых интерфейсов. Языки Java в последнее время стал все более широко употребляться для разработки интерфейсов. Тем не менее Си++ позволяет разрабатывать наиболее сложные программы, поскольку с его помощью можно использовать всю мощь базовой графической среды.

Обработка сложных структур данных — текста, бизнес-информации, графических образов — еще одна область применения Си++, обусловленная его объектно-ориентированными свойствами. Объекты языка отлично подходят для представления сложных структур данных и манипуляций с ними. Для хранения сложных структур данных можно использовать объектно-ориентированные системы управления базами данных, интерфейсы которых опять-таки рассчитаны на Си++.

Построение распределенных систем, сетей, сопряжение различных программ с помощью таких стандартов, как COM/DCOM, CORBA, идеально подходит для Си++.

В общем, везде, где требуется высокоэффективная программа, работающая со сложными структурами данных, естественно выбрать язык Си++.

Кроме эффективности, мощности и выразительности языка, преимуществом Си++ является возможность многократного использования программ и модулей. Для каждой из перечисленных областей и многих других, не упомянутых здесь, на языке Си++ можно создать классы и библиотеки классов, реализующие функциональность часто встречающихся объектов данной области. (Мы будем обсуждать принципы и некоторые возможности создания библиотек классов в последующих главах.)

Например, для обработки текстов можно создать классы для работы со строками, словами, параграфами, шрифтами и т.д. Для графических систем — библиотеки различных форм, окон, движений и т.п. Объектная модель обуславливает удобство создания многократно используемых элементов и, что еще более существенно, легкость их адаптации к конкретной задаче.

Сказанное не означает, что другие языки не имеют права на существование. Во-первых, существует большой класс задач, для которых самое главное — быстрота разработки, а эффективность, а иногда и даже надежность, не играют большой роли. Для их решения имеется большое количество средств быстрой разработки графических интерфейсов (уже упоминавшиеся Powerbuilder и другие), интерпретируемые языки разработки командных файлов — Perl, Tcl и пр. Во-вторых, существуют специализированные языки в определенных предметных областях. В-третьих, для составления программ, которые должны работать на разных машинах без перекомпиляции (особенно для Интернета) все больше используется Java.

Чтобы не тратить слишком много места, скажем, что существует масса языков программирования, и почти каждый из которых способен решить поставленную задачу. Выбор зависит от самой задачи, квалификации и знаний программистов, других программ, с которыми данная программа взаимодействует и других факторов. Просто Си++ при прочих равных лучше.
Как мы уже отмечали, язык Си++ — это универсальный язык программирования. С его помощью можно разрабатывать программы, используя разные стили программирования: процедурный, объектно-ориентированный, параметрический. Процедурное программирование поддерживается тем, что программа на языке Си++ состоит из функций, которые вызывают друг друга. В таком варианте Си++ используется как "улучшенный" Си. Тем не менее в определенных обстоятельствах процедурный подход имеет право на существование. Параметрическое программирование поддерживается механизмом шаблонов. Объектно-ориентированный подход обеспечивается механизмом классов. В данной книге мы будем уделять основное внимание объектно-ориентированному стилю программирования на языке Си++.

1.3. Стандарт языка

Принятие международного стандарта языка Си++ — чрезвычайно важное событие. Использование любого языка программирования страдает от того, что вместо одного общего языка начинают использовать массу дилектов. Хотя благодаря активной позиции и высокому авторитету Страуструпа, вариантов языка Си++ было не так уж много, тем не менее почти каждая компания, разрабатывая компилятор Си++, добавляла или исключала те или иные возможности. Страдал в конечном итоге рядовой программист, для которого перехож д одного компилятора на другой, а иногда даже переход с одной версии того же самого компилятора на другую, выливался в серьезную переработку программы. С наличием стандарта, если и не положен конец вариантам, то во всяком случае становиться понятно, какие свойства языка можно ожидать реализованными на всех компиляторах, а какие — только на каком-то особенном. Поэтому, если и используются нестандартные свойства, то используются осмысленно, с пониманием всех последствий.

Не менее важным результатом стандартизации явилось то, что в процессе выработки и утверждения стандарта, язык был уточен и дополнен рядом существенных возможностей. В обсуждении стандарта приняло участие огромное количество программистов, как практиков, так и теоретиков. Комитет по стандарту языка объединил действительно талантливых людей, обладающих, что немаловажно, чувством стиля и меры. В целом процесс стандартизации способствовал популяризации Си++, а сам язык стал мощнее.

Последнее техническое заседание комитета состоялось в конце 1997 года. На настоящее время стандарт утвержден Международной организацией по стандартизации ISO. Его номер ISO/IEC 14882. ISO распространяет стандарты только за плату, поэтому официально утвержденного стандарта не имеется в свободном доступе на Интернете. За плату его можно получить на узле американского национального комитета по стандартам в информационных технологиях www.ncits.org. Однако проекты стандарта, в том числе и одни из самых последних, имеются, и к ним можно обратиться по адресу:

или же через страницу вопросов и ответов Бьерна Страуструпа:

http://www/research.att.com/~bs/bs_faq.html

Россия является членом Международной организации по стандартизации ISO. Поэтому принятие стандарта ISO означает автоматическое принятие его в России. Процессом стандартизации и распространением стандартов в России занимается ВНИИ Сертификации (адрес узла в Интернете: www.vniis.ru).
2. Объектная модель языка Си++

2.1. Понятие объекта

Язык программирования Си++ реализует объектно-ориентированный подход к программированию. В сознании значительной части людей объектно-ориентированный подход к программированию ассоциируется прежде всего с используемым языком программирования. Если проект написан на Си++, Яве или Смолток - значит используется объектный подход. Я бы сказал иначе - если проект пишется на языке Си++ или Смолток, значит возможно использование объектного подхода.

Что-же составляет объектно-ориентированный подход к программированию? Это последовательное использование объектной модели предметной области на всех этапах разработки программного обеспечения. Ниже мы описем основные понятия объектной модели. Эти понятия не зависят ни от используемого языка программирования, ни от применяемой технологии разработки программных систем. Разумеется, базовые понятия по-разному реализуются в разных языках программирования, поэтому говорят не просто об объектной модели, но объектной модели конкретного языка программирования (в нашем случае – язык Си++).

В центре объектной модели естественно стоит понятие объекта. Объект — это настолько общее понятие, что довольно сложно дать ему четкое и приемлемое для всех определение. Ивар Якобсон [6] дает следующее определение: объект — это сущность, способная сохранять свое состояние (информацию) и обеспечивающая набор операций (поведение) для проверки и изменения этого состояния. Объект характеризуется набором операций и состоянием, которое сохраняет результат этих операций.

Предположим, мы разрабатываем библиотечную систему. Одним из объектов в этой системе будет читатель. Читатель характеризуется своим именем и списком книг, взятых в библиотеке. Читатель может сдать книгу или взять новую. В такой модели объект читатель имеет состояние, состоящее из двух атрибутов — имя (неизменяемое) и список книг (изменяемый) — и две операции — сдать и взять. Операции соответственно изменяют список книг у данного читателя.

Целым система будет состоять из множества объектов: читатели, книги, хранилище, библиотекарь и т.д. Эти объекты взаимодействуют между собой, посылая друг другу сообщения. Например, читатель может получить сообщение о необходимости сдать книгу (поскольку подходит к концу срок, на который она выдана), и в ответ он выполнит операцию "сдать". Другим примером сообщения может быть вопрос библиотекаря к читателю: "Как Вас зовут?". Чтобы правильно отреагировать на такое сообщение, объекту "читатель" необходимо иметь операцию "называть свое имя".

В системе обычно функционируют множество объектов одного класса. Читатели Иванов, Петров и Сидоров — экземпляры класса "читатель". Все экземпляры одного и того же класса имеют один и тот же
набор операций и могут реагировать на один и те же сообщения. Класс иногда называют типом объектов. Класс в определенном смысле соответствует понятию абстрактного типа данных, введенного в программировании довольно давно — еще в языке Модула-2.

Класс могут наследовать свойства других классов. В нашем примере два класса: библиотекарь и читатель — имеют общее. Они являются людьми, т.е. они принадлежат к классу "человек". Атрибут "имя" и операция "сказать свое имя" — свойства человека. Эти свойства наследованы классами библиотекарь и читатель от класса "человек".

Предположим, что нашей библиотекой пользуются студенты и преподаватели университета, при этом у них немного разные права. Например, преподаватели могут одновременно брать больше книг, чем студенты. В рамках объектной модели этот факт можно отобразить следующим образом: введем два класса — "читатель-преподаватель" и "читатель-студент". Они наследуют (являются подклассами, или выведены из) класса "читатель". У них общий интерфейс — набор операций — однако действия при выполнении этих операций могут быть различны. Кроме того, при дальнейшей разработке системы нам понадобятся различные операции или атрибуты для студентов и преподавателей.

Таким образом классы образуют иерархию, показанную на рис. 1.

Рис.1. Пример иерархии классов и библиотечной системы.

На рис. 1 видно, что операции "Взять" и "Вернуть" определены в классе "читатель" и в классах "студент" и "профессор". Это построение отражает тот факт, что операции, хотя и обладают один и тем же интер-
файлом, реализованы по-разному: Например, у студента имеется ограничение на количество взятых книг, а у профессора его нет.

Различие операций "Взять" и "Вернуть" в классах "профессор" и "студент" иллюстрирует еще одно важное свойство объектной модели — полиморфизм. Полиморфизм позволяет работать с неким объектом, не зная конкретного класса, к которому этот объект относится. Для того, чтобы послать сообщение "сдать" нам не важно, кем является данный читатель — студентом или преподавателем. Однако сама операция сдачи книг будет происходить по-разному в зависимости, кто именно этот читатель.

2.2. Что такое объектно-ориентированное программирование?

Простейший ответ на вопрос "Что такое объектно-ориентированное программирование?" — это программирование с использованием объектов. Но пожалуй такой ответ слишком краток, чтобы реально прояснить, почему объектно-ориентированное программирование так популярно и почему оно позволяет лучше решать задачи современного программирования.

Что такое программа? Это модель некой другой реально существующей или искусственной системы. Если мы разрабатываем библиотечную систему, цель которой учитывать взятые и сданные книги в библиотеке и вести каталог, то мы моделируем реальную библиотеку с книгами и живых читателей. Если мы разрабатываем программу решения систем уравнений, то мы моделируем абстрактную математическую систему.

Модель в данном случае — это описание задачи и способа ее решения на языке программирования. Как и на всяком другом языке, описывать можно по-разному. Можно описывать в терминах действий. Чтобы решить поставленную задачу, надо сделать то-то и то-то. В случае решения системы уравнений, подобное описание работает хорошо. Мы точно знаем последовательность всех действий, необходимых для решения системы уравнений, точно знаем алгоритм решения. В случае библиотечной системы — это не так. Действия вполне разнородны и разные действия делают разные действия. В математике: действия изменяются с течением времени, а значит, они не являются постоянными.

Попробуйте описать толпу на Тверском бульваре. Кто-то сидит на скамейке, кто-то идет. Две читают газеты, четверо книг, а пятьдесят человек просто глядят на прохожих. Действия ежесекундно меняются и тот кто читал книгу уже идет покупать мороженое. Конечно, можно разделить людей по группам — кто идет, кто читает, как мороженое покупает, но такое разделение будет неполным. Действия меняются, нужно же описать, как происходит взаимодействие с машинами, магазинами, киосками и т.д. Одним словом, начинает ощущаться отсутствие литературного таланта, поскольку мы еще не дошли до того, чем это толпа отличается от толпы на Крещатике, Невском проспекте или Елисеевских полях.

Попробуем подойти с другой стороны. Толпа состоит из людей. У всех людей есть совокупность общих черт и индивидуальные отличия.
Есть провокация части и тротуар. Есть машины и регулировщики, киоски и т.д. Если мы начнем описывать, начиная с находящихся на бульваре людей и объектов, то это описание будет менее увлекательное, зато более простое, более узнаваемое, даже при отсутствии особых литературных дарований.

Аналогичное сравнение применимо к разработке сложной программы. Когда мы разрабатываем программную систему, мы разрабатываем программную модель предметной области — части реального мира. Самая большая сложность состоит в так называемом семантическом разрыве между реальностью и программой. Программа и предметная область описываются в разных терминах, в разных понятиях.

Объектная модель уменьшает этот разрыв. Реальный мир естественно описывать как набор взаимодействующих объектов. Такое описание с одной стороны, понятно конечному потребителю и эксперту, с другой стороны, легко ложится на объектную модель, а следовательно, его легко реализовать.

Программная система должна постоянно изменяться, приспосабливаясь к изменяющимся требованиям и задачам. Чаще всего изменяются не сами объекты, участвующие в системе, а методы их взаимодействия. В этом случае основа системы остается без изменения и тем самым реализация изменений оказывается существенно проще. Если же изменения касаются какого-либо типа объектов, они будут локальны к данному классу.

В современном программировании одна из самых насущных проблем заключается в необходимости ускорения разработки. Объектно-ориентированный подход стимулирует многократное использование программного обеспечения. Количество существенно различных объектов не так уж и велико. Иногда можно использовать имеющиеся классы в готовом виде. Можно использовать объекты в качестве компонентов других объектов. Механизм наследования классов позволяет использовать имеющиеся классы в качестве базовых, из которых выводятся новые, специализированные на конкретное применение. Кроме того, широкое распространение объектно-ориентированного подхода привело к огромному числу предлагаемых готовых библиотек классов, как универсального характера, так и ориентированных на различные сферы бизнеса.

Существенный уровень ускорения разработки обеспечивается ещё и путём использования готовых, многократно проверенных классов. Это уменьшает количество ошибок. Структура программ, основанная на объектах, отображающих объекты реального мира, уменьшает вероятность появления побочных неожиданных эффектов от изменений. Также здесь можно повторить аргументы в справочниках программиста.

Разумеется, все это не делается бесплатно. При разработке системы необходимо приложить определенные усилия, особенно направленные на многократное использование кода.

И здесь опять объектная ориентация показывает свое преимущество тем, что существует объектно-ориентированные методы для всех этапов разработки — начиная от анализа требований заказчика и кончая тестированием и поддержкой исходных текстов.
Использование самой передовой методики не гарантирует от неудач. Один из самых больших провалов в истории разработки программных комплексов — система обработки багажа в новом аэропорту города Денвера, которая должна была быть реализована как полностью объектно-ориентированная система.

Основные проблемы объектно-ориентированной разработки — трудоемкость и сложность, особенно на начальном этапе перехода от, например, структурного программирования.

Объектно-ориентированные языки программирования сложнее процедурных. Для освоения языка Си++ требуется по крайней мере в 2-3 раза больше времени, чем для освоения языка Си. Библиотеки классов экономят время разработки, но опы-таки требуют время на изучение.

Тем не менее, опыт показывает, что преодоление всех этих трудностей себя полностью оправдывает. Компании, переходящие на объектно-ориентированную разработку, вначале вкладывают дополнительные средства на обучение. По мере освоения нового и накопления собственного багажа проектных решений и программного обеспечения, процесс разработки ускоряется и удешевляется, и выгоды намного перекрывают начальные затраты.

2.3. Класс, тип и объект в языке Си++

В параграфе 2.1 мы рассмотрели общие понятия объектно-ориентированного программирования. В каждом конкретном языке программирования эти понятия реализованы по-разному. Иногда говорят, что у каждого языка имеется своя объектная модель.

Каковы же основные свойства объектной модели языка Си++?

1. Язык Си++ — это строго типизированный язык. Каждая величина, с которой имеет дело программа, принадлежит к какому-либо типу. Тип объекта определен статически, т.е. объект, будучи однажды созданным, не может изменять свой тип. Преобразования типов в выражениях допустимы, но правила преобразования достаточно строгие.

2. Тип в языке Си++ — либо встроенный тип, т.е. заранее определенный (целые и вещественные числа, логические значения и т.д.) либо тип, определенный программистом — класс.

3. При определении нового типа — класса, его можно сконструировать таким образом, что использование объектов этого класса в программе практически ничем не будет отличаться от использования стандартных встроенных типов. Это достигается с помощью механизма переопределения операций, таких как сложение, присваивание и т.d.

4. Язык Си++ поддерживает как обычное одинарное наследование, так и множественное наследование. У класса может быть несколько базовых классов.

5. Полиморфизм в языке Си++ реализован с помощью механизма виртуальных методов. Базовый класс определяет лишь интерфейс объекта, задавая набор виртуальных методов. Реализация же виртуальных методов происходит в порожденных классах.
Как неоднократно отмечал создатель Си++ Бьёрн Страуструп, язык Си++ поддерживает не только объектно-ориентированное программирование, но и другие стили программирования. Он может быть процедурно-ориентированным, когда в качестве основного элемента программы выступают процедуры или функции. Он поддерживает параметрическое программирование с помощью шаблонов. При таком подходе в программе определяются обобщенные процедуры и структуры данных, в которые затем подставляют параметры — конкретные типы и объекты — для получения нужных результатов. Во многом именно своей широте и универсальности язык Си++ и завоевал такое широкое признание.

В данной книге мы опишем все свойства и возможности языка, однако постараемся это сделать именно с позиций применимости в объектно-ориентированном программировании.
3. Начальные сведения о языке

3.1. Простейшая программа

Самая короткая программа на языке C++ выглядит так:

```cpp
// Простейшая программа
define main() { }
```

Первая строчка в программе — комментарий, который служит лишь для пояснения. Признаком комментария являются два знака деления подряд (//).

`main` — это имя главной функции программы. С функции `main` всегда начинается выполнение. У функции есть имя (`main`), после имени в круглых скобках перечисляются аргументы или параметры функции (в данном случае у функции `main` нет аргументов). У функции может быть результат или возвращаемое значение. Если функция не возвращает никакого значения, то это обозначается ключевым словом `void`. В фигурных скобках записывается тело функции — действия, которые выполняет данная функция. Пустые фигурные скобки означают, что никаких действий не производится. Таким образом, вышеприведенная программа ничего не делает!

Если мы говорим об объектно-ориентированном программе (а именно и таких программах мы будем говорить в этой книге), то программа должна создать объект какого-либо класса и послать ему сообщение. Чтобы не усложнять программу, мы воспользуемся одним из готовых, предопределенных классов — классом `iostream` (поток ввода-вывода). Этот класс определен в файле заголовков "iostream.h". Поэтому первое, что надо сделать — включить файл заголовков в нашу программу:

```cpp
#include <iostream.h>
definite main() { }
```

Кроме класса, файл заголовков определяет глобальный объект этого класса `cout`. Объект называется глобальным, поскольку к нему возможен доступ из любой части программы. Этот объект выполняет вывод на консоль. В функции `main` мы можем к нему обратиться и послать ему сообщение:

```cpp
#include <iostream.h>
definite main()
{
    cout << "Hello world!" << endl;
}
```
Операция сдвига << для класса ostream определена как "вывести". Таким образом, программа посылает сообщения объекту cout "вывести строку Hello world!" и "вывести перевод строки" (endl обозначает новую строку). В ответ на эти сообщения объект cout выведет строку "Hello world!" на консоль и переведет курсор на следующую строку.

В гл. 9 мы рассмотрим операции ввода-вывода с потоками более подробно.

3.2. Компиляция и выполнение программы

Программа на языке Си++ (как, впрочем, и на большинстве современных языков программирования) – это прежде всего текст. С помощью произвольного текстового редактора программист записывает инструкцию, в соответствии с которой компьютер будет работать, выполняя данную программу.

Для того, чтобы компьютер мог выполнить программу, написанную на языке Си++, её нужно перевести на язык машинных инструкций. Эту задачу выполняет компилятор. Компилятор читает файл с текстом программы, анализирует её, проверяет на предмет возможных ошибок и, если таковых не обнаружено, создает выполняемый файл, т.е. файл с машинными инструкциями, который можно выполнять.

Один раз откомпилировав программу, её можно выполнять многократно, с различными исходными данными.

Мы настоятельно рекомендуем выполнять упражнения и проверять примеры на компьютере во время чтения этой книги. Способ компиляции и выполнения программ естественно зависит от того, какую операционную систему и компилятор вы используете.

Не имея возможности описать все варианты, остановимся только на двух наиболее часто встречающихся.

3.2.1. Компилярирование и выполнение программ в среде Windows

Если Вы используете персональный компьютер с операционной системой Microsoft® Windows 98™, Windows NT™ или Windows 2000™, то компилятор у Вас скорее всего Visual C++®. Этот компилятор представляет из себя интегрированную среду программирования, т.е. объединяет текстовый редактор, компилятор, отладчик и еще ряд дополнительных программ. Мы предполагаем, что читатель работает с версией 5.0 или старше. Версии младше 4.2 не имеет смысла изучать, поскольку реализация слишком сильно отличается от стандарта языка.

В среде Visual C++ прежде всего Вам необходимо создать новый проект. Для этого выберите в меню File атрибут New. Появится новый диалог. В закладке "Projects" в списке различных типов выполняемых файлов выберите "W32 Consol Application". Убедитесь, что отмечена кнопка "Create new workspace". Затем наберите имя проекта (например test) в поле "Project name" и наберите имя каталога, в котором Вы будете
хранить все файлы, относящиеся к данному проекту, в поле “Location”. По-
сле этого нажмите кнопку “OK”.

Теперь необходимо создать файл. Опять в меню “File” выберите атрибут “New”. В появившемся диалоге в закладке “File” отмquete “text file”. По умолчанию новый файл будет добавлен к текущему проекту test, в чем
Вы можете убедиться, взглянув на поле “Add to project”. В поле “File name”
наберите имя файла. Пусть это будет main.cpp. Расширение .cpp – это
стандарт для файлов с исходными текстами на языке C++. Поле “Location” должно показывать на каталог C:\Work. Нажмите кнопку “OK”.

На экране появится пустой файл. Наберите текст программы из
предыдущего параграфа.

Компиляция выполняется с помощью меню “Build”. Выберите пункт
“Build test.exe” (этому пункту меню соответствует функциональная клави-
ша F7). В нижней части экрана появятся сообщения компиляции. Если Вы
сделали какую-нибудь опечатку, двойной щелчок мышью по строке с
ошибкой переводит курсор в окне текстового редактора на соответствую-
щую строку кода. После исправления всех ошибок и повторной компиля-
ции, Вы получите сообщение об успешной компиляции и компоновки (пока
мы не будем углубляться в подробности, просто вы увидите сообщение
“Linking”).

Готовую программу можно выполнить с помощью меню “Build”,
пункт “Execute test.exe”. То же самое можно сделать нажав одновременно
клавиши CTRL и F5. На экране монитора появится консольное окно и в
нем будет выведена строка “Hello world!”. Затем появится надпись “Press
any key to continue”. Эта надпись означает, что программа завершилась и
лишь ожидает нажатия произвольной клавиши для того, чтобы закрыть
консольное окно.

3.2.2. Компиляция и выполнение программ в среде
Unix

Если Вы работаете в операционной системе Unix, то скорее всего у Вас нет интегрированной среды разработки программ. Вы будете пользу-
ватьсяся любым доступным вам текстовым редактором для набивки текстов
программ.

Редактор Emacs предпочтительнее, поскольку в нем есть специ-
альный режим редактирования программ на языке C++. Этот режим
включается автоматически при редактировании файла с именем оканчи-
вающимся на “.cpp” или “.h”. Но при отсутствии Emacs сгодится любой
текстовый редактор.

Первое, что надо сделать – поместить текст программы в файл. В
редакторе создайте файл с именем “main.cpp” (расширение .cpp-использу-
ется для текстов программ на языке C++): Наберите текст программы из
предыдущего параграфа и сохраните файл.

Теперь программу надо откомпилировать. Команда вызова компи-
лятора зависит от того, какой компилятор C++ установлен на Вашем
компьютере. Если используется компилятор GNU C++, команда компиля-
ции выглядит так:
Вместо gcc может использоваться g++, c++, cc. Уточнить Вы можете у Вашего системного администратора. Отметим, что у широко распространенного компилятора GNU C++ есть ряд отличий от стандарта ISO.

В случае каких-либо ошибок в программе, компилятор выдаст сообщение на терминал с указанием номера строки, где обнаружена ошибка. Если Вы не сделали никаких опечаток в программе, компилятор должен создать выполняемый файл с именем a.out. Выполнить его можно просто набрав имя a.out в ответ на подсказку интерпретатора команд:

```bash
> a.out
```

Результатом выполнения будет вывод строки экран терминала:

```
Hello world!
```

3.3. Структура программы на языке C++

В данном разделе даются самые начальные сведения о структуре программы. Читатели, знакомые с языком Си, большей частью могут его пропустить.

3.3.1. Запись программ

Для записи программ используются большие и маленькие латинские буквы, цифры, знаки препинания и ряд специальных символов. Знаки других алфавитов, в том числе русского, можно использовать в качестве комментариев, символьных констант и литералов (см. ниже).

Кроме инструкций компьютеру, в текст программы принято вставлять инструкции человеку, который будет читать эту программу. Любой текст, начинающая с двух знаков деления и до конца строки является комментарием, никак не анализируются компьютером и служат лишь для пояснений. Кроме того, любой текст, заключенный между символами /* и */ также является комментарием.

```cpp
текст программы // комментарий
текст программы
*/
комментарий
*/
текст программы /* комментарий */ текст программы
```

Написание комментариев — не просто причуда и даже не признак хорошего тона. Это необходимость. Программа должна жить, т.е. изменяться, припособливаться к новым условиям, в ней должны исправляться найденные ошибки. Даже если сама по себе программа не изменяется, а
лишь используется, часто необходимо понимать её особенности и почему она делает то или другое.

Комментарии и призваны помочь программисту в таких случаях, особенно если изменяет программу не тот, кто её изначально написал.

### 3.3.2. Имена

Для символического обозначения величин, имен функций и т.п. используются имена или идентификаторы.

Идентификаторы в языке C++ — это последовательность знаков, начинающаяся с буквы. В идентификаторах можно использовать заглавные и строчные латинские буквы, цифры и знак подчеркивания. Длина идентификаторов произвольна. Примеры правильных идентификаторов:

```
abc  A12  NameOfPerson  BITES_PER_WORD
```

Отметим, что abc и Abc — два разных идентификатора, т.е. заглавные и строчные буквы различаются. Примеры неправильных идентификаторов:

```
12x   a-b
```

Ряд слов в языке C++ имеет особое значение и не может использоваться в качестве идентификаторов. Такие зарезервированные слова называются ключевыми. Один пример ключевого слова мы уже встречали: `void`. Полный список ключевых слов приведен в раздел 12.1.5.

Далее в этой книге мы будем выделять ключевые слова жирным шрифтом. Некоторые C++ ориентированные редакторы (например, emacs или редакторы, входящие в состав Visual C++ или Borland C++) выделяют ключевые слова другим цветом. Это делается только для наглядности и удобства чтения, в самих же файлах с программами эти слова никак не выделяются.

### 3.3.3. Переменные и константы

Программа манипулирует с информацией, представленной в виде различных объектов и величин. Переменная — это символическое обозначение величины в программе. Как ясно из названия, значение переменной (или величина, которую она обозначает) может изменяться во время выполнения программы.

С точки зрения архитектуры компьютера, переменная — это символическое обозначение ячейки оперативной памяти программы, в которой хранятся данные. Содержание этой ячейки — это текущее значение переменной.

В языке C++ прежде чем использовать переменную, её необходимо объявить. Объявить переменную с именем `x` можно так:

```
int x;
```
В объявлении первым стоит название типа переменной int (целое число), а затем идентификатор x — имя переменной. У переменной x есть тип — в данном случае целое число. Тип переменной определяет, какие возможные значения эта переменная может принимать и какие операции можно выполнять над данной переменной. Тип переменной изменить нельзя, т.е. пока переменная x существует, она всегда будет целого типа.

Язык C++ — это строго типизированный язык. Любая величина, используемая в программе, принадлежит к какому-либо типу. При любом использовании переменных в программе проверяется, что выражение или операция применима к типу переменной. Довольно часто смысл выражения зависит от типа участвующих в нем переменных.

Например, если мы запишем x+y, где x — объявлена выше переменная, то переменная y обязана быть одного из числовых типов.

Соответствие типов проверяется во время компиляции программы. Если компилятор обнаруживает несоответствие типа переменной и ее использования, он выдаст ошибку (или предупреждение). Однако во время выполнения программы типы не проверяются. Такой подход, с одной стороны, позволяет обнаружить и исправить большое количество ошибок на стадии компиляции, а с другой стороны не замедляет выполнения программы.

Переменной можно присвоить какое-либо значение с помощью операции присваивания. Присвоить — это значит установить текущее значение переменной. По другому можно объяснить, что операція присваивания запоминает новое значение в ячейке памяти, которая обозначена переменной.

```c++
int x; // объявить целую переменную x
int y; // объявить целую переменную y
x = 0;  // присвоить x значение 0
y = x + 1; // присвоить y значение x + 1, т.е. 1
x = 1;  // присвоить x значение 1
y = x + 1; // присвоить y значение x + 1, теперь уже 2
```

В программе можно явно записать величину — число, символ и т.п. Например, мы можем записать выражение x + 4 — сложить текущее значение переменной x и число 4. В зависимости от того, при каких условиях мы будем выполнять программу, значение переменной x может быть различно. Однако целое число четыре всегда останется четырьмя. Это неизменяемая величина или константа.

Таким образом, явная запись значения в программе — это константа.

Далеко не всегда удобно явно записывать константы в тексте программы. Гораздо чаще используются символические константы. Например, если мы запишем:

```c++
const int BITS_IN_WORD = 32;
```
то затем имя BITS_IN_WORD можно использовать вместо целого числа 32.

Выгоды от такого использования очевидны. Во-первых, имя
BITS_IN_WORD (битов в машинном слове) дает хорошую подсказку, для
чего используется данное число. Без всяких комментариев понятно, что
выражение

\( b / \text{BITS_IN_WORD} \)

(значение \( b \) разделить на число 32) вычисляет количество машинных
слов, необходимых для хранения \( b \) битов информации. Во-вторых, если
по каким-либо причинам нам надо изменить эту константу, нужно будет
изменить только одно место в программе — определение константы, оста-вив
все случаи ее использования как есть. (Например, мы переносим про-
грамму на компьютер с другой длиной машинного слова)

3.3.4. Операторы

Запись действий, которые должен выполнить компьютер, состоит из
операторов. Различаются операторы объявления имен, операторы управ-
ления и операторы-выражения.

3.3.4.1. Объявления имен

С операторами объявления имен мы уже сталкивались в предыдущем параграфе. Эти операторы объявляют имена, т.е. делают их известными программе. Все идентификаторы или имена, используемые в программе на языке Си++ должны быть объявлены (за исключением имени функции main, с которой всегда начинается выполнение).

Оператор объявления состоит из названия типа и объявляемого имен:

```c
int x; // объявить целую переменную x
double f; // объявить переменную f типа double
cost float pi = 3.1415;
// объявить константу pi типа float со значением 3.1415
```

Оператор объявления заканчивается точкой с запятой.

3.3.4.2. Операторы-выражения

Программа оперирует с данными. Собственно, это и есть цель любой программы — преобразовать входную информацию в выходную. Числа можно складывать, вычитать, умножать, делить. Знаки можно сравнивать и т.д.. То есть из разных величин можно составлять выражения, результат вычисления которых — новая величина. Приведем примеры выражений:

\[ x \times 12 + y \] // значение Х умножить на 12 и к результату
// прибавить значение Y
val < 3 // сравнить значение val с 3
-9 // константное выражение -9

Выражение, после которого стоит точка с запятой — это оператор-выражение. Его смысл состоит в том, что компьютер должен выполнить все действия, записанные в данном выражении, иначе говоря, вычислить выражение.

x + y - 12; // сложить значения x и y и затем,
// вычесть 12
a = b + 1; // прибавить единицу к значению b и
// запомнить результат в переменной a

Выражения — это переменные, функции и константы, называемые операндами, объединенные знаками операций. Операции могут быть унарными — с одним операндом, например минус. Могут быть бинарные — с двумя операндами, например сложение, деление. В Си++ есть даже однонарочную операцию — условное выражение. В табл. 2 приведен список всех операций языка Си++ для встроенных типов данных. Подробно каждая операция разбирается в разделе посвященном тому или иному типу данных. Кроме того, ряд операций будет рассмотрен в разделе, посвященном определению операторов для классов. Пока что мы ограничиваемся лишь общим описанием способов записи выражений:

В типизированном языке, которым является Си++, у переменных и констант есть тип. Есть тип и у результата выражения. Например, операции сложения (+), умножения (*), вычитания (-) и деления (/), примененные к целым числам, выполняются по общепринятым математическим правилам и дают в результате целое значение. Т.e. же операции можно применить к вещественным числам и получить вещественное значение.

Операция сравнения: больше (>), меньше (<), равно (=), не равно (!=) сравнивает значения чисел и выдает логическое значение: истина (true) или ложь (false).

Присваивание — это тоже операція и является частью выражения. Значение правого операнда присваивается левому операнду.

x = 2; // переменной x присвоить значение 2
cond = x < 2; // переменной cond присвоить значение true,
// если x меньше 2, в противном случае
// присвоить значение false
3 = 5; // ошибка, число 3 непосредственно изменять своё
// значение

Последний пример иллюстрирует требование в левом операнду операции присваивания. Он должен быть способен хранить и изменять свое значение. Перменные, объявленные в программе, обладают подобным свойством. В следующем фрагменте программы

int x = 0;
x = 3;
вначале объявляется переменная x с начальным значением 0. Затем значение x изменяется на 3, 4 и затем 5. Опять таки, обратим внимание на последнюю строку. При вычислении операции присваивания сначала вычисляется левый operand, а затем правый. Когда вычисляется выражение x + 1, значением переменной x является 4. Поэтому значение выражения x + 1 равно 5. После вычисления операции присваивания (или говоря проще, после присваивания) значение переменной x становится 5.

У операции присваивания тоже есть результат. Он равен значению правого операнда. Таким образом операция присваивания может участвовать в более сложном выражении:

```
z = (x = y + 3);
```

В приведенном примере переменным x и z присваивается значение y + 3.

Очень часто в программе приходится увеличивать или уменьшать значение переменной на единицу. Для того, чтобы сделать эти действия наиболее эффективными и удобными для использования, в C++ имеются специальные знаки операций: ++ (увеличить на единицу) и — (уменьшить на единицу). Кроме того, имеются две формы этих операций: префиксная и постфиксная. Рассмотрим их на примерах.

```cpp
int x = 0;
++x;
```

Значение x увеличивается на единицу и становится равным 1.

```
--x;
```

Значение x уменьшается на единицу и становится равным 0.

```cpp
int y = ++x;
```

Значение x опять увеличивается на единицу. Результат операции ++ — новое значение x, т.е. переменной y присваивается значение 1.

```cpp
int z = x++;
```

Здесь используется постфиксная запись операции увеличения на единицу. Значение переменной x до выполнения операции равно 1. Сама операция так же — значение x увеличивается на единицу и становится равным 2. Однако результат постфиксной операции — значение аргумента до увеличения. Таким образом, переменной z присваивается значение 1. Аналогично, результатом постфиксной операции уменьшения на единицу
является начальное значение операнда, а префиксной — конечное значение операнда.

Подобными мотивами оптимизации и сокращения записи руководствовались создатели языка Си (а затем и Си++), вводя новые знаки операций типа "выполнить операцию и присвоить". Довольно часто одна и та же переменная используется в левой и правой части операции присваивания, например:

\[
\begin{align*}
x &= x + 5;  \\
y &= y * 3;  \\
z &= z - (x + y);
\end{align*}
\]

В Си++ эти выражения можно записать короче:

\[
\begin{align*}
x &= 5;  \\
y &= 3;  \\
z &= x + y;
\end{align*}
\]

т.е. запись oper= означает, что левый operand вначале используется как левый operand операции oper, а затем, как левый operand операции присваивания результата операции oper. Кроме краткости выражения, такая запись облегчает оптимизацию программы компилятором.

Наряду с общепринятыми арифметическими и логическими операциями, в языке Си++ имеется набор операций для работы с битами — порядочные И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и НЕ, а также сдвига.

Особняком стоит операція sizeof. Эта операція позволяет определить, сколько памяти занимает то или иное значение. Например

\[
\begin{align*}
\text{sizeof(long);} & \quad \text{// сколько байтов занимает тип long}  \\
\text{sizeof b;} & \quad \text{// сколько байтов занимает переменная b}
\end{align*}
\]

Операция sizeof в качестве аргумента берёт имя типа или выражение. Аргумент заключается в скобки (если аргумент — выражение, скобки не обязательны). Результат операции — целое число, равное количеству байтов, необходимых для хранения в памяти заданной величины.

Ниже приводится сводная таблица операций языка Си++.

<table>
<thead>
<tr>
<th>Операция</th>
<th>Действие</th>
<th>Тип operandов</th>
<th>Тип результата</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ сложение, - вычитание, * умножение, / деление</td>
<td>Сложение, вычитание, умножение и деление чисел</td>
<td>Целые или вещественные числа, байты</td>
<td>Соответствует типу наиболее &quot;длинного&quot; operanda, т.е. operanda, с помощью которого можно выразить наибольшее число</td>
</tr>
<tr>
<td>% остаток</td>
<td>Остаток от деления первого operanda на второй</td>
<td>Целые числа или байты, преобразованные в целые числа</td>
<td>Целое число</td>
</tr>
<tr>
<td>Операция</td>
<td>Описание</td>
<td>Результаты операции</td>
<td>Тип операнда</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>- минус</td>
<td>Изменение знака числа</td>
<td>Знаковые числа и байты</td>
<td>Совпадает с типом операнда</td>
</tr>
<tr>
<td>+ плюс</td>
<td>Унарный плюс, результат операции — неизмененный операнд</td>
<td>Любые числа и байты</td>
<td>Совпадает с типом операнда</td>
</tr>
<tr>
<td>++ увеличить на единицу, префиксная форма</td>
<td>Увеличить значение операнда на единицу, результат — новое, увеличенное значение операнда</td>
<td>Целые числа</td>
<td>Тип результата совпадает с типом операнда</td>
</tr>
<tr>
<td>-- уменьшить на единицу, префиксная форма</td>
<td>Уменьшить значение операнда на единицу, результат — новое, уменьшенное значение операнда</td>
<td>Целые числа</td>
<td>Тип результата совпадает с типом операнда</td>
</tr>
<tr>
<td>++ увеличить на единицу, постфиксная форма</td>
<td>Увеличить значение операнда на единицу, результат — старое, неизмененное значение операнда</td>
<td>Целые числа</td>
<td>Тип результата совпадает с типом операнда</td>
</tr>
<tr>
<td>-- уменьшить на единицу, постфиксная форма</td>
<td>Уменьшить значение операнда на единицу, результат — старое, неизмененное значение операнда</td>
<td>Целые числа</td>
<td>Тип результата совпадает с типом операнда</td>
</tr>
<tr>
<td>==, = равно, не равно</td>
<td>Сравнение двух operandов на равенство или неравенство</td>
<td>Логическое значение</td>
<td></td>
</tr>
<tr>
<td>&gt;, &lt;, &lt;=, &gt;= меньше, больше, меньше или равно, больше или равно</td>
<td>Сравнение двух operandов</td>
<td>Логическое значение</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp; логическое И</td>
<td>Конъюнкция двух логических условий</td>
<td>Логические значения</td>
<td>Логическое значение</td>
</tr>
<tr>
<td></td>
<td></td>
<td>логическое ИЛИ</td>
<td>Дизъюнкция двух логических значений</td>
</tr>
<tr>
<td>! логическое НЕ</td>
<td>Логическое отрицание</td>
<td>Логическое значение</td>
<td>Логическое значение</td>
</tr>
<tr>
<td>&amp; битовое И</td>
<td>Побитовое выполнение операции И</td>
<td>Целые числа</td>
<td>Целое число</td>
</tr>
<tr>
<td></td>
<td>битовое ИЛИ</td>
<td>Побитовое выполнение операции ИЛИ</td>
<td>Целые числа</td>
</tr>
<tr>
<td>^ битовое ИСКЛЮЧАЮЩЕЕ ИЛИ</td>
<td>Побитовое выполнение операции ИСКЛЮЧАЮЩЕЕ ИЛИ</td>
<td>Целые числа</td>
<td>Целое число</td>
</tr>
<tr>
<td>~ битовое НЕ</td>
<td>Изменение каждого бита в числе на противоположный</td>
<td>Целые числа</td>
<td>Целое число</td>
</tr>
<tr>
<td>&lt;&lt; сдвиг влево</td>
<td>Побитовый сдвиг левого операнда на количество разрядов, соответствующее значению правого операнда</td>
<td>Целые числа</td>
<td>Целое число</td>
</tr>
<tr>
<td>Сдвиг вправо</td>
<td>Побитовый сдвиг левого операнда на количество разрядов, соответствующее значению правого операнда</td>
<td>Целые числа</td>
<td>Целое число</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------</td>
<td>------------</td>
<td>--------------</td>
</tr>
<tr>
<td>? : условное выражение</td>
<td>Трехарыная операция; если значение первого операнда — истина, то результат — второй операнд; если ложь — результат — третий операнд</td>
<td>Первый операнд — логическое значение, второй и третий операнды могут иметь любой, но один и тот же тип</td>
<td>Тип второго и третьего operandov</td>
</tr>
<tr>
<td>, последовательность</td>
<td>Выполнить выражение до запятой, затем выражение после запятой</td>
<td>Произвольные</td>
<td>Тип второго выражения</td>
</tr>
<tr>
<td>= присваивание</td>
<td>Присвоить значение правого операнда левому</td>
<td>Произвольные, но для типа левого операнда должна существовать операция присваивания с типом правого операнда</td>
<td>Тип левого операнда</td>
</tr>
<tr>
<td>+=, -=, *=, /=, %=,</td>
<td>Выполнить соответствующую операцию с левым operandom и правым operandom и присвоить результат левому operandu</td>
<td>Типы operandов должны быть такими, что, во-первых, соответствующая арифметическая операция определена для них, а во-вторых, результат может быть присвоен левому операнду</td>
<td>Тип левого операнда</td>
</tr>
</tbody>
</table>

### 3.3.4.2.1. Порядок вычисления выражений

У каждой операции имеется приоритет. Если в выражении несколько операций, то первой будет выполнена операция с более высоким приоритетом. Если же операции одного и того же приоритета, они выполняются слева направо.

Например, в выражении

$$2 + 3 \times 6$$

сначала будет выполнено умножение, а затем сложение, соответственно значение этого выражения число 20.

В выражении

$$2 \times 3 + 4 \times 5$$

сначала будет произведено умножение, а затем сложение. В каком порядке будет произведено умножение — сначала $2 \times 3$, а затем $4 \times 5$ или наоборот, вообще говоря не определено. Т.е. для операции сложения не определено в каком порядке вычисляются её operandы.

В выражении
\[ x = y + 3 \]

начале выполняется сложение, а затем присваивание, поскольку приоритет операции присваивания ниже сложения.

Для данного правила существует исключение: если в выражении несколько операций присваивания, то они выполняются справа налево. Например в выражении

\[ x = y = 2 \]

начала выполняется операция присваивания значения 2 переменной \( y \). Затем результат этой операции – значение 2 – будет присвоено переменной \( x \).

Ниже приведен список всех операций в порядке понижения приоритета. У операций, перечисленных в одной строчке, одинаковый приоритет. Операции, перечисленные в каждой последующей строчке менее приоритетны (т.е. выполняются после) операций, перечисленных в предыдущей строке.

<table>
<thead>
<tr>
<th>:: (разрешение области видимости имен)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (обращение к элементу класса)</td>
</tr>
<tr>
<td>-&gt; (обращение к элементу класса по указателю)</td>
</tr>
<tr>
<td>[] (индексирование)</td>
</tr>
<tr>
<td>вызов функции</td>
</tr>
<tr>
<td>++ (постфиксное увеличение на единицу)</td>
</tr>
<tr>
<td>-- (постфиксное уменьшение на единицу)</td>
</tr>
<tr>
<td>typeid (нахождение типа)</td>
</tr>
<tr>
<td>dynamic_cast static_cast reinterpret_cast const_cast (преобразования типа)</td>
</tr>
<tr>
<td>sizeof (определение размера)</td>
</tr>
<tr>
<td>++ (префиксное увеличение на единицу)</td>
</tr>
<tr>
<td>-- (префиксное уменьшение на единицу)</td>
</tr>
<tr>
<td>~ (битовое НБ)</td>
</tr>
<tr>
<td>! (логическое НБ)</td>
</tr>
<tr>
<td>- (изменение знака)</td>
</tr>
<tr>
<td>+ (плюс)</td>
</tr>
<tr>
<td>&amp; (взятие адреса)</td>
</tr>
<tr>
<td>* (обращение по адресу)</td>
</tr>
<tr>
<td>new (создание объекта)</td>
</tr>
<tr>
<td>delete (удаление объекта)</td>
</tr>
<tr>
<td>(type) (преобразование типа)</td>
</tr>
</tbody>
</table>

| * (умножение) |
| / (деление) |
| % (остаток) |
| + (сложение) |
| - (вычитание) |
| << |
| >> (сдвиг) |
| <= >= (сравнения на больше или меньше) |
| == != (равно, неравно) |
Для того, чтобы изменить последовательность вычисления выражений, можно воспользоваться круглыми скобками. Часть выражения, заключенная в скобки, вычисляется до вычисления остального выражения. Значением

\[(2 + 3) \times 6\]

будет 30.

Скобки могут быть вложенными, соответственно самые внутренние выполняются самыми первыми:

\[(2 + (3 \times (4 + 5)) - 2)\]

**Упражнения**

1. Модифицируйте функцию `main` из предыдущего раздела, добавив туда вычисление выражений, приведенных в данном параграфе. Выведите их значения на терминал. Для вывода используйте операцию `<<`, т.е. конструкция `cout << x;` выведет значение переменной `x`.

2. Проверьте порядок вычислений, записав какое-либо сложное выражение со скобками и без них. Выведите значения выражений на терминал.

**3.3.4.3. Операторы управления**

Операторы управления определяют, в какой последовательности выполняется программа. Если бы их не было, операторы программы выполнялись всегда последовательно, в том порядке, в котором они записаны.

**Условные операторы**

Условные операторы позволяют выбрать один из вариантов выполнения действий в зависимости от каких-либо условий. Условие — это
логическое выражение, т.е. выражение, результатом которого является логическое значение true (истина) или false (ложь).
Оператор if выбирает один из двух вариантов последовательности вычислений.

```cpp
if (условие)
    оператор1
else
    оператор2
```

Если условие истинно, выполняется оператор1, если условие ложно, то выполняется оператор2.

```cpp
if (x > y)
    a = x;
else
    a = y;
```

В данном примере переменной а присваивается значение максимума из двух величин х и у.
Конструкция else необязательна. Можно записать:

```cpp
if (x < 0)
    x = -x;
абс = x;
```

В данном примере оператор x = -x; выполняется только в случае, если значение переменной x было отрицательно. Присваивание переменной abs выполняется в любом случае. Таким образом, приведенный фрагмент программы изменит значение переменной x на его абсолютное значение и присвоит переменной abs новое значение x.
Если в случае истинности условия необходимо выполнить несколько операторов, их можно заключить в фигурные скобки:

```cpp
if (x < 0) {
    x = -x;
    cout << "Изменить значение x на противоположное по знаку";
}
абс = x;
```

Теперь, если x отрицательно, не только его значение измениться на противоположное, но и соответствующее сообщение будет выведено на экран терминала. Фактически, заключая несколько операторов в фигурные скобки, мы сделали из них один сложный оператор или блок. Прием заключения нескольких операторов в блок работает везде, где нужно поместить несколько операторов вместо одного.
Условный оператор можно расширить для проверки нескольких условий:
if (x < 0)
    cout << "Отрицательная величина";
else if (x > 0)
    cout << "Положительная величина";
else
    cout << "Ноль";

Конструкция else if может быть несколько.
Хотя любые комбинации условий можно выразить с помощью оператора if, довольно часто запись становится неудобной и запутанной. Оператор выбора switch используется, когда для каждого из нескольких возможных значений выражения нужно выполнить определенные действия. Например, предположим, что в переменной code хранится целое число от 0 до 2 и нам нужно выполнить различные действия в зависимости от её значения:

switch (code) {
    case 0:
        cout << "код ноль";
        x = x + 1;
        break;
    case 1:
        cout << "код один";
        y = y + 1;
        break;
    case 2:
        cout << "код два";
        z = z + 1;
        break;
    default:
        cout << "Необработываемое значение";
    }

В зависимости от значения code управление передаётся на одну из меток case. Выполнение оператора заканчивается по достижении либо оператора break, либо конца оператора switch. Таким образом, если code равно 1, выводится "код один" и затем переменная y увеличивается на единицу. Если бы после этого не стоял оператор break, то управление "провалялось" бы дальше, вывелась бы фраза "код два" и переменная z тоже бы увеличилась на единицу.

Если значение переключателя не совпадает ни с одним из значений меток case, то выполняются операторы, записанные после метки default. Метка default может быть опущена, что эквивалентно записи

default:
    ; // пустой оператор, не выполняющий никаких действий
логическое выражение, т.е. выражение, результатом которого является логическое значение true (истина) или false (ложь).

Оператор if выбирает один из двух вариантов последовательности вычислений.

```cpp
if (условие)
    оператор1
else
    оператор2
```

Если условие истино, выполняется оператор1, если условие ложно, то выполняется оператор2.

```cpp
if (x > y)
    a = x;
else
    a = y;
```

В данном примере переменной a присваивается значение максимума из двух величин x и y.
Конструкция else необязательна. Можно записать:

```cpp
if (x < 0)
    x = -x;
abs = x;
```

В данном примере оператор x = -x; выполняется только в случае, если значение переменной x было отрицательно. Присваивание переменной abs выполняется в любом случае. Таким образом, приведенный фрагмент программы изменит значение переменной x на его абсолютное значение и присвоит переменной abs новое значение x.
Если в случае истинности условия необходимо выполнить несколько операторов, их можно заключить в фигурные скобки:

```cpp
if (x < 0) {
    x = -x;
    cout << "Изменить значение x на противоположное по знаку";  
}
abs = x;
```

Теперь, если x отрицательно, не только его значение измениться на противоположное, но и соответствующее сообщение будет выведено на экран терминала. Фактически, заключая несколько операторов в фигурные скобки, мы сделали из них один сложный оператор или блок. Прием заключения нескольких операторов в блок работает везде, где нужно поместить несколько операторов вместо одного.
Условный оператор можно расширить для проверки нескольких условий:
if (x < 0)
    cout << "Отрицательная величина";
else if (x > 0)
    cout << "Положительная величина";
else
    cout << "Ноль";

Конструкция else if может быть несколько.
Хотя любые комбинации условий можно выразить с помощью оператора if, довольно часто запись становится неудобной и запутанной.
Оператор выбора switch используется, когда для каждого из нескольких возможных значений выражения нужно выполнить определенные действия. Например, предположим, что в переменной code хранится целое число от 0 до 2 и нам нужно выполнить различные действия в зависимости от её значения:

```cpp
switch (code) {
    case 0:
        cout << "код ноль";
        x = x + 1;
        break;
    case 1:
        cout << "код один";
        y = y + 1;
        break;
    case 2:
        cout << "код два";
        z = z + 1;
        break;
    default:
        cout << "Необработываемое значение";
}
```

В зависимости от значения code управление передаётся на одну из меток case. Выполнение оператора заканчивается по достижении либо оператора break, либо конца оператора switch. Таким образом, если code равно 1, выводится "код один" и затем переменная y увеличивается на единицу. Если бы после этого не стоял оператор break, то управление "провалилось" бы дальше, вывелась бы фраза "код два" и переменная z тоже бы увеличилась на единицу.

Если значение переключателя не совпадает ни с одним из значений меток case, то выполняются операторы, записанные после метки default. Метка default может быть опущена, что эквивалентно записи

```cpp
default:
; // пустой оператор, не выполняющий никаких действий
```
Как легко понять, приведенный пример можно переписать с помощью оператора if:

```c++
if (code == 0) {
    cout << "код ноль";
    x = x + 1;
} else if (code == 1) {
    cout << "код один";
    y = y + 1;
} else if (code == 2) {
    cout << "код два";
    z = z + 1;
} else {
    cout << "Необрабатываемое значение";
}
```

Пожалуй, запись с помощью оператора переключения switch более наглядна. Особенно часто переключатель используется, когда значение выражения имеет тип набора (см. пункт 4.1.5).

**Упражнение**

Напишите программу вывода целого числа от 0 до 9 в виде слова (один, два и т.д.). Значение числа задавайте в программе с помощью явного присваивания переменной.

**Операторы цикла**

Сила компьютеров заключается в неутомимости при повторениях. Предположим, нам нужно вычислить сумму всех целых чисел от 0 до 100\(^1\). Для этого воспользуемся оператором цикла for:

```c++
int sum = 0;
int i;

for (i = 1; i <= 100; i = i + 1) // заголовок цикла
    sum = sum + i;       // тело цикла
```

Оператор цикла состоит из заголовка цикла и тела цикла. Тело цикла — это оператор, который будет повторно выполняться (в данном случае — увеличение значения переменной sum на величину переменной i). Заголовок — это ключевое слово for, после которого в круглых скобках записаны три выражения, разделенные точкой с запятой. Первое выражение вычисляется один раз до начала выполнения цикла. Второе — это ус-

---

\(^1\) Когда юному Гауссу учителю математики задал сложить все числа от 0 до 100, будущий великий математик не стал напрямую складывать все числа, а вычислил результат за одну минуту, найдя простую закономерность. При наличии компьютера и операторов цикла, быть может он бы и не стал великим математиком.
ловие цикла. Тело цикла будет повторяться до тех пор, пока условие цикла истино. Третье выражение вычисляется после каждого повторения тела цикла.

Оператор for реализует фундаментальный принцип вычислений в программировании — итерацию. Тело цикла повторяется для разных, в данном случае последовательных, значений переменной i. Повторение иногда называется итерацией. Мы как бы проходим по последовательности значений переменной i, выполняя с текущим значением одно и то же действие, тем самым постепенно вычисляя нужное значение. С каждой итерацией мы подходим к нему все ближе и ближе. С другим принципом вычислений в программировании — рекурсией, мы познакомимся в разделе, описывающем функции.

Любое из трех выражений в заголовке цикла может быть опущено (в том числе и все три). Тоже самое можно записать:

```c
int sum = 0;
int i = 1;

for (; i <= 100; ) {
    sum = sum + i;
    i = i + 1;
}
```

Заметим, что вместо одного оператора цикла мы записали несколько операторов, заключенных в фигурные скобки—блок. Другой вариант:

```c
int sum = 0;
int i = 1;
for (; ; ) {
    if (i > 100)
        break;

    sum = sum + i;
    i = i + 1;
}
```

В последнем примере мы опять встречаем оператор break. Оператор break завершает выполнение цикла. Еще одним вспомогательным оператором при выполнении циклов служит оператор продолжения continue. Оператор continue заставляет пропустить остаток тела цикла и перейти к следующей итерации (повторению). Например, если мы хотим найти сумму всех целых чисел от нуля до ста, которые не делятся на 7, можно записать это так:

```c
int sum = 0;
for (int i = 1; i <= 100; i = i+1) {
    if (i % 7 == 0)
        continue;
    sum = sum + i;
}
```
```c
int digits = 0;
while (N > 0) {
    digits = digits + 1;
    N = N / 10;
}
```

Еще одно полезное свойство цикла for: в первом выражении заголовка цикла можно объявить переменную. Эта переменная будет действительна только в пределах цикла.

Другой формой оператора цикла является оператор while. Его форма следующая:

```c
while (условие) оператор
```

Условие — как и в условном операторе if — это выражение, которое принимает логическое значение истина или ложь. Выполнение оператора повторяется до тех пор, пока значением условия является true (истина). Условие пересчитывается перед каждой итерацией. Подсчитать, сколько десятичных цифр нужно для записи целого положительного числа $N$ можно с помощью следующего фрагмента:

```c
int digits = 0;
while (N > 0) {
    digits = digits + 1;
    N = N / 10;
}
```

Если число $N$ меньше либо равно нулю, тело цикла не выполнится ни разу.

Третьей формой оператора цикла является цикл do while. Он имеет форму:

```c
do { операторы } while ( условие);
```

Отличие от предыдущей формы цикла while заключается в том, что условие проверяется после выполнения тела цикла. Предположим, надо прочитать символы с терминала до тех пор, пока не будет введён символ звездочки.

```c
char ch;
do {
    ch = getch(); // функция getch возвращает символ, введённый с клавиатуры
} while (ch != '*');
```

В операторах while и do также можно использовать операторы break и continue.

Как легко заметить, операторы цикла взаимозаменяются. Оператор while соответствует оператору for:

```c
for ( ; условие ; )
```
Пример чтения символов с терминала можно переписать в виде:

```c
char ch;
ch = getch();
while (ch != '***') {
    ch = getch();
}
```

Разные формы нужны для удобства и наглядности записи.

**Упражнение**

Перепишите все примеры данного раздела, используя все три формы циклов.

**Операторы перехода**

Последовательность выполнения операторов в программе можно изменить с помощью оператора перехода `goto`. Он имеет вид

```c
goto метка;
```

Метка ставится в программе, записывая ее имя и затем двоеточие. Например, вычислить абсолютную величину значения переменной `x` можно следующим способом

```c
if (x >= 0)
    goto positiv;
    x = -x; // переменить знак x
positiv:
    // объявление метки
    abs = x; // присвоить переменной abs
    // положительное значение
```

При выполнении `goto` вместо следующего оператора выполняется оператор, следующий за меткой `positiv`. Если значение `x` положительно, оператор `x = -x` не будет выполняться.

В настоящее время общепринято, что оператор `goto` очень легко запутывает программу, без него вообще говоря можно обойтись, поэтому лучше его не использовать, а уж если использовать, то лишь в самом крайнем случае.

**3.3.5. Функции**

Функции являются составными частями выражений. Функции в языке Си++ имеют практически тот же смысл, что и функции в математике
Программа вызывает функцию с определенными аргументами, функция выполняет необходимые действия и возвращает результат.

Программа на языке Си++ состоит из по крайней мере одной функции — функции main. С нее всегда начинается выполнение программы. Встретив имя функции в выражении, программа вызовет эту функцию, т.е. передаст управление на ее начало и начнет выполнять операторы. Достигнув конца функции или оператора return — выход из функции, управление вернется в ту точку, откуда функция была вызвана, подставив вместо нее вычисленный результат. На рис. 2 стрелочками показана передача управления при вызове функции func и затем возврат управления в точку вызова. Функция func вычисляет квадрат своего аргумента, таким образом переменной x будет присвоено значение 100.

Прежде всего, функцию необходимо объявить. Объявление функции, аналогично объявлению переменной, определяет имя функции и ее тип — типы и количество ее аргументов и тип возвращаемого значения.

```c++
// функция sqrt с одним аргументом — вещественным
// числом двойной точности, возвращает результат
// типа double
double sqrt(double x);
```

```c++
// функция sum от трех целых аргументов
// возвращает целое число
int sum(int a, int b, int c);
```

Объявление функции называют иногда прототипом функции. После того, как функция объявлена, её можно использовать в выражениях:

```c++
double x = sqrt(3) + 1;
sum(k, l, m) / 15;
```

![Diagram](image)

Рис. 2. Передача управления при вызове функции.
Если функция не возвращает никакого результата, т.е. её результат объявлен типа void, её вызов не может быть использован как operand более сложного выражения, а должен быть записан сам по себе:

    func(a, b, c);

Определение функции описывает как она работает, т.е. какие действия надо выполнить, чтобы получить искомый результат. Для функции sum, объявленной выше, определение может выглядеть следующим образом:

    int
    sum(int a, int b, int c)
    {
        int result;
        result = a + b + c;
        return result;
    }

Первая строка — это заголовок функции, он совпадает с объявлением функции, за исключением того, что объявление заканчивается точкой с запятой. Далее в фигурных скобках заключено тело функции — действия, которые данная функция выполняет.

Аргументы a, b и c называются формальными параметрами. Это переменные, которые определены в теле функции (т.е. к ним можно обращаться только внутри фигурных скобок). При написании определения функции, программа не знает их значения. При вызове функции вместо них подставляются фактические параметры — значения, с которыми функция вызывается. Выше, в примере вызова функции sum, фактическими параметрами (иногда говорят фактическими аргументами) являлись значения переменных k, l и m.

Формальные параметры принимают значения фактических аргументов, заданных при вызове, и функция выполняется.

Первое, что мы делаем в теле функции — объявляем внутреннюю переменную result типа целое. Переменные, объявленные в теле функции также называются локальными. Это связано с тем, что переменная result существует только во время выполнения тела функции sum. После завершения выполнения функции она уничтожается — её имя становится неизвестным и память, занимаемая этой переменной, освобождается.

Вторая строка определения тела функции — вычисление результата. Сумма всех аргументов присваивается переменной result. Отметим, что до присваивания значение result было неопределенным. (Непредельно — это значит значение переменной было неким произвольным числом, которое никак нельзя определить заранее).

Последняя строкка функции возвращает вычисленное значение в качестве результата. Оператор return завершает выполнение функции и возвращает выражение, записанное после ключевого слова return, в ка-
sum = sum + 1;
}

Еще одно полезное свойство цикла for: в первом выражении заголовка цикла можно объявить переменную. Эта переменная будет действительна только в пределах цикла.

Другой формой оператора цикла является оператор while. Его форма следующая:

while (условие)
оператор

Условие – как и в условном операторе if – это выражение, которое принимает логическое значение истина или ложь. Выполнение оператора повторяется до тех пор, пока значением условия является true (истина). Условие пересчитывается перед каждой итерацией. Подсчитать, сколько десятичных цифр нужно для записи целого положительного числа N можно с помощью следующего фрагмента:

```c
int digits = 0;
while (N > 0) {
    digits = digits + 1;
    N = N / 10;
}
```

Если число N меньше либо равно нулю, тело цикла не выполнится ни разу.

Третьей формой оператора цикла является цикл do while. Он имеет форму:

```c
do { операторы } while ( условие);
```

Отличие от предыдущей формы цикла while заключается в том, что условие проверяется после выполнения тела цикла. Предположим, надо прочитать символы с терминала до тех пор, пока не будет введён символ звездочка.

```c
char ch;
do {
    ch = getch();  // функция getch возвращает
    // символ, введённый с клавиатуры
} while (ch != '*');
```

В операторах while и do также можно использовать операторы break и continue.

Как легко заметить, операторы цикла взаимозаменяемы. Оператор while соответствует оператору for:

```c
for ( ; условие ; )
```
оператор

Пример чтения символов с терминала можно переписать в виде:

```c
char ch;
ch = getch();
while (ch != '*') {
    ch = getch();
}
```

Разные формы нужны для удобства и наглядности записи.

**Упражнение**

Перепишите все примеры данного раздела, используя все три формы циклов.

**Операторы перехода**

Последовательность выполнения операторов в программе можно изменить с помощью оператора перехода `goto`. Он имеет вид

```
goto метка;
```

Метка ставится в программе, записывая ее имя и затем двоеточие. Например, вычислить абсолютную величину значения переменной `x` можно следующим способом

```c
if (x >= 0) goto positiv;
goto positiv;
// переменить знак x
x = -x;
// объявление метки
positiv: // присвоить переменной abs
abs = x; // положительное значение
```

При выполнении `goto` вместо следующего оператора выполняется оператор, следующий за меткой `positiv`. Если значение `x` положительно, оператор `x = - x` не будет выполняться.

В настоящее время общепринято, что оператор `goto` очень легко запутывает программу, без него вообще говоря можно обойтись, поэтому лучше его не использовать, а уж если использовать, то лишь в самом крайнем случае.

**3.3.5. Функции**

Функции являются составными частями выражений. Функции в языке Си++ имеют практически тот же смысл, что и функции в математике.
честве выходного значения. В следующем фрагменте программы переменной s присваивается значение 10:

```c
int k = 2;
int l = 3;
int m = 5;
int s = sum(k, l, m);
```

### 3.3.5.1. Имена функций

В языке Си++ допустимо иметь несколько функций с одним и тем же именем, потому что функции различаются не только по именам, но и по типам аргументов. Если в дополнение к функции `sum`, определенной выше, мы определим еще одну функцию с тем же именем:

```c
double
sum(double a, double b, double c)
{
    double result;
    result = a + b + c;
    return result;
}
```

это будет считаться новой функцией. Иногда говорят, что у этих функций разные подписи. В следующем фрагменте программы первый раз будет вызвана первая функция, а во второй раз — вторая:

```c
int x, y, z, ires;
double p, q, s, dres;
...
// вызвать первое определение функции sum
ires = sum(x, y, z);
// вызвать второе определение функции sum
dres = sum(p, q, s);
```

При первом вызове функции `sum` все фактические аргументы имеют тип `int`. Поэтому вызывается первая функция. Во втором вызове все аргументы имеют тип `double`, соответственно вызывается вторая функция.

Кроме типа аргументов важно и их количество. Можно определить функцию `sum`, суммирующую четыре аргумента:

```c
int
sum(int x1, int x2, int x3, int x4)
{
    return x1 + x2 + x3 + x4;
}
```
Отметим, при определении функций имеет значения тип и количество аргументов, но не тип возвращаемого значения. Попытка определения двух функций с одним и тем же именем, одними и теми же аргументами, но с разными возвращаемыми значениями, приведет к ошибке компиляции:

```c
int foo(int x);
double foo(int x); // ошибка — двукратное определение имени
```

### 3.3.5.2. Необязательные аргументы функций

При объявлении функций в языке Си++ имеется возможность задать значения аргументов по умолчанию. Первый случай применения этой возможности языка — сокращение записи. Если функция вызывается с одним и тем же значением аргумента в 99 процентов случаев из ста и это значение достаточно очевидно, можно задать его по умолчанию. Предположим, функция `exp` возводит число в произвольную целую положительную степень. Часто всего она используется для возведения в квадрат. Её объявление можно записать так:

```c
double exp(double x, unsigned int e = 2);
```

Определение функции:

```c
double exp(double x, unsigned int e)
{
    double result = 1;
    for (unsigned int i = 0; i < e; i++)
        result *= x;
    return result;
}
```

В главной функции:

```c
main()
{
    double y = exp(3.14); // 3.14 в квадрате
    double x = exp(2.9, 5); // 2.9 в пятой степени
}
```

Использование аргументов по умолчанию удобно при изменении функции. Если при изменении программы нужно добавить новый аргумент, то для того, чтобы не изменять все вызовы этой функции, можно новый аргумент объявить со значением по умолчанию. В таком случае старые вызовы будут использовать значение по умолчанию, а новые — указанные при вызове.

Необязательных аргументов может быть несколько. Если указан один необязательный аргумент, то либо он должен быть последним в про-
тотипе, либо все аргументы после него должны также иметь значение по умолчанию. (Объясните почему.)
Если у функции задан необязательный аргумент, то фактически заданы несколько подписей этой функции. Например, попытка определения двух функций:

```c
double exp(double x, unsigned int e = 2);
double exp(double x);
```

приведет к ошибке компиляции — неоднозначности определения функции. Это происходит потому, что вызов

```c
double x = exp(4.1);
```

подходит как для первой, так и для второй функции.

3.3.5.3. Рекурсия

Определения функций не могут быть вложенными, т.е. нельзя внутри тела одной функции определить тело другой функции. Разумеется, можно вызывать одну функцию из другой. В том числе функция может вызвать сама себя.

Рассмотрим функцию вычисления факториала целого числа. Её можно реализовать двумя способами. Первый способ использует итерацию:

```c
int fact(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
        result = result * i;
    return result;
}
```

Второй способ:

```c
int fact(int n)
{
    if (n == 1)
        return 1;  // факториал 1 равен 1
    else
        // факториал числа n равен
        // факториалу n-1 умноженному на n
        return n * fact(n -1);  
}
```
Функция fact вызывает сама себя с модифицированными аргументами. Такой способ вычислений называется рекурсией. Рекурсия — это очень мощный метод вычислений. Значительная часть математических функций определяется в рекурсивных терминах. В программировании алгоритмы обработки сложных структур данных также часто рекурсивны. Рассмотрим, например, структуру двоичного дерева. Дерево состоит из узлов и направленных связей. С каждым узлом может быть связано один или два узла, называемые сыновьями этого узла. Соответственно для сыновей узел, из которого к ним идут связи, называется отцом. Узел, у которого нет отца, называется корень. У дерева есть ровно один корень. Узлы, у которых нет сыновей, называются листьями. Пример дерева приведен на рис. 3.

В этом дереве узел A — корень дерева, узлы B и C — сыновья узла A, узлы D и E — сыновья узла B, узел F — сын узла C. Узлы D, E и F — листья. Узел B является корнем поддерева, состоящего из трех узлов B, C и D. Обход дерева (прохождение по всем его узлам) можно описать таким образом:
1. Посетить корень дерева.
2. Обойти поддеревья с корнями — сыновьями данного узла, если у узла есть сыновья.
3. Если у узла нет сыновей — обход закончен.

Очевидно, что реализация такого алгоритма с помощью рекурсии не представляет труда.

Довольно часто рекурсия и итерация взаимозаменяемы (как в примере с факториалом). Выбор между ними может быть обусловлен разными факторами. Чаще рекурсия более наглядна и легче реализуется. Итерация в большинстве случаев более эффективна.

Рис. 3. Структура двоичного дерева.
Упражнение

Напишите рекурсивную функцию вычисления суммы чисел от 1 до n.

3.3.5.4. Библиотека стандартных функций

Ряд функций, которые часто используются во многих программах, образуют библиотеку стандартных функций языка Си++. На самом деле большинство из них пришли из языка Си.

Библиотека стандартных функций является частью практически любого компилятора языка Си++ и позволяет программистам использовать готовые функции в своих программах. В состав библиотеки входят математические функции (вычисление логарифма, квадратного корня, возведение в произвольную степень и т.д.), функции ввода-вывода, функции обработки строк символов и ряд других. С некоторыми из этих функций мы познакомимся по ходу изучения языка Си++.

3.4. Стиль записи программ

При записи программ важно придерживаться определенного стиля записи. Прежде всего стиль выражается в системе именования и в расположении операторов программы. Наличие одного стиля облегчает чтение и модификацию разных файлов. Если же разные программисты по-разному называют переменные и функции, то и разбираться в таких программах сложнее. Всегда полезно помнить, что чтение программ — не менее важное дело, чем их написание, а довольно часто на чтение программы уходит времени гораздо больше, чем на её первоначальное составление.

Правила записи программ меняются от организации к организации, от проекта к проекту. Они зависят от сложившейся практики, личных пристрастий программистов и руководителей проектов, от используемых средств программирования и дополнительных библиотек. Главное, чтобы эти правила существовали, все разработчики были с ними знакомы и их придерживались. В качестве примера ниже мы приведем ряд правил, в соответствии с которыми записаны примеры в данной книге. По мере знакомства с новыми конструкциями, мы будем дополнять и расширять эти правила.

Прежде всего каждый оператор пишется на отдельной строке. Не пытайтесь экономить количество строк, делая их длинными, такие строки неудобно читать и модифицировать. Пустые строки обычно разделяют однородные группы операторов. Например, после объявления переменных перед первым выполняемым оператором желательно поставить пустую строку. Пустые строки часто ставятся перед логическими операторами, такими как цикл или условный оператор.
При написании программ важно подчеркнуть структуру программы. Важнейшим атрибутом является отступ от левого края и расположение фигурных скобок. С левого края начинаются описания функций, классов, глобальных переменных. Начало блока (открывающая фигурная скобка) увеличивает отступ от левого края:

```c
int
sum(int x, int y, int z)
{
    return x + y + z;  // запись с отступом
}
```

Как видно из примера, тип возвращаемого значения записан на отдельной строке, на следующей строке записано имя функции с формальными параметрами, затем на следующей строке открывающаяся фигурная скобка. Тело функции, в данном случае состоящее из одного оператора, записано с отступом. Закрывающаяся фигурная скобка убирает отступ.

Зависимость операторов подчеркивается отступом (обычно на табуляцию или на 3-4 пробела) зависимого оператора от левого края.

Например, выполняемый оператор в операторе if зависит от условия, поэтому он записывается с отступом.

```c
int
abs(int x)
{
    if (x >= 0)        // одинарный отступ
        return x;     // отступ для оператора if
    else
        return -x;
}
```

Если в операторе if используется блок, то открывающаяся фигурная скобка ставится на той же строке с условием, а закрывающаяся — на новой строке. Если после этого следует конструкция else или else if, она начинается на той же строке через пробел.

```c
if (условие 1) {
    операторы
} else if (условие 2) {
    операторы
} else {
    операторы
}
```

Аналогичное правило используется для операторов цикла — тело цикла пишется с отступом по сравнению с заголовком цикла, закрывающаяся фигурная скобка убирает дополнительный отступ. Особенно важно придерживаться правил отступа при использовании вложенных циклов и
условных операторов. Закрывающаяся фигурная скобка должна находиться на том же уровне, что и начало соответствующего условного оператора или цикла:

```java
for (int i = 0; i < n; i++) {
    if (x > 0) {
        while (true) {
            // операторы
        } // конец while
    } // конец if
} // конец цикла for
```

Немного отличается написание оператора switch. Несмотря на то, что после заголовка оператора ставится открывающаяся фигурная скобка, все метки case или default располагаются на том же уровне, что и switch. Однако операторы, соответствующие меткам, пишутся с отступом:

```java
switch (code) {
    case 1:
        x = 1;
        break;
    case 2:
        x = 2;
        break;
    default:
        ;
}
```

Заметим, что подавляющее большинство современных редакторов и сред программирования помогают программисту расположить программу красиво и по правилам. В частности, редакторы подсказывают отступ для очередного оператора, увеличивают отступ при открытии фигурной скобки, уменьшают при закрытии и т.д. Так что на практике соблюдение правил намного легче, чем кажется с первого взгляда.

Ещё раз подчеркнем, что приведенные правила — лишь примеры (хотя и очень часто употребляющиеся). Главная цель состоит в том, чтобы текст программы выглядел понятно и красиво.

Те же цели — легкость понимания и удобства чтения — преследуют правила именования. Использование одно или двух буквенных идентификаторов в качестве имен переменных допустимо только, если это временные локальные переменные, смысл которых и так ясен из контекста. Однобуквенные идентификаторы i, j, k, l также часто используют в качестве переменных цикла. Часто всего имена функций, классов, переменных и т.д. составляются из английских слов, описывающих их смысл. Имена функций мы будем писать с маленькой буквы, также как и имена переменных. Имена констант мы будем составлять из заглавных букв. Если имена констант состоят из нескольких слов, в качестве разделителя используется подчеркивание:

```java
fact, width, length, BITS_IN_BYTE
```

Правила именования классов рассматриваются в разделе, посвященном классам.
4. Типы данных, классы и объекты

Программа на языке С++ оперирует с величинами: числами, символами, объектами. Каждая величина имеет тип. Тип величины определяет возможные значения и набор определенных для них операций. Например, тип short — короткое целое число со знаком — может принимать значения от -32768 до 32767. Для таких чисел определены обычные арифметические операции и определены правила выполнения этих операций при, например, переполнении, т.е. при выходе результата за границы допустимых значений этого типа.

В языке С++ существуют простейшие типы: целые числа, вещественные числа и т.д.. В программе можно определить более сложные типы — классы, состоящие из атрибутов и методов.

Величины могут быть постоянными, т.е. неизменяемыми во время выполнения программы. Тогда для их обозначения используются константы. Величины могут быть изменяемыми и в программе создаются переменные для хранения изменяемых величин.

В этой главе мы рассмотрим основные величины, с которыми оперирует программа на языке С++.

4.1. Встроенные типы данных

Встроенные типы данных предопределены в языке. Это самые простые величины, из которых составляют все производные типы, в том числе и классы. Различные реализации и различные компиляторы могут определять различные диапазоны значений целых и вещественных чисел.

В табл. 2 перечислены простейшие типы данных, которые определяет язык С++ и приведены наиболее типичные диапазоны их значений.

4.1.1. Целые числа

Для представления целых чисел в языке С++ существует несколько типов — char, short, int и long. Они отличаются друг от друга диапазоном возможных значений (см. табл. 2): Каждый из этих типов может быть знаковым или беззнаковым. По умолчанию, тип целых величин — знаковый. Если перед определением типа стоит ключевое слово unsigned, то тип целого числа беззнаковый. Для того, чтобы определить переменную x типа короткого целого числа, нужно записать:

```c++
short x;
```

2 Полное название типов: short int, long int, unsigned long int и т.д.. Поскольку описатель int можно опустить, мы используем сокращенные названия.
Число без знака принимает только положительные значения и значение ноль. Число со знаком принимает положительные значения, отрицательные значения и значение ноль.

Таблица 2. Встроенные типы данных языка Си++.

<table>
<thead>
<tr>
<th>Название</th>
<th>Обозначение</th>
<th>Диапазон значений</th>
</tr>
</thead>
<tbody>
<tr>
<td>Байт</td>
<td>char</td>
<td>от -128 до +127</td>
</tr>
<tr>
<td>Байт баз знака</td>
<td>unsigned char</td>
<td>от 0 до 255</td>
</tr>
<tr>
<td>Короткое целое число</td>
<td>short</td>
<td>от -32768 до +32767</td>
</tr>
<tr>
<td>Короткое целое число без знака</td>
<td>unsigned short</td>
<td>от 0 до 65535</td>
</tr>
<tr>
<td>Целое число</td>
<td>int</td>
<td>от -2147483648 до +2147483647</td>
</tr>
<tr>
<td>Целое число без знака</td>
<td>unsigned int (или просто unsigned)</td>
<td>от 0 до 4294967295</td>
</tr>
<tr>
<td>Длинное целое число</td>
<td>long</td>
<td>от -2147483648 до +2147483647</td>
</tr>
<tr>
<td>Длинное целое число без знака</td>
<td>unsigned long</td>
<td>от 0 до 4294967295</td>
</tr>
<tr>
<td>Вещественное число одинарной точности</td>
<td>float</td>
<td>от ±3.4е-38 до ±3.4е+38 (7 значащих цифр)</td>
</tr>
<tr>
<td>Вещественное число двойной точности</td>
<td>double</td>
<td>от ±1.7е-308 до ±1.7е+308 (15 значащих цифр)</td>
</tr>
<tr>
<td>Вещественное число увеличенной точности</td>
<td>long double</td>
<td>от ±1.2е-4932 до ±1.2е+4932</td>
</tr>
<tr>
<td>Логическое значение</td>
<td>bool</td>
<td>значения true (истина) или false (ложь)</td>
</tr>
</tbody>
</table>

Целое число может быть непосредственно записано в программе в виде константы. Запись чисел соответствует общепринятой нотации. Примеры целых констант: 0, 125, -37. По умолчанию целые константы принадлежат к типу int. Если необходимо указать, что целое число - это константа типа long, можно добавить символ L или l после числа. Если константа беззнаковая, т.е. относится к типу unsigned long, unsigned short или unsigned int, после числа записывается символ U или u. Например 34U, 700034L, 7654ul.

Кроме стандартной десятичной записи, число можно записывать в восьмеричной или шестнадцатеричной системе счисления. Признаком восьмеричной системы счисления является цифра 0 в начале числа. Признаком шестнадцатеричной - 0x или 0X перед числом. Для шестнадцатеричных цифр используются латинские буквы от A до F (неважно большие или маленькие).

Таким образом, следующий фрагмент программы:

```c
const int x = 240;
```
const int y = 0360;
const int z = 0xF0;

определяет три целые константы x, y и z с одинаковыми значениями.
Отрицательные числа предваряются знаком минус -. Приведем еще несколько примеров:

// ошибка в записи восьмеричного числа
const unsigned long 11 = 0678;
// правильная запись
const short a = 0xa4;
// ошибка в записи десятичного числа
const int x = 23F3;

Для целых чисел определены стандартные арифметические операции сложения (+), вычитания (-), умножения (*), деления (/), нахождение остатка от деления (%), изменение знака (-). Результатом этих операций также является целое число. При делении остаток отбрасывается. Примеры выражений с целыми величинами:

x + 4;
30 - x;
x * 2;
-x;
10 / x;
x % 3;

Кроме стандартных арифметических операций, для целых чисел определен набор битовых (или поразрядных) операций. В них целое число рассматривается как строка битов (нулей и единиц при записи числа в двоичной системе счисления или разрядов машинного представления).

К этим операциям относятся поразрядные операции И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ, поразрядное отрицание и сдвиги. Поразрядная операція ИЛИ, обозначаемая знаком | , выполняет операцию ИЛИ над каждым индивидуальным битом двух своих operandов. Например 1 | 2 в результате дают 3 поскольку в двоичном виде 1 это 01, 2 - это 10, соответственно операция ИЛИ дает 11 или 3 в десятичной системе (мы опустили нули слева).

Аналогично выполняются поразрядные операции И, ИСКЛЮЧАЮЩЕЕ ИЛИ и отрицание.

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>результат</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 &amp; 7</td>
<td>результат</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4 ^ 7</td>
<td>результат</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0 &amp; 0xF</td>
<td>результат</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>~0x00F0</td>
<td>результат</td>
<td>0xFF0F</td>
<td></td>
</tr>
</tbody>
</table>

Операция сдвига сдвигает двоичное представление левого операнда на количество битов, соответствующее значению правого операнда.
Например, двоичное представление короткого целого числа 3 - 00000000000000011. Результатом операции 3 << 2 (сдвиг влево) будет двоичное число 0000000000001100 или в десятичной записи 12. Аналогично, сдвинув число 9 (в двоичном виде 0000000000001001) вправо на 2 разряда (записывается 5 >> 2) получим 0000000000000010, т.е. 2.

При сдвиге влево, число дополняется нулями справа. При сдвиге вправо бит, которым дополняется число, зависит от того, знаковое оно или беззнаковое. Для беззнаковых чисел при сдвине вправо они всегда дополняются нулевым битом. Если же число знаковое, то значение самого левого бита числа используется для дополнения. Это объясняется тем, что самый левый бит как раз и является знаком - 0 означает плюс и 1 означает минус. Таким образом, если:

```c
short x = 0xFF00;
unsigned short y = 0xFF00;
```

tо результатом x >> 2 будет 0xFFC0 (двоичное представление 111111111000000), а результатом y >> 2 будет 0x3FC0 (двоичное представление 001111111000000).

Рассмотренные арифметические и поразрядные операции выполняются над целыми числами и в результате дают целое число. В отличие от них операции сравнения выполняются над целыми числами, но в результате дают логическое значение истина (true) или ложь (false).

Для целых чисел определены операции сравнения: равенства (==), неравенства (!=), больше (>), меньше (<), больше или равно (>=) и меньше или равно (<=).

Последний вопрос, который мы рассмотрим в отношении целых чисел — это преобразование типов. В языке C++ допустимо смешивать различные целые типы в выражении. Например, вполне допустимо записать x + y, где x типа short, а y — типа long. При выполнении операции сложения величина переменной x преобразуется к типу long. Такое преобразование всегда можно произвести и оно безопасно, т.е. при нем мы не теряем никаких значащих цифр. Общее правило преобразования целых типов состоит в том, что более короткий тип при вычислениях преобразуется к более длинному. Только при выполнении присваивания, длинный тип может преобразовываться к более короткому. Например:

```c
short x;
long y = 15;

x = y; // преобразование длинного типа к более короткому
```

Такое преобразование не всегда безопасно, при нем могут потеряться значащие цифры. Обычно компиляторы выдают предупреждение или ошибку, встречающая такое преобразование.
Упражнения

1. Попробуйте, разрешает ли Ваш компилятор C++ преобразования из более длинного типа в более короткий.

2. Напишите программу, которая выполняет битовые операции И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и НЕ над числами 5 и 12.

3. Напишите программу, которая демонстрирует отличия операции сдвига для знаковых и беззнаковых целых чисел.

4.1.2. Вещественные числа

Вещественные числа в C++ могут быть одного из трех типов: с одинаковой точностью float, с двойной точностью — double и с расширенной точностью — long double.

```c
float x;
double e = 2.9;
long double s;
```

В большинстве реализаций языка представление и диапазоны значений соответствуют стандарту IEEE (Institute of Electrical and Electronics Engineers) для представления вещественных чисел. Именно в соответствии с этим стандартом в табл. 2 и приведены диапазоны значений вещественных чисел. Точность представления чисел составляет 7 десятичных значащих цифр для типа float, 15 значащих цифр для double и 19 для типа long double.

Вещественные числа записываются либо в виде десятичных дробей: 1.3, 3.1415, 0.0005, либо в виде мантиссы и экспоненты: 1.2E0, 0.12e1. Отметим, что обе предыдущие записи изображают одно и тоже число 0.12.

По умолчанию вещественная константа принадлежит к типу double. Чтобы обозначить, что константа на самом деле float, нужно добавить символ f или F после числа: 2.7f. Символ l или L означает, что записанное число относится к типу long double.

```c
const float pi_f = 3.14f;
const double pi_d = 3.1415;
const long double pi_l = 3.1415L;
```

Для вещественных чисел определены все стандартные арифметические операции сложения (+), вычитания (-), умножения (*), деления (/) и изменения знака (-). В отличие от целых чисел, операция взятия по модулю не определена. Аналогично, все битовые операции и сдвиги не применямы к вещественным числам; они работают только с целыми числами.

Примеры операций:

```c
2 * pi;
(x - e) / 4.0
```
Число без знака принимает только положительные значения и значение ноль. Число со знаком принимает положительные значения, отрицательные значения и значение ноль.

Таблица 2. Встроенные типы данных языка C++.

<table>
<thead>
<tr>
<th>Название</th>
<th>Обозначение</th>
<th>Диапазон значений</th>
</tr>
</thead>
<tbody>
<tr>
<td>Байт</td>
<td>char</td>
<td>от -128 до +127</td>
</tr>
<tr>
<td>Байт без знака</td>
<td>unsigned char</td>
<td>от 0 до 255</td>
</tr>
<tr>
<td>Короткое целое число</td>
<td>short</td>
<td>от -32768 до +32767</td>
</tr>
<tr>
<td>Короткое целое число без знака</td>
<td>unsigned short</td>
<td>от 0 до 65535</td>
</tr>
<tr>
<td>Целое число</td>
<td>int</td>
<td>от -2147483648 до +2147483647</td>
</tr>
<tr>
<td>Целое число без знака</td>
<td>unsigned int (или просто unsigned)</td>
<td>от 0 до 4294967295</td>
</tr>
<tr>
<td>Длинное целое число</td>
<td>long</td>
<td>от -2147483648 до +2147483647</td>
</tr>
<tr>
<td>Длинное целое число без знака</td>
<td>unsigned long</td>
<td>от 0 до 4294967295</td>
</tr>
<tr>
<td>Вещественное число одинарной точности</td>
<td>float</td>
<td>от ±3.4e-38 до ±3.4e+38 (7 значащих цифр)</td>
</tr>
<tr>
<td>Вещественное число двойной точности</td>
<td>double</td>
<td>от ±1.7e-308 до ±1.7e+308 (15 значащих цифр)</td>
</tr>
<tr>
<td>Вещественное число увеличённой точности</td>
<td>long double</td>
<td>от ±1.2e-4932 до ±1.2e+4932</td>
</tr>
<tr>
<td>Логическое значение</td>
<td>bool</td>
<td>значения true (истина) или false (ложь)</td>
</tr>
</tbody>
</table>

Целое число может быть непосредственно записано в программе в виде константы. Запись чисел соответствует общей принятой нотации. Примеры целых констант: 0, 125, -37. По умолчанию целые константы принадлежат к типу int. Если необходимо указать, что целое число - это константа типа long, можно добавить символ L или l после числа. Если константа беззнаковая, т.е. относится к типу unsigned long, unsigned short или unsigned int, после числа записывается символ U или u. Например 34U, 700034L, 7654ul.

Кроме стандартной десятичной записи, числа можно записывать в восьмеричной или шестнаадцатеричной системе числения. Признаком восьмеричной системы счисления является цифра 0 в начале числа. Признаком шестнаадцатеричной - Ox или 0X перед числом. Для шестнаадцатеричных цифр используются латинские буквы от A до F (неважно большие или маленькие).

Таким образом, следующий фрагмент программы:

```c++
const int x = 240;
```
const int y = 0360;
const int z = 0xF0;

Определяет три целые константы х, у и z с одинаковыми значениями.

Однородительные числа предваряются знаком минус -. Приведем еще несколько примеров:

// ошибка в записи восьмеричного числа
const unsigned long ll = 0678;
// правильная запись
const short a = 0xa4;
// ошибка в записи десятичного числа
const int x = 23F3;

Для целых чисел определены стандартные арифметические операции сложения (+), вычитания (-), умножения (*), деления (/), нахождение остатка от деления (%), изменение знака (-). Результатом этих операций также является целое число. При делении остаток отбрасывается.

Примеры выражений с целыми величинами:

```
x + 4;
30 - x;
x * 2;
-x;
10 / x;
x % 3;
```

Кроме стандартных арифметических операций, для целых чисел определен набор битовых (или поразрядных) операций. В них целое число рассматривается как строка битов (нулей и единиц при записи числа в двоичной системе счисления или разрядов машинного представления).

К этим операциям относится поразрядные операции И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ, поразрядное отрицание и сдвиги. Поразрядная операция ИЛИ, обозначаемая знаком | , выполняет операцию ИЛИ над каждым индивидуальным битом двух своих operandов. Например 1 | 2 в результате дают 3 поскольку в двоичном виде 1 это 01, 2 - это 10, соответственно операция ИЛИ даёт 11 или 3 в десятичной системе (мы опускали нули слева).

Аналогично выполняются поразрядные операции И, ИСКЛЮЧАЮЩЕЕ ИЛИ и отрицание.

```
3 | 1  результат 3
4 & 7  результат 4
4 ^ 7  результат 3
0 & 0xF  результат 0
~0x00F0 результат 0xFF0F
```

Операция сдвига сдвигает двоичное представление левого операнда на количество битов, соответствующее значению правого операнда.
Вещественные числа можно сравнивать на равенство (==), неравенство (!=), больше (>), меньше (<), больше или равно (>=) и меньше или равно (<=). В результате операции сравнения получается логическое значение истина или ложь.

Если арифметическая операция применяется к двум вещественным числам разных типов, то менее точное число преобразуется к более точному, т.е. float преобразуется в double и double преобразуется к long double. Очевидно, что такое преобразование всегда можно выполнить без потери точности.

Если вторым операндом в операции с вещественным числом выступает целое число, то целое число преобразуется в вещественное представление.

Хотя любую целую величину можно представить в виде вещественного числа, при таком преобразовании возможна потеря точности (для больших чисел). Существуют специальные приемы работы с большими числами, когда необходимо сохранить высокую точность. Хорошим стартом для поиска необходимой информации будет монография Д. Кнута [2].

4.1.3. Логические величины

В языке C++ существует специальный тип для представления логических значений bool\(^2\). Для величин этого типа существует только два возможных значения: true (истина) и false (ложь). Объявление логической переменной выглядит следующим образом:

```cpp
bool condition;
```

Соответственно, существуют только две логические константы - истина и ложь. Они обозначаются соответственно true и false.

Для типа bool определены стандартные логические операции: логическое И (&&), ИЛИ (||) и НЕ (!).

```cpp
// истино, если обе переменные cond1 и cond2 истины
cond1 && cond2
// истино, если хотя бы одна из переменных истина
cond1 || cond2
// результат противоположен значению cond1
!cond1
```

Как мы уже отмечали ранее, логические значения получаются в результате операций сравнения. Кроме того, в языке C++ принято следующее правило преобразования чисел в логические значения: ноль соответствует значению false и любое отличное от нуля число преобразуется в значение true. Поэтому можно записать, например:

```cpp
int k = 100;
```

---

\(^2\) Этот тип появился сравнительно недавно и, например, компилятор Visual C++ полностью поддерживает его только в версиях 5.0 и старше.
while (k) { // выполнить цикл 100 раз
    k--; 
}

4.1.4. Символы и байты

Символьный или байтовый тип в языке C++ относится к целым числам, однако мы выделили их в отдельный раздел, потому что запись знаков имеет свои отличия.

Итак, для записи знаков в языке Си++ имеются типы char и unsigned char. Первый — это целое число со знаком, хранящееся в одном байте, второй — беззнаковое байтовое число. Эти типы чаще всего используются для манипулирования с символами, поскольку коды символов как раз помещаются в байт.

Пояснение. Единственное, что может хранить компьютер — это числа. Поэтому, для того, чтобы хранить и манипулировать символами, им присвоены коды — целые числа. Существует несколько стандартов, определяющих, какие коды соответствуют каким символам. Для английского алфавита и знаков препинания используется стандарт ASCII. Этот стандарт определяет коды от 0 до 127. Для представления русских букв используется стандарт KOI-8 или CP-1251. В этих стандартах русские буквы кодируются числами от 128 до 255. Таким образом все символы могут быть представлены в одном байте (максимальное число, помещающееся в одном байте — 255). Для работы с китайским, японским, корейским и рядом других алфавитов одного байта недостаточно и используется кодировка с помощью двух байтов и, соответственно, тип wchar_t (подробнее см. следующих подраздел).

Чтобы объявить переменную байтового типа, нужно записать:

    char c; // байтовое число со знаком
    unsigned char u; // байтовое число без знака

Поскольку байты — это целые числа, то все операции с целыми числами применимы и к байтам. Также и стандартная запись целочисленных констант применима к байтам, т.е. можно записать:

    c = 45;

где c - байтовая переменная. Однако для байтов существует и другая запись констант. Знак алфавита (буква, цифра, знак препинания), заключенный в апострофы является байтовой константой, например:

    'S', '€', '8', 'Ф'

Числовым значением такой константы является код данного символа, принятый в Вашей операционной системе.

В кодировке ASCII два следующих оператора эквивалентны:

    char c = 68;
Первый из них присваивает байтовой переменной с значение числа 65. Второй присваивает этой переменной код латинской буквы D, который в кодировке ASCII равен 68.

Для обозначения ряда непечатных символов используются так называемые экранированные последовательности — знак обратной дробной черты после которого стоит бука. Эти последовательности стандартны и заранее предопределены в языке:

\a звонок
\b возврат на один символ назад
\f перевод страницы
\n новая строка
\r перевод каретки
\t горизонтальная табуляция
\v вертикальная табуляция
\' апостроф
\" двойные кавычки
\/ обратная дробная черта
\? вопросительный знак

Для того, чтобы записать произвольное байтовое значение, также используется экранированная последовательность — после обратной дробной черты записывается целое число от 0 до 255.

```c
const char zero = '\0';
const unsigned char bitmask = '\0xFF';
const char tab = '\010';
```

Следующая программа выведет все печатные коды ASCII в порядке их кодов:

```c
for (char c = 32; c < 128; c++)
    cout << c << " ";
```

Однако напомним еще раз, что байтовые величины — это прежде всего целые числа, поэтому вполне допустимы выражения вида

'F' + 1
'a' < 23

и тому подобные. Тип char был придуман для языка C, от которого C++ достались все базовые типы данных. Язык C предназначен для программирования на достаточно "низком" уровне, приближенном к тому, как работает процессор ЭВМ, именно поэтому символ в нем — это лишь число.

В языке C++ в большинстве случаев для работы с текстом используются специально разработанные классы строк, о которых мы будем говорить позже.
Упражнение

Определите, каковы коды русских букв, применимые на Вашем компью-
тере. Для вывода числовых значений символов можно воспользоваться конструк-
цией:

```c
int i;
char c;
i = c;
cout << i;
```

4.1.4.1. Кодировка, многобайтовые символы

Мы уже кратко упомянули наличие разных кодировок букв, цифр,
знаков препинания и т.д. Алфавит большинства европейских языков мо-
жет быть представлен однобайтовыми числами (т.е. кодами в диапазоне
от 0 до 255). В большинстве кодировок принято, что первые 127 кодов
отводятся для символов, входящих в набор ASCII: ряд специальных сим-
волов, латинские заглавные и строчные буквы, арабские цифры и знаки
препинания. Вторая половина кодов — от 128 до 255 отводится под буквы
того или иного языка. Фактически вторая половина кодовой таблицы ин-
терпретируется по-разному, в зависимости от того, какой язык считается
"текущим". Один и тот же код может соответствовать русской букве 'З' для
кодировки русского языка, знаку '@' для кодировки английского и букве '✓'
французского языка.

Однако для таких языков, как китайский, японский и некоторые дру-
гие, одного байта недостаточно — алфавиты этих языков насчитывают бо-
лее 255 символов.

Перечисленные выше проблемы привели к созданию многобайто-
вых кодировок символов. Двухбайтовые символы в языке C++ предста-
вляются с помощью типа wchar_t:

```c
wchar_t wch;
```

Тип wchar_t иногда называют расширенным типом символов и де-
тали его реализации могут варьироваться от компилятора к компилятору,
в том числе может варьироваться и количество байт, отводимых под один
символ. Тем не менее, в большинстве случаев используется именно двух-
байтовое представление.

Константы типа wchar_t записываются в виде L'ab'.

4.1.5. Наборы перечислимых значений

Достаточно часто в программе удобно ввести тип, состоящий лишь
из нескольких заранее известных значений. Например, в программе нуж-
но манипулировать со времёнем суток и мы решили, что будем различать
ночь, утро, день и вечер. Конечно, можно договориться обозначить их
числами от 1 до 4. Во-первых, это не наглядно. Во-вторых, что даже более существенно, очень легко сделать ошибку и, например использовать число 5, которое не соответствует никакому времени дня. Гораздо удобней и надежнее определить набор значений с помощью типа enum языка Си++:

```cpp
e num DayTime { morning, day, evening, night };
```

Теперь можно определить переменную

```cpp
DayTime current;
```

которая хранит текущее время дня, а затем присваивать ей одно из допустимых значений типа DayTime:

```cpp
current = day;
```

Контроль, который производит компилятор, при использовании этой переменной в программе, существенно более строгий, чем при использовании целого числа.

Для наборов определены операции сравнения на равенство (==) и неравенство (!=) с атрибутами этого же типа, т.е.

```cpp
if (current != night)
  // выполнить работу
```

Вообще говоря, внутреннее представление значений набора — целые числа. По умолчанию элементам набора соответствуют последовательные целые числа начиная с 0. Этим можно пользоваться в программе. Во-первых, можно задать, какое число будет соответствовать какому атрибуту набора:

```cpp
enum { morning = 4, day = 3, evening = 2, night = 1};
```

// последовательные числа начиная с 1
```cpp
enum { morning = 1, day, evening, night };
```

// используются числа 0, 2, 3 и 4
```cpp
enum { morning, day = 2, evening, night };
```

Во-вторых, атрибуты наборов можно использовать в выражениях вместо целых чисел. Преобразования из набора в целое и наоборот разрешены.

Однако мы не рекомендуем такую практику. Для работы с целыми константами лучше использовать символические обозначения констант, а наборы использовать как наборы.

### 4.2. Классы и объекты

До сих пор мы говорили о встроенных типах, т.е. типах определенных в самом языке. Классы — это типы, определенные в конкретной программе. Определение класса состоит из определения, из каких составных
частей или атрибутов он состоит и какие операции определены для класса.

Предположим, в программе необходимо оперировать с комплексными числами. Комплексное число состоит из вещественной и мнимой части и с ними можно выполнять арифметические операции.

```cpp
class Complex {
public:
    int real;                // вещественная часть
    int imaginary;          // мнимая часть
    void Add(Complex x);    // прибавить комплексное число
};
```

Приведенный выше пример — упрощенное определение класса Complex, представляющее комплексное число. Комплексное число состоит из вещественной части — целого числа real и мнимой части; которая представлена целым числом imaginary. real и imaginary — это атрибуты класса. Для класса Complex определена одна операция или метод — Add.

Определив класс, мы можем создать переменную типа Complex:

```cpp
Complex number;
```

Переменная с именем number содержит значение типа Complex, содержит объект класса Complex. Имея объект, мы можем установить значения атрибутов объекта:

```cpp
number.real = 1;
number.imaginary = 2;
```

Операция "." обозначает обращение к атрибуту объекта. Создав еще один объект класса Complex, мы можем прибавить его к первому:

```cpp
Complex num2;
number.Add(num2);
```

Как легко заметить, метод Add выполняется с объектом. Имя объекта (или переменной, содержащей объект, что в общем одно и то же) — в данном случае number, записано первым. Через точку записано имя метода — Add с аргументом — значением другого объекта класса Complex, который прибавляется к number. Если Вы вспомните предыдущую главу, методы часто называются сообщениями. Но чтобы послать сообщение, необходим получатель. Таким образом, объекту number посылается сообщение Add с аргументом num2. Объект number принимает это сообщение и складывает свое значение со значением аргумента сообщения.

Данные рассуждения будут яснее, если мы определим, как выполняется операция сложения.
void Complex::Add(Complex x)
{
    this->real = this->real + x.real;
    this->imaginary = this->imaginary + x.imaginary;
}

Первые две строки говорят, что это метод Add класса Complex. В фигурных скобках записано определение операции или метода Add. Это определение говорит следующее: для того, чтобы прибавить значение объекта класса Complex к данному объекту, надо сложить вещественные части и запомнить результат в атрибуте вещественной части текущего объекта. Точно также надо сложить мнимые части двух комплексных чисел и запомнить результат в атрибуте текущего объекта, обозначающем мнимую часть.

Запись this-> говорит, что атрибут принадлежит к тому объекту, который выполняет метод Add (объекту, который получил сообщение Add). В большинстве случаев this-> можно опустить. В записи определения метода какого-либо класса упоминание атрибута класса без какой дополнительной информации означает, что речь идет об атрибуте текущего объекта.

Теперь приведем полностью этот небольшой пример:

```cpp
// определение класса комплексных чисел
class Complex {
public:
    int real;       // вещественная часть
    int imaginary;  // мнимая часть
    void Add(Complex x); // прибавить комплексное число
};

// определение метода сложения
void Complex::Add(Complex x)
{
    real = real + x.real;
    imaginary = imaginary + x.imaginary;
}

void main()
{
    Complex number;       // первый объект класса Complex
    number.real = 1;
    number.imaginary = 3;
    Complex num2;         // второй объект класса Complex
    num2.real = 2;
    num2.imaginary = 1;
}
number.Add(num2);  // прибавить значение второго  
// объекта к первому  
}

В языке Си++ можно сделать так, что класс будет практически не-  
отличим от предопределенных встроенных типов при использовании в  
выражениях. Для класса можно определить операции сложения, умноже- 
ния и т.д. пользуясь стандартной записью таких операций, т.е. x + y. В  
языке Си++ считается, что подобная запись — это также вызов метода с  
именем operator+ того класса, к которому принадлежит переменная x.  
Перепишем определение класса Complex:

```
// определение класса комплексных чисел  
class Complex
{
  public:
    int real;  // вещественная часть
    int imaginary;  // мнимая часть
    // прибавить комплексное число
    Complex operator+(const Complex x) const;
};
```

Вместо метода Add появился метод operator+. Изменилось и его  
opределение. Во-первых, этот метод возвращает значение типа Complex  
(операция сложения в результате дает новое значение того же типа, что и  
tипы операндов). Во-вторых, перед аргументом метода появилось ключе- 
вое слово const. Это слово обозначает, что при выполнении данного ме- 
tода аргумент изменяться не будет. Также const появилось после объя- 
влении метода. Второе ключевое слово const означает, что объект,  
выполняющий метод, не будет изменен. (При выполнении операции сло- 
жения x + y над двумя величинами x и y сами эти величины не изменя- 
ются). Теперь запишем определение определение операции сложения:

```
Complex  
Complex::operator+(const Complex x) const
{
  Complex result;
  result.real = real + x.real;
  result.imaginary = imaginary + x.imaginary;
  return result;
}
```

**Упражнение**

Определите операции вычитания, умножения и деления для объектов класса  
Complex.

**4.2.1. Подписи методов и необязательные аргументы**

Как и при объявлении функций, язык Си++ допускает определение  
в одном классе нескольких методов с одним и тем же именем, но разными  
tипами и количеством аргументов. (Определение методов или атрибутов с
одинаковыми именами в разных классах не вызывает проблем, поскольку пространства имен разных классов не пересекаются. Подробнее см. параграф 7.4).

// определение класса комплексных чисел
class Complex
{
public:
    int real;           // вещественная часть
    int imaginary;      // мнимая часть
    // прибавить комплексное число
    Complex operator+(const Complex x) const;
    // прибавить целое число
    Complex operator+(long x) const;
};

В следующем примере вначале складываются два комплексных числа и вызывается первая операция +. Затем к комплексному числу прибавляется целое число, и тогда выполняется вторая операция сложения.

    Complex c1;
    Complex c2;
    long x;
    c1 + c2;
    c2 + x;

Аналогично можно задавать значения аргументов методов по умолчанию. За более подробным описанием отсылаем читателя к п. 3.3.5.2.

4.2.2. Запись классов

Как уже отмечалось раньше, выбор имен — это не праздный вопрос. Существует множество систем именования классов. Опишем ту, которой мы придерживаемся в данной книге.

Имена классов, их методов и атрибутов составляются из английских слов, описывающих их смысл, при этом, если слов несколько, они пишутся слитно. Имена классов начинаются с заглавной буквы, если название составлено из нескольких слов, каждое слово начинается с заглавной буквы, остальные буквы маленькие.

    Complex, String, StudentLibrarian

Имена методов классов также начинаются с большой буквы:

    Add, Concat

Имена атрибутов класса начинаются с маленькой буквы, однако если имя состоит из нескольких слов, последующие слова начинаются с большой:

    real, classElement

При записи определения класса мы придерживаемся той же системы расположения, что и при записи функций. Ключевое слово class и имя класса записываются в первой строке, открывающаяся фигурная скобка — на следующей строке, методы и атрибуты класса — на последующих строках с отступом.
5. Производные типы данных

5.1. Массивы

Массив – это коллекция нескольких величин одного и того же типа. Простейшим примером массива может служить набор из двенадцати целых чисел соответствующих числу дней в каждом календарном месяце:

```c
int days[12];

days[0] = 31; // январь
days[1] = 28; // февраль
days[2] = 31; // март
days[3] = 30; // апрель
days[4] = 31; // май

days[5] = 30; // июнь
days[6] = 31; // июль
days[7] = 31; // август
days[8] = 30; // сентябрь
days[9] = 31; // октябрь
days[10] = 30; // ноябрь
days[11] = 31; // декабрь
```

В первой строчке мы объявили массив из 12 элементов типа int и дали ему имя days. Остальные строки примера – присваивания значений элементам массива. Для того, чтобы обратиться к определенному элементу массива, используется операция индексации []. Как видно из примера, первый элемент массива имеет индекс 0, соответственно последний - 11.

При объявлении массива, его размер должен быть известен в момент компиляции, поэтому в качестве размера можно указывать только целую константу. При обращении же к элементу массива, в качестве значения индекса может выступать любая переменная или выражение, которое вычисляется во время выполнения программы и преобразуется к целому значению.

Предположим, мы хотим распечатать все элементы массива days. Для этого удобно воспользоваться циклом for.

```c
for (int i = 0; i < 12; i++) {
    cout << days[i];
}
```

Следует отметить, что при выполнении программы границы массива не контролируются. Если мы ошиблись и вместо 12 в вышеприведенном цикле написали 13, то компилятор не выдаст ошибки. При выполнении программы попытается напечатать 13-ое число. Что при этом
случится, вообще говоря не определено. Быть может программа выдаст собой. Более вероятно, что будет напечатано какое-то случайное 13-ое число. Выход индексов за границы массива — довольно часто встречающаяся ошибка, которую иногда очень трудно обнаружить. В дальнейшем, при изучении классов, мы рассмотрим, как можно переопределить операцию [] и добавить контроль за индексами.

Отсутствие контроля индексов налагает большую ответственность на программиста. С другой стороны, индексация — настолько часто используемая операция, что наличие контроля несомненно серьезно повлияло бы на производительность программы.

Рассмотрим еще один пример. Предположим, что имеется массив из 100 целых чисел и его необходимо отсортировать, т.е. расположить в порядке возрастания. Сортировка методом “пузырька” — наиболее простая и распространенная — будет выглядеть следующим образом:

```c
int array[100];

for (int i = 0; i < 99; i++) {
    int max = array[i];
    for (int j = i + 1; j < 100; j++) {
        if (array[j] > max)
            max = array[j];
    }
    array[i] = max;
}
```

В приведенных примерах у массивов имеется только один индекс. Такие одномерные массивы часто называются векторами. Имеется возможность определить массивы с несколькими индексами или размерностями. Например объявление

```c
int m[10][5];
```

объявляет матрицу целых чисел размером 10 на 5. По другому интерпретировать вышеприведенное объявление можно как массив из 10 элементов, каждый из которых — вектор целых чисел длиной 5. Общее количество целых чисел в массиве `m` равно 50.

Обращение к элементам многомерных массивов аналогично обращению к элементам векторов: `m[1][2]` обращается к третьему элементу второй строки матрицы `m`.

Количество размерностей в массиве может быть произвольным. Также как и с вектором, при объявлении многомерного массива все его размеры должны быть заданы константами.

При объявлении массива можно присвоить начальные значения его элементам (инициализовать массив). Для вектора это будет выглядеть следующим образом:

```c
```
При инициализации многомерных массивов каждая размерность должна быть заключена в фигурные скобки

double temp[2][3] = {
    {3.2, 3.3, 3.4},
    {4.1, 3.9, 3.9}};

Интересной особенностью инициализации многомерных массивов является возможность не задавать размеры всех измерений массива, кроме самого последнего. Вышеприведенный пример можно переписать так:

double temp[][3] = {
    {3.2, 3.3, 3.4},
    {4.1, 3.9, 3.9}};

    // Вычислить размер пропущенной размерности
const int size_first = sizeof temp / sizeof (double[3]);

Упражнения
1. Напишите программу, в которой определяется двухмерная матрица целых чисел. Инициализируется какими-либо значениями, затем она сортируется по строкам и выводится на терминал также по строкам. Как нужно изменить программу для сортировки и вывода по столбцам?
2. Напишите программу перестановки элементов вектора в обратном порядке.

5.2. Структуры

Структуры — это ничто иное, как классы, у которых разрешен доступ ко всем их элементам (доступ к определенным атрибутам класса может быть ограничен, о чем мы узнаем в главе 6). Пример структуры:

```c
struct Record {
    int number;
    char name[20];
};
```

Также, как и для классов, операция "." обозначает обращение к элементу структуры.

В отличие от классов, можно определить переменную-структуру без определения отдельного типа:

```c
struct {
    double x;
    double y;
} coord;
```

Обратиться к атрибутам переменной coord можно coord.x и coord.y.
5.2.1. Битовые поля

В структуре можно определить размеры атрибута с точностью до бита. Традиционно структуры использовались в системном программировании для описания регистров аппаратуры. В них каждый бит имеет своё значение. Не менее важным является возможность экономии памяти — ведь минимальный тип атрибута структуры это байт (char) который занимает 8 битов. До сих пор, несмотря на мегабайты и даже гигабайты оперативной памяти, доступные в современных компьютерах, существует немало задач, где каждый бит на счету.

Если после описания атрибута структуры поставить двоеточие и затем целое число, то это число задает количество битов, выделенных под данный атрибут структуры. Такие атрибуты называют битовыми полями. Следующая структура хранит в компактной форме дату и время дня с точностью до секунды.

```c
struct TimeAndDate {
    unsigned hours : 5;  // часы от 0 до 24 (5 битов)
    unsigned mins : 6;   // минуты (6 битов)
    unsigned secs : 6;   // секунды от 0 до 60 (6 битов)
    unsigned weekDay : 3; // день недели
    unsigned monthDay : 6; // день месяца от 1 до 31
    unsigned month : 5;   // месяц от 1 до 12
    unsigned year : 8;    // год от 0 до 100
};
```

Одна структура TimeAndDate требует всего 39 битов, т.е. 5 байтов (один байт - 8 битов). Если бы использовали для каждого атрибута этой структуры тип char, то нам бы потребовалось 7 байтов.

5.3. Объединения

Особым видом структур данных является объединение. Форма определения объединения напоминает определение структуры, только вместо ключевого слова struct используется union:

```c
union number {
    short sx;
    long lx;
    double dx;
};
```

В отличие от структуры, все атрибуты объединения располагаются по одному адресу. Под объединение выделяется столько памяти, сколько нужно для хранения наибольшего атрибута объединения. Объединения применяются в тех случаях, когда в один момент времени используется только один атрибут объединения и прежде всего для экономии памяти. Предположим, нам нужно определить структуру, которая хранить
"универсальное" число, т.е. число одного из предопределенных типов и признак типа. Это можно сделать следующим образом:

```c
struct Value {
    enum NumberType { ShortType, LongType, DoubleType };
    NumberType type;
    short sx;       // если type равен ShortType
    long lx;        // если type равен LongType
    double dx;      // если type равен DoubleType
};
```

Атрибут type содержит тип хранимого числа, а соответствующий атрибут структуры – значение числа.

```c
Value shortVal;
shortVal.type = Value::ShortType;
shortVal.sx = 15;
```

Хотя память выделяется под все три атрибута sx, lx и dx, реально используется только один из них. Съэкономить память можно, используя объединение:

```c
struct Value {
    enum NumberType { ShortType, LongType, DoubleType };
    NumberType type;
    union number {
        short sx;       // если type равен ShortType
        long lx;        // если type равен LongType
        double dx;      // если type равен DoubleType
    } val;
};
```

Теперь память выделена только для максимального из этих трех атрибутов (в данном случае dx). Однако и обращаться с объединением надо осторожно. Поскольку все три атрибута делят одну и ту же область памяти, изменение одного из них означает изменение всех остальных. Рис. 4 поясняет выделение памяти под объединение. При обоих вариантах мы предполагаем, что структура расположена по адресу 1000. Объединение располагает все три своих атрибута по одному и тому же адресу.

**Замечание.** Объединения существовали в языке Си, откуда без изменений и перешли в Си++. Использование наследования классов, описанное в следующей главе, позволяет во многих случаях достигнуть того же эффекта без использования объединений, причем программа будет более надежной.

**Упражнение**

Напишите программу, использующую тип `Value` неправильно, т.е. присваивающую, например, значение атрибуту lx при type `DoubleType`. Напечатайте значения sx и dx.
5.4. Указатели

Указатель – это производный тип, который означает адрес какого-либо значения. В языке C++ широко используется понятие адреса переменных. Работа с адресами досталась C++ в наследство от языка Си. Предположим, что в программе определена переменная типа int:

```
int x;
```

Можно определить переменную типа указатель на целое число:

```
int* xptr;
```

и присвоить переменной xptr адрес переменной x:

```
xptr = &x;
```
Операция & — примененная к переменной — это операция взятия адреса. Операция * — примененная к адресу — это операция обращения по адресу. Таким образом, два оператора эквивалентны:

```c
int y = x;      // присвоить переменной y значение x
int y = *xpstr; // присвоить переменной y значение, находящееся по адресу xpstr
```

С помощью операции обращения по адресу можно записывать значения:

```c
*xpstr = 10; // записать число 10 по адресу xpstr
```

После выполнения этого оператора, значение переменной x станет равным 10, поскольку xpstr указывает на переменную x.

Указатель — это не просто адрес, а адрес величины определенного типа. Указатель xpstr — адрес целой величины. Определить адреса величин других типов можно следующим образом:

```c
unsigned long* lpstr; // указатель на целое число без знака
char* cp;             // указатель на байт
Complex* p;           // указатель на объект класса Complex
```

Если указатель указывает на класс, то операция обращения к атрибуту класса вместо точки обозначается "->", например p->real. Если вспомнить один из предыдущих примеров:

```c
void
Complex::Add(Complex x)
{
    this->real = this->real + x.real;
    this->imaginary = this->imaginary + x.imaginary;
}
```

tо this — это указатель на текущий объект, т.е. объект, который выполняет метод Add. Запись this-> означает обращение к атрибуту текущего объекта.

Можно определить указатель на любой тип, в том числе на функцию или метод класса. Если имеется несколько функций одного и того же типа:

```c
int foo(long x);
int bar(long x);
```

можно определить переменную типа указатель на функцию и вызывать эти функции не напрямую, а косвенно, через указатель:

```c
int (*funcptr)(long x);
funcptr = &foo;
(funcptr)(2);
```
Для чего нужны указатели? Указатели появились прежде всего для нужд системного программирования. Поскольку язык Си предназначался для "низкоуровневого" программирования, на нем нужно было обращаться к, например, регистрам устройств. У этих регистров вполне определенные адреса, т.е. необходимо было прочитать или записать значение по определенному адресу. С помощью механизма указателей такие операции не требуют никаких дополнительных средств языка.

```c
int* hardwareRegister = 0x80000;
*hardwareRegister = 12;
```

Однако использование указателей далеко не ограничивается нуждами системного программирования. Указатели позволяют существенно упростить и ускорить ряд операций. Предположим, в программе имеется область памяти для хранения промежуточных результатов вычислений. Эту область памяти используют разные модули программы. Вместо того, чтобы каждый раз при обращении к модулю копировать эту область памяти, мы можем передавать указатель в качестве аргумента вызова функции, тем самым упрощая и ускоряя вычисления.

```c
struct TempResults {
    double x1;
    double x2;
} tempArea;

// Функция calc возвращает истину, если вычисления были успешны и ложь при наличии ошибки.
// Вычисленные результаты записываются на место аргументов по адресу, переданному в указателе trPtr
bool calc(TempResults* trPtr)
{
    // вычисления
    if (noerrors) {
        trPtr->x1 = res1;
        trPtr->x2 = res2;
        return true;
    } else {
        return false;
    }
}
```

```c
void fun1( void)
{
    ...
}```
В приведенном примере проиллюстрированы сразу две возможности использования указателей: передача адреса общей памяти и возможность функции иметь более одного значения в качестве результата. Структура TempResults используется для хранения каких-то данных. Вместо того, чтобы передавать эти данные по-отдельности, в функцию calc передается указатель на структуру. Этим достигаются две цели: большая наглядность и большая эффективность (не надо копировать элементы структуры по одному). Функция calc возвращает булевское значение — признак того, завершились ли вычисления успешно. Сами же результаты вычислений записываются в структуру, указатель на которую передан в качестве аргумента.

Упомянутые нами примеры применения указателей никак не связаны с объектно-ориентированным программированием. Казалось бы, объектно-ориентированное программирование должно уменьшить зависимость от низкоуровневых конструкций типа указателей. На самом деле программирование с классами нисколько не уменьшило нужду в указателях и даже наоборот, нашло им еще большее применение, о чем мы будем рассказывать по ходу изложения.

5.4.1. Адресная арифметика

С указателями можно выполнять не только операции присваивания и обращения по адресу, но и ряд арифметических операций. Прежде всего, указатели одного и того же типа можно сравнивать с помощью стандартных операций сравнения. При сравнении указателей сравниваются их значения, а не значения величин, на которые данные указатели указывают. Так в нижеприведенном примере результат первой операции сравнения будет ложным:

```cpp
int x = 10;
int y = 10;
int* xptr = &x;
int* yptr = &y;

// сравниваем указатели
if (xptr == yptr) {
    cout << "Указатели равны" << endl;
} else {
    cout << "Указатели неравны" << endl;
}
```
// сравниваем значения, на которые указывают указатели
if (*xptr == *y.ptr) {
    cout << "Значения равны" << endl;
} else {
    cout << "Значения неравны" << endl;
}

Однако результат второй операции сравнения будет истинным, поскольку переменные x и y имеют одно и то же значение.

Кроме того, над указателями можно выполнять ограниченный набор арифметических операций. К указателю можно прибавить или вычесть целое число. Результатом прибавления единицы к указателю является адрес следующей величины типа, на который указывает указатель, в памяти. Поясним это на рисунке. Пусть x_Ptr - указатель на целое число типа long, а ср - указатель на тип char. Начиная с адреса 1000 в памяти расположены два целых числа. Адрес второго - 1004 (в большинстве реализаций Си++ под тип long выделяется четыре байта). Начиная с адреса 2000 расположены объекты типа char.

![Рис. 5. Адресная арифметика](image)

Размер памяти, выделяемой для числа типа long и для char, различен. Поэтому изменение адреса для при увеличении x_Ptr и ср тоже различно. Однако и в том и в другом случае увеличение указателя на единицу означает переход к следующей в памяти величине того же типа. Прибавление или вычитание любого целого числа работает по тому же принципу, что и увеличение на единицу. Указатель сдвигается вперед (при прибавлении положительного числа) или назад (при вычитании положи-
тельного числа) на соответствующее количество объектов того типа, на который указатель указывает. Вообще говоря неважно, объекты какого типа на самом деле находятся в памяти - адрес просто увеличивается или уменьшается на необходимую величину. На самом деле значение указателя ptr всегда изменяется на число, равное sizeof(*ptr).

Указатели одного и того же типа можно вычитать друг из друга. Разность указателей показывает, сколько объектов соответствующего типа может поместиться между указанными адресами.

5.4.2. Связь между массивами и указателями

Существует определенная связь между указателями и массивами. Предположим, имеется массив из 100 целых чисел. Запишем двумя способами программу суммирования элементов этого массива:

```c
long array[100];
long sum = 0;
for (int i = 0; i < 100; i++)
    sum += array[i];
```

То же самое можно сделать с помощью указателей:

```c
long array[100];
long sum = 0;
for (long* ptr = &array[0]; ptr < &array[99] + 1; ptr++)
    sum += *ptr;
```

Элементы массива расположены последовательно в памяти и увеличение указателя на единицу означает смещение к следующему элементу массива. Упоминание имени массива без индексов преобразуется к адресу его первого элемента:

```c
for (long* ptr = array; ptr < &array[99] + 1; ptr++)
    sum += *ptr;
```

Хотя смещение указателей и массивов вполне законно, мы бы не стали рекомендовать такой стиль, особенно начинающим программистам.

При использовании многомерных массивов указатели позволяют обращаться к срезам или подмассивам. Если мы объявим трехмерный массив `exmpl`:

```c
long exmpl[5][6][7]
```

5.4.3. Бестиповый указатель

Особым случаем указателей является бестиповый указатель. Ключевое слово void используется для того, чтобы показать, что указатель означает просто адрес памяти, независимо от типа величины, находящейся по этому адресу:

```c
void* ptr;
```

Для указателя на тип void не определена операция ->, не определена операция обращения по адресу *, не определена адресная арифметика. Использование бестиповых указателей ограничено работой с памятью при использовании ряда системных функций, передачей адресов в функции, написанные на языках программирования более низкого уровня, например на ассемблере.

В программе на языке C++ бестиповый указатель может применяться там, где адрес интерпретируется по-разному, в зависимости от каких-либо динамически вычисляемых условий. Например, приведенная ниже функция будет печатать целое число, содержащееся в одном, двух или четырех байтах, расположенных по передаваемому адресу:

```c
void printbytes(void* ptr, int nbytes)
{
    if (nbytes == 1) {
        char* cptr = (char*)ptr;
        cout << *cptr;
    } else if (nbytes == 2) {
        short* sptr = (short*)ptr;
        cout << *sptr;
    } else if (nbytes == 4) {
        long* lptr = (long*)ptr;
        cout << *lptr;
    } else {
        cout << "Неверное значение аргумента";
    }
}
```

Как видите, в примере используется операция явного преобразования типа. Имя типа, заключенное в круглые скобки, стоящее перед выражением, преобразует значение этого выражения к указанному типу. Разумеется, эта операция может применяться к любым указателям.

5.4.4. Нулейвой указатель

В программах на языке C++ значение указателя, равное нулю, используется в качестве "неопределенного" значения. Например, если какая-то функция вычисляет значение указателя, то чаще всего возвращение нулевого значения означает ошибку.
В языкe Си++ определена символьическая константа NULL для обозначения нулевого значения указателя.
Такое использование нулевого указателя было основано на том, что по адресу 0 никогда не могут располагаться данные программы, он зарезервирован операционной системой для своих нужд. Однако во многом нулевой указатель — просто удобное соглашение, которого все придерживаются.

5.5. Строки и литералы

Для того, чтобы работать c текстом, в языке Си++ не существует особого встроенного типа данных. Текст представляется в виде последовательности знаков (байтов), закончивающихся нулевым байтом. Иногда такое представление называют Си-строки, поскольку оно появилось в языке Си. Кроме того, в Си++ можно создать классы для более удобной работы с текстами (готовые классы для представления строк имеются в стандартной библиотеке шаблонов, с которой мы познакомимся в последующих главах).

Строки представляются в виде массива байтов:

```c
char string[20];

string[0] = 'H';
string[1] = 'e';
string[2] = 'l';
string[3] = 'l';
string[4] = 'o';
string[5] = 0;
```

В массиве string записана строка "Hello". При этом фактически мы использовали только 6 из 20 элементов массива.

Для записи строковых констант в программе используются литералы. Литерал - последовательность знаков, заключенная в двойные кавычки:

"Это строка"
"0123456789"
"++"
Заметим, что символ, заключенный в двойные кавычки, отличается от символа, заключенного в апострофы. Литерал "*" обозначает два байта: первый байт содержит символ звездочки, второй байт содержит ноль. Константа '*' обозначает один байт, содержащий знак звёздочки.
С помощью литералов можно инициализировать массивы:

```c
char alldigits[] = "0123456789";
```

Размер массива явно не задан, он определяется исходя из размера инициализирующего его литерала, в данном случае 11 (10 символов плюс нулевой байт).
При работе со строками особенно часто используется связь между массивами и указателями. Значение литерала — это массив неизменяемых байтов необходимого размера. Строковый литерал может быть присвоен указателю на char:

```c
const char* message = "Сообщение программы";
```

Значение литерала — это адрес его первого байта, указатель на начало строки. В следующем примере функция CopyString копирует первую строку во вторую:

```c
void
CopyString(char* src, char* dst)
{
    while (*dst++ = *src++)
    {
        *dst = 0;
    }
}
```

```c
void
main()
{
    char first[] = "Первая строка";
    char second[100];
    CopyString(first, second);
}
```

Указатель на байт (тип char*) указывает на начало строки. Предположим, нам нужно подсчитать количество цифр в строке, на которую указывает указатель str:

```c
#include <ctype.h>

int count = 0;
while (*str != 0) { // признак конца строки - ноль
    if (isdigit(*str++)) // проверить байт, на который
        count++;
    // указывает str и сдвинуть // указатель на следующий байт
```
При выходе из цикла while переменная count содержит количество цифр в строке str, а сам указатель str указывает на конец строки — нулеевой байт. Для проверки того, что текущий символ — цифра, используется функция isdigit. Это одна из многих стандартных функций языка для работы с символами и строками.

С помощью функций стандартной библиотеки языка реализованы многие часто используемые операции над символьными строками. В большинстве своём в качестве строк они воспринимают указатели. Приведем ряд наиболее употребительных. Прежде, чем использовать их в программе, нужно подключить их описания с помощью операторов #include <string.h> и #include <ctype.h>.

char* strcpy(char* target, const char* source);

Копировать строку source по адресу target, включая завершающий нулеевой байт. Функция предполагает, что память, выделенная по адресу target, достаточна для копируемой строки. В качестве результата функция возвращает адрес первой строки.

char* strcat(char* target, const char* source);

Присоединить вторую строку с конца первой, включая завершающий нулеевой байт. На место завершающего нулевого байта первой строки переписывается первый символ второй строки. В результате по адресу target получается строка, образованная слиянием первой со второй. В качестве результата функция возвращает адрес первой строки.

int strcmp(const char* string1, const char* string2);

Сравнить две строки в лексикографическом порядке (по алфавиту). Если первая строка должна стоять раньше по алфавиту, чем первая, результат функции меньше нуля, если позже — больше нуля и ноль, если две строки равны.

size_t strlen(const char* string);

Определить длину строки в байтах, не считая завершающего нулеового байта.

В следующем примере, использующем приведенные функции, в массиве result будет образована строка "1 января 1998 года, 12 часов".

char result[100];
char* date = "1 января 1998 года";
char* time = "12 часов";
strcpy(result, date);
strcat(result, ", ", time);
Как видно из этого примера, литералы можно непосредственно использовать в выражениях.
Определить массив строк можно с помощью следующего объявления:

```c
char* StrArray[5] =
{ "one", "two", "three", "four", "five" };
```

**Упражнение**

1. Напишите программу сортировки массива строк по алфавиту.
2. Как бы вы реализовали функцию strcat?

**5.6. Распределение памяти**

В языке Си++ существует три способа выделения памяти для используемых в программе данных: автоматическое, статическое и динамическое.

Самый простой метод — это объявление переменных внутри функций. Если переменная объявлена внутри функции, каждый раз, когда функция вызывается, под переменную автоматически отводится память. Когда функция завершается, память, занимаемая переменными, автоматически освобождается. Такие переменные называют автоматическими.

При создании автоматических переменных они никак не инициализируются, т.е. значение автоматической переменной сразу после ее создания не определено, нельзя предсказать каково будет это значение. Соответственно, перед использованием автоматических переменных их необходимо либо явно инициализировать, либо присвоить какое-либо значение.

```c
int funct()
{
    double f;       // значение f неопределено
    f = 1.2;        // теперь значение f определено

    // явная инициализация автоматической переменной
    bool result = true;
}
```

Аналогично автоматическим переменным, объявленным внутри функции, автоматические переменные, объявленные внутри блока (последовательности операторов, заключенных в фигурные скобки) создают-ся при входе в блок и уничтожаются при выходе из блока. В программе сортировки (параграф 5.1) переменная max создается заново на каждой итерации цикла.
Замечание. Распространенной ошибкой является использование адреса автоматической переменной после выхода из функции. Конструкция типа:

```c
int*
func()
{
  int x;
  // ...
  return &x;
}
```

dает непредсказуемый результат.

Другим способом выделения памяти является статическое.
Если переменная определена вовне функции, память для нее отводится статически, один раз в начале выполнения программы, и переменная уничтожается только тогда, когда программа завершается. Можно статически выделить память и под переменную, определенную внутри функции или блока. Для этого нужно использовать ключевое слово `static` в его определении:

```c
double globalMax; // переменная определена вовне функции
void
func(int x)
{
  static bool visited = false;

  if (!visited) {
    // инициализация
    visited = true;
  }
  // ...
}
```

В данном примере переменная `visited` создается в начале выполнения программы. Ее начальное значение — `false`. При первом вызове функции `func` условие в операторе `if` будет истинным, выполнится инициализация и переменной `visited` будет присвоено значение `true`. Поскольку статическая переменная создается только один раз, то и её значения сохраняются между вызовами функции. При втором и последующих вызовах функции `func` инициализация не будет производиться.

Если бы переменная `visited` не была объявлена `static`, то инициализация происходила бы при каждом вызове функции.

Третьим способом выделения памяти с языке C++ является динамический. Память для величины какого-либо типа можно выделить выполнив операцию `new`. В качестве операнда выступает название типа, а результатом является адрес выделенной памяти.
long* lp;
lp = new long;    // создать новое целое число
Complex* cp;
cp = new Complex; // создать новый объект типа Complex

Созданный таким образом объект существует до тех пор, пока па-
мять не будет явно освобождена с помощью операции delete. В качестве
операнда delete должен быть задан адрес, возвращенный операцией new:

delte lp;
delte cp;

Динамическое распределение памяти используется прежде всего
тогда, когда заранее не известно, сколько объектов понадобится в про-
грамме и понадобятся ли они вообще. С помощью динамического распреде-
ления памяти можно гибко управлять временем жизни объектов, на-
пример выделить память не в самом начале программы (как для
глобальных переменных), но тем не менее сохранять нужные данные в
эй памяти до конца программы.

Если необходимо динамически создать массив, то нужно использо-
ватель немного другой форму new:

new int[100];

В отличии от определения переменной типа массив, размер масси-
ва в операции new может быть произвольным, в том числе вычисляемым
в ходе выполнения программы. (Напомним, что при объявлении пере-
менной типа массив, размер массива должен быть константой.)

Освобождение памяти, выделенной под массив, должно быть вы-
полнено с помощью следующей операции delete:

delte [] address;

В следующем фрагменте программы мы динамически выделяем
память под строку переменной длины и копируем туда исходную строку

// стандартная функция strlen подсчитывает
// количество символов в строке
int length = strlen(src_str);

// выделить память и добавить один байт
// для завершающего нулевого байта
char* buffer = new char[length + 1];
strcpy(buffer, src_str);    // копирование строки

Операция new возвращает адрес выделенной памяти. Однако не
гарантировано, что new всегда завершится успешно. Объем оперативной
памяти ограничен, и может случиться так, что найти еще один участок
свободной памяти невозможно. В таком случае new возвращает нулевой
указатель (адрес 0). Результат new необходимо проверять:

    char* newstr;
    newstr = new char[length];
    if (newstr == NULL) {  // проверить результат
        // обработка ошибок
    }
    // память выделена успешно

Замечание. Указатели и динамическое распределение памяти — очень
мощные средства языка. С их помощью можно разрабатывать гибкие и очень
эффективные программы. В частности, одна из областей применения Си++ — сис-
темное программирование — практически было бы невозможно без возможности
работы с указателями. Однако возможности, которые получает программист при
работе с указателями, накладывают на него и большую ответственность. Наи-
большее количество ошибок в программу вносится при работе с указателями. Как
правило, эти ошибки являются наиболее трудными для обнаружения и исправле-
ния.

Приведем несколько характерных примеров.

Использование неверного адреса в операции delete. Результат такой опера-
ции непредсказуем. Вполне возможно, что сама операция пройдет успешно, одн.
нако внутренняя структура памяти будет испорчена, что приведет либо к ошибке
в следующей операции new, либо к порче какой-нибудь информации.

Пропущенное освобождение памяти, т.e. программа многократно выделяет
память под данные, но "забывает" её освобождать. Такие ошибки называют
утечками памяти. Во-первых, программа использует ненужную ей память, тем
самым понижая производительность. Кроме того, вполне возможно что в 99 слу-
чаях из 100 программа успешно выполнится. Однако, если потеря памяти ока-
жется слишком большой, программе не хватит памяти под какие-нибудь данные и,
соответственно, произойдет сбой.

Запись по неверному адресу. Скорее всего будут испорчены какие-либо
данные. Как проявится такая ошибка — неверным результатом, сбоем программы
или еще каким-нибудь образом — предсказать трудно.

Примеры ошибок можно приводить бесконечно. Общие их черты, обу-
славливающие сложность обнаружения, это, во-первых, непредсказуемость ре-
зультата и, во-вторых, проявление не в момент совершения ошибки, а позже,
быть может в месте программы, которое само по себе совершенно правильно
(неверная операция delete — сбой в последующей операции new, запись по невер-
ному адресу — использование испорченных данных в другой части программы и
т.п.).

Отнюдь не призывая отказаться от применения указателей (впрочем это
практически невозможно с Си++), мы хотим подчеркнуть, что их использование
требует дисциплины и внимательности. Несколько общих рекомендаций:
1. Используйте указатели и динамическое распределение памяти только там,
где это действительно необходимо. Проверьте, можно ли выделить память
статически или использовать автоматическую переменную.

75
2. Старайтесь локализовать распределение памяти. Если какой-либо метод выделяет память (в особенности под временные данные), он же и должен ее освободить.
3. Там, где это возможно, используйте ссылки вместо указателей.
4. Проверяйте программы с помощью специальных средств контроля за памятью (Purify компании Rational, Bounce Checker компании Nu-Mega и др.)

5.7. Ссылки

Ссылка — это еще одно имя переменной. Если имеется какая-либо переменная, например

```c
Complex x;
```

можно определить ссылку на переменную `x` как

```c
Complex& y = x;
```

и тогда `x` и `y` обозначают одну и ту же величину. Если выполнены операторы

```c
x.real = 1;
x.imaginary = 2;
```

tо `y.real` равно 1 и `y.imaginary` равно 2. Фактически, ссылка — это адрес переменной (поэтому при определении ссылки используется символ `&` — знак операции взятия адреса) и в этом смысле сходна с указателем, однако у ссылок есть свои существенные особенности.

Во-первых, определяя переменную типа ссылки, её необходимо инициализировать, указав на какую переменную она ссылается. Нельзя определить ссылку

```c
int& xref;
```

можно только

```c
int& xref = x;
```

Во-вторых, нельзя переопределить ссылку, т.е. изменить на что она ссылается. Если после определения ссылки `xref` мы выполняем присваивание

```c
xref = y;
```

то выполнится присваивание значения переменной `y` той переменной, на которую ссылается `xref`. Ссылка `xref` по-прежнему будет ссылаться на `x`. В результате выполнения следующего фрагмента программы:
int x = 10;
int y = 20;
int& xref = x;
xref = y;
x += 2;
cout << "x = " << x << endl;
cout << "y = " << y << endl;
cout << "xref = " << xref << endl;

будет выведено:

x = 22
y = 20
xref = 22

В-третьих, синтаксически обращение к ссылке аналогично обращению к переменной. Если для обращения к атрибуту объекта, на который указывает указатель, применяется операция ->, то для подобной же операции со ссылкой применяется точка "."

Complex a;
Complex* aptr = &a;
Complex& aref = a;

aptr->real = 1;
aref.imaginary = 2;

Аналогично указателю, ссылка сама по себе не представляет никакого значения. Ссылка должна на что-то ссылаться, тогда как указатель должен на что-то указывать.

5.7.1. Распределение памяти при передаче аргументов функции

Когда мы говорили о функциях, мы говорили, что у функций (также как и у методов классов) есть аргументы, фактические значения которых передаются при вызове функции.

Рассмотрим более подробно метод Add класса Complex. Изменим его немного, с тем, чтобы он вместо изменения состояния объекта возвращал результат операции сложения:

Complex
Complex::Add(Complex x)
{
Complex result;
result.real = real + x.real;
result.imaginary = imaginary + x.imaginary;
return result;
При вызове этого метода

```
Complex n1;
Complex n2;
...
Complex n3 = n1.Add(n2);
```

значение переменной n2 передается в качестве аргумента. Компилятор создает временную переменную типа Complex, копирует значение n2 в эту временную переменную и передает её в метод Add. Такая передача аргумента называется передачей по значению. У передачи аргументов по значению имеются два свойства. Во-первых, эта операція достаточно неэффективная, особенно если объект сложный и требует много памяти или создание объекта сопряжено с выполнением сложных действий (о конструкторах объектов см. раздел 6.3). Во-вторых, изменения аргумента функции не сохраняются. Если бы метод Add был бы определен как

```
Complex
Complex::Add(Complex x)
{
    Complex result;

    x.imaginary = 0; // изменение аргумента метода

    result.real = real + x.real;
    result.imaginary = imaginary + x.imaginary;
    return result;
}
```

tо при вызове n3 = n1.Add(n2) результат был бы конечно другой, но переменная n2 не изменила бы своего значения. Хотя в данном примере изменяется значение аргумента метода Add, этот аргумент — лишь копия объекта n2, а не сам объект. По завершению выполнения метода Add его аргументы просто уничтожаются без всякого эффекта на первоначальные значения фактических параметров.

При возврате результата функции происходят те же действия, т.е. создается временная переменная, в которую копируется результат и уже затем значение временной переменной копируется в переменную n3. Временные переменные потому и называют временными, что компилятор сам их создает на ограниченное время выполнения метода и сам их уничтожает.

Другим способом передачи аргументов является передача по ссылке. Изменив описание метода Add на следующее:

```
Complex
Complex::Add(Complex& x)
{
    Complex result;
```
result.real = real + x.real;
result.imaginary = imaginary + x.imaginary;
return result;
}

при вызове n3 = n1.Add(n2) компилятор будет создавать ссылку на переменную n2 и передавать её методу Add. В большинстве случаев это намного эффективнее, так как для ссылки требуется немного памяти и её создание проще. Однако мы получим нежелательный в данном случае эффект. Метод

```cpp
Complex
Complex::Add(Complex& x)
{
    Complex result;

    x.imaginary = 0;    // изменение значения
    result.real = real + x.real;  // по переданной ссылке
    result.imaginary = imaginary + x.imaginary;
    return result;
}
```

изменит значение переменной n2. Операция Add не предусматривает изменения собственных operandov. Чтобы избежать ошибок, лучше записать аргумент с описателем const, который определяет соответствующую переменную как неизменяемую.

```cpp
Complex::Add(const Complex& x)
```

В таком случае попытка изменить значение аргумента будет обнаружена на этапе компиляции и компилятор выдаст ошибку. Передачей аргумента по неконстантной ссылке можно воспользоваться тогда, когда функция действительно должна изменить свой аргумент. Например, метод Coord класса Figure записывает координаты некой фигуры в свои аргументы:

```cpp
void
Figure::Coord(int& x, int& y)
{
    x = coordx;
    y = coordy;
}
```

При вызове

```cpp
int cx, cy;
Figure fig;
...
```
переменным $cx$ и $cy$ будет присвоено значение координат фигуры $fig$.

Вернемся к методу $Add$ и попробуем оптимизировать передачу вычисленного значения. Простое на первый взгляд решение возвращать ссылку на результат не работает:

```cpp
Complex&
Complex::Add(const Complex& x)
{
    Complex result;
    result.real = real + x.real;
    result.imaginary = imaginary + x.imaginary;
    return result;
}
```

При выходе из метода автоматическая переменная $result$ уничтожается и память, выделенная для неё, освобождается. Поэтому результат $Add$ — ссылка на несуществующую память. Результат подобных действий непредсказуем. Иногда программа будет работать как ни в чем не бывало, иногда может произойти сбой, иногда результат будет испорчен. Однако возвращение результата по ссылке возможно, если объект, на который эта ссылка ссылается, не уничтожается по выходу из функции или метода. Если метод $Add$ прибавляет значение аргумента к текущему значению объекта и возвращает новое значение в качестве результата, то его можно записать:

```cpp
Complex&
Complex::Add(const Complex& x)
{
    real += x.real;
    imaginary += x.imaginary;
    return *this;  // передать ссылку на текущий объект
}
```

Как это было с аргументом, передача ссылки на текущий объект позволяет использовать метод $Add$ слева от операции присваивания, например, в следующем выражении:

```cpp
x.Add(y) = z;
```

К значению объекта $x$ прибавляется значение $y$, а затем результату присваивается значение $z$. (Фактически это эквивалентно $x = z$.) Чтобы запретить подобные конструкции, достаточно добавить описатель $const$ перед типом возвращаемого значения:

```cpp
const Complex&
Complex::Add(const Complex& x)
```
Передача аргументов и результата по ссылке аналогична предаче указателя в качестве аргумента:

```c
Complex* Complex::Add(Complex* x)
{
    real += x->real;
    imaginary += x->imaginary;
    return this;
}
```

Если нет существенных причин использовать в качестве аргумента или результата именно указатель, передача по ссылке выглядит предпочтительней. Во-первых, запись операций проще, а во-вторых, обращения по ссылке легче контролировать.

**Сводка примерных правил передачи аргументов**

1. Встроенные типы лучше передавать по значению. С точки зрения эффективности разницы практически нет, поскольку встроенные типы занимают минимальную память, создание временных переменных и копирование их значений выполняется быстро.
2. Если в функции или методе используется значение аргумента, но оно не изменяется, передавайте аргумент по неизменяемой ссылке.
3. Передача изменяемой ссылки должна применяться только тогда, когда функция должна изменить переменную, ссылка на которую передается.
4. Передача по указателю используется только если функции нужен именно указатель, а не значение объекта.

**5.8. Использование описателя const**

Во многих примерах мы уже использовали ключевое слово const для обозначения того, что та или иная величина не изменяется. В данном параграфе приводятся подробные правила употребления описателя const.

Если в начале описания переменной стоит описатель const, то описываемый объект не изменяется во время выполнения программы:

```c
const double pi = 3.1415;
const Complex one(1,1);
```

Если const стоит перед определением указателя или ссылки, то это означает, что не изменяется объект на который данный указатель или ссылка указывает:

```c
// указатель на неизменяемую строку
```
const char* ptr = &string;

char x = *ptr; // обращение по указателю — допустимо
ptr++; // изменение указателя — допустимо
*ptr = '0'; // попытка изменения объекта, на
// который указатель указывает —
// ошибка

Если нужно объявить указатель, значение которого не изменяется, то такое объявление выглядит следующим образом:

char* const cptr = &string; // неизменяемый указатель
char x = *ptr; // обращение по указателю — допустимо
ptr++; // изменение указателя — ошибка
*ptr = '0'; // изменение объекта, на который
// указатель указывает — допустимо

При описании метода класса как const, выполнение метода не мо-
жет изменять значение объекта, который этот метод выполняет.

class A
{
  public:
    int GetValue (void) const;
    int AddValue (int x) const;
  private:
    int value;
}

int
A::GetValue(void) const
{
  return value; // объект не изменяется
}

int
A::AddValue(int x) const
{
  value += x; // попытка изменить атрибут объекта
              // приводит к ошибке компиляции
  return value;
}

Таким образом, использование определителя const дает возможность
программисту контролировать возможность изменения информации в про-
грамме, тем самым предупреждая возможные ошибки.
5.9. Переименование типов

При определении переменных сложных типов часто пользуются сокращением или псевдонимом типа. Любому типу, простому или сложному, можно присвоить имя и пользоваться новым именем. Например,

```c
typedef char* byte_ptr;
byte_ptr cp;
```

Переменная `cp` определена с типом `byte_ptr`, который на самом деле соответствует указателю на байт. Использование переименования типов или псевдонимов типов преследует две цели.

Первая, и наиболее очевидная — сокращение записи. В программах на языке Си++ могут быть очень сложные производные типы. Каждый раз их записывать полностью, во-первых долго, а во-вторых легко ошибиться. Поэтому гораздо удобнее и надежнее определить псевдоним для такого сложного типа и использовать его.

Вторая цель та же, что и при использовании символических констант. Определения псевдонима типа и использование его в программе облегчает изменения. Предположим, что для представления денежных величин мы решили использовать тип `long`. Определив для него псевдоним

```c
typedef long money;
```
мы не только облегчим понимание программы (тип `long` используется слишком часто, чтобы нести какую-либо смысловую нагрузку), но и существенно упростим изменения, если вдруг придется держать дробные доли копеек и перейти на тип `double`, а то и вообще создать особый класс для представления денежных величин.

Для аналогичных целей переименования типов используются в стандартной библиотеке языка Си++. С одним из них мы уже встречались: тип `size_t` используется для представления размеров строк и других объектов в памяти. В большинстве случаев он определен как `long`, однако для ранних версий Си++, которые работали в системе Windows 3.1, где оперативной памяти было мало, тип `size_t` мог быть определен как `short`.  

...
6. Классы

В данном разделе мы рассмотрим все те возможности, которыми программист обладает при определении классов.

6.1. Контроль доступа к объекту

Основной характеристикой класса с точки зрения его использования является его интерфейс, т.е. перечень методов, с помощью которых можно обратиться к объекту данного класса. Кроме интерфейса, объект обладает текущим значением или состоянием, которое он хранит в атрибутах класса. В Си++ имеются богатые возможности по контролю за тем, к каким частям класса можно обращаться извне, т.е. при использовании объектов, и какие части являются "внутренними", необходимыми лишь для реализации интерфейса.

Определение класса может быть поделено на три части — внешняя, внутренняя и защищенная. Внешняя часть предваряется ключевым словом public после которого ставиться двоеточие. Внешняя часть — это определение интерфейса. Методы и атрибуты, определенные во внешней части класса доступны как объектам данного класса, так и любым функциям и объектам других классов. Определением внешней части мы контролируем, как можно обратиться к объекту. Предположим, мы хотим определить класс для работы со строками текста. Прежде всего нам надо соединить строки, заменять заглавные буквы на строчные и знать длину строк. Соответственно, эти операции мы поместим во внешнюю часть класса:

```cpp
class String
{
public:
    // добавить строку в конец текущей строки
    void Concat(const String& str);
    // заменить заглавные буквы на строчные
    void ToLower(void);
    int GetLength(void) const; // сообщить длину строки
};
```

Внутренняя и защищенная части класса доступны только при реализации методов этого класса. Внутренняя часть предваряется ключевым словом private, защищенная — ключевым словом protected.

```cpp
class String
{
public:
    // добавить строку в конец текущей строки
    void Concat(const String& str);
```
// заменить заглавные буквы на строчные
void ToLower(void);
int GetLength(void) const; // сообщить длину строки

private:
    char* str;
    int length;
};

В большинстве случаев, атрибуты не помещаются во внешнюю часть класса, поскольку они представляют состояние объекта и возможности его использования и изменения должны быть ограничены. Представьте себе, что произойдет, если в классе String будет изменен указатель на строку без изменения длины строки, которая хранится в атрибуте length.

Объявляя атрибуты str и length как private, мы говорим, что непосредственно к ним обращаться можно только при реализации методов класса, как бы изнутри класса (private по-английски — частный, личный). Например:

```cpp
int String::GetLength(void) const
{
    return length;
}
```

Внутри определения методов класса можно обращаться не только к внутренним атрибутам текущего объекта, но и к внутренним атрибутам любых других известных данному методу объектов того же класса. Реализация метода Concat будет выглядеть следующим образом:

```cpp
void String::Concat(const String& x)
{
    length += x.length;
    char* tmp = new char[length + 1];
    ::strcpy(tmp, str);
    ::strcat(tmp, x.str);
    delete [] str;
    str = tmp;
}
```

Однако если в программе будет сделана попытка обратиться к внутреннему атрибуту или методу класса вовне определения метода, компилятор выдаст ошибку, например

```cpp
main()
{
    String s;
    if (s.length > 0) // ошибка
        ...
```
Разница между защищенными (protected) и внутренними атрибутами будет описана в разделе 6.5.3, где рассматривается создание иерархий классов.

При записи классов мы помещаем внешнюю часть первой, после нее защищенную часть и последней — внутреннюю часть. Причина состоит в том, что внешняя часть определяет интерфейс, использование объектов данного класса. Соответственно при чтении программы эта часть нужна прежде всего. Защищенная часть нужна при разработке зависимых от данного класса новых классов. И внутреннюю часть нужно изучать реже всего — при разработке самого класса.

Как известно, любое правило имеет исключения. Если никому нельзя, но кому-то очень надо, можно разрешить и доступ к внутренним переменным. Предположим, мы хотим в дополнение к интерфейсу класса String создать функцию, которая создает новую строку, являющуюся результатом слияния двух строк, но не изменяет сами аргументы. (Особенно часто подобный интерфейс необходимо создавать при определении операций — см. ниже). Для того, чтобы эта функция работала быстро, желательно чтобы она имела доступ к внутренним атрибутам класса String. Разрешить доступ можно, объявив функцию "другом" класса String используя ключевое слово friend:

```cpp
class String
{

  friend String concat(const String& s1, const String& s2);

};
```

Тогда функция concat может быть реализована следующим образом:

```cpp
String concat(const String& s1, const String& s2)
{
  String result;
  result.length = s1.length + s2.length;
  result.str = new char[result.length + 1];
  if (result.str == 0) {
    // обработка ошибки
  }
  strcpy(result.str, s1.str);
  strcat(result.str, s2.str);
  return result;
}
```
С помощью механизма friend можно разрешить обращение к внутренним элементам класса как отдельной функции, отдельному методу другого класса или всем методам другого класса:

class String
{
    // все методы класса StringParser обладают правом
    // доступа ко всем атрибутам класса String
    friend class StringParser;

    // из класса Lexer только метод CharCounter может
    // обращаться к внутренним атрибутам String
    friend int Lexer::CharCounter(const String& s, char c);
};

Конечно, не следует злоупотреблять механизмом friend. Каждое решение по использованию friend должно быть осознано. Если только одному методу какого-либо класса действительно необходим доступ, не следует объявлять весь класс как friend.

6.1.1. Доступ по чтению и по записи

Кроме контроля доступа к атрибутам класса с помощью разделения класса на внутреннюю, защищенную и внешнюю части, можно (и нужно) контролировать, с помощью каких методов можно изменить текущее значение объекта, а с помощью каких — нельзя.

В описании класса String один из методов — GetLength — описан как неизменяемый (в конце описания метода стоит слово const). Это означает, что вызов данного метода не изменяет текущее значение объекта. Остальные методы изменяют значение объекта. Контроль за правильностью использования тех или иных методов ведется на стадии компиляции. Например, если аргументом какой-либо функции объявлена ссылка на неизменяемый объект, то, соответственно эта функция может вызывать только методы, объявленные как const:

int
Lexer::CharCounter(const String& s, char c)
{
    int n = s.GetLength(); // допустимо

    s.Concat("ab"); // ошибка - Concat изменяет значение s
}

Общим правилом является объявление всех методов как неизменяемых, за исключением тех, которые действительно изменяют значение объекта. Иными словами, объявляйте как можно больше методов как const. Такое правило соответствует правилу объявления аргументов как const. Объявление константных аргументов запрещает изменение объек-
6.2. Оптимизация записи простых методов

Вернемся еще раз к реализации класса String. Реализация метода GetLength состоит из одного оператора. Язык C++ позволяет встраивать вызов функции в оператор, чтобы избежать создания дополнительных объектов во время выполнения функции и тем самым предотвращает случайные ошибки.
Еще раз подчеркнем, что это преобразование — результат оптимизации, выполняемой компилятором, и оно никак не видно программисту. Запись определения метода в теле класса лишь дает указание компилятору, что можно применять подстановку, но совершенно не обязательно компилятор будет подставлять текст метода.

То, что метод реализуется методом подстановки, можно указать и при определении метода вовне описания класса. В таком случае перед определением метода ставиться ключевое слово `inline`:

```cpp
class String
{
public:
    int GetLength(void) const;
    ...;
};

inline int String::GetLength(void) const
{
    return length;
}
```

### 6.3. Конструкторы и деструкторы

При определении класса имеется возможность задать для объекта начальное значение. Специальный метод класса, называемый конструктором, выполняется каждый раз, когда создается новый объект этого класса. Конструктор — это метод, имя которого совпадает с именем класса. Конструктор не возвращает никакого значения.

Для класса `String` имеет смысл в качестве начального значения использовать пустую строку:

```cpp
class String
{
public:
    String(); // объявление конструктора
};

// определение конструктора
String::String()
{
    str = 0;
    length = 0;
}
```

Определив такой конструктор мы гарантируем, что, даже при создании автоматической переменной, объект будет соответствующим образом инициализирован (в отличие от переменных встроенных типов).
Конструктор без аргументов называется стандартным конструктором или конструктором по умолчанию. Возможно определить несколько конструкторов с различными наборами аргументов. Возможности инициализации объектов в таком случае расширяются. Для нашего класса строк будет логичным инициализировать переменную с помощью указателя на строку.

```cpp
class String
{
public:
    String();                  // стандартный конструктор
    String(const char* p);    // дополнительный конструктор
};

// определение второго конструктора
String::String(const char* p)
{
    length = strlen(p);
    str = new char[length + 1];
    if (str == 0)
    {
        // обработка ошибок
    }
    strcpy(str, p);  // копирование строки
}
```

Теперь можно, создавая переменные типа String, инициализировать их различным образом:

```cpp
char* cp;
// выполняется стандартный конструктор
String s1;
// выполняется второй конструктор
String s2(“Начальное значение”);
// выполняется стандартный конструктор
String* sptr = new String;
// выполняется второй конструктор
String* ssptr = new String(cp);
```

Остановимся чуть подробнее на одном из видов конструктора с аргументом, в котором в качестве аргумента выступает объект того же самого класса. Такой конструктор часто называют копирующим конструктором, поскольку предполагается, что при его выполнении создается объект-копия другого объекта. Для класса String он может выглядеть следующим образом:

```cpp
class String
{
public:
    String(const String& s);
```
Как легко понять, новый объект будет копией своего аргумента. При этом новый объект независим от первоначального в том смысле, что изменение значения одного не изменяет значение другого.

// первый объект с начальным значением “Astring”
String a(“Astring”);

// новый объект - копия первого,
// т.е. со значением “Astring”
String b(a);

// изменение значения b на “AstringAstring”,
// значение объекта a не изменяется
b.Concat(a);

Столь логичное поведение объектов класса String на самом деле обусловлено наличием копирующего конструктора. Если бы его не было, компилятор создал бы его по умолчанию и такой конструктор просто бы копировал все атрибуты класса, т.е. был бы эквивалентен:

String::String(const String& s)
{
    length = s.length;
    str = s.str;
}

При вызове метода Concat для объекта b произошло бы следующее: объект b перераспределил бы память под строку str, выделив новый участок памяти и удалив предыдущий (см. определение метода выше). Однако указатель str объекта а по-прежнему указывает на первоначальный участок памяти, только что освобожденный объектом b. Соответственно, значение объекта а испорчено.

Для класса Complex, который мы рассматривали ранее, кроме стандартного конструктора можно задать конструктор, строящий комплексное число из целых чисел:

class Complex
{
    public:
        Complex();

Complex(int rl, int im = 0);
Complex(const Complex& c);
// прибавить комплексное число
Complex operator+(const Complex x) const;
private:
    int real; // вещественная часть
    int imaginary; // мнимая часть
};

// Стандартный конструктор создает число (0, 0)
//
Complex::Complex() : real(0), imaginary(0)
{}

// Создать комплексное число из действительной и
// мнимой частей. У второго аргумента есть значение
// по умолчанию - мнимая часть равна нулю по умолчанию
//
Complex::Complex(int rl, int im) :
    real(rl), imaginary(0)
{}

// Скопировать значение комплексного числа
//
Complex::Complex(const Complex& c) :
    real(c.real), imaginary(c.imaginary)
{}

Теперь при создании комплексных чисел происходит их инициализация:

Complex x1; // начальное значение - ноль
Complex x2(3); // мнимая часть по умолчанию равна 0
// создается действительное число 3
Complex x3(0, 1); // мнимая единица
Complex y(x3); // мнимая единица

Конструкторы, особенно копирующие конструкторы, довольно часто
выполняются неявно. Предположим, мы бы описали метод `Concat` чуть-
чуть по-другому:

Concat(String s);

вместо

Concat(const String& s);
т.е. использовали бы передачу аргумента по значению вместо передачи по ссылке. Конечный результат бы не изменился, однако при вызове метода

```cpp
b.Concat(a)
```

компилитор создал бы временную переменную типа String — копию объекта a — и ее бы передал в качестве аргумента. При выходе из метода String эта переменная была бы уничтожена. Прикиньте, насколько быстродействие метода было бы ниже!

Второй пример вызова конструктора — неявное преобразования типа. Допустима запись вида:

```cpp
b.Concat("LITERAL");
```

хотя сам метод определен только для аргумента — объекта типа String. Поскольку в классе String есть конструктор с аргументом — указателем на байт (а литерал — как раз константа такого типа), компилитор произведет автоматическое преобразование. Будет создана автоматическая переменная типа String с начальным значением "LITERAL", ссылка на нее передается в качестве аргумента метода String, а по завершению Concat временная переменная будет уничтожена.

Если мы хотим избежать подобного неэффективного преобразования, можно определить отдельный метод для работы с указателями:

```cpp
class String
{
public:
    void Concat(const String& s);
    void Concat(const char* s);
};

void
String::Concat(const char* s)
{
    length += strlen(s);
    char* tmp = new char[length + 1];
    if (tmp == 0) {
        // обработка ошибки
    }
    strcpy(tmp, str);
    strcat(tmp, s);
    delete [] str;
    str = tmp;
}
```

Аналогично тому, что при создании объекта выполняется конструктор, при уничтожении объекта выполняется специальный метод класса,
называемый деструктором. Обычно деструктор освобождает ресурсы, используемые данным объектом.

У класса может быть только один деструктор. Его имя — это имя класса перед которым добавлен знак тильда ‘~’. Для объектов класса String деструктор должен освободить память, используемую для хранения строки:

    class String
    {
        ~String();
    };

    String::~String()
    {
        if (str)
            delete str;
    }

Если деструктор не объявлен в определении класса, то при уничтожении объекта никаких действий не производится.

Деструктор вызывается всегда перед тем, как освобождается память, выделенная под объект. Если объект типа String был создан с помощью операции new, то при вызове

    delete sptr;

выполняется деструктор ~String(), а затем освобождается память, занимаемая этим объектом. Предположим, в некой функции объявлена автоматическая переменная типа String:

    int funct(void)
    {
        String str;
        ...
        return 0;
    }

При выходе из функции funct по оператору return переменная str будет уничтожена: выполниться деструктор и затем освободиться память, занимаемая этой переменной.

В особых случаях деструктор можно вызвать явно:

    sptr->~String();

Такие вызовы встречаются довольно редко, некоторые примеры приводятся ниже, в разделе посвященном переопределению операций new и delete.
Упражнение

1. Объясните, почему не имеет смысла создавать деструктор для класса 
Complex.
2. Как мог бы работать конструктор класса String с аргументом — целым 
числом.

6.4. Переопределение операций

Определяя классы часто желательно сделать так, чтобы выраже-
ния с объектами этих классов имели привычный вид выражений со встро-
енными типами. Для класса строк было бы естественно иметь возмож-
ность записать if (s1 < s2) или s1 = s2 + s3, где s1, s2, s3 — 
объекты типа String. Язык Си++ позволяет это делать с помощью пере-
определения операций <, = и +. Таким образом можно определить класс, 
который по своему использованию мало чем отличается от встроенных 
типов Си++, как бы расширить набор базовых типов языка.

Переопределение операций — очень мощное средство языка. Оно 
позволяет писать простые и ясные программы, оперирующие со сложны-
ми типами данных.

Разумеется, применять переопределение операций надо разумно. 
Например, для класса читателей определение операции сложения скорее 
всего только запутает программиста. Что такое сложить читателя X с чи-
тателем Y? Однако для класса, представляющего временные интервалы, 
интуитивно абсолютно ясно, что значит к промежутку времени в два часа 
прибавить интервал времени в 22 часа — мы получим сутки, 24 часа. Для 
класса String операція сложения тоже ясна — это конкатенация, слияние 
двух строк. Но определять операцию умножения для строк пожалуй не 
стоит, хотя язык это и позволяет.

Для начала определим две операции в классе String — сравнение 
на меньше и сложение:

class String
{
public:

   String operator+(const String& s) const;
   bool operator<(const String& s) const;
};

Признаком того, что переопределяется операция, служит ключевое 
слово operator, после которого стоит знак операции. В остальном опера-
ция мало чем отличается от обычного метода класса. Теперь в программе 
можно записать

String s1, s2;

s1 + s2
Объект \texttt{s1} выполнить метод \texttt{operator+} с объектом \texttt{s2} в качестве аргумента.

Результатом операции сложения является объект типа \texttt{String}. Ни какой из аргументов операции не изменяется. Описатель \texttt{const} при описании аргумента говорит о том, что \texttt{s2} не может измениться при выполнении сложения, а описатель \texttt{const} в конце определения операции говорит то же самое об объекте, выполняющем сложение.

Реализация может выглядеть следующим образом:

```c++
String
String::operator+(const String s) const
{
    String result;

    result.length = length + s.length;
    result.str = new char[result.length + 1];
    strcpy(result.str, str);
    strcat(result.str, s.str);
    return String;
}
```

При сравнении на меньше мы будем сравнивать строки в лексикографической последовательности. Проще говоря, меньше та строка, которая должна стоять раньше по алфавиту:

```c++
bool
String::operator<(const String& s) const
{
    char* cp1 = str;
    char* cp2 = s.str;

    while (true) {
        if (*cp1 < *cp2)
            return true;
        else if (*cp1 > *cp2)
            return false;
        else {
            cp1++;
            cp2++;
            if (*cp2 == 0) // конец строки
                return false;
            else if (*cp1 == 0) // конец строки
                return true;
        }
    }
}
```
Для того, чтобы объекты этого класса действительно могли участвовать в разнообразных выражениях, необходимо определить еще несколько операций. Во-первых, надо определить операцию присваивания. Операция присваивания в качестве аргумента использует объект того же класса и копирует значение этого объекта. Однако, в отличие от копирующего конструктора, у объекта уже имеется какое-то свое значение и его нужно аккуратно уничтожить.

class String
{
public:

    // объявление операции присваивания
    String& operator=(const String& s);
};

// Реализация присваивания
String&
String::operator=(const String& s)
{
    if (this == &s)
        return *this;

    if (str != 0) {
        delete [] str;
    }

    length = s.length;
    str = new char[length + 1];
    if (str == 0) {
        // обработка ошибок
    }
    strcpy(str, s.str);
    return *this;
}

Обратим внимание на несколько важных особенностей операции присваивания. Во-первых, в качестве результата операции присваивания объект возвращает ссылку на самого себя. Это позволяет использовать строки в выражениях типа:

s1 = s2 = s3;

Во-вторых, в начале операции проверяется, что аргумент не равен самому объекту. Таким образом присваивание s1 = s1 выполнится правильно и быстро.

В-третьих, перед тем как скопировать новое значение, операция присваивания освобождает память, занимаемую старым значением.
Аналогично операции присваивания можно определить операцию +=. В качестве упражнения предлагаем реализовать весь набор операций, позволяющих использовать класс String в различных выражениях:

class String
{
public:
    String();
    String(const String& s);
    String(const char*);
    String& operator=(const String& s);
    String& operator+=(const String& s);
    bool operator==(const String& s) const;
    bool operator!=(const String& s) const;
    bool operator<(const String& s) const;
    bool operator>(const String& s) const;
    bool operator<=(const String& s) const;
    bool operator>=(const String& s) const;
    String operator+(const String& s) const;
};

Во всех приведенных операциях в качестве обеих operandов выступают объекты класса String. Но что же делать, если мы хотим в качестве operand'a использовать указатель или даже один символ? Конечно, компилятор может автоматически преобразовать указатель на байт в String, но, во-первых это неэффективно, а во-вторых, что же делать с одним символом. Одним из решений является определения операции, имеющей в качестве аргумента символ:

class String
{
public:
    // добавить символ с в конец строки
    String operator+(char c) const;
};

Мы определили второй метод с именем operator+, используя то, что методы, как и функции, различаются не только по имени, но и по типу и количеству аргументов (вспомниме пункт 3.3.5).

Это позволит записывать выражения типа s + 'x'. Однако "естественная" запись 'x' + s уже не будет работать — у типа char нет операции сложения с объектом класса String. Выходом из положения является определение операции как отдельной функции, а не метода класса:

class String
{
public:
    friend String operator+(char c, const String& s);
};
String
operator+(char c, const String& s)
{
    String result;
    result.length = s.length + 1;
    result.str = new char[length + 1];
    result.str[0] = c;  // поместить символ с
                      // в начало строки
    strcpy(result.str + 1, s.str);
    return result;
}

Теперь при записи выражения 'x' + s вызовется вышеприведенная функция.

Заметим, что поскольку операция сложения не является методом класса, она не выполняется от имени какого-либо объекта и указателя this в ней не существует.

Объявление friend позволило операции + обращаться в внутренним атрибутам объектов класса String.

Последней операцией, которую мы рассмотрим в этом разделе, является операция самоувеличения ++. Определим ее для класса Interval, который хранит интервал времени, измеряемый в часах, минутах и секундах.

class Interval
{
public:
    Interval();
    Interval(short hours, short min, short sec);
    Interval& operator++();
private:
    // в качестве внутреннего представления используем
    // число секунд
    long seconds;
};

// Конструктор по умолчанию создает
// интервал длиной 0 секунд
Interval::Interval()
{
    seconds = 0;
}

// Конструктор преобразует часы, минуты, секунды
// в количество секунд
Interval::Interval(short hour, short min, short sec)
{
seconds = 24*60*hour + 60*min + sec;

// Операция ++ увеличивает интервал на одну секунду
Interval&
Interval::operator++()
{
    seconds++;
    return *this;
}

Операция увеличения может применяться как в префиксной записи (до операнда) так и в постфиксной записи (после операнда). Приведенное выше — определение для префиксной формы операции. Вначале значение объекта увеличивается и результат операции — новое значение. Для постфиксной операции в Си++ имеется специальное соглашение. Она записывается с аргументом типа int, хотя на самом деле у операции нет никакого дополнительного аргумента. Этот аргумент введен исключительно для того, чтобы различить две формы операции увеличения. (Аналогичное правило существует и для операции уменьшения — ).

class Interval
{
    Interval& operator++();
    Interval operator++(int);
};

// Увеличить значение интервала на единицу,
// однако в качестве результата использовать
// начальное значение объекта
Interval
Interval::operator++(int)
{
    Interval result(*this);
    seconds++;
    return result;
}

Как видите, в качестве результата возвращается не ссылка, а значение — новый объект класса Interval.

Правила, которые приведены в данном разделе, помогают построить удобный в использовании класс. Разумеется можно запрограммировать операцию сложения, возвращающую тип bool и операцию меньше, которая делит строку пополам. Синтаксически все будет правильно, но подобные трюки не увеличивают ценность программы, а уменьшают её. Перепределение операций было введено в язык Си++ специально для того, чтобы создаваемые классы своим поведением были как можно более "привычны", похожи на встроенные типы данных.
Упражнения

1. Разработайте класс для представления времени дня и определите все разумные для него операции.
2. К классу времени добавьте возможность прибавлять к времени дня интервал времени. Обратите внимание на правильность перехода в следующий день.

6.5. Производные классы, наследование

Важнейшим свойством объектно-ориентированного программирования является наследование. Для того, чтобы показать, что класс B наследует классу A (класс A выведен из класса B), в определении класса B после имени класса ставится двоеточие и затем перечисляются классы, из которых B наследует:

class A
{
    public:
        A();
        ~A();
        MethodA();
};
class B: public A
{
    public:
        B();

};

Название "наследование" происходит от того, что класс B обладает всеми свойствами класса A, он их унаследовал. У объекта производного класса есть все атрибуты и методы базового класса. Разумеется, новый класс может добавить свои собственные атрибуты и методы.

V b;
b.MethodA(); // вызов метода базового класса

Часто выведенный класс называют подклассом, а базовый класс – суперклассом. Из одного базового класса можно породить сколько угодно подклассов. В свою очередь, производный класс может служить в качестве базового для других классов. Изображая отношения наследования часто рисуют их в виде иерархии или дерева (см. рис.7).
Иерархия классов может быть сколь угодно глубокой. Если нужно различить, о каком именно классе идет речь, класс C называют непосредственным или прямым базовым классом класса D, а класс A – косвенным базовым классом класса D.

Предположим, что для библиотечной системы, которую мы разрабатываем, необходимо создать классы, описывающие различные книги, журналы и т.п., которые хранятся в библиотеке. Книга, журнал, газета и микрофильм обладают как общими так и различными свойствами. У книги имеется автор или авторы, название и год издания. У журнала есть название, номер и содержание – список статей. Однако у всех них имеются и общие свойства – они "единицы хранения" в библиотеке, у них есть инвентарный номер, они могут быть в читальном зале, на руках или в фонде хранения. Их можно выдать и соответственно сдать в библиотеку. Эти общие свойства удобно объединить в одном базовом классе. Введем класс Item, который описывает единицу хранения в библиотеке:

class Item
{
    public:
        Item();
        ~Item();
        // истина, если единица хранения на руках
        bool IsTaken() const;
        // истина, если этот предмет имеется в библиотеку
        bool IsAvailable() const;
        long GetInvNumber() const;  // инвентарный номер
        void Take();  // операция "взять"

Рис. 7. Пример иерархии классов
void Return(); // операция "вернуть"
private:
    // инвентарный номер - целое число
    long invNumber;
    // хранит состояние объекта - взят на руки
    bool taken;
};

Когда мы разрабатываем часть системы, которая имеет дело с процессом выдачи и возврата книг, вполне достаточно того интерфейса, который представляет базовый класс. Например

// выдать на руки
void
TakeAnItem(Item& i)
{
    ...;
    if (i.IsAvailable())
        i.Take();
}

Конкретные свойства книги будут представлены классом Book.

class Book : public Item
{
public:
    String Author(void) const; // автор
    String Title(void) const; // название
    String Publisher(void) const; // издательство
    long YearOfPublishing(void) const; // год выпуска
    String Reference(void) const; // полная ссылка
    // на книгу
private:
    String author;
    String title;
    String publisher;
    short year;
};

Для журнала класс Magazin представляет другие сведения:

class Magazin : public Item
{
public:
    String Volume(void) const; // том
    short Number(void) const; // номер
    String Title(void) const; // название
    Date DateOfIssue() const; // дата выпуска
private:
    String volume;
    short number;
    String title;
Ключевое слово public перед именем базового класса, определяет, что внешний интерфейс базового класса становиться внешним интерфейсом порожденного класса. Это наиболее часто употребляемый тип наследования. Описание защищенного и внутреннего наследования приведено в параграфе 6.4.1.

У объекта класса Book имеются методы, непосредственно определенные в классе Book и методы, определенные в классе Item:

```java
Book b;
long in = b.GetInvNumber();
String t = b.Reference();
```

Производный класс имеет доступ к методам и атрибутам базового класса, объявленным в внешней и защищенной части базового класса, однако доступ к внутренней части базового класса не разрешен. Предположим в качестве части полой ссылки на книгу решено использовать инвентарный номер. Метод Reference класса Book будет выглядеть следующим образом:

```java
String
Book::Reference(void) const
{
    String result = author + "\n"
        + title + "\n"
        + String(GetInvNumber());
    return result;
}
```

(Предполагается, что у класса String есть конструктор, который преобразует целое число в строку.) Запись:

```java
String result = author + "\n"
    + title + "\n"
    + String(invNumber);
```

не разрешена, поскольку invNumber — внутренний атрибут класса Item. Однако если бы мы поместили invNumber в защищенную часть класса:

```java
class Item
{
    ...  
    protected:
        long invNumber;
};
```

то методы классов Book и Magazin могли бы непосредственно использовать этот атрибут.
Назначение защищенной (protected) части класса состоит в том, чтобы закрыть доступ "извне" к определенным атрибутам и методам, разрешить пользоваться ими производным классам.

Если одно и то же имя атрибута или метода встречается как в базовом классе, так и в производном, то производный класс прерывает базовый.

```java
class A
{
    public:
        ...
        int foo();
        ...
    }

class B : public A
{
    public:
        int foo();
        void bar();
    }

void B::bar()
{
    x = foo();     // вызывается метод foo класса B
}
```

Однако метод базового класса не исчезает. Просто при поиске имени foo вначале просматриваются атрибуты и методы самого класса. Если бы имя не было найдено, начался бы просмотр имен в базовом классе, затем внешних имен. В данном случае имя foo существует в самом классе, поэтому оно и используется.

С помощью записи A::foo() можно явно указать, что нас интересует имя, определенное в классе A и тогда запись:

```java
x = A::foo();
```

вызывает метод базового класса.

Вообще запись класс::имя уже многократно нами использовалась. Она при поиске имени означает, что имя относится к заданному классу.

6.5.1. Виртуальные методы

В обоих классах, выведенных из класса Item имеется метод Title, выдающий в качестве результата заглавие книги или название журнала. Кроме этого метода, полезно было бы иметь метод, выдающий полное название любой единицы хранения. Реализация этого метода раз-
лична, поскольку название книги и журнала состоит из разных частей. Однаково вид метода — возвращаемое значение и аргументы — и его общий смысл один и тот же. Название — это общее свойство всех единиц хранения в библиотеке и логично поместить метод, выдающий название, в базовый класс.

class Item
{
    public:
        virtual String Name(void) const;
        ...
};

class Book : public Item
{
    public:
        virtual String Name(void) const;
        ...
};

class Magazin : public Item
{
    public:
        virtual String Name(void) const;
        ...
};

Реализация метода Name для базового класса тривиальна: поскольку название известно только производному классу мы будем возвращать пустую строку.

String
Item::Name(void) const
{
    return "";
}

Для книги название состоит из фамилии автора, названия книги, издательства и года издания:

String
Book::Name(void) const
{
    return author + title + publisher + String(year);
}

У журнала полное название состоит из названия журнала, года и номера:
Методы Name определены как виртуальные с помощью описателя virtual, стоящего перед определением метода. Виртуальные методы реализуют идею полиморфизма в языке C++. Если в программе используется указатель на базовый класс Item и с его помощью вызывается метод Name:

Item* ptr;
...
String name = ptr->Name();

tо по виду вызова метода невозможно определить, какая из трех выше-приведенных реализаций Name будет выполнена. Все зависит от того, на какой конкретный объект указывает указатель ptr.

Item* ptr;
...
if (type == "Book")
    ptr = new Book;
else if (type == "Magazin")
    ptr = new Magazin;
...
String name = ptr->Name();

В данном фрагменте программы если переменная type, обозначающая тип библиотечной единицы, была равна “Book”, то будет вызван метод Name класса Book. Если же она была равна “Magazin”, то будет вызван метод класса Magazin.

Виртуальные методы позволяют программировать действия, общие для всех производных классов, в терминах базового класса. Динамически, во время выполнения программы, будет вызван метод нужного конкретного класса.

Приведем еще один пример виртуального метода. Предположим, в графическом редакторе при нажатии определенной клавиши нужно перерисовать текущую форму на экране. Форма может быть квадратом, кругом, эллипсом и т.д. Мы введем базовый класс для всех форм Shape. Конкретные фигуры, с которыми работает редактор, будут представлены классами Square (квадрат), Circle (круг), Ellipse (эллипс), производными от класса Shape. Класс Shape определяет виртуальный метод Draw для рисования формы на экране.

```cpp
class Shape {
```
public:
    Shape();
    virtual void Draw(void);
};

    //
    // квадрат
    //
    class Square : public Shape
    {
        public:
            Square();
            virtual void Draw(void);
        private:
            double length; // длина стороны
    }

    //
    // круг
    //
    class Circle : public Shape
    {
        public:
            Circle();
            virtual void Draw(void);
        private:
            short radius;
    }...

Конкретные классы реализуют этот метод, и разумеется реализуют по-разному. Однако в функции перерисовки текущей формы, если у нас имеется указатель на базовый класс, достаточно лишь записать вызов виртуального метода и динамически будет вызван нужный алгоритм рисования конкретной формы в зависимости от того, к какому из классов Square, Circle и т.д. принадлежит объект, на который указывает указатель shape:

    Repaint(Shape* shape)
    {
        shape->Draw();
    }

Замечание. В чем разница между виртуальными методами и переопределением методов?

Что изменилось, если бы метод Name не был описан как виртуальный? В таком случае решение, какой именно метод был бы выполнен, было бы принято статически, во время компиляции программы. В примере с методом Name, поскольку мы работаем с указателем на базовый класс, был бы вызван метод Name
класса Item. При определении метода как virtual, решение, какой именно метод будет выполняться, принимается во время выполнения.

Свойство виртуальности проявляется только тогда, когда обращение к методу идет через указатель или ссылку на объект. Потенциально, указатель или ссылка может указывать как на объект базового класса, так и на объект производного класса. Если же в программе имеется сам объект, то уже во время компиляции известно, какого он типа и соответственно виртуальность не используется.

```cpp
func(Item item)
{
    item.Name();  // вызывается метод Item::Name()
}

func(Item& item)
{
    item.Name();  // вызывается метод в соответствии
    // с типом того объекта, на который
    // ссылается item
}
```

**Упражнения**

1. Напишите программу печати списка книг, имеющихся в библиотеке, используя только интерфейс базового класса Item, т.е. используя полиморфизм классов, выведенных из класса Item.

2. Преобразуйте интерфейс класса Item так, чтобы все методы стали виртуальными и производные классы не добавляли новых собственных методов.

**6.5.2. Преобразование базового и производного классов**

Рис. 8 иллюстрирует соотношение между базовым и производным классом.

![Базовый класс | Производный класс](image)

**Рис. 8. Структура объекта производного класса**

Объект базового класса является частью объекта производного класса. Если в программе используется указатель на производный класс, то его всегда можно преобразовать к указателю на базовый класс без по-
тери информации. Поэтому во многих случаях компилятор может выпол- нить такое преобразование автоматически.
Обратное же всегда верно. Преобразование из базового класса к производному не всегда можно выполнить. Поэтому говорят, что преоб- разование:

```c
Item* iPtr;
...
Book* bPtr = (Book*)iPtr;
```

небезопасно. Такое преобразование можно выполнять только тогда, когда абсолютно точно известно, что `iPtr` указывает на объект класса `Book`.

6.5.3. Внутреннее и защищенное наследование

До сих пор мы использовали только внешнее наследование. Однако в языке C++ имеется также внутреннее и защищенное наследование. Если перед именем базового класса ставиться ключевое слово `private`, то наследование называется внутренним.

```c
class B : private A
{
    ...
};
```

В случае внутреннего наследования внешняя и защищенная части базового класса становятся внутренней частью производного класса. Внутренняя часть базового класса остается недоступной для производно- го класса.
Если перед именем базового класса поставить ключевое слово `protected`, то будет использоваться защищенное наследование. При нем внешняя и защищенная части базового класса становятся защищенной частью производного класса. Внутренняя часть базового класса остается недоступной для производного класса.
Фактически, при защищенном и внутреннем наследовании произ- водный класс исключает из своего интерфейса интерфейс базового клас- са, но сам может имользоваться. Разницу между защищенным и внут- ренним наследованием почувствует только класс, выведенный из производного.

6.6. Абстрактные классы

Вернемся к примеру из предыдущего параграфа. Мы ввели базо- вый класс `Item`, который представляет общие свойства всех единиц хра- нения в библиотеке. Но существуют ли объекты класса `Item`? Т.е. суще- ствует ли в действительности "единица хранения" сама по себе. Конечно, каждая книга (класс `Book`), журнал (класс `Magazin`) и т.д. принадлежит к классу `Item`, поскольку они выведены из него, однако объект самого базо-
вого класса вряд ли имеет смысл. Базовый класс — это некоторое абстрактное понятие, описывающее общие свойства других, конкретных объектов.

Тот факт, что в данном случае объекты базового класса не могут существовать сами по себе, обусловливается еще одним обстоятельством. Ряд методов базового класса не могут быть реализованы в нем, а должны быть реализованы в порожденных классах. Тот же метод Name. Его реализация в базовом классе довольно условна, она не имеет особого смысла. Было бы гораздо логичнее вообще не реализовывать этот метод в базовом классе, а возложить ответственность за его реализацию на производные классы.

С другой стороны нам важен факт наличия метода Name во всех производных классах и то, что этот метод виртуален. Именно поэтому мы можем работать с указателями (или ссылками) на объекты базового класса не зная точно, на какой именно из производных классов этот указатель указывает. Виртуальный механизм по времени выполнения программы сам разберется и вызовет нужную реализацию метода Name.

Такая ситуация складывается довольно часто в объектно-ориентированном программировании. (Вспомните пример с различными формами в графическом редакторе. Рисование некой обобщенной формы невозможно.) В подобных случаях используется механизм абстрактных классов. Запишем базовый класс Item немного по-другому:

```cpp
class Item
{
public:

    virtual String Name() const = 0;
};
```

Теперь мы определили метод Name как чисто виртуальный. Класс, у которого есть хотя бы один чисто виртуальный метод, называется абстрактным.

То, что метод объявлен чисто виртуальным, требует, что этот метод обязан быть определен во всех классах, производных от Item. Наличие чисто виртуального метода запрещает создание объекта типа Item. В программе можно использовать указатели или ссылки на тип Item. Записи

```cpp
Item it;
Item* itptr = new Item;
```
не разрешены, компилятор сообщит об ошибке. Однако вполне можно записать:

```cpp
Book b;
Item* itptr = &b;
Item& itref = b;
```
Отметим, что определив чисто виртуальный метод в классе Book, в следующем уровне наследования его уже не обязательно переопределять (в классах, производных из Book).

Если по каким-либо причинам в производном классе чисто виртуальный метод не определен, то этот класс тоже будет абстрактным, и любые попытки создать объект данного класса будут вызывать ошибку. Таким образом, забыть определить чисто виртуальный метод просто невозможно. Абстрактный базовый класс навязывает определенный интерфейс всем производным из него классам. Собственно в этом и состоит главное назначение абстрактных классов — в определении интерфейса для всей иерархии классов. Разумеется, это не означает, что в абстрактном классе не может быть определенных методов или атрибутов.

Вообще говоря, класс можно сделать абстрактным, даже если все его методы определены. Иногда это необходимо сделать для того, чтобы быть уверенным в том, что объект данного класса никогда не будет создан. Можно определить один из методов как чисто виртуальный, но тем не менее определить его реализацию. Обычно для этих целей выбирается деструктор:

```cpp
class A
{
    public:
        virtual ~A() = 0;
    
};

A::~A()
{
    .
    ...
}
```

Класс A — абстрактный, и объект типа A создать невозможно. Однако деструктор его определен и будет вызван при уничтожении объектов производных классов (см. ниже о порядке выполнения конструкторов и деструкторов).

**Упражнение**

Перепишите иерархию классов, выведенных из класса Item, определив как можно больше методов как чисто виртуальные.

6.7. Инициализация объектов

Рассмотрим более подробно, как создаются объекты. Предположим, создается объект типа Book.

Во-первых, под объект выделяется необходимое количество памяти — либо динамически, если объект создается с помощью операции new,
либо автоматически при создании автоматической переменной, либо статически при создании статической переменной.

Класс Book — производный от класса Item. Поэтому вначале вызывается конструктор Item.

У объекта класса Book имеются атрибуты — объекты других классов, в частности String. После завершения конструктора базового класса будут созданы все атрибуты, т.е. вызваны их конструкторы. По умолчанию используются стандартные конструкторы как для базового класса, так и для атрибутов.

И только теперь подошла очередь до вызова конструктора класса Book.

В самом конце, после завершения конструктора Book, создаются структуры, необходимые для работы виртуального механизма (отсюда следует, что в конструкторе нельзя использовать виртуальный механизм).

Вызов конструкторов базового класса и конструкторов для атрибутов класса можно задать явно. Особенно это важно, если есть необходимость либо использовать нестандартные конструкторы, либо присвоить начальные значения атрибутам класса. Вызов конструкторов записывается после имени конструктора класса после двоеточия. Друг от друга вызовы разделяются запятой. Такой список называется списком инициализации или просто инициализацией:

```java
Item::Item() : taken(false), invNumber(0) {}
```

В данном случае атрибутам объекта присваиваются начальные значения. Для класса Book конструктор может выглядеть следующим образом:

```java
Book::Book() : Item(), title("<None>"),
author("<None>"), publisher("<None>"),
year(-1)
{}
```

Вначале выполняется стандартный конструктор класса Item, а затем создаются атрибуты объекта с некими начальными значениями. Теперь предположим, что у классов Item и Book есть не только стандартные конструкторы, но и конструкторы, которые задают начальные значения атрибутов. Для класса Item конструктор задает инвентарный номер единицы хранения.

```java
class Item
{
public:
    Item(long in) { invNumber = in; }
};
```
class Book
{
    public:
        Book(long in, const String& a, const String& t);
        ...
    
    
    Тогда конструктор класса Book имеет смысл записать так:

    Book::Book(long in, const String& a, const String& t) :
        Item(in), author(a), title(t)
    
    {} 

    Такого же результата можно добиться и при другой записи:

    Book::Book(long in, const String& a, const String& t) :
        Item(in)
    
    { 
        author = a;
        title = t;
    } 

    Однако предыдущая реализация лучше. Во втором случае вначале для атрибутов author и title объекта типа Book вызываются стандартные конструкторы. Затем программа выполнит операции присваивания новых значений. В первом же случае для каждого атрибута выполняться лишь один копирующий конструктор. Посмотрев на реализацию класса String, можете убедиться, насколько эффективнее первый вариант конструктора класса Book.

    Имеется еще один случай, когда без инициализации невозможно обойтись. В качестве атрибута класса можно определить ссылку. Однако при создании ссылки ее необходимо инициализировать, поэтому в конструкторе подобного класса нужно применять инициализацию.

    class A
    {
        public:
            A(const String& x);
        private:
            String& str_ref;
    };

    A::A(const String& x) : str_ref(x)
    {} 

    Создавая объект класса A, мы задаем строку, на которую он будет ссылаться. Ссылка инициализируется во время конструирования объекта. Поскольку ссылку нельзя переопределить, все время жизни объект класса A будет ссылаться на одну и ту же строку. Выбор ссылки в качестве атри-
бута класса обычно как раз и определяется тем, что ссылка инициализируется при создании объекта и никогда не изменяется. Тем самым дается гарантия использования ссылки на одну и ту же переменную. Значение переменной может изменяться, но сама ссылка — никогда.

Рассмотрим еще один пример использования ссылки в качестве атрибута класса. Предположим, что в нашей библиотечной системе книги, журналы, альбомы и т.д. могут храниться в разных хранилищах. Хранилище описывается объектом класса Repository. У каждого элемента хранения есть атрибут, указывающий на его хранилище. Возможно два варианта. Первый вариант — элемент хранения хранится всегда в одном и том же хранилище, нельзя переместить книгу из одного хранилища в другое. В данном случае использование ссылки полностью оправдано

```cpp
class Repository
{
    // ...
};

class Item
{
    public:
        Item(Repository& rep) : myRepository(rep) {};
    // ...

    private:
        Repository& myRepository;
};
```

При создании объекта необходимо задать, где он хранится. Изменить хранилище нельзя пока данный объект не уничтожен. Атрибут myRepository всегда ссылается на один и тот же объект.

Второй вариант заключается в том, что книги можно перемещать из хранилища в хранилище. Тогда в качестве атрибута класса Item лучше использовать указатель на Repository:

```cpp
class Item
{
    public:
        Item() : myRepository(0) {};
        Item(Repository* rep) : myRepository(rep) {};
        void MoveItem(Repository* newRep);
    // ...

    private:
        Repository* myRepository;
};
```

Создавая объект Item можно задать, где он хранится, а можно и не задавать. Впоследствии можно изменить хранилище, например, с помощью метода MoveItem.
При уничтожении объекта вызов деструкторов происходит в обратном порядке. Вначале вызывается деструктор самого класса, затем деструкторы атрибутов этого класса, а затем деструктор базового класса.

В создании и уничтожении объектов имеется одно существенное отличие. Создавая объект, мы всегда точно знаем, какого именно класса этот объект. При уничтожении это не всегда известно.

```c++
Item* itptr;

if (type == "book")
    itptr = new Book();
else
    itptr = new Magazine();

...
delete itptr;
```

Во время компиляции неизвестно, какое именно будет значение переменной type и, соответственно, объект какого класса удаляется операцией delete. Поэтому компилятор может вставить вызов только деструктора базового класса.

Для того, чтобы все необходимые деструкторы были вызваны, нужно воспользоваться виртуальным механизмом — объявить деструктор как в базовом классе, так и производном, как virtual.

```c++
class Item
{
    virtual ~Item();
};

class Book
{
    public:
        virtual ~Book();
};
```

Возникает вопрос — почему бы всегда не объявлять деструкторы виртуальными? Единственная плата за это — небольшое увеличение памяти для реализации виртуального механизма. Таким образом, не объявлять деструктор виртуальным имеет смысл только, если во всей иерархии классов нет виртуальных функций и удаление объекта никогда не происходит через указатель на базовый класс.

**Упражнение**

Разработайте структуру классов с базовым классом `Number`, который является интерфейсом для всех возможных числам — целым, вещественным и комплексным. Пример использования `union` для подобной цели был приведен в параграфе 5.3. Теперь вместо `union` используйте наследование и виртуальные методы.
6.8. Множественное наследование

В языке Си++ имеется возможность в качестве базовых задать несколько классов. В таком случае производный класс наследует методы и атрибуты всех его родителей. Пример иерархии классов в случае множественного наследования приведен на рисунке 9.

![Diagram](image)

Рис. 9. Иерархия классов при множественном наследовании

В данном случае класс C наследует двум классам А и B.
Множественное наследование — мощное средство языка. Приведем некоторые примеры использования множественного наследования.
Предположим, имеющуюся библиотечную систему решено установить в университете и интегрировать ее с другой системой учета преподавателей и студентов. В библиотечной системе имеются классы, описывающие читателей и работников библиотеки. В системе учета кадров имеются классы, хранящие информацию о преподавателях и студентах. Используя множественное наследование можно создать классы студентов-читателей, преподавателей-читателей и студентов, подрабатывающих библиотекарями.

В графическом редакторе для некоторых фигур может быть предусмотрен текст пояснений. При этом все алгоритмы форматирования и печати пояснений работают с классом Annotation. Тогда те фигуры, которые могут содержать пояснение, будут представлены классами, производными от двух базовых классов:

```java
class Annotation
```
public:
    String GetText(void);
private:
    String annotation;
};

class Shape
{
public:
    virtual void Draw(void);
};

class AnnotatedSquare : public Shape, public Annotation
{
public:
    virtual void Draw();
};

У объекта класса AnnotatedSquare имеется метод GetText, унаследованный от класса Annotation, и он определяет виртуальный метод Draw, унаследованный от класса Shape.

При применении множественного наследования возникает ряд проблем. Первая из них — возможный конфликт имен методов или атрибутов нескольких базовых классов.

class A
{
public:
    void fun();
    int a;
};
class B
{
public:
    int fun();
    int a;
};
class C : public A, public B
{
};

При записи:

C* cp = new C;
cp->fun();

невозможно определить к какому из двух методов fun происходит обращение. Ситуация называется неоднозначной и компилятор выдаст ошиб-
ку. Заметим, что ошибка выдастся не при определении класса с, в котором заложена возможность возникновения неоднозначной ситуации, а лишь при попытке вызова метода fun.

Неоднозначность можно разрешить, явно указав, к которому из базовых классов происходит обращение:

cp->A::fun();

Вторая проблема заключается в возможности многократного включения базового класса. В упомянутом выше примере интеграции библиотечной системы и системы кадров вполне вероятна ситуация, при которой классы для работников библиотеки и для студентов были введены из одного и того же базового класса Person — человек:

class Person
{
  public:
    String name();
};

class Student : public Person
{
  ...
};

class Librarian : public Person
{
  ...
};

Если теперь создать класс для представления студентов, подрабатывающих в библиотеке

class StudentLibrarian : public Student, public Librarian
{
};

то объект данного класса будет содержать объект базового класса Person дважды (см. рис. 10).

Рис. 10. Структура объекта StudentLibrarian.
Кроме того, что подобная ситуация — нерациональное использование памяти, никаких других неудобств в данном случае она не вызывает. Возможную неоднозначность можно разрешить с помощью явного указания класса:

```cpp
StudentLibrarian* sp;

// ошибка - неоднозначное обращение, непонятно к // какому именно экземпляру типа Person обращаться
sp->Person::name();

// правильное обращение
sp->Student::Person::name();
```

Тем не менее иногда необходимо, чтобы объект базового класса содержался ровно один раз в производном. Для этих целей применяется виртуальное наследование, речь о котором пойдет в следующем подразделе.

### 6.8.1. Виртуальное наследование

Базовый класс можно объявить виртуальным базовым классом используя запись:

```cpp
class Student : virtual Person
{
};

class Librarian : virtual Person
{
};
```

Гарантированно, что объект виртуального базового класса будет содержаться ровно один раз в объекте выведенного класса (см. рис. 11). Платой за виртуальность базового класса являются дополнительные надкладные расходы при обращениях к атрибутам и методам этого класса.

![Diagram](image)

**Рис. 11. Структура объекта StudentLibrarian при виртуальном множественном наследовании.**

120
6.9. Операции new и delete

Выделение памяти под объекты некого класса производится либо при создании переменных типа этого класса, либо с помощью операции new. Было бы странно, если бы язык C++, позволяющий переопределить почти все операции для определенного класса, не разрешал бы переопределение и операций выделения и освобождения памяти. Разумеется, он позволяет.

Прежде всего рассмотрим модификацию операции new, которая уже определена в самом языке. (Точнее, она определена в стандартной библиотеке языка C++. Это определение не выделяет память, а лишь создает объект на заранее выделенном участке памяти. Форма определения следующая:

```c++
new (адрес) имя_класса (аргументы_конструктора)
```

Перед именем класса в круглых скобках указывается адрес, по которому должен располагаться создаваемый объект. Фактически, такая операция new не выделяет память, а лишь создает объект по указанному адресу, выполняя его конструктор. Соответственно, можно не выполнять операцию delete для этого объекта, а лишь вызвать его деструктор перед тем, как поместить новый объект на то же место памяти.

```c++
char memory_chunk[4096];
Book* bp = new (memory_chunk) Book;
... bp->~Book();
Magazin* mp = new (memory_chunk) Magazin;
... mp->~Magazin();
```

В этом примере никакой потери памяти не происходит. Память выделена один раз, объявлением массива memory_chunk. Операции new создают объекты в начале этого массива (разумеется, мы предполагаем, что 4096 байтов достаточно для объектов). Когда объект становится не нужным, явно вызывается его деструктор и на том же месте создается новый объект.

Любой класс может использовать два вида операций new и delete — глобальную и определенную для класса. Если класс и ни один из его базовых классов, как прямых, так и косвенных, не определяет операцию new, то используется глобальная операция new. Глобальная операция new всегда используется для выделения памяти под встроенные типы и под массивы (независимо от того, объекты какого класса составляют массив).

Если класс определяет операцию new, то для всех экземпляров этого класса и любых классов, производных от него, глобальная операция будет переопределена и будет использоваться new данного класса. Если
нужно использовать именно глобальную операцию, можно перед new по-
ставить два двоеточия ::new.

Вид стандартной операции new следующий:

class A
{
    void* operator new(size_t size);
};

Аргумент size задает размер необходимой памяти в байтах.
size_t — это тип целого, подходящий для задания размера объектов в
данной реализации языка, определенный через typedef. Чаще всего это
tип long. Аргумент операции new не задается явно при ее вызове. Компи-
лятор сам его подставляет, зная размер создаваемого объекта.

Реализация операции new, которая совпадает со стандартной, вы-
глядит просто:

void*
A::operator new(size_t size)
{
    return ::new char[size];
}

В классе может быть определено несколько операций new с раз-
личными дополнительными аргументами. При вызове new эти аргументы
указываются сразу после ключевого слова new в скобках до имени типа.
Компилятор добавляет еще один аргумент от себя — размер памяти, и
затем вызывает соответствующую операцию. Описанная выше модифи-
кация new, помещающая объект по определенному адресу, имеет вид:

void* operator new(void* addr, size_t size);

Предположим, мы хотим определить такую операцию, которая бу-
дет инициализировать каждый байт выделенной памяти каким-либо чис-
лом.

class A
{
    void* operator new(char init, size_t size);
};

void*
A::operator new(char init, size_t size)
{
    char* result = ::new char[size];
    if (result) {
        for (size_t i = 0; i < size; i++)
            result[i] = init;
    }
}
return result;
}

Вызов такой операции имеет вид:

A* aptr = new (32) A;

Память под объект класса A будет инициализирована числом 32 (что, кстати, является кодом пробела).

Отметим, что если класс определяет хотя бы одну форму операции new, глобальная операция становится переопределена. Например, если бы в классе A была определена только операция new с инициализацией, то вызов

A* ptr = new A;

привел бы к ошибке компиляции, поскольку подобной формы new в классе не определено. Поэтому, если вы определяете new, определяйте все её формы, включая стандартную (быть может просто вызывая глобальную операцию).

В отличие от операции new, для которой можно определить разные модификации в зависимости от числа и типов аргументов, операция delete существует только в единственном варианте:

void operator delete (void* addr);

В качестве аргумента ей передается адрес, который в своё время возвратила операция new для данного объекта. Соответственно, для класса можно определить только одну операцию delete. Напомним, что операция delete ответственна только за освобождение занимаемой памяти. Деструктор объекта вызывается отдельно. Операция delete, которая будет вызывать стандартную, выглядит следующим образом:

void
A::operator delete(void* addr)
{
    ::delete [] (char*)addr;
}

Упражнение

Переопределением операций new и delete можно воспользоваться для отладки правильности распределения памяти в программе, т.е. соответствия выделения и освобождения памяти. Создайте такие операции для класса Item.

6.10. Преобразования типов

Определяя класс, программист задает методы и операции, которые применимы к объектам этого класса. Например, при определении класса комплексных чисел была определена операція сложения двух комплексных чисел. При определении класса строк, мы определили операцию конкатенации двух строк. Что же происходит, если в выражении мы попыта-
емся использовать ту же операцию сложения с типами, для которых она явно не задана? Компилятор пытается преобразовать величины, участвующие в выражении к типам, для которых операция задана. Это преобразование, называемое преобразованием типов, выполняется в два этапа.

Первый этап — попытка воспользоваться стандартными преобразованиями типов, определенными в языке Си++ для встроенных типов. Если подобными преобразованиями добиться результата не удаётся, тогда компилятор пытается воспользоваться преобразованиями, определенными пользователем. "Помочь" компилятору правильно преобразовать типы величин можно, явно задав преобразования типов.

6.10.1. Явные преобразования типов

Если перед выражением указать имя в круглых скобках, то значение выражения будет преобразовано к указанному типу:

double x = (double)1;
void* addr;
Complex* cptr = (Complex*) addr;

Такие преобразования типов использовались в языке Си. Их основным недостатком является полное отсутствие контроля. Явные преобразования типов традиционно использовались очень часто в программах на языке Си и, к сожалению, продолжались использоваться в Си++, что приводит к ошибкам и к запутанным программам. В большинстве своём их можно избежать в Си++. Тем не менее иногда явные преобразования типов необходимы.

Для того, чтобы преобразовывать типы, но хотя бы с минимальным контролем, можно записать

static_cast<тип > (выражение)

Операция static_cast позволяет преобразовывать типы, основываясь лишь на сведениях о типах выражений, известных во время компиляции. Иными словами, static_cast никак не проверяет типы выражений во время выполнения. С одной стороны, это позволяет большую ответственность на программиста, а с другой ускоряет выполнение программ. С её помощью можно как выполнять стандартные преобразования, так и нестандартные. Операция static_cast позволяет преобразовывать типы, связанные отношением наследования, указатель к указателю, один числовой тип к другому, перечислимое значение к целому. В частности, с помощью операции static_cast можно преобразовывать не только указатель на производный класс к базовому классу, но и указатель на базовый класс к производному, что в общем случае небезопасно.

Однако попытка преобразовать целое число к указателю приведет к ошибке компиляции. Если все же необходимо преобразовать совершенно не связанные между собой типы, можно вместо static_cast записать reinterpret_cast:
void* addr;
int* intptr = static_cast<int*>(addr);
Complex* cPtr = reinterpret_cast<Complex*>(2000);

Если нужно ограниченное преобразование типа, которое только преобразует неизменяемый тип к изменяемому (убирает описатель const), можно воспользоваться операцией const_cast:

const char* addr1;
char* addr2 = const_cast<char*>(addr);

Использование static_cast, const_cast и reinterpret_cast вместо явного преобразования в форме (тип) имеет существенные преимущества. Во-первых, можно всегда применить "минимальное" преобразование, т.е. преобразование, которое меньше всего изменяет тип. Во-вторых, все преобразования можно легко обнаружить в программе. В-третьих, национального программиста намного легче распознать, что важно при модификации программы. Сразу можно будет отличить неконтролируемое преобразование от преобразования неизменяемого указателя к изменяемому.

6.10.2. Стандартные преобразования типов

К стандартным преобразованиям относятся преобразования целых типов и преобразования указателей. Они выполняются компилятором автоматически. Часть правил преобразования мы уже рассмотрели ранее. Преобразования целых величин, при которых не теряется точность, сводятся к следующим:

- Величины типа char, unsigned char, short или unsigned short преобразуются к типу int, если точность типа int достаточна, в противном случае они преобразуются к типу unsigned int.
- Величины типа wchar_t и константы перечисленных типов преобразуются к первому из типов int, unsigned int, long и unsigned long, точность которого достаточна для представления данной величины.
- Битовые поля преобразуются к типу int, если точность типа int достаточна или к unsigned int, если точность unsigned int достаточна. В противном случае преобразование не производится.
- Логические значения преобразуются к типу int, false становится 0 и true становится 1.

Эти четыре типа преобразований мы будем называть безопасными преобразованиями.

Язык Си (от которого Си++ унаследовал большинство стандартных преобразований) часто критикоали за излишне сложные правила преобразования типов и за их автоматическое применение без ведома пользователя. Основной рекомендацией является рекомендация избегать неявных преобразований типов, в особенности тех, при которых возможна потеря точности или знака.
Коротко правила стандартных преобразований при выполнении арифметических операций сводятся к следующим:
Вначале, если в выражении один из операндов имеет тип long double, то другой преобразуется также к long double;
в противном случае, если один из операндов имеет тип double, то другой преобразуется также к double;
в противном случае, если один из операндов имеет тип float, то другой преобразуется также к float;
в противном случае, производится безопасное преобразование.
Затем, если в выражении один из операндов имеет тип unsigned long, то другой преобразуется также к unsigned long;
в противном случае, если один из операндов имеет тип long, а другой — unsigned int и тип long может представить все значения unsigned int, то unsigned int преобразуется к long, иначе оба операнда преобразуются к unsigned long;
в противном случае, если один из операндов имеет тип long, то другой преобразуется также к long;
в противном случае, если один из операндов имеет тип unsigned, то другой преобразуется также к unsigned;
в противном случае оба операнда типа int.

(1L + 2.3) — результат типа double
(8u + 4) — результат типа unsigned long

Все приведенные преобразования типов производятся компилятором автоматически и обычно при компиляции даже не выдается никакого предупреждения, поскольку при них не теряются значащие цифры или точность результата.
Как мы уже отмечали ранее, при выполнении операции присваивания со стандартными типами, может происходить потеря точности. Большинство компиляторов при попытке такого присваивания выдают предупреждение или даже ошибку. Например, при попытке присваивания

```c
long x;
char c;
c = x;
```

если значение x равно 20, то и c будет равно 20. Но если x равно 500, значение c будет равно -12 (при условии выполнения на персональном компьютере), поскольку старшие биты, не помещающиеся в char, будут обрезаны. Именно поэтому большинство компиляторов выдают ошибку и не будут транслировать подобные конструкции.

6.10.3. Преобразования указателей и ссылок

При работе с указателями и ссылками компилятор выполняет автоматически только два вида преобразований.
1. Если имеется указатель или ссылка на производный тип, а требуется соответственно указатель или ссылка на базовый тип.
2. Если имеется указатель или ссылка на изменяемый объект, а требуются указатель или ссылка на неизменяемый объект того же типа.

```cpp
size_t strlen(const char* s); // прототип функции

class A { }

class B : public A { }

char* cp;
strlen(cp);  // автоматическое преобразование из
            // char* в const char*

B* bObj = new B;
A* aObj = bObj;  // преобразование из указателя на
                 // производный класс к указателю на
                 // базовый класс
```

Если требуются какие-то другие преобразования, их необходимо указывать явно, но в этом случае вся ответственность за правильность преобразования лежит на программисте.

6.10.4. Преобразования типов, определенных в программе

В языке Си++ можно определить гораздо больше типов, чем в Си. Казалось бы и правила преобразования новых типов станут намного сложнее. К счастью, этого не произошло. Все дело в том, что при определении классов программист может контролировать какие преобразования допустимы и как они выполняются при преобразовании в данный тип или из данного типа в другой.

Прежде всего, выполнение тех или иных операций с аргументами разных типов можно регулировать с помощью методов и функций с разными аргументами. Для того, чтобы определить операцию сложения комплексного числа с целым, нужно определить две функции в классе Complex:

```cpp
class Complex {

    friend Complex operator+(const Complex& x, int y);
    friend Complex operator+(int y, const Complex& x);
};
```

При наличии таких функций никаких преобразований типа не происходит в следующем фрагменте программы:

```cpp
int x;
Complex y;
...
Complex z = x + y;
```
Тем не менее в других ситуациях преобразования типа производятся. Прежде всего компилятор старается обойтись стандартными преобразованиями типа. Если их не хватает, то выполняются преобразования либо с помощью конструкторов, либо с помощью определенных программистом операций преобразования.

Задав конструктор класса, имеющий в качестве аргумента величину другого типа, программист тем самым определяет правило преобразования:

```cpp
class Complex
{
public:
    // неявное правило преобразования
    // из целого типа в тип Complex
    Complex(int x);
};
```

Операции преобразования имеют вид:

```cpp
operator имя_типа();
```

Например, преобразование из комплексного числа в целое можно записать:

```cpp
class Complex
{
public:
    // операция преобразования из типа
    // Complex в целый тип
    operator int();
};
```

При записи:

```cpp
Complex cmpl;
int x = cmpl;
```

будет вызвана функция `operator int()`.

**6.11. Динамическое определение типов**

Средства динамического определения типов в языке Си++ преследуют две основные цели: возможность определения в ходе выполнения программы к какому именно типу принадлежит данная величина и возможность проведения безопасных преобразований типов.

Как мы помним, указатель на величину какого-либо класса можно всегда преобразовать к указателю на базовый класс. Однако обратное неверно — имей указатель на базовый класс, преобразование к указателю
на производный класс небезопасно, т.е. компилятор не может гарантировать, что оно правильно. Программист, исходя из логики программы, может это сделать, но тогда он должен предусмотреть дополнительные средства в своих классах, которые хранят информацию о том, какого именно типа данная величина. Приведем пример.

Предположим для более эффективной работы библиотеки мы решили воспользоваться готовой программой распределения нагрузки между библиотекарями. Эта программа распределяет сообщения по обработчикам, причем сообщения должны быть выведены из класса Message, а обработчики — из класса Receiver. Чтобы приспособить программу для нашей библиотечной системы, применим множественное наследование и создадим класс BookMessage и LibrarianReceiver. В классе LibrarianReceiver метод Process получает указатель на сообщение и в нем можно реализовать обработку книги.

```cpp
class Message {
    ...
};

class BookMessage : public Message, public Book {
}

class Receiver {
    virtual void Process(Message* m) = 0;
};

class LibrarianReceiver : public Receiver,
    public Librarian {
    virtual void Process(Message* m);    // обработка книги
};
```

Система распределения сообщения выбирает очередной приемник и пересылает ему сообщение:

```cpp
void ScheduleMessage(Message* m) {
    Receiver* pr;
    // выбрать очередной приемник
    pr->Process(m);
}
```

В функции обработки книги (методе Process класса LibrarianReceiver) мы получим указатель на класс Message. Можно ли
проверить, что полученное сообщение — действительно книга, а не что-нибудь еще? Без изменения класса Message нельзя.

Подобные ситуации при написании программ хотя и не очень часты, тем не менее вполне реальны. Для облегчения их программирования в язык Си++ были добавлены средства динамического определения типов (английская аббревиатура RTTI — Run-Time Type Information). Эти средства применимы к указателям и ссылкам и включают три основных элемента:

- операция dynamic_cast;
- класс type_info;
- операция typeid.

**Операция dynamic_cast**

Выражение `dynamic_cast<имя_типа>(выражение)` преобразует значение выражения к указанному типу. Имя_типа должно быть указателем или ссылкой на имеющийся и определенный в данном контексте тип или бестиповым указателем (void*). При выполнении преобразования производится проверка, каков в действительности тип выражения. Если преобразование невозможно, в случае попытки преобразовать к указателю, результатом является нулевой указатель, а в случае ссылки, операция возбуждает исключительную ситуацию bad_cast.

```cpp
class A { ... };
class B : public A { ... };
class C { ... };

main()
{
    B* pb = new B; // указатель на B
    foo(pb);       // вызов функцииfoo
}

void foo(A* pa)
{
    B* pb = dynamic_cast<B*>(pa); // допустимое
                                    // преобразование
    C* pc = dynamic_cast<C*>(pa); // результат null
}

Используя dynamic_cast, метод Process класса LibrarianReceiver можно записать:

```cpp
void LibrarianReceiver::Process(Message* m)
{
    Book* bookPtr = dynamic_cast<Book*>(m);
    if (bookPtr == 0) {
        // это не книга - обработка не нужна
        return;
    }
```
Класс type_info

Класс type_info служит для представления информации о типе. Он определен в файле заголовков typeinfo.h и имеет вид:

class type_info
{
public:
    virtual ~type_info();
    int operator==(const type_info& t) const;
    int operator!=(const type_info& t) const;
    int before(const type_info& t) const;
    const char* name() const;
    const char* raw_name() const;
};

Объекты данного класса можно сравнивать на равенство и неравенство. Методы name и raw_name выдают имена типов. Причем метод raw_name выдает внутреннее представление имени в том виде, в каком оно хранится в программе. Метод name выдает имя в "нормальном", удобочитаемом виде. Получить информацию о типе (объект класса type_info) можно с помощью операции typeid.

Операция typeid

Операция typeid служит для получения ссылки на объект класса type_info, который описывает тип данного выражения. Её формат

typeid(имя_типа)
или
typeid(выражение)

Важно отметить, что операция typeid выдает как можно более точную информацию о типе. Это значит, например, что из указателя на базовый класс извлекается информация о действительном типе объекта. В приведенном ниже примере, t1 содержит информацию о производном классе. Заметьте, что мы применили операцию * к указателю для того, чтобы получить информацию о классе, а не об указателе на класс.

    class B { ... };
    class D : public B { ... };

    void foo()
    {
D* pd = new D;
B* pb = pd;
...
const type_info& t = typeid(pb);
// t содержит информацию о типе указателя
const type_info& tl = typeid(*pb);
// tl содержит информацию о типе D
...

Использование RTTI должно быть оправдано. Во многих случаях механизма виртуальных функций вполне достаточно для выполнения разных действий для разных классов.

Упражнение

Если в примере, приведенном в данном параграфе, можно было бы изменить класс Message, как можно было бы обойтись без средств динамического определения типов? (Подсказка – какую виртуальную функцию надо определить в классе Message и как её реализовать в классе BookMessage)

6.12. Указатели на элементы класса

Предположим в классе Task имеется несколько однотипных методов.

class Task
{
    void init(char* cp);
    void run(char* cp);
    void suspend(char* cp);
    void resume(char* cp);
    void stop(char* cp);
};

Мы должны разработать некий промежуточный слой программного обеспечения, который вызывает один из этих методов, для простоты изложения – функцию с именем call_method. В качестве аргументов этой функции передается указатель на объект типа Task, аргумент метода и некоторый указатель, какой именно метод вызвать. Для подобных целей в языке C++ имеется особый тип указателей – указатель на элемент класса.

Указатель на элемент класса – это особый тип указателя. В отличие от обычного указателя, который указывает на один определенный объект, указатель на элемент класса содержит информацию как о классе, так и о его элементе (атрибуте или методе). Указатель на элемент класса – это значение, которое идентифицирует атрибут или метод класса. Мож
но его интерпретировать как позицию атрибута или метода в объекте класса (разумеется, это лишь способ более наглядно представить указатель на элемент). Объявление указателя на элемент будет выглядеть так:

```c
typedef void (Task::*PtrMethod)(char* cp) ();
```

Тип `PtrMethod` — это указатель на метод класса `Task`, который берёт в качестве аргумента значение типа `char*` и не возвращает никакого значения (возвращаемое значение типа `void`). Переменная этого типа может принимать одно из значений `Task::init`, `Task::run` и т.д.

Теперь мы можем записать функцию `call_method`:

```c
void call_method(Task* tPtr, PtrMethod ptrmem, char* arg)
{
    tPtr->*ptrmem(arg);
}
```

```c
main()
{
    PtrMethod p = &Task::init;
    Task* task = new Task;
    call_method(task, p, "server");
}
```

Можно определить не только указатель на метод, но и указатель на атрибут класса:

```c
class Interface
{
    long x;
    long y;
};
typedef long Interface::*lPtr;
```

Нужна в подобных функциях возникает при обработке ошибок, при сопряжении различных библиотек классов и т.д. Ряд функций стандартной библиотеки шаблонов реализованы с использованием указателей на атрибуты классов.
7. Структура сложных программ

До сих пор мы рассматривали небольшие примеры программ или даже фрагменты программ. Но современный язык программирования должен поддерживать производство больших программных продуктов, состоящих из многих десятков, сотен или даже тысяч классов. Программа на языке Си++ может быть написана коллективом программистов на протяжении нескольких лет. В этой главе мы рассмотрим свойства языка, позволяющие создавать большие программы.

7.1. Компоновка нескольких файлов в одну программу

Мы уже говорили, что программа — это прежде всего текст на языке Си++. С помощью компилятора текст преобразуется в выполняемый файл — форму, позволяющую компьютеру выполнять программу.

Если мы рассмотрим этот процесс чуть более подробно, то выяснится, что обработка исходных файлов происходит в три этапа. Сначала файл обрабатывается препроцессором, который выполняет операторы #include, #define и еще несколько других. После этого этапа программа все еще представлена в виде текстового файла, хотя и измененного по сравнению с первоначальным. Затем, на втором этапе, компилятор создает так называемый объектный файл. Программа уже переведена в машинные инструкции, однако она еще не полностью готова к выполнению. В объектном файле имеются ссылки на различные системные функции и на стандартные функции языка Си++. Например, выполнение операции new заключается в вызове определенной системной функции. Даже если в программе явно не упомянута ни одна функция, необходим по крайней мере один вызов системной функции — завершение программы и освобождение всех принадлежащих ей ресурсов.

На третьем этапе компиляции к объектному файлу подсоединяются все функции, на которые он ссылается. Функции тоже должны быть оттранслированы, т.е. переведены на машинный язык в форму объектных файлов. Этот процесс называется компоновкой и как раз его результат и есть выполняемый файл.

Системные функции и стандартные функции языка Си++ заранее откомпилированы и хранятся в виде библиотек. Библиотека — это некий архив объектных модулей, с которым удобно компоновать программу.

Основная цель многоэтапной компиляции программ — возможность компоновать программу из многих файлов. Каждый файл представляет собой некоторый законченный фрагмент программы, который может ссылаться на функции, переменные или классы, определенные в других файлах. Компоновка объединяет фрагменты в одну программу, которая "самодостаточна", т.е. содержит всё необходимое для выполнения.
Современные системы программирования во многом автоматизировали процесс компоновки. Например в системе Microsoft Visual C++ достаточно лишь включить нужные файлы в один проект для того, чтобы они компоновались вместе. В системе Unix обычно требуется больше ручного труда для компоновки многих файлов в один выполняемый. За подробными инструкциями мы отсылаем читателя к руководству по той конкретной системе программирования, которую он использует. В последующих параграфах мы рассмотрим средства создания больших программ самого языка Си++, не зависящие от той или иной системы программирования.

7.2. Включаемые файлы

В языке Си++ существует строгое правило о том, что в программе прежде, чем использовать имя или идентификатор, его необходимо определить. Рассмотрим для начала функции. Для того, чтобы имя функции стало известным программе, его нужно либо объявить, либо определить.

Объявление функции состоит лишь из её прототипа, т.е. имени, типа результата и списка аргументов. Объявление функции задает её формат, не говоря ничего, как она выполняется. Примеры объявлений функций:

```cpp
double sqrt(double x); // функция sqrt
long fact(long x); // функция fact

// функция PrintBookAnnotation
void PrintBookAnnotation(const Book& book);
```

Определение функции – это определение того, как функция выполняется. Оно включает в себя тело функции, программу её выполнения.

```cpp
// функция вычисления факториала
// целого положительного числа
long fact(long x)
{
    if (x == 1)
        return 1;
    else
        return x * fact(x - 1);
}
```

Определение функции выполняет роль объявления её имени, т.е. если в начале файла определена функция fact, в последующем тексте функций и классов ей можно пользоваться. Однако если в программе функция fact используется в нескольких файлах, такое построение программы уже не подходит. В программе должно быть только одно определение функции.

Удобно было бы поместить определение функции в отдельный файл, а в других файлах в начале помещать лишь объявление, прототип функции.
// начало файла main.cpp

long fact(long); // прототип функции

main()
{

    int x10 = fact(10); // вызов функции

}

// конец файла main.cpp

// начало файла fact.cpp

// определение функции
// вычисления факториала целого положительного числа

long fact(long x)
{
    if (x == 1)
        return 1;
    else
        return x * fact(x - 1);
}

// конец файла fact.cpp

Компоновщик объединит оба файла в одну программу.
Аналогичная ситуация существует и для классов. Любой класс в языке C++ состоит из двух частей: объявления и определения. В объявлении класса говорится, каков интерфейс класса, какие методы и атрибуты составляют объекты этого класса. Объявление класса состоит из ключевого слова class, за которым следует имя класса, затем список наследования и затем в фигурных скобках методы и атрибуты класса. Заканчивается объявление класса точкой с запятой.

class Book : public Item
{
    public:
        Book();
}
Определение класса – это определение всех его методов.

// определение метода Title
String Book::String()
{
    return title;
}

Определение класса должно быть ровно одно и, если класс используетя во многих файлах, его удобно поместить в отдельный файл. В остальных файлах для того, чтобы использовать класс Book, например определить переменную класса Book, в начале файла необходимо поместить объявление класса.

Таким образом, в начале каждого файла будут сосредоточены прототипы всех используемых функций и объявлении всех используемых классов. Программа будет работать, однако писать такую программу не очень удобно.

В начале каждого файла нам придется повторять довольно большие одинаковые куски текста. Мало того, что это утомительно, но и легко допустить ошибку. Если по каким-то причинам потребуется изменить объявление класса, то потребуется изменять все файлы, в которых он используется (хотя возможно, что изменение никак не затрагивает интерфейс класса, то как этот класс используется. Например, добавление нового внутреннего атрибута непосредственно не влияет на использование внешних методов и атрибутов этого класса.)

Язык С++ предлагает удобное решение. Можно поместить объявление классов и функций в отдельный файл и включать этот файл в начало других файлов с помощью оператора #include:

```
#include "Book.h"
...
Book b;
```

Фактически оператор #include подставляет содержимое файла Book.h в текущий файл перед тем, как начать его компиляцию. Эта подстановка осуществляется во время первого прохода компилятора по программе – препроцессора. Файл Book.h называется файлом заголовков.

В такой же файл заголовков можно поместить прототипы используемых функций и включать его в другие файлы, там где функции используются.
Таким образом, текст программы на языке Си++ помещается в файлы двух типов — файлы заголовков и файлы программ. В большинстве случаев имеет смысл каждый класс помещать в отдельный файл, вернее два файла — файл заголовков для объявления класса и файл программы для определения класса. Имя файла обычно состоит из имени класса. Для файла заголовков к нему добавляется окончание ".h" (иногда, особенно в системе Unix, ".hh" или ".h"). Имя файла программы — опять-таки имя класса с окончанием ".cpp" (иногда ".cc" или ".c").

Объединять несколько классов в один файл стоит лишь в том случае, если они очень тесно связаны и один без другого не используются.

Включение файлов может быть вложенным, т.е. файл заголовков может сам использовать оператор #include. Файл Book.h выглядит следующим образом:

```c++
#ifndef __BOOK_H__
#define __BOOK_H__

//
//

// включить файл с объявлением используемого
// здесь базового класса
#include "Item.h"
#include "String.h"    // объявление класса String

// объявление класса Book
class Book : public Item {
    public:
    ...
    private:
    String title;
    ...
};
#endif
```

Обратите внимание на первые две и последнюю строки этого файла. Оператор #ifndef начинает блок так называемой условной компиляции, который заканчивается оператором #endif. Блок условной компиляции — это кусок текста, который будет компилироваться только если выполнено определенное условие. В данном случае условие заключается в том, что символ __BOOK_H__ неопределен. Если этот символ определен, текст между #ifndef и #endif не будет включен в программу. Первым оператором в блоке условной компиляции стоит оператор #define, который определяет символ __BOOK_H__ как пустую строку.

Давайте посмотрим, что случится, если в какой-либо .cpp файл будет дважды включен файл Book.h:

```c++
#include "Book.h"
```


```
#include "Book.h"

Перед началом компиляции текст файла Book.h будет подставлен вместо оператора #include:

```
#ifdef __BOOK_H__
define __BOOK_H__
.
.
class Book
{
  .
};
#endif
```

```
#ifdef __BOOK_H__
define __BOOK_H__
.
.
class Book
{
  .
};
#endif
```

В самом начале символ __BOOK_H__ неопределён и блок условной компиляции обрабатывается. В нем определяется символ __BOOK_H__. Теперь условие для второго блока условной компиляции уже не выполняется и он будет пропущен. Тем самым объявление класса Book будет вставлено в файл только один раз. Разумеется написание два раза подряд оператор #include одного и того же заголовка легко поправить. Однако структура заголовков может быть очень сложна. Чтобы избежать необходимости отслеживать все вложенные заголовки и искать, почему какой-либо файл оказался вставленным дважды, можно применить изложенный выше прием и существенно упростить себе жизнь.

Еще одно замечание по составлению заголовков. Включайте в заголовок как можно меньше других заголовков. Например, в заголовок Book.h необходимо включить заголовки Item.h и String.h поскольку класс Book их использует. Однако если используется лишь имя класса без упоминания его содержимого, можно обойтись лишь объявлением этого имени:

```
#include "Item.h"
#include "String.h"

class Annotation;  // Annotation - имя некого класса

class Book : public Item
{
  public:
  ```
Объявление класса Item требуется знать целиком, для того, чтобы обработать объявление класса Book, т.е. компилятору надо знать все методы и атрибуты Item, чтобы включить их в класс Book. Объявление класса String также требуется знать целиком, по крайней мере для того, чтобы правильно вычислить размер экземпляра класса Book. Что же касается класса Annotation, то ни размер его объектов, ни его методы не важны для определения содержимого объекта класса Book. Единственно, что надо знать, это то, что Annotation — это имя некоего класса, который будет определен в другом месте.

Общее правило таково, что если объявление класса использует указатель или ссылку на другой класс и не использует никаких методов или атрибутов этого класса, достаточно объявление имени класса. Разумеется, полное объявление класса Annotation понадобится в определении метода CreateAnnotation.

Компилятор поставляется с набором файлов заголовков, которые описывают все стандартные функции и классы. При включении стандартных файлов обычно используют немного другой синтаксис:

```c++
#include <string.h>
```

В языке Си++ имеется несколько операторов, начинающихся со знака #: #include, #define, #undef, #ifdef, #else, #if, #pragma. Все они обрабатываются так называемым препроцессором и их детальное описание имеется в разделе 12.7.

### 7.3. Файлы и переменные

Мы уже довольно подробно говорили о автоматических переменных и упоминали статические переменные. Все они определены внутри какой-либо функции или метода класса.

Назначение автоматических переменных — хранение каких-либо данных во время выполнения функции или метода. По завершению этой функции автоматические переменные уничтожаются и данные теряются. С этой точки зрения автоматические переменные — это временные переменные.

Иногда временное хранилище данных нужно еще на более короткое время, чем выполнение всей функции. Во-первых, поскольку в Си++ необязательно, чтобы все используемые переменные были определены в самом начале функции или метода, переменную можно определить непосредственно перед тем, как она используется. Во-вторых, переменную можно определить внутри блока — группы операторов, заключенных в фигурные скобки. При выходе из блока, такая переменная уничтожается, не дожидаясь конца функции. Третьей возможностью временного использо-
вания переменной является определение переменной в заголовке цикла for только для итераций этого цикла

```cpp
funct(int N, Book[] & bookArray)
{
    int x; // автоматическая переменная x

    for (int i = 0; i < N; i++) {
        // переменная i определена только на время
        // выполнения цикла for
        String s;
        // новая автоматическая переменная создается
        // при каждой итерации цикла заново
        s.Append(bookArray[i].Title());
        s.Append(bookArray[i].Author());
        cout << s;
    }

    cout << s; // ошибка, переменная s не существует
}
```

Если переменную, определенную внутри функции или блока, описать как статическую, она не будет уничтожаться при выходе из этого блока и будет хранить свое значение между вызовами функции. Однако при выходе из соответствующего блока эта переменная станет недоступна, иными словами невидима для программы. В следующем примере переменная allAuthors накапливает список авторов книг, переданных в качестве аргументов функции funct за все ее вызовы:

```cpp
funct(int n, Book[] & bookArray)
{
    for (int i = 0; i < n; i++) {
        static String allAuthors;
        allAuthors.Append(bookArray[i].Author());
        cout << allAuthors;
        // авторы всех ранее обработанных книг,
        // в том числе в предыдущих вызовах функции
    }

    cout << allAuthors; // ошибка, переменная недоступна
}
```

Иногда необходимо, чтобы к одной переменной можно было обращаться из разных функций. Предположим, в нашей программе используемый генератор случайных чисел. Мы хотим инициализировать его один раз, в начале выполнения программы, а затем обращаться к нему из разных частей программы. Рассмотрим несколько возможных реализаций.

Во-первых, определим класс RandomGenerator с двумя методами: Init для инициализации генератора и GetNumber для получения следующего числа.

```cpp
//
// Файл RandomGenerator.h
//
```
class RandomGenerator
{
public:
    RandomGenerator();
    ~RandomGenerator();
    void Init(unsigned long start);
    unsigned long GetNumber();
private:
    unsigned long previousNumber;
};

//
// файл RandomGenerator.cpp
//
#include "RandomGenerator.h"
#include <time.h>

void
RandomGenerator::Init(unsigned long x)
{
    previousNumber = x;
}

unsigned long
RandomGenerator::GetNumber(void)
{
    unsigned long ltime;

    // получить текущее время в секундах, прошедших с
    // полуночи 1 января 1970 года
    time(&ltime);
    ltime <= 16;
    ltime >>= 16;        // взять младшие 16 битов
    previousNumber = previousNumber * ltime;
    return previousNumber;
}

Первый вариант состоит в создании объекта класса RandomGenerator в функции main и передачи ссылки на него во все функции и методы, где он потребуется.

        // файл main.cpp
        // файл main.cpp
#include "RandomGenerator.h"

main()
{
    RandomGenerator rgen;

142
rangen.Init(1000);

fun1(rangen);
  
Class1 b(rangen);
  
fun2(rangen);
}

void
fun1(RandomGenerator& r)
{
  unsigned long x = r.GetNumber();
  
}


///////////
// файл class.cpp
///////////
#include "RandomGenerator.h"

Class1::Class1(RandomGenerator& r)
{
  
}

void
fun2(RandomGenerator& r)
{
  unsigned long x = r.GetNumber();
  
}

Поскольку завершение функции main завершает выполнение программы, то все необходимые условия выполнены: генератор случайных чисел создается в самом начале программы, все объекты и функции обра-щаются к одному и тому же генератору, и генератор уничтожается по завершении программы. Такой стиль программирования допустимо ис-пользовать только, если передавать ссылку на используемый экземпляр объекта нужно не часто. В противном случае этот способ крайне неудо-бен. Передавать ссылку на один и тот же объект утомительно, загромож-дает интерфейс классов и крайне не гибко.

Язык C++ предоставляет возможность определения глобальной переменной. Если переменная определена вне функции, она создается в самом начале выполнения программы (еще до начала выполнения main). Эта переменная доступна во всех функциях того файла, где она определена. Аналогично прототипу функции, имя глобальной переменной можно объявить в других файлах и тем самым дать возможность к ней обра-щаться и в других файлах:
# файл main.cpp

```cpp
#include "RandomGenerator.h"

// определение глобальной переменной
RandomGenerator rgen;

main()
{
    rgen.Init(1000);
}

void
fun1(void)
{
    unsigned long x = rgen.GetNumber();
    . . .
}
```

# файл class.cpp

```cpp
#include "RandomGenerator.h"

// объявление глобальной переменной,
// внешней по отношению к данному файлу
extern RandomGenerator rgen;

Class1::Class1()
{
    . . .
}

void
fun2()
{
    unsigned long x = rgen.GetNumber();
    . . .
}
```

Объявление внешней переменной можно поместить в файл-заголовок. Тогда не нужно будет повторять объявление переменной с описателем extern в каждом файле, который её использует.

Модификацией определения глобальной переменной является добавление описателя static. Для глобальной переменной описатель static означает то, что эта переменная доступна только в одном файле - в том, в котором она определена. (Правда в данном примере такая модификация
недопустима — нам-то как раз нужно, чтобы к глобальной переменной rgen можно было обращаться из разных файлов.)

Определение глобальной переменной намного более удобно, чем передача ссылки на генератор случайных чисел в каждый метод и функцию в качестве аргумента. Достаточно описать внешнюю глобальную переменную (включив соответствующий файл заголовков с помощью оператора #include) и генератор становится доступен. Не нужно менять интерфейс, если вдруг понадобиться обращение к генератору. Не нужно передавать один и тот же объект в разные функции.

Тем не менее использование глобальных переменных легко может привести к ошибкам. В нашем случае с генератором при его использовании нужно твердо помнить, что глобальная переменная уже определена. Простая забывчивость может привести к тому, что будет определен второй объект — генератор случайных чисел, например с именем randomGen. Поскольку с точки зрения правил языка никаких ошибок не допущено, компиляция пройдет нормально. Однако результат работы программы будет не тот, который мы ожидаем. (Исходя из определения класса, определите, почему).

При составлении программ самым лучшим решением будет то, которое не позволит ошибиться, т.е. неправильная программа не будет компилироваться. Не всегда это возможно, но в данном случае, как и во многих других, соответствующие средства имеются в языке Си++.

Изменим описание класса RandomGenerator:

```cpp
class RandomGenerator {

public:
    static void Init(unsigned long start);
    static unsigned long GetNumber(void);

private:
    static unsigned long previousNumber;
};
```

Определения методов Init и GetNumber не изменяются. Единственно, что надо будет добавить в файл RandomGenerator.cpp, это определение переменной previousNumber:

```cpp
//
// файл RandomGenerator.cpp
//
#include "RandomGenerator.h"
#include <time.h>

unsigned long RandomGenerator::previousNumber;
```

Методы и атрибуты класса, описанные static, существуют независимо от существования объектов этого класса. Вызов статического метода имеет вид

имя_класса::имя_метода
Например

```
RandomGenerator::Init(x)
```

У статического метода не существует указателя this, таким образом он имеет доступ либо к статическим атрибутам класса, либо к атрибутам передаваемых ему в качестве аргументов объектов. Например:

```c++
class A
{
    public:
        static void Method(const A& a);
    private:
        static int a1;
        int a2;
};

void A::Method(const A& a)
{
    int x = a1;    // обращение к статическому атрибуту
    int y = a2;    // ошибка, a2 неопределен
    int z = a.a2;  // правильно
}
```

Статический атрибут класса во многом аналогичен глобальной переменной, но доступ к нему контролируется классом. Ровно один статический атрибут класса создается в начале программы для всех объектов данного класса (и даже если ни одного объекта не было создано). Можно считать, что статический атрибут — это атрибут класса, а не объекта.

Теперь программа, использующая генератор случайных чисел будет выглядеть так:

```c++
#include "RandomGenerator.h"

main()
{
    RandomGenerator::Init(1000);
}

void fun1(void)
{
    unsigned long x = RandomGenerator::GetNumber();
    ...
}
```
Такое построение программы и удобно и надежно. В отличие от глобальной переменной, второй раз определить генератор невозможно – мы и первый-то раз его определили лишь фактом включения класса RandomGenerator в программу, а два раза определить один и тот же класс нам не даст компилятор.

Разумеется, существуют и другие способы обеспечения того, чтобы существовал ровно один объект какого-либо класса.

Кратко суммируем результаты этого параграфа:

1. Автоматические переменные заново создаются каждый раз, когда управление передается в соответствующую функцию или блок.
2. Статические и глобальные переменные создаются один раз, в самом начале выполнения программы.
3. К глобальным переменным можно обращаться из всей программы.
4. К статическим переменным, определенных вовне функций, можно обращаться из всех функций данного файла.
5. Хотя использование глобальных переменных иногда может быть удобно, делать это необходимо с большой осторожностью из-за того, что легко допустить ошибку (к ним нет контроля доступа, можно переопределить глобальную переменную).
6. Статические атрибуты класса существуют в единственном экземпляре и создаются в самом начале программы. Статические атрибуты применяют тогда, когда нужно иметь одну переменную, к которой могут обращаться все объекты данного класса. Доступ к статическим атрибутам контролируется теми же правилами, что и к обычным атрибутам.
7. Статические методы класса используются для функций, по своей сути являющихся глобальными, но логически относящихся к какому-либо классу.
7.4. Область видимости имен

Между именами переменных, функций, типов и т.п. могут возникать конфликты при использовании одного и того же имени в разных частях программы. Для того, чтобы эти конфликты разрешать, в языке имеется такое понятие как область видимости имени.

Минимальной областью имен является блок. Все имена, определяемые в блоке, должны быть различны. При попытке объявить две переменные с одинаковым именем, произойдет ошибка. Имена, определенные в блоке, видимы (доступны) в этом блоке и во всех вложенных блоках. Аргументы функции, описанные в ее заголовке, рассматриваются как определенные в теле этой функции.

Имена, объявленные в классе, видимы внутри этого класса, т.е. во всех методах этого класса. Для того, чтобы обратиться к атрибуту класса, нужно использовать операции „." и „->" или „::".

Имена, объявленные вовне блоков, имеют область видимости весь текст файла, следующий за объявлением.

Объявление может перекрывать такое же имя, объявленное во внешней области.

```cpp
int x = 7;

class A
{
    public:
        void foo(int y);
        int x;
};

main()
{
    A a;
    a.foo(x); // используем глобальная переменная x
    // и передается значение 7
    cout << x;
}

void A::foo(int y)
{
    x = y + 1; // x - атрибут объекта типа A

    double x = 3.14; // новая переменная x перекрывает атрибут класса x

    cout << x;
}

cout << x;
```


В результате выполнения приведенной программы будет напечатано 3.14, 8 и 7.

Несмотря на то, что имя во внутренней области видимости перекрывает имя, объявленное во внешней области, перекрываемая переменная продолжает существовать. В некоторых случаях к ней можно обратиться, явно указав область видимости с помощью квалификатора "::". Обозначение ::имя говорит о том, что имя относится к глобальной области видимости. (Попробуйте поставить :: перед переменной x в вышеприведенном примере.) Два двоеточия часто употребляют перед именами стандартных функций библиотеки языка Си++, чтобы, во-первых, подчеркнуть, что это глобальные имена, и, во-вторых, избежать возможных конфликтов с именами методов класса, в котором они употребляются.

Если перед квалификатором поставить имя класса, то имя будет искаться в указанном классе. Например, обозначение А::x показало бы, что речь идет об атрибуте класса А. Аналогично можно обращаться к атрибутам структур и объединений. Поскольку определения классов и структур могут быть вложенными, у имени может быть несколько квалификаторов:

class Klass
{
public:
    enum Color { RED, WHITE, BLUE }
    struct Structure
    {
        static int Flag;
        int x;
    };
    int y;
    void Method();
};

Следующие обращения допустимы извне класса:

Klass::BLUE
Klass::Structure::Flag

При реализации метода Method обращения к тем же именам могут быть проще:

void
Klass::Method()
{
    Color x = BLUE;
    y = Structure::flag;
}
При попытке обратиться извне класса к атрибуту набора BLUE компилятор выдаст ошибку, поскольку имя BLUE определено только в контексте класса.

Отметим одну особенность типа enum. Его атрибуты как бы экспортируются во внешнюю область имен. Несмотря на наличие фигурных скобок, к атрибутам перечисленного типа Color не обязательно (хотя и не воспрещается) обращаться Color::BLUE.

7.5. Оператор определения контекста namespace

Несмотря на столь развитую систему областей видимости имен, иногда и ее недостаточно. В больших программах возможность возникновения конфликтов на глобальном уровне достаточно реальна. Имена всех классов верхнего уровня должны быть различны. Хорошо, если вся программа разрабатывается одним человеком. А если большой группой? Особенно при использовании готовых библиотек классов (см. ниже). Чтобы избежать конфликтов, обычно договариваются о системе имен классов. Договариваться о стиле имен всегда полезно, однако проблема остается, особенно в случае разработки классов, которыми будут пользоваться другие.

Одно из сравнительно поздних добавлений к языку Си++ — контексты, определяемые с помощью оператора namespace. Они позволяют заключить группу объявлений классов, переменных и функций в отдельный контекст со своим именем. Предположим мы разработали набор классов для вычисления различных математических функций. Все эти классы, константы и функции можно заключить в контекст math для того, чтобы разрабатывая программу, использующую наши классы, другой программист не должен был бы выбирать имена, обязательно отличные от того, что мы использовали.

```cpp
namespace math {
  double const pi = 3.1415;
  double sqrt(double x);
  class Complex {
    public:
      ...
  }
};
```

Теперь к константе pi обращаться надо math::pi.

Контекст может содержать как объявлния, так и определения переменных, функций и классов. Если функция или метод определяется вовне контекста, ее имя должно быть полностью квалифицированно

```cpp
double math::sqrt(double x) {
```

150
Контексты могут быть вложенными, соответственно имя должно быть квалифицировано несколько раз:

```cpp
namespace first
{
  int i;

  namespace second
  {
    int i;
    int whati() { return first::i; }
    // возвращается значение первого i

    int anotherwhat { return i; }
    // возвращается значение второго i

  }
}
first::second::whati();  // вызов функции
```

Если в каком-либо участке программы интенсивно используется определенный контекст и все имена уникальны по отношению к нему, можно сократить полные имена, объявив контекст текущим с помощью оператора using.

```cpp
double x = pi;  // ошибка, надо использовать math::pi
using namespace math;  // использовать контекст math
double y = pi;  // теперь правильно
```
8. Обработка ошибок

8.1. Ошибки неизбежны

Полное и окончательное исчезновение ошибок — мечта любого программиста. К сожалению, этой мечте не суждено осуществиться, во всяком случае в обозримом будущем. Поэтому существенной частью любой реальной программы является обработка ошибок.

Прежде чем перейти к рассмотрению средств языка Си++ для обработки ошибки, остановимся немного на том, какие собственны ошибки мы будем обрабатывать.

Пропустим ошибки компиляции. Пока они все не исправлены, программа не готова и ее нельзя запустить. Здесь мы будем рассматривать только ошибки, происходящие во время выполнения программы.

Первый вид ошибок, который всегда приходит в голову — это ошибки программирования. Сюда относятся и ошибки в алгоритме, в логике программы, и чисто программистские ошибки. Ряд возможных ошибок мы указывали ранее (например, при работе с указателями), гораздо больше вы узнаете на собственном горьком опыте.

Теоретически возможно написать программу без таких ошибок. Во многом язык Си++ помогает предотвратить ошибки во время выполнения программы, осуществляя строгий контроль на стадии компиляции. Вообще, чем больший контроль на стадии компиляции, тем меньше ошибок остается при выполнении программы.

Перечислим некоторые средства языка, помогающие избежать ошибок:
1. Контроль типов. Использование недопустимых операций, смещение несовместимых типов будет обнаружен компилятором.
2. Обязательное объявление имен до их использования. Невозможно вызвать функцию с неверным числом аргументов. При изменении определения переменной или функции легко обнаружить все места, где она используется.
3. Ограничение видимости имен, контексты имен. Уменьшается возможность конфликтов имен, неправильного переопределения имен.

Самым важным средством уменьшения вероятности ошибок является собственно объектно-ориентированный подход к программированию, который поддерживает язык Си++. Наряду с преимуществами объектного программирования, о которых мы говорили ранее, построение программы из классов позволяет отлаживать классы по отдельности и строить программы из надежных составных "кирпичиков", используя одни и те же классы многократно.

Несмотря на все эти положительные качества языка, остается "простор" для написания ошибочных программ. По мере рассмотрения свойств языка, мы стараемся давать рекомендации, какие возможности использовать, что бы уменьшить вероятность совершить ошибку.
Понимание того, что идеальных программ не существует, помогает разрабатывать более надежные программы. Самым важным является контроль данных — правило проверять в программе все, что может содержать ошибку. Если в программе предполагается какое-то условие, не лишним бывает его проверить, хотя бы в начальной версии программы, до того, как мы на опыте убедимся, что это условие действительно выполняется. Хорошим правилом является проверка указателей, передаваемых в качестве аргументов, на равенство нулю, проверка того, что индексы не выходят за границы массива и т.п.

Ну и решающими качествами, уменьшающими количество ошибок, являются внимательность, аккуратность и опыт.

Вторым видом ошибок являются "предусмотренные", запланированные ошибки. Если разрабатывается программа диалога с пользователем, это программа обязана разумно реагировать и обрабатывать неправильные нажатия клавиш. Программа чтения текста должна учитывать возможные синтаксические ошибки. Программы передачи данных по телефонной линии должна обрабатывать помехи и возможные сбои при передаче. Такие ошибки — это вообще говоря не ошибки с точки зрения программы, а плановые ситуации, которые она обрабатывает.

Третий вид ошибок тоже в какой-то мере предусмотрен. Это исключительные ситуации, которые могут иметь место даже если в программе нет ошибок. Например, нехватка памяти для создания нового объекта. Или сбой диска при извлечении информации из базы данных.

Именно обработка этих двух последних видов ошибок и рассматривается в последующих параграфах. Граница между ними довольно условна. Например, для большинства программ сбой диска — исключительная ситуация, но для операционной системы сбой диска должен быть предусмотрен и должен обрабатываться. Скорее два типа можно разграничивать по тому, какая реакция программы должна быть предусмотрена. Если после плановых ошибок программа должна продолжать работать, то после исключительных ситуаций надо лишь сохранить уже вычисленные данные и завершить программу.

8.2. Возвращаемое значение как признак ошибки

Простейший способ сообщения об ошибках — возвращаемое значение функции или метода. Функция сохранения объекта в базе данных может возвращать логическое значение: true в случае успешного сохранения, false при ошибке.

    class Database
    {
    public:
        bool SaveObject(const Object* obj);
    };

Соответственно, вызов метода должен выглядеть так

    if (database.SaveObject(my_obj) == false) {
        ...
// обработка ошибки

Обработка ошибки, разумеется, зависит от конкретной программы. Типичная ситуация, когда при многократно вложенных вызовах функций, обработка происходит на несколько уровней выше, чем уровень, где ошибка произошла. В таком случае результат, сигнализирующий об ошибке, придется передавать во всех вложенных вызовах.

```cpp
main()
{
    if (fun1() == false)
        // обработка ошибки
}

bool fun1()
{
    if (fun2() == false)
        return false;

    return true;
}

bool fun2()
{
    if (database.SaveObject(obj) == false)
        return false;

    return true;
}
```

Если функция или метод должны возвращать какую-то величину в качестве результата, особое, недопустимое значение этой величины используется в качестве признака ошибки. Если метод возвращает указатель, выдача нулевого указателя применяется в качестве признака ошибки. Если функция вычисляет положительное число, возврат -1 можно использовать в качестве признака ошибки.

Иногда невозможно вернуть признак ошибки в качестве возвращаемого значения. Примером является конструктор объекта, который не может вернуть значение. Как же сообщить о том, что во время инициализации объекта что-то было не так?

Распространенным решением является дополнительный атрибут объекта – флаг, отражающий правильность состояния объекта. Предположим конструктор класса Database должен соединиться с сервером базы данных.

```cpp
class Database
```
public:
    Database(const char* serverName);
    
    bool Ok(void) const { return okFlag; };
private:
    bool okFlag;
};

Database::Database(const char* serverName)
{
    if (connect(serverName) == true)
        okFlag = true;
    else
        okFlag = false;
}

void main()
{
    Database database("db-server");
    if (!database.Ok()) {
        cerr << "Ошибка соединения с базой данных" << endl;
        return;
    }
}

Лучше вместо метода Ok, возвращающего значение флага okFlag, переопределить операцию ! (отрицание).

class Database
{
public:
    bool operator!() const { return !okFlag; };
};

Тогда проверка успешности соединения с базой данных будет выглядеть так:

if (!database) {
    cerr << "Ошибка соединения с базой данных" << endl;
}

Следует отметить, что лучше избегать такого построения классов, при котором возможны ошибки в конструкторе. Из конструктора можно выделить соединение с сервером базы данных в отдельный метод Open:

class Database
{
public:
и тогда отпадает необходимость в операции ! или методе Ok().

Использование возвращаемого значения в качестве признака ошибки — достаточно универсальный метод. Он используется прежде всего для обработки запланированных ошибочных ситуаций. Его недостатками являются, во-первых, необходимость передавать признак ошибки через вложенные вызовы функций. Во-вторых, возникают неудобства, если метод или функция уже возвращают значение и приходится либо модифицировать интерфейс, либо придумывать специальное "ошибочное" значение. В-третьих, логика программы оказывается запутанной из-за сплошных условных операторов if с проверкой на ошибочное значение.

8.3. Исключительные ситуации

Язык Си++ предлагает специальный механизм для сообщения об ошибках — механизм исключительных ситуаций. Название несомненно наводит на мысль, что предназначен этот механизм прежде всего для оповещения об ошибках-исключительных ситуациях, о которых мы говорили в первом параграфе этой главы. Однако механизм исключительных ситуаций может применяться и для обработки плановых ошибок.

Исключительная ситуация возбуждается при выполнении оператора throw. В качестве аргумента throw задается любое значение. Это может быть значение одного из встроенных типов (число, строка символов и т.п.) или объект любого определенного в программе класса.

При возникновении исключительной ситуации выполнение текущей функции или метода немедленно прекращается, созданные к этому моменту автоматические переменные уничтожаются и управление передается в точку, откуда была вызвана текущая функция или метод. В точке возврата возбуждается та же самая исключительная ситуация, прекращается выполнение текущей функции или метода, уничтожаются автоматические переменные и управление передается в точку, откуда была вызвана эта функция или метод. Происходит своего рода откат всех вызовов до тех пор, пока не завершиться функция main и тем самым не завершиться все программа.

Предположим из main была вызвана функция foo, которая вызвала метод Open, который в свою очередь возбудил исключительную ситуацию:

```cpp
class Database
{
    public:
        void Open(const char* serverName);
};

void
Database::Open(const char* serverName)
```
if (connect(serverName) == false)
    throw 2;
}

foo()
{
    Database database;
    database.Open("db-server");
    String y;
    ...
}

main()
{
    String x;
    foo();
}

В этом случае управление вернется в функцию foo, будет вызван деструктор объекта database, управление вернется в main, где будет вызван деструктор объекта x и программа завершится. Таким образом, исключительные ситуации предоставляют возможность аварийного завершения программы с некоторыми возможностями очистки переменных.

В таком виде оператор throw используется для действительно исключительных ситуаций, которые практически никак не обрабатываются. Гораздо чаще даже исключительные ситуации нужно обрабатывать.

В программе можно объявить блок, в котором мы будем отслеживать исключительные ситуации с помощью операторов try и catch:

try {
    ...
}

catch (тип_исключительной_операции) {
    ...
}

Если внутри блока try произошла исключительная ситуация, то она прежде всего передается в оператор catch. Тип исключительной ситуации — это тип аргумента throw. Если тип возбужденной исключительной ситуации совместим с типом аргумента catch, выполняется блок catch. Тип аргумента catch совместим, если он либо совпадает с типом ситуации, либо является одним из ее базовых типов. Если тип несовместим, то происходит откат вызовов, описанный выше до тех пор, пока либо не завершится программа, либо не встретится блок catch с подходящим типом аргумента.

В блоке catch происходит обработка исключительной ситуации.

foo()
{
    Database database;
    int attempCount = 0;
}
again:

try {
    database.Open("dbserver");
} catch (int& ex) {
    cerr << "Ошибка соединения номер " << x << endl;
    if (++attemptCount < 5)
        goto again;
    throw;
}

String y;
...

Ссылка на аргумент throw передается в блок catch. Этот блок гасит исключительную ситуацию. Во время обработки можно либо повторно возбудить ту же самую исключительную ситуацию с помощью оператора throw без аргументов, либо возбудить любую другую, либо не возбуждать никакой. В последнем случае исключительная ситуация считается погашенной и выполнение программы продолжается после блока catch.

С одним блоком try может быть связано несколько блоков catch с разными аргументами. В этом случае исключительная ситуация последовательно "применяется" к каждому catch до тех пор, пока аргумент не окажется совместимым. Этот блок и выполняется. Специальный вид catch

catch (...)
совместим с любым типом исключительной ситуации. Но, правда, в него нельзя передать аргумент.

8.4. Обработка исключительных ситуаций

Механизм исключительных ситуаций предоставляет гибкие возможности для обработки ошибок, однако им надо уметь правильно воспользоваться. В этом параграфе мы рассмотрим некоторые приемы обработки исключительных ситуаций.

Прежде всего, имеет смысл определить специальный класс для исключительных ситуаций. Простейшим вариантом является класс, который может хранить код ошибки:

```c++
class Exception {
public:
    enum ErrorCode {
        NO_MEMORY,
        DATABASE_ERROR,
        INTERNAL_ERROR,
        ILLEGAL_VALUE
```
Возбуждение исключительной ситуации будет выглядеть:

```cpp
if (connect(serverName) == false)
    throw Exception(Exception::DATABASE_ERROR, serverName);
```

а проверка на исключительную ситуацию:

```cpp
try {
    
    } catch (Exception & e) {
        cerr << "Произошла ошибка " << e.GetErrorKind()
            << " Дополнительная информация: "
            << e.GetErrorMessage();
    }
```

Преимущества класса перед простым целым числом состоят во-первых, что передается дополнительная информация и, во-вторых, в операторах catch можно реагировать только на ошибки определенного вида. Если была возбуждена исключительная ситуация другого типа, например

```cpp
throw AnotherException;
```

то блок catch будет пропущен – он ожидает только исключительные ситуации типа Exception. Это особенно существенно при сопряжении нескольких различных программ и библиотек – каждый набор классов отвечает только за собственные ошибки.

В данном случае код ошибки записывается в объекте типа Exception. Если в одном блоке catch мы ожидает несколько разных исключительных ситуаций и для них необходима разная обработка, то в программе придется анализировать код ошибки с помощью операторов if или switch.

```cpp
try {
    
    } catch (Exception & e) {
        cerr << "Произошла ошибка " << e.GetErrorKind()
            << " Дополнительная информация: "
            << e.GetErrorMessage();
    }
```
Другим методом разделения различных исключительных ситуаций является создание иерархии классов — по отдельному классу на каждый тип исключительной ситуации.

В приведенной на рис.12 структуре классов все исключительные ситуации делятся на ситуации, связанные с работой базы данных (класс DatabaseException) и внутренние ошибки программы (класс InternalException). В свою очередь ошибки базы данных бывают двух типов: ошибки соединения (представленные классом ConnectDbException) и ошибки чтения (ReadDbException). Внутренние исключительные ситуации разделены на недостаток памяти (NoMemoryException) и недопустимые значения (IllegalValException).

Рис. 12. Пример иерархии классов для представления исключительных ситуаций

Теперь блок catch может быть записан в следующем виде:

```c++
try {
    } catch (ConnectDbException& e) {
        // обработка ошибки соединения с базой данных
    } catch (ReadDbException& e) {
        // обработка ошибок чтения из базы данных
    } catch (DatabaseException& e) {
        // обработка других ошибок базы данных
    } catch (NoMemoryException& e) {
```
// обработка нехватки памяти
} catch (...) {
    // обработка всех остальных исключительных ситуаций
}

Напомним, что когда при проверке исключительной ситуации на соответствие аргументу оператора catch, проверка идет последовательно до тех пор, пока не найдется подходящий. Поэтому, например, нельзя ставить catch для класса DatabaseException впереди catch для класса ConnectDbException — исключительная ситуация типа ConnectDbException совместима с классом DatabaseException (это ее базовый класс) и она будет обработана в catch для DatabaseException и не дойдет до блока с ConnectDbException.

Построение системы классов для разных исключительных ситуаций более трудоемко на стадии описания ошибок, приходится создавать новый класс для каждого типа исключительной ситуации. Однако с точки зрения обработки она гибче и позволяет писать легче читаемые программы.

Чтобы облегчить обработку ошибок и сделать запись о них более наглядной, описания методов и функций можно дополнить информацией, какого типа исключительные ситуации они могут возбуждать:

class Database
{
    public:
    Open(const char* serverName) throw ConnectDbException;
};

Такое описание говорит, что метод Open класса Database может возбудить исключительную ситуацию типа ConnectDbException. Соответственно, при использовании этого метода желательно предусмотреть обработку возможной исключительной ситуации.

В заключение приведем несколько рекомендаций по использованию исключительных ситуаций.

1. При возбуждении исключительной ситуации остаток функции или метода не выполняется. Более того, при обработке ее не всегда известно, где конкретно возникла исключительная ситуация. Поэтому прежде чем выполнить оператор throw, освободите ресурсы, зарезервированные в текущей функции. Например, если какой-либо объект был создан с помощью new, необходимо явно вызвать delete для него.

2. Используйте исключительных ситуаций в деструкторах. Деструктор может быть вызван в результате уже случившейся исключительной ситуации при отката вызовов функций и методов. Повторная исключительная ситуация не обрабатывается и завершает выполнение программы.

3. Если исключительная ситуация возникла в конструкторе объекта, считается, что объект не полностью сформирован и деструктор для него вызван не будет.
9. Ввод-вывод

Ввод-вывод — это операции обмена данными между программой и внешними устройствами. Типичным внешним устройством является терминал. На терминале можно напечатать информацию. Можно ввести информацию с терминала, напечатав ее на клавиатуре. Другим типичным устройством является жесткий или гибкий диск, на котором расположены файлы. Программа может создавать файлы, в которых храниться информация. Другая (или эта же) программа может читать информацию из файла.

В языке Си, предшественнике Си++, операции ввода-вывода были реализованы не в виде специальных операторов языка, а с помощью набора стандартных функций. Аналогичный подход применен и в Си++. В нем нет особых операторов для ввода или вывода данных. Вместо этого имеется набор классов, стандартно поставляемых вместе с компилятором, которые и реализуют основные операции ввода-вывода.

Причиной является как слишком большое разнообразие операций ввода и вывода в разных операционных системах, особенно графических, так и возможность определения новых типов данных в языке Си++. Вывод даже простой строки текста в MS DOS, MS Windows и в X Window настолько различен, что пытаться придумать общие для всех них операторы было бы слишком небыло и на самом деле затруднило бы работу. Что же говорить о классах, определенных программистом, у которых могут быть совершенно специфические требования к их вводу-выводу.

Библиотека классов для ввода-вывода решает две задачи. Во-первых, она обеспечивает эффективный ввод-вывод всех встроенных типов и простое, но тем не менее гибкое, определение операций ввода-вывода для новых типов, разрабатываемых программистом. Во-вторых, сама библиотека позволяет при необходимости развивать её и модифицировать.

В задачу данной книги не входит описание программирования в графических системах типа MS Windows. То, что описано в данной главе, относится к самому простому алфавитно-цифровому выводу на терминал, который будет работать на консольном окне MS Windows, в MS DOS, в системе Unix.

Для того, чтобы узнать структуру библиотеки потоков и как её можно изменить, лучше обратиться к [13] и [15].

Поскольку Си++ совместим с Си, в нем доступны стандартные функции ввода-вывода языка Си, такие как printf. Мы же будем рассматривать классы, составляющие библиотеку потоков ввода-вывода.

9.1. Потоки

Механизм для ввода-вывода в Си++ называется потоком. Название произошло от того, информация вводится и выводится в виде потока байтов - символ за символом.
Класс istream реализует поток ввода, класс ostream — поток вывода. Эти классы определены в файле заголовков iostream.h. Библиотека потоков ввода-вывода определяет три глобальных объекта: cout, cin и cerr. cout называется стандартным выводом, cin — стандартным вводом, cerr — стандартным потоком сообщений об ошибках. cout и cerr выводят на терминал и принадлежат к классу ostream, cin имеет тип istream и вводит с терминала. Разница между cout и cerr существенна в Unix — они используют разные дескрипторы для вывода. В других системах они существуют больше для совместимости.

Вывод осуществляется с помощью операции >>, ввод с помощью операции <<. Выражение

    cout << "Пример вывода: " << 34;

напечатает на терминале строку "Пример вывода", за которым будет выведено число 34. Выражение

    int x;
    cin >> x;

введёт целое число с терминала в переменную x. (Разумеется, для того, чтобы ввод произошел, на терминале нужно напечатать какое-либо число и нажать клавишу возврат каретки.)

В классах iostream операции >> и << определены для всех встроенных типов языка C++ и для строк (тип char*). Если мы хотим использовать такую же запись для ввода и вывода других классов, определенных в программе, для них нужно определить эти операции.

```cpp
class String
{
public:
    friend ostream& operator<<(ostream& os, const String& s);
    friend istream& operator>>(istream& is, String& s);
private:
    char* str;
    int length;
};

ostream& operator<<(ostream& os, const String& s)
{
    os << s.str;
    return os;
}

istream& operator>>(istream& is, String& s)
{
    // Предполагаем, что строк длинее 1024 байтов не будет
    char tmp[1024];
    is >> tmp;
```
if (str != 0) {
    delete [] str;
}

length = strlen(tmp);
str = new char[length + 1];
if (str == 0) {
    // обработка ошибок
    length = 0;
    return is;
}

strcpy(str, tmp);
return is;

Как показано в примере класса String, операция «», во-первых, не метод класса String, а отдельная функция. Она и не может быть методом класса String, поскольку её правый операнд — объект класса ostream. С точки зрения записи, она могла бы быть методом класса ostream, но тогда с добавлением нового класса приходилось бы модифицировать класс ostream, что невозможно — каждый бы модифицировал стандартные классы, поставляемые вместе с компилятором. Когда же операция «» реализована как отдельная функция, достаточно в каждом новом классе её определить и можно использовать запись:

String x;
.
.
cout << "this is a string: " << x;

Во-вторых, операция «» возвращает в качестве результата ссылку на поток вывода. Это позволяет её использовать в выражениях типа приведенного выше, соединяющих несколько операций вывода в одно выражение.

Аналогично реализована операция ввода. Для класса istream она определена для всех встроенных типов языка Си++ и указателей на строку символов. Если необходимо, чтобы класс, определенный в программе, позволял ввод из потока, для него нужно определить операцию «» в качестве функции-friend:

9.1.1. Манипуляторы и форматирование ввода-вывода

Часто необходимо вывести строку или число в определенном формате. Для этого используются так называемые манипуляторы.

Манипуляторы — это объекты особых типов, которые управляют тем, как ostream или istream обрабатывают последующие аргументы.
Некоторые манипуляторы могут также выводить или вводить специальные символы.

С одним манипулятором мы уже сталкивались: `endl`. Он вызывает вывод символа новой строки. Другие манипуляторы позволяют задавать формат вывода чисел:

- `endl` при выводе перейти на новую строку;
- `ends` вывести нулевой байт (признак конца строки символов);
- `flush` немедленно вывести, опустошить все промежуточные буфера;
- `dec` выводить числа в десятичной системе (действует по умолчанию);
- `oct` выводить числа в восьмеричной системе;
- `hex` выводить числа в шестнадцатеричной системе счисления;
- `setw(int n)` установить ширину поля вывода в n символов (n – целое число);
- `setfill(int n)` установить символ-заполнитель; этим символом выводимое значение будет дополняться до необходимой ширины;
- `setprecision(int n)` установить количество цифр после запятой при выводе вещественных чисел;
- `setbase(int n)` установить систему счисления для вывода чисел; n может принимать значения 0, 2, 8, 10, 16, причем 0 означает систему счисления по умолчанию, т.е. 10.

Использование манипуляторов просто — их надо вывести в выходной поток. Предположим, мы хотим вывести одно и то же число в разных системах счисления:

```cpp
int x = 53;
cout << "Десятичный вид: " << dec << x << endl
    << "Восьмеричный вид: " << oct << x << endl
    << "Шестнадцатеричный вид: " << hex << x << endl;
```

Аналогично используются манипуляторы с параметрами. Вывод числа с разным количеством цифр после запятой:

```cpp
double x;
// вывести число в поле общей шириной 6 символов
// (3 цифры до запятой, десятичная точка
// и 2 цифры после запятой)
cout << setw(6) << setprecision(2) << x << endl;
```

...
Те же самые манипуляторы (за исключением end1 и ends) могут использоваться и при вводе. В этом случае они описывают, как представлены вводимые числа. Кроме того имеется манипулятор, работающий только при вводе: ws. Этот манипулятор переключает вводной поток в такой режим, при котором все пробелы (включая табуляции, переводы строки, переводы каретки и переводы страницы) будут вводиться. По умолчанию эти символы воспринимаются как разделители между атрибутами ввода.

```cpp
int x;
// ввести шестнадцатиричное число
cin >> hex >> x;
```

**Упражнение**

Разработайте программу перевода целых чисел из одной системы счисления в другую. Системы счисления для вводимых и выводимых чисел должны задаваться в начале работы программы, а затем программа читает числа одной системы счисления и выводит их в другую.

**9.2. Строковые потоки**

Специальным случаем потоков являются строковые потоки, представленные классом `stringstream`. Отличие этих потоков состоит в том, что все операции происходят в памяти. Фактически такие потоки формируют форматированную строку символов, заканчивающуюся нулевым байтом. Строковые потоки применяются прежде всего для того, чтобы облегчить форматирование данных в памяти.

Например, в приведенном в предыдущей главе классе `Exception` для исключительной ситуации можно добавить сообщение. Если мы хотим составить сообщение из нескольких частей, то может возникнуть необходимость форматирования этого сообщения:

```cpp
// произошла ошибка
stringstream ss;
ss << "Ошибка ввода-вывода, регистр: " << oct << reg1;
ss << "Системная ошибка номер: " << dec << errno << ends;
String msg(ss.str());
ss.rdbuf()->freeze(0);
Exception ex(Exception::INTERNAL_ERROR, msg);
throw ex;
```

Прежде всего создается объект типа `stringstream` с именем ss. Затем созданный строковый поток выводятся сформированные нужным образом данные. Отметим, что в конце мы вывели манипулятор ends, который добавил необходимый для символьной строки байтов нулевой байт. Метод `str()` класса `stringstream` дает доступ к сформированной строке (тип его возвращаемого значения — `char*`). Следующая строка освобождает память, занимаемую строковым потоком (подробности см. ниже). Последние две строки создают объект типа `Exception` с типом
ошибки INTERNAL_ERROR и сформированным сообщением и возбуждают исключительную ситуацию.

Важным свойством класса stringstream является то, что он автоматически выделяет нужное количество памяти для хранения строк. В следующем примере функция split_numbers выделяет числа из строки, состоящей из нескольких чисел, разделенных пробелом, и печатает их по одному на строке.

```
#include <sstream.h>

void
split_numbers(const char* s)
{
    stringstream iostream;
    iostream << s << ends;
    int x;
    while (s >> x)
        cout << x << endl;
}

void
main()
{
    split_numbers("123 34 56 932");
}
```

Замечание. В среде Visual C++ файл заголовков называется stringstream.

Как видно из этого примера, независимо от того, какова на самом деле длина входной строки, объект iostream автоматически выделяет память и при выходе из функции split_numbers, когда объект уничтожается, память будет освобождена.

Однако имеется одно исключение из данного правила. Если программа обращается непосредственно к хранимой в объекте строке с помощью метода str(), то объект перестает контролировать эту память, что означает, что при уничтожении объекта память не будет освобождена. Для того, чтобы память все-таки была освобождена, необходимо вызов метода rdbuf()->freeze(0) (см. предыдущий пример).

9.3. Ввод-вывод файлов

Ввод-вывод файлов может осуществляться как с помощью стандартных функций библиотеки C, так и с помощью потоков ввода-вывода. Функции библиотеки C являются функциями низкого уровня, без всякого контроля типов.

Прежде, чем перейти к рассмотрению собственно классов, остановимся немного на том, как происходят операции ввода-вывода с файлами. Файл рассматривается как последовательность байтов. Чтение или запись происходит последовательно. Например, при чтении мы начинаем с начала файла. Предположим, первая операция чтения ввела 4 байта, интер-
протиправленное как целое число. Тогда следующая операция чтения начнёт ввод с пятого байта, и так далее до конца файла.

Аналогично происходит запись в файл — по умолчанию первая запись производится в конец имеющегося файла, а все последующие операции записи последовательно пишут данные друг за другом. При операціях чтения-записи говорят, что существует текущая позиция, начиная с которой будет производиться следующая операция.

Большинство файлов обладают возможностью прямого доступа. Это означает, что можно производить операции ввода-вывода не последовательно, а в произвольном порядке: после чтения первых 4-х байтов прочесть с 20 по 30, затем два последних и т.п. При написании программ на языке Си++ возможность прямого доступа обеспечивается тем, что текущую позицию чтения или записи можно явно установить.

В библиотеке Си++ для ввода-вывода файлов существуют классы ofstream (вывод) и ifstream (ввод). Оба они выведены из класса fstream. Сами операции ввода-вывода выполняются так же, как и для других потоков — операции >> и << определены для класса fstream как ввести и вывести соответствующее значение. Различия заключаются в том, как создаются объекты и как они привязываются к нужным файлам.

При выводе информации в файл прежде всего нужно определить в какой файл будет производиться вывод. Для этого можно использовать конструктор класса ofstream в виде:

```cpp
ofstream(const char* szName, int nMode = ios::out,
int nProt = filebuf::openprot);
```

Первый аргумент — имя выходного файла и это единственный обязательный аргумент. Второй аргумент задает режим, в котором открыывается поток. Этот аргумент — битовое ИЛИ следующих величин:

- ios::app — при записи данные добавляются в конец файла, даже если текущая позиция была перед этим перемещена;
- ios::ate — при создании потока текущая позиция помещается в конец файла; однако в отличие от режима app запись ведётся в текущую позицию;
- ios::in — поток создается для ввода; если файл уже существует, он сохраняется;
- ios::out — поток создается для вывода (режим по умолчанию);
- ios::trunc — если файл уже существует, его предыдущее содержимое уничтожается и длина файла становится равной нулю; режим действует по умолчанию, если не заданы ios::ate, ios::app или ios::in;
- ios::binary — ввод-вывод будет происходить в двоичном виде, по умолчанию используется текстовое представление данных.
Третий аргумент используется только в том случае, если создается новый файл и он определяет параметры создаваемого файла.

Можно создать поток вывода с помощью стандартного конструктора без аргументов, а позднее выполнить метод _open_ с такими же аргументами, как и у предыдущего конструктора:

```cpp
void open(const char* szName, int nMode = ios::out,
          int nProt = filebuf::openprot);
```

Только после того, как поток создан и соединен с определенным файлом (либо с помощью конструктора с аргументами, либо с помощью метода _open_) можно выполнять вывод. Выводятся данные операцией <<. Кроме того, данные можно вывести с помощью методов _write_ или _put_:

```cpp
ostream& write(const char* pch, int nCount);
ostream& put(char ch);
```

Метод _write_ выводит указанное количество байтов (nCount), расположенных в памяти, начиная с адреса pch. Метод _put_ выводит один байт.

Для того, чтобы переместить текущую позицию, используется метод _seekp_:

```cpp
ostream& seekp(streamoff off, ios::seek_dir dir);
```

Первый аргумент — целое число, смещение позиции в байтах. Второй аргумент определяет откуда отсчитывается смещение и он может принимать одно из трех значений:

- `ios::beg` — смещение от начала файла
- `ios::cur` — смещение от текущей позиции
- `ios::end` — смещение от конца файла

Сместив текущую позицию, операции вывода продолжаются с нового места файла.

После завершения вывода можно выполнить метод _close_, который завершает внутренние буфера в файл и отсоединяет поток от файла. Также может произойти при ошибке объекта.

Класс _ifstream_, осуществляющий ввод из файлов, аналогичен. При создании объекта типа _ifstream_ в качестве аргумента конструктора можно задать имя существующего файла

```cpp
ifstream(const char* szName, int nMode = ios::in,
         int nProt = filebuf::openprot);
```

Можно воспользоваться стандартным конструктором, а подсоединиться к файлу с помощью метода _open_.

Чтение из файла производится операцией >> или методами _read_ или _get_.
Метод read вводит указанное количество байтов (nCount) в память, начиная с адреса pch. Метод get вводит один байт.

Также, как и для вывода, текущую позицию ввода можно изменить с помощью метода seekr, а по окончании операций закрыть файл с помощью close или просто уничтожить объект.

В заключении отметим, что описание операций ввода-вывода файлов в данном параграфе намеренно неполное. Во-первых, некоторые операции ввода-вывода файлов имеют специфику, зависящую от конкретной операционной системы и компилятора, поэтому за деталями лучше обращаться к технической документации Вашего компилятора. Во-вторых, при разработке стандартной библиотеки языка Си++ определения операций ввода-вывода были изменены (см. раздел 11.5.3).

Упражнения

1. Реализуйте для классов Book и Magazine операции ввода-вывода.
2. Напишите программу копирования одного файла в другой.
3. Разработайте программу сохранения и восстановления состояния объекта в файле. При первом вызове программа должна вначале создать объект типа Book, установить его атрибуты и затем записать его состояние в файл. При последующих вызовах программа должна восстанавливать состояние объекта из файла. Попытайтесь воспользоваться операторами << и >>, определенными для данного класса.
10. Шаблоны

10.1. Зачем нужны шаблоны

Достаточно часто можно записать алгоритм выполнения какого-либо действия независимо от того, какого типа данные мы обрабатываем. Простейшим примером будет определение максимума из двух величин.

```c
if (a > b)
    x = a;
else
    x = b;
```

Независимо от того, к какому именно типу принадлежат переменные a, b и x, если это один и тот же тип, для которого определена операція больше, запись будет одна и та же. Было бы естественно определить функцию `max`, возвращающую максимум из двух своих аргументов. Возникает вопрос, как описать аргументы этой функции? Конечно, можно определить `max` для всех известных типов, однако, во-первых, пришлось бы повторять одну и ту же запись многократно, а во-вторых, с добавлением новых классов добавлять новые функции.

Аналогичная ситуация встречается и со многими сложными структурами данных. В классе, реализующем связанный список целых чисел, алгоритмы добавления нового атрибута списка, поиска нужного атрибута и так далее, не зависят от того, что атрибуты списка — целые числа. Точно такие же алгоритмы нужно будет реализовать для списка вещественных чисел или указателей на класс `Book`.

Механизм шаблонов в языке C++ позволяет эффективно решать такие и многие подобные задачи.

10.2. Функции-шаблоны

Запишем алгоритм нахождения максимума двух величин, где в качестве параметра выступает тип этих величин.

```c
template<class T>
const T& max(const T& a, const T& b)
{
    if (a > b)
        return a;
    else
        return b;
}
```
Эта запись еще не создала ни одной функции, это лишь шаблон для конкретной функции. Только тогда, когда происходит обращение к функции с аргументами конкретного типа, произойдет генерация конкретной функции.

```cpp
int x, y, z;
String s1, s2, s3;

// генерация функции max для класса String
s1 = max(s2, s3);

// генерация функции max для типа int
x = max(y, z);
```

Первое обращение к функции max генерирует функцию

```cpp
const String& max(const String& a, const String& b);
```

Второе обращение генерирует функцию

```cpp
const int& max(const int& a, const int& b);
```

Объявление шаблона функции max говорит, что конкретная функция зависит от одного параметра — типа T. Первое обращение к max в программе использует аргументы типа String. В шаблон функции подставляется тип String вместо T. Получается функция:

```cpp
const String& max(const String& a, const String& b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Эта функция компилируется и используется в программе. Аналогичные действия происходят и при втором обращении, только теперь вместо параметра T подставляется тип int. Как видно из приведенных примеров, компилятор сам определяет, какую функцию надо использовать и автоматически генерирует необходимое определение.

У функции-шаблона может быть несколько параметров. Так, например, функция `find` библиотеки STL (стандартной библиотеки шаблонов), которая ищет первый элемент, равный заданному, в интервале значений, имеет вид:

```cpp
template <class InIterator, class T>
InIterator
find(InIterator first, InIterator last, const T& val);
```
Класс T — это тип элементов интервала. Тип iterator — тип указателя на начало и конец интервала.

10.3. Шаблоны классов

Поскольку объектно-ориентированное программирование предполагает объединение данных с алгоритмами их обработки в классы, естественным является наличие классов-шаблонов в языке C++. Шаблон класса имеет вид:

```cpp
template <листинг параметров>
class объявление_класса
```

Список параметров класса-шаблона аналогичен списку параметров функции-шаблона: список классов и переменных, которые подставляются в объявление класса при генерации конкретного класса.

Очень часто шаблоны используются для создания коллекций, т.е. классов, представляющих из себя набор объектов одного и того же типа. Простейшим примером коллекции будет массив, который мы рассматривали в гл. 5. Массив, несомненно, очень удобная структура данных, однако у него имеется ряд существенных недостатков, к которым, например, относится необходимость задания размера массива при его определении и отсутствие контроля за значениями индексов при обращении к атрибутам массива.

Попробуем при помощи шаблонов устранить два отмеченных недостатка у одномерного массива. При этом, по мере возможности, попытаемся сохранить синтаксис обращения к атрибутам массива. Назовем новую структуру данных вектор vector.

```cpp
template <class T>
class vector
{
public:
    vector() : nItem(0), items(0) {}; 
    ~vector() { delete items; };

    void insert(const T& t)
    { T* tmp = items;
        items = new T[nItem + 1];
        memcpY(items, tmp, sizeof(T)* nItem);
        item[++nItem] = t;
        delete tmp; }
    void remove(void)
    { T* tmp = items;
        items = new T[--nItem];
        memcpY(items, tmp, sizeof(T) * nItem);
        delete tmp; }

    const T& operator[](int index) const
```
{  
if ((index < 0) || (index >= nItem))  
    throw IndexOutOfBoundsException;  
return items[index];  
}

T& operator[](int index)  
{  
if ((index < 0) || (index >= nItem))  
    throw IndexOutOfBoundsException;  
return items[index];  
}

private:  
T* items;  
int nItem;  
};

Кроме конструктора и деструктора у нашего вектора есть только три метода: метод insert добавляет в конец вектора новый элемент, увеличивая длину вектора на единицу, метод remove удаляет последний элемент вектора, уменьшая его длину на единицу, и операция [] обращается к n-ому элементу вектора.

vector<int> IntVector;

IntVector.insert(2);  
IntVector.insert(3);  
IntVector.insert(25);  
// получили вектор из трех атрибутов 2, 3 и 25

// переменная x получает значение 3  
int x = IntVector[1];  
// будет возбуждена исключительная ситуация  
int y = IntVector[4];  
// изменить значение второго атрибута вектора.  
IntVector[1] = 5;

Обратите внимание, что операция [] определена в двух вариантах – как константный метод и как неконстантный. Если операция [] используется справа от операции присваивания (в первых двух присваиваниях), то используется неконстантный её вариант, если слева (в последнем присваивании) – константный. Использование операции индексирования [] слева от операции присваивания означает, что значение объекта изменяется, соответственно нужна неконстантная операция.

Параметр шаблона vector – любой тип, у которого определены операция присваивания и стандартный конструктор. (Стандартный конструктор необходим при операции new для массива.)

Также, как и с функциями-шаблонами, автоматически, при задании первого объекта типа vector<int>, происходит генерация конкретного класса из шаблона. Если далее в программе будет использован вектор
вещественных чисел или строк, соответственно будут сгенерированы конкретные классы и для них. Генерация конкретного класса означает, что генерируются все его методы, соответственно, размер исходного кода растет. Поэтому из небольшого шаблона может получиться большая программа. Ниже мы рассмотрим одну возможность сократить размер программы, используя при этом безотносительные шаблоны.

Сгенерировать конкретный класс из шаблона можно явно, записав:

```cpp
template vector<int>;
```

Этот оператор не создаст никаких объектов типа vector<int> но тем не менее вызовет генерацию класса со всеми его методами.

### 10.3.1. Интеллектуальный указатель

Рассмотрим еще один пример использования класса-шаблона. С помощью него мы попытаемся "улучшить" указатели языка Си++. Если указатель указывает на объект, выделенный с помощью операции new, необходимо явно вызывать операцию delete тогда, когда объект становится ненужен. Однако далеко не всегда просто определить, нужен объект или нет, особенно если на него могут ссылаться несколько разных указателей. Разработаем класс, который ведет себя очень похоже на указатель, но автоматически уничтожает объект, когда уничтожается последняя на него ссылка. Назовем этот класс "интеллектуальный указатель" (Smart Pointer). Идея заключается в том, что настоящий указатель мы окружим специальной оболочкой. Вместе со значением указателя мы будем хранить счетчик – сколько других объектов на него ссылается. Как только этот счетчик станет равным нулю, объект, на который указатель указывает, пора уничтожать.

Структура Ref хранит исходный указатель и счетчик ссылок.

```cpp
template <class T>
struct Ref {
    T* realPtr;
    int counter;
};
```

Теперь определим интерфейс интеллектуального указателя:

```cpp
template <class T>
class SmartPtr {
public:
    // конструктор из обычного указателя
    SmartPtr(T* ptr = 0);
    // копирующий конструктор
    SmartPtr(const SmartPtr& s);
    ~SmartPtr();
```
У класса `SmartPtr` определены операции обращения к элементу `->`, взятия по адресу `"*"` и операции присваивания. С объектом класса `SmartPtr` можно обращаться практически также, как и с обычным указателем.

```cpp
struct A
{
    int x;
    int y;
};

SmartPtr<A> aPtr(new A); // создать новый указатель
int x1 = aPtr->x; // обратиться к элементу A
(*aPtr).y = 3; // обратиться по адресу
```

Рассмотрим реализацию методов класса `SmartPtr`. Конструктор инициализирует объект указателем. Если указатель равен нулю, то `refPtr` устанавливается в ноль. Если же конструктору передается ненулевой указатель, то создается структура `Ref`, счетчик обращений в которой устанавливается в 1 и указатель — в переданный указатель:

```cpp
template <class T>
SmartPtr<T>::SmartPtr(T* ptr)
{
    if (ptr == 0)
        refPtr = 0;
    else {
        refPtr = new Ref<T>;
        refPtr->realPtr = ptr;
        refPtr->counter = 1;
    }
}
```

Деструктор уменьшает количество ссылок на 1 и, если он достигает 0, уничтожает объект:

```cpp
template <class T>
SmartPtr<T>::~SmartPtr()
{
    if (refPtr != 0) {
```
refPtr->counter--;  
if (refPtr->counter <= 0) {
    delete refPtr->realPtr;
    delete refPtr;
}

Реализация операций -> и * довольно проста:

template <class T>
T*  
SmartPointer<T>::operator->() const
{
    if (refPtr != 0)
        return refPtr->realPtr;
    else
        return 0;
}

template <class T>
T&  
SmartPointer<T>::operator*() const
{
    if (refPtr != 0)
        return *refPtr->realPtr;
    else
        throw bad_pointer;
}

Самые сложные для реализации – это копирующий конструктор и операции присваивания. При создании объекта SmartPtr – копии имеющегося, мы не будем копировать сам исходный объект. Новый интеллигентный указатель будет ссылаться на тот же объект, мы лишь увеличив счетчик ссылок.

template <class T>
SmartPointer<T>::SmartPointer(const SmartPtr& s):refPtr(s.refPtr)  
{
    if (refPtr != 0)
        refPtr->counter++;
    }

При выполнении присваивания прежде всего нужно отсоединиться от имеющегося объекта, а затем присоединиться к новому аналогично тому, как это сделано в копирующем конструкторе.

template <class T>
SmartPointer&
```cpp
SmartPointer<T>::operator=(const SmartPtr& s)
{
    // отсоединиться от имеющегося указателя
    if (refPtr != 0) {
        refPtr->counter--;
        if (refPtr->counter <= 0) {
            delete refPtr->realPtr;
            delete refPtr;
        }
    }

    // присоединить к новому указателю
    refPtr = s.refPtr;

    if (refPtr != 0)
        refPtr->counter++;
}

В следующей функции по выходу из функции объект класса Complex будет уничтожен:

```cpp
void foo(void)
{
    SmartPtr<Complex> complex(new Complex);
    SmartPtr<Complex> ptr = complex;
    return;
}
```

### 10.3.2. Задание свойств класса

Одним из интересных методов использования шаблонов является уточнение поведения с помощью дополнительных параметров шаблона. Предположим, мы пишем функцию сортировки вектора:

```cpp
template <class T>
void sort_vector(vector<T>& vec)
{
    for (int i = 0; i < vec.size() - 1; i++)
        for (int j = i; j < vec.size(); j++)
            if (vec[i] < vec[j]) {
                T tmp = vec[i];
                vec[i] = vec[j];
                vec[j] = tmp;
            }
}
```
Эта функция будет работать хорошо с числами, но если мы захотим её использовать для массива указателей на строки (char*), то результат будет не совсем ожидаемый. Сортировка произойдет не по значению строк, а по их адресам (операция меньше для двух указателей — это сравнение значений этих указателей, т.е. адресов величин, на которые они указывают, а не самих величин). Чтобы исправить недостаток, добавим второй параметр к шаблону:

```cpp
template <class T, class Compare>
void sort_vector(vector<T>& vec)
{
    for (int i = 0; i < vec.size() - 1; i++)
        for (int j = i; j < vec.size(); j++)
            if (Compare::less(vec[i], vec[j]))
            {
                T tmp = vec[i];
                vec[i] = vec[j];
                vec[j] = tmp;
            }
}
```

Класс Compare должен реализовывать статическую функцию `less`, сравнивающую два значения типа T. Для целых чисел этот класс может выглядеть следующим образом:

```cpp
class CompareInt
{
    static bool less(int a, int b) { return a < b; };
};
```

и сортировка вектора будет выглядеть

```cpp
vector<int> vec;
sort<int, CompareInt>(vec);
```

Для указателей на байт (строк) можно создать класс

```cpp
class CompareCharStr
{
    static bool less(char* a, char* b)
    {
        return strcmp(a, b) >= 0;
    }
};
```

и, соответственно, сортировать с помощью вызова

```cpp
vector<char*> svec;
sort<char*, CompareCharStr>(svec);
```
Как легко заметить, для всех типов, для которых операция меньше имеет нужный нам смысл, можно написать шаблон класса сравнения:

```cpp
template<class T> class Compare {
    static bool less(T a, T b) { return a < b; }
};
```

и использовать его в сортировке (обратите внимание на пробел между закрывающимися угловыми скобками в параметрах шаблона; если его не поставить, компилятор спутает две скобки с операцией сдвига):

```cpp
vector<double> dvec;
sort<
double, Compare<double> >(dvec);
```

Чтобы не загромождать запись, воспользуемся возможностью задания значения параметра по умолчанию. Так же, как и для аргументов функций и методов, для параметров шаблона можно определить значения по умолчанию. Окончательный вид функции сортировки будет следующий:

```cpp
template <class T, class C = Compare<T> >
void sort_vector(vector<T>& vec) {
    for (int i = 0; i < vec.size() - 1; i++)
        for (int j = i; j < vec.size(); j++) {
            if (C::less(vec[i], vec[j])) {
                T tmp = vec[i];
                vec[i] = vec[j];
                vec[j] = tmp;
            }
        }
}
```

Второй параметр шаблона иногда называют параметром-штрихом, поскольку он лишь модифицирует поведение класса, который манипулирует с типом, определяемым первым параметром.

**Упражнения**

1. Напишите функцию-шаблон для нахождения минимального элемента в векторе vector.

2. Разработайте класс-шаблон для вектора, в котором элементы упорядочены по возрастанию.
11. Библиотеки классов

Одним из важнейших преимуществ объектно-ориентированного программирования является возможность многократного использования разработанного программного обеспечения. Реальное ускорение процесса разработки возможно только тогда, когда конкретная программная система разрабатывается не с нуля, а используя готовые составные части. Разумеется, и раньше, для процедурно-ориентированных языков программирования типа Форtran, Паскаля или Си создавались библиотеки функций, реализующих наиболее употребляемые алгоритмы обработки данных. В данной главе мы рассмотрим возможности многократного использования программного обеспечения, написанного на языке Си++ и принципы построения библиотек классов.

11.1. Многократное использование функций и классов

Мы уже неоднократно упоминали библиотеки функций по ходу этой книги. Библиотеки функций создавались, прежде всего для расширения возможностей языка Си, а поскольку Си++ совместим с Си, то эти функции можно использовать и в программах на языке Си++.

Цель создания, как библиотек функций, так и библиотек классов одна – многократное использование готовых решений. Однако принципы их организации разны, что связано, прежде всего, с разницей между процедурным и объектно-ориентированным подходами к программированию.

Библиотека функций, как очевидно из названия, состоит из функций. Функция реализует какой-либо алгоритм обработки данных. Имеется только один способ приспособления функции к конкретной программе – задание различных аргументов.

Для хорошо определенных математических функций этого вполне достаточно. Поэтому, например, библиотека математических функций на языке ФорTRAN широко используется. Примером таких функций в стандартной библиотеке Си являются функции sqrt (вычисление квадратного корня), exp (возведение в произвольную степень) и др. В случае этих функций и аналогичных им тип аргументов фиксирован.

Иногда заранее тип аргумента не известен, и тогда приходится пользоваться "универсальным" типом данных void*. Например, в функции qsort, реализующей алгоритм быстрой сортировки, указатель на начало данных имеет тип void*, а другой аргумент задает размер одного атрибута данных:

```c
void qsort(void* base, size_t num, size_t width,
           int (*compare)(const void* elem1,
                           const void* elem2));
```
Других способов настройки стандартных функций на специфику конкретных задач практически не существует. Соответственно, применение библиотек функций весьма ограничено. Либо это стандартные математические функции, либо вспомогательные функции достаточно низкого уровня (например, qsort, printf), либо узкоспециализированные библиотеки функций решения определенного класса задач.

Возможности приспособления классов и объектов к конкретной задаче гораздо больше и они намного гибче.

Первый способ аналогичен настройке функций. Классы, реализующие какие-либо объекты, пригодные для использования в различных программах, помещаются в библиотеку и настраиваются с помощью параметров. Те же математические функции могут быть реализованы с помощью классов. В виде классов более гибко могут быть реализованы, например обработка комплексных чисел, работа с датами, интервалами времени и тому подобными универсальными объектами общего назначения.

Объектно-ориентированный подход позволяет создавать новые классы и даже системы классов на основе поставляемых в библиотеке.

Наиболее часто применяемый способ — уточнение с помощью наследования. Базовые классы реализуют основные алгоритмы обработки, а в программе из них выводятся конкретные классы, изменяющие и приспосабливающие их к текущей задаче. Практически вся библиотека классов MFC (Microsoft Foundation Classes — библиотека базовых классов Microsoft) построена по этому принципу. Конкретная программа состоит из классов, выведенных из классов, составляющих эту библиотеку.

Другой способ приспособления — реализация некоторого интерфейса для того, чтобы класс мог работать в наборе с другими классами. Особенно часто такое приспособление используется при работе с шаблонами. Например, для того, чтобы использовать объекты некоторого класса в качестве атрибута массива переменной длины (пример в п. 10.3), этот класс должен иметь стандартный конструктор и оператор присваивания.

11.2. Принципы построения библиотеки классов

При составлении библиотеки классов прежде всего необходимо определить ее цель и принцип построения.

По целям библиотеки классов прежде всего делятся на библиотеки общего назначения и специализированные по областям применения. Библиотеки общего назначения должны включать классы, используемые для разработки самих разнообразных программ, независимо от области их применения. Состав специализированных библиотек естественно зависит от области применения.

По принципам построения библиотеки делятся на библиотеки конкретных классов, библиотеки шаблонов, библиотеки абстрактных классов и среды разработки.
Библиотеки конкретных классов пользователю предоставляют набор “готовых к употреблению” классов. Их уже не надо подстраивать под конкретную программу, достаточно создать объект определенного класса.

Библиотеки шаблонов определяют шаблоны классов, которые нужно доопределить в конкретных программах, используя в качестве аргументов шаблонов классы, используемые в этой программе. Ниже мы более подробно рассмотрим стандартную библиотеку шаблонов.

Библиотеки аBSTрактных классов перед их использованием требуют определение конкретных классов, выведенных из классов библиотеки. Довольно часто используется смешанный подход – библиотека определяет конкретные классы, однако заложенная в них функциональность слишком общая для использования в более-менее сложных программах. Поэтому выведение из библиотечных классов все равно используется, несмотря на то, что классы не являются формально аBSTрактными.

Среды разработки – наиболее сложные формы библиотек классов. Основным отличием её от простой библиотеки классов является то, что среда разработки задает не только (и не столько) набор классов, как форму их использования. Например, библиотека MFC, входящая в состав Visual C++ компании Microsoft, во многом именно среда разработки, реализующая модель "документ-вид". В какой-то мере стандартная библиотека шаблонов тоже задает способ использования классов, поэтому может быть отнесена к средам разработки.

Деление библиотек на различные виды во многом условно. Каждая конкретная библиотека использует разные подходы.

В библиотеке классов необходимо соблюдать последовательный способ именования классов, переменных и констант.

Прежде всего имена, используемые в библиотеке, не должны пересекаться с именами, используемыми в программах, которые включают данную библиотеку. В настоящее время наиболее надежным методом является использование контекстов имен. До появления контекстов наиболее потребительный метод заключался в добавлении одного или того же префикса перед всеми доступными именами классов, констант и переменных. Например в библиотеке Tool.h++ компании Rogue Wave имена всех классов начинаются с двух букв "RW": RWDate, RWString и т.д. Часто в префик сах используется подчеркивание для того, чтобы его лучше выделить. Система имен библиотеки ODMG использует префикс "d_"; d_Time, d_Database.

При выборе имени важно быть последовательным. Если состояние объекта для класса X определяется с помощью метода OK, то непоследовательно использовать для аналогичных целей метод IsOK в классе Y и метод is_OK для класса Z. То же самое относится к использованию прописных и строчных букв в именах, использованию подчеркиваний и так далее. Одна и та же система именования должна использоваться во всех библиотеках.

Для того, чтобы какая-либо программа могла использовать библиотечные классы и функции, их описание, записанное во включающих файлах, должно быть включено с помощью оператора #include. Независимо от организации включающих файлов внутри библиотеки, должна быть...
продумана дисциплина использования этих файлов в пользовательских программах.

Фактически существует только одно правило — для использования одного класса нужно подключить не более одного включаемого файла. Например, для того, чтобы использовать класс RWDate нужно записать оператор

```
#include "rw/rwdate.h"
```

Если использование класса требует определения других классов, пользователь не должен включать дополнительные файлы в свою программу. Сам файл (в нашем случае rwdate.h) должен об этом позаботиться. Крайним случаем такой дисциплины является вариант, при котором в библиотеке есть один включаемый файл, упоминание которого в операторе `#include` дает возможность использовать все классы библиотеки. Такой файл, например, может сам включать все файлы заголовков библиотеки.

### 11.3. Библиотека классов Tools.h++

Библиотека Tools.h++ разработана компанией Rogue Wave и является одной из первых библиотек классов общего назначения. Библиотека широко используется благодаря продуманным интерфейсам и эффективной реализации.

Она состоит из трех частей. Первая часть — это набор конкретных классов общего назначения. К ним относятся классы, реализующие строки (RWCString), календарные даты (RWDate), время (RWTime), файлы (RWFile), ряд констант для математических вычислений. Класс RWCString, реализующий строки символов переменной длины, предоставляет широкий набор операций и эффективную реализацию. В частности, для строк реализованы расширенные регулярные выражения, позволяющие сравнивать строки с разнообразными шаблонами.

Пример использования класса RWDate:

```c++
// Начало века
RWDate d(1, "January", 1900);

// Сегодняшняя дата
RWDate today;

cout << "С начала века прошло "
    << today - d << " дней, " << endl;
```

Пример использования класса RWCString и класса RWCEnexpr — регулярного выражения, задающего шаблон, по которому можно осуществлять поиск в строке:

```c++
RWCString a("Штирлиц идет по коридору");
```
В результате переменная `a` получит значение "Исаев идет по коридору".

Вторая часть — классы для построения коллекций, моделирующих коллекции языка Smalltalk. Эта часть библиотеки позволяла строить коллекции пользовательских объектов без использования механизма шаблонов (фактически она была разработана в то время, когда шаблоны еще не являлись частью языка Си++). Реализация коллекций основана на том, что все атрибуты коллекций должны быть выведены из класса RWCollectable. После появления шаблонов ценность подобных коллекций сильно уменьшилась.

Третья часть библиотеки Tools.h++ — реализация коллекций с помощью шаблонов. Библиотека реализует все основные типы коллекций: линейные и двусвязанные списки, деревья, вектора и т.д. Библиотека использует разные коллекции для объектов и указателей на объекты. В последних версиях библиотеки Tools.h++ реализация собственных коллекций Rogue Wave основана на Стандартной Библиотеке Шаблонов (см. следующий параграф).

Для обоих типов коллекций библиотека предоставляет средства легкого сохранения объектов и коллекций на диске с последующим восстановлением сохраненного состояния объектов, включая ссылки на другие объекты.

Подробнее библиотека описана в [16].

### 11.4. Стандартная библиотека шаблонов STL

Стандартная библиотека шаблонов была разработана в исследовательской лаборатории компании Hewlett-Packard Александром Степановым, Менгом Ли и Давидом Мюстером. Принципы построения библиотеки состоят в том, что библиотека включает ортогональные реализации коллекций, итераторов и алгоритмов. Ортогональность или независимость означает, что коллекции, итераторы и алгоритмы могут сочетаться практически в любой комбинации, например алгоритм не зависит от того, какой именно тип коллекции используется. Кроме того, библиотека STL практически не использует наследования. Рассмотрим её возможности более подробно.

#### 11.4.1. Коллекции и алгоритмы

STL включает реализации списков, векторов, очередей (деков по терминологии перевода Д. Кнута [1]), наборов, множественных наборов, таблиц, упорядоченных таблиц, упорядоченных наборов. Все эти классы — шаблоны и могут генерировать коллекции, содержащие объекты любого
типа. Например, вектор целых чисел `vector<int>` можно использовать вместо массивов целых чисел, при этом не нужно заботиться о динамическом распределении памяти.

```c++
vector<int> vec(3); // определить вектор
    // из трех элементов
vec[0] = 4;
vec[1] = vec[0] + 3;
vec[2] = vec[0] + vec[1];
```

Библиотека STL также содержит широкий набор алгоритмов, реализованных с помощью функций-шаблонов для манипулирования с данными, содержащимися в коллекциях. Например, переставить атрибуты коллекции в обратном порядке можно с помощью алгоритма `reverse`:

```c++
reverse(v.begin(), v.end());
```

Отметим две особенности вызова функции. Во-первых, функция `reverse` — это глобальная функция, а не атрибут класса. Во-вторых, функция манипулирует не с отдельными атрибутами, а с интервалом. В данном случае интервал включает все атрибуты вектора.

Эти две особенности и составляют суть ортогональности построения библиотеки — алгоритмы отделены от классов. Функцию `reverse` можно использовать для перестановки элементов любой коллекции, для которой можно задать интервал, а это все виды коллекций STL — очереди, списки и т.п.. С её помощью можно даже переставить элементы стандартного массива.

```c++
double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
reverse(A, A + 6);
for (int i = 0; i < 6; ++i)
    cout << "A[" << i << "] = " << A[i];
```

Этот пример также использует интервал, первый аргумент — начало интервала, второй аргумент — конец интервала, причем второй аргумент показывает элемент, не включенный в интервал.

### 11.4.2. Итераторы

Аргументами функции `reverse` являются объекты специального вида, называемые *итераторами*. Итератор — это в какой-то степени обобщение понятия указателя. Итератор хранит положение элемента в коллекции, к которому происходит доступ.

Для итераторов определена операция "++" — переход к следующему элементу коллекции. Как легко заметить, эта операция для указателей тоже означает переход к следующему элементу массива. Также для ите-
раторов определена операция "*" — обращение к значению, на которое указывает итератор (сравните со смыслом операции "**" для указателей)).

Обычно итераторы связаны с определенной коллекцией. В приведенном выше примере методы vec.end() и vec.begin() возвращают объекты типа vector<int>::iterator. Метод begin возвращает итератор, указывающий первый элемент коллекции, метод end — итератор, указывающий на конец коллекции.

Итераторы — это тот механизм, с помощью которого происходит отделение алгоритмов от коллекций. Алгоритмы — это функции-шаблоны, параметризованные классом итератора. Определение алгоритма поиска атрибута в коллекции выглядит примерно так:

```cpp
template <class InputIterator, class T>
InputIterator find(InputIterator first,  
        InputIterator last,    
        const T& value)
{
    while (first != last && *first != value) ++first;
    return first;
}
```

Функция find использует три аргумента: первый аргумент — начало интервала, в котором происходит поиск, второй аргумент — конец этого интервала, и третий атрибут — искомое значение. Тип первого и второго аргументов — InputIterator, который и является параметром функции-шаблона. При генерации конкретной функции вместо него будет подставлен какой-то определенный тип.

11.4.3. Моделирование концепций

Как и при использовании любого шаблона, важным вопросом является то, какие типы можно использовать в качестве параметров шаблонов. Например, тип double* можно использовать в качестве параметра find, а тип double — нельзя. В STL для описания возможных параметров шаблонов используется понятие концепции. Есть концепция InputIterator. Какой-либо тип моделирует концепцию InputIterator, если этот тип реализует определенный набор методов и операций с конкретной семантикой. В частности, эти типы должны реализовывать упомянутые выше операции ++ и *. Концепция — это не понятие языка C++, это понятие введено в библиотеке STL для удобства определения.

Концепция может уточняться. Для функции reverse (перестановка элементов коллекции в обратном порядке) кроме операции ++ необходима операция —. Параметр функции reverse должен моделировать концепцию BidirectionalIterator. Концепция BidirectionalIterator уточняет концепцию InputIterator, поскольку BidirectionalIterator требует все то, что требует InputIterator плюс еще операцию —.
Как мы уже упоминали ранее, в 1994 году библиотека STL была утверждена комитетом ANSI/ISO в качестве части стандартной библиотеки языка Си++. Авторы STL получили престижную премию ACM за вклад в развитие вычислительной техники. Существует целый ряд книг, посвященных этой библиотеке, в частности [12].

Под влиянием того, что описание библиотеки STL были приняты в качестве части стандарта языка, многие другие библиотеки, в частности Tools.h++, описанная в предыдущем параграфе, стали модифицироваться для того, чтобы используемые коллекции стали соответствовать STL.

### 11.5. Стандартная библиотека языка Си++

Стандартная библиотека языка Си++ — это библиотека, определенная стандартом языка. Реализация языка Си++, соответствующая стандарту, должна реализовывать эту библиотеку. На самом деле стандартная библиотека Си++ — это набор, состоящий из следующих частей:

1. Адаптированная стандартная библиотека шаблонов (STL).
2. Библиотека ввода-вывода потоков iostream.
3. Средства определения национальных особенностей реализации языка Си++ (обозначение денежной единицы, представление чисел (десятичная точка или запятая, запись больших чисел), способ записи времени и даты и т.д.).
4. Класс-шаблон для представления строк символов.
5. Класс-шаблон для представления комплексных чисел.
6. Возможности описания конкретной вычислительной среды, в частности диапазонов чисел.
7. Средства управления памятью.
8. Средства поддержки разных языков, в частности сообщений на разных языках.
9. Предопределенные классы исключительных ситуаций.
10. Стандартная библиотека Си.

Ряд этих возможностей мы уже рассматривали. Следует отметить, что все части, несмотря на то, что многие из них уже существовали задолго до появления стандартной библиотеки Си++, при включении в библиотеку подверглись существенной переработке. Была переработана как система файлов-заголовков (даже имена стали другими — исчез суффикс ".h"), так и сами функции и классы.

Разработчики добились, чтобы библиотека выглядела не эклектичным собиранием всего, что было доступно, а действительно единой библиотекой. В частности этим объясняется широкое использование шаблонов, сообщением обошибках с помощью исключительных ситуаций, использование именованного контекста std.

Для того, чтобы лучше понять, что именно было переделано, рассмотрим немного подробнее три части: иерархию классов для исключительных ситуаций, класс строк и изменений в классах ввода-вывода потоков.
11.5.1. Классы исключительных ситуаций

Основная задача при проектировании этих классов состояла в том, чтобы, во-первых, иметь механизм сообщения об ошибках для самой библиотеки, и, во-вторых, дать программисту, использующему библиотеку, основу для создания собственных классов ошибок.

Все исключительные ситуации выведены из базового класса `exception`, который имеет вид:

```cpp
class exception
{
 public:
   exception(); throw();
   exception(const exception&) throw();
   exception& operator=(const exception&) throw();
   virtual ~exception() throw;
   virtual const char* what() const throw();

 private:

};
```

Как мы видим, кроме конструктора, копирующего конструктора, оператора присваивания и деструктора, имеется только один информативный метод `what`, который выдает описание ошибки.

Все исключительные ситуации разбиты на три группы: логические ошибки (`logic_error`), ошибки выполнения (`runtime_error`) и прочие. В табл. 4 приведены возможные причины исключительных ситуаций, а на рис. 13 показана их иерархия.

Таблица 4. Список исключительных ситуаций стандартной библиотеки Си++

<table>
<thead>
<tr>
<th>Имя исключительной ситуации</th>
<th>Возможная причина</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad_alloc</td>
<td>Ошибка в процессе выделения памяти</td>
</tr>
<tr>
<td>bad_cast</td>
<td>Невозможность преобразовать к типу ссылки в операции <code>dynamic_cast</code></td>
</tr>
<tr>
<td>bad_typeid</td>
<td>Задание неверного типа в операции <code>typeid</code></td>
</tr>
<tr>
<td>bad_exception</td>
<td>Задание несуществующей исключительной ситуации в спецификации <code>throw</code> или определении метода или функции</td>
</tr>
<tr>
<td>domain_error</td>
<td>Аргумент или значение не соответствуют области определения типа</td>
</tr>
<tr>
<td>invalid_argument</td>
<td>Неверный аргумент</td>
</tr>
<tr>
<td>ios_base::failure</td>
<td>Ошибка операции ввода-вывода</td>
</tr>
<tr>
<td>length_error</td>
<td>Неверная длина</td>
</tr>
<tr>
<td>out_of_range</td>
<td>Недопустимое значение индекса</td>
</tr>
<tr>
<td>overflow_error</td>
<td>Арифметическое переполнение при преобразовании битового вектора в целое число</td>
</tr>
<tr>
<td>range_error</td>
<td>Недопустимый интервал</td>
</tr>
<tr>
<td>underflow_error</td>
<td>Ошибка исчезновения значащих цифр при преобразованиях чисел</td>
</tr>
</tbody>
</table>
Всего определено 12 классов, которые закладывают хорошую основу для иерархии исключительных ситуаций в любой программе. Разумеется, это не означает, что программист не может сам определять классы для своих исключительных ситуаций, причем совсем не обязательно, чтобы они были производными от класса exception.

11.5.2. Стандартные коллекции

Стандартная библиотека Си++ включает в себя переработанную библиотеку STL, основой которой является набор шаблонов для построения коллекций. Рассмотрим более подробно как эти коллекции устроены.

Стандартная библиотека определяет два вида коллекций: последовательности и ассоциативные коллекции. Основное различие между ними заключается в том, что в последовательностях доступ к элементу осуществляется по его позиции, тогда как в ассоциативных коллекциях доступ к элементу осуществляется по ключу.

Тем не менее, большая часть интерфейсов у всех коллекций совпадает. Это следует принципу ортогональности, заложенному в библиотеке STL (на которой, напомним, основана стандартная библиотека Си++). Коллекции взаимозаменяются и большинство процедур обработки коллекций (алгоритмов) работает независимо от конкретного вида коллекции. Тем не менее, каждая конкретная коллекция оптимизирована на определенный вид доступа, что позволяет разрабатывать эффективные программы с помощью стандартной библиотеки Си++.

Основными последовательностями являются вектор, список и дк (классы vector, list и deque). На их базе построены очереди, стеки и
в очереди с приоритетами (классы queue, stack и priority_queue), которые представляют из себя интерфейсы к основным последовательностям, называемые иногда адапторами.

К ассоциативным коллекциям относятся словари (классы map и multimap) и наборы (классы set и multiset).

Мы подробно рассмотрим возможности, предоставляемые классами vector и map, как наиболее употребимыми, и кратко отметим особенности остальных видов коллекций.

11.5.2.1. Вектор

Класс-шаблон vector представляет из себя вектор однотипных элементов. Его интерфейс (приведены не все, а только основные методы):

template <class T, class A = allocator<T> >
class std::vector
{
public:
    // определение типов
    typedef T value_type;       // тип элемента
    typedef A allocator_type;

    typedef A::size_type size_type;
    // тип для размера вектора
    typedef A::difference_type difference_type;
    // тип для расстояния между элементами

    typedef A::pointer pointer;
    // тип указателя на элемент
    typedef A::const_pointer const_pointer;

    typedef A::reference reference;
    // тип ссылки на элемент
    typedef A::const_reference const_reference;

    // определение итератора зависит от вида коллекции и от реализации
    typedef value_type* iterator;
    typedef const value_type* const_iterator;
    typedef std::reverse_iterator<iterator>
              reverse_iterator;
    typedef std::reverse_iterator<const iterator>
              const_reverse_iterator;

    // итераторы
    iterator begin();       // начало вектора
    const_iterator begin() const;
iterator end(); // конец вектора
cnst_iterator end() const;
reverse_iterator rbegin();
    // начало обратной последовательности
const_reverse_iterator rbegin() const;
reverse_iterator rend();
    // конец обратной последовательности
const_reverse_iterator rend() const;

    // доступ к элементам
    // обращение к элементу по индексу
const_reference at(size_type n) const;
reference at(size_type n);
const_reference operator[](size_type n) const;
reference operator[](size_type n);

    // первый элемент
reference front();
cnst_reference front() const;
    // последний элемент
reference back();
cnst_reference back() const;

    // добавить новый элемент в конец последовательности
void push_back(const T& x);
void pop_back(); // удалить последний элемент

    // вставить элемент в определенной позиции
iterator insert(iterator pos, const T& x = T());
void insert(iterator pos, size_type m, const T& x);
    // удалить элементы из последовательности
iterator erase(iterator pos);
iterator erase(iterator first, iterator last);
    // удалить все элементы из последовательности
void clear();

    // конструкторы и дополнительные методы
explicit vector(const A& a = A());
vector(const vector& x);
~vector();
vector& operator=(const vector& x);

size_type size() const; // количество элементов

    // операции сравнения коллекций
bool operator==(const vector& x) const;
bool operator!=(const vector& x) const;
bool operator<(const vector& x) const;
void swap(vector& x); // обменяться элементами
Определеение типов

Прежде всего стандартная коллекция определяет псевдонимы для типов элементов вектора и итераторов. Псевдонимы типов приводятся для того, чтобы с коллекциями можно было работать, не зная конкретного типа элементов коллекции. В программе, работающей с вектором, можно обратиться к его элементам с помощью псевдонимов. Например, функцию нахождения минимального элемента в векторе можно было бы написать следующим образом:

```cpp
template <class V> typename V::value_type min(const V& v)
{
    typename V::value_type m = v[0];

    for (typename V::const_iterator it = v.begin();
         it != v.end(); ++it) {
        if (m > *it)
            m = *it;
    }

    return m;
}
```

Заметим, что здесь предполагается, что у элемента вектора определена операція больше, что вообще говоря необязательно для вектора. Как видно из данного примера, единственным параметром функции является сам класс вектор, а к его элементам обращаются через псевдоним типа V::value_type. Ключевое слово typename дает указание компилятору, что V::value_type — это название типа.
Вызов такого такого алгоритма потребует только одного аргумента — вектора:

```cpp
class T { ... };
vector<T> v;
... T min_elem = min(v);
```

**Итераторы**

Для вектора определены методы, создающие итераторы, использующиеся для навигации по коллекции. Для вектора определены как стандартный итератор, продвигающийся от первого элемента к последнему, так и обратный (reverse_iterator), продвигающийся от последнего элемента к первому.

Метод `begin()` создает итератор, указывающий на первый элемент коллекции. Операция `++` продвигает итератор к следующему элементу коллекции до тех пор, пока итератор не достигнет значения `end()`. Метод `end()` возвращает итератор, указывающий на позицию после последнего элемента коллекции, как бы на специальную позицию "конец коллекции".

Обращение к значению элемента, на который указывает итератор, осуществляется операцией итератора `*`. Для константных итераторов (`const_iterator, const_reverse_iterator`) эта операция позволяет только извлекать значение элемента, т.е. стоять в правой части операции присваивания. Для неконстантных итераторов она может применяться как в правой, так и левой части операции присваивания.

```cpp
vector<int> vi;
... vector<int>::iterator it = vi.begin();
while (it != vi.end()) {
   *it = *it + 10;
   ++it;
}
```

Для обратного итератора, создаваемого методами `rbegin()` и `rend()`, смысл операции `++` заключается в сдвиге к предыдущему элементу коллекции. Отметим, что пары `begin() — rend()` и `end() — rbegin()` возвращают одну и ту же позицию в коллекции.

**Доступ к элементам**

У вектора имеются две группы методов для доступа к элементам. Первая — общая для всех видов коллекций. К ней относятся методы `front()` и `back()`, позволяющие обращаться соответственно к первому и последнему элементам вектора. `push_back()` добавляет новый элемент
в конец коллекции, `pop_back()` удаляет последний элемент из коллекции.

Методы `insert()`, `erase()` и `clear()` позволяют вставлять и удалять элементы из произвольной позиции вектора.

Кроме того, вектор оптимизирован для доступа к элементам коллекции по их индексу (аналогично обычному массиву). Для этого в векторе определены операции `[]` и `at()`. Операция индексирования `[]` полностью аналогична операции индексирования в массиве, включая то, что она не проверяет выход за границы вектора:

```cpp
vector<long> vl;
long sum = 0;
for (int i = 0; i < vl.size(); ++i) {
    sum += vl[i];
}
```

Метод `at()` при попытке обратиться к несуществующему элементу вектора возбуждает исключительную ситуацию `out_of_range`:

```cpp
void print_elem(vector<string>& s, int k)
{
    try {
        cout << s.at(k) << endl;
    } catch(out_of_range) {
        cout << "недерный индекс" << endl;
    }
}
```

### 11.5.3.2. Словарь

Класс-шаблон `map` представляет последовательность пар, состоящих из ключа и значения. Он предназначен для быстрого доступа к значению элемента по его ключу. Название словарь как раз и отражает пример использования подобной коллекции.

Как мы уже отмечали, большая часть интерфейса всех коллекций одинакова. У словаря имеется обычный набор псевдонимов типов и методов для коллекций и ряд определений, относящихся только к ассоциативным коллекциям.

```cpp
template <class K, class T, class Cmp = less<K>,
class A = allocator<T> >
class std::map
{
public:
    // типы
```
typedef K key_type;
typedef T mapped_type;
typedef pair<const K, T> value_type;
typedef Cmp key_compare;

// доступ к элементам
mapped_type& operator[](const key_type& k);
// найти элемент по ключу
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;

У шаблона map имеется четыре параметра. Первый — это тип ключа, второй — тип значения. Заметим, что элемент коллекции map — это пара (ключ, значение), поэтому и определение псевдонима value_type — это пара, а не одно значение. Следствием такого определения является, например то, что все итераторы и стандартные методы обращения к элементам, работают с парами. У каждой пары есть первый и второй элементы к которым можно обратиться как к first и second. Подсчет суммы значений элементов коллекции будет выглядеть следующим образом:

map<string, int> m;

m<string, int>::iterator it = m.begin();
int sum = 0;
while (it != m.end()) {
    sum += it->second;
}

Третий, необязательный параметр шаблона map — это класс, определяющий способ сравнения ключей. Так же, как и у уже знакомого нам четвертого параметра, описывающего способ выделения памяти, у него есть значение по умолчанию. Стандартная библиотека включает класс-шаблон less, который определяет, что для сравнения ключей будет использоваться операция меньше.

Важнейшие дополнительные методы класса map позволяют индексировать элементы по ключу (вместо индексации элемента по номеру позиции в векторе) и находить элемент по ключу: Если мы хотим создать словарь, который хранит соответствие названия месяца и количества дней в нем, то это может выглядеть следующим образом:

map<string, int> daysInMonth;
daysInMonth["январь"] = 31;
daysInMonth["февраль"] = 28;
daysInMonth["март"] = 31;
daysInMonth["апрель"] = 30;
daysInMonth["май"] = 31;
daysInMonth["июнь"] = 30;
daysInMonth["июль"] = 31;
daysInMonth["август"] = 31;
daysInMonth["сентябрь"] = 30;
daysInMonth["октябрь"] = 31;
daysInMonth["ноябрь"] = 30;
daysInMonth["декабрь"] = 31;

for (map<string, int>::iterator p = daysInMonths.begin();
   p != daysInMonths.end(); ++p) {
   cout << "В месяце " << p->first
        << " имеется " << p->second << " дней" << endl;
}

int d = daysInMonth["март"];

В классе map гарантировано, что все значения ключа уникальны. Класс multimap используется тогда, когда в коллекции возможно более одной пары с одним и тем же значением ключа.

Наборы значений: классы set и multiset — это выраженные случаи ассоциативных коллекций, когда для ключа не задается никакого значения.

11.5.3.3. Другие коллекции

Принцип оптимизации каждой коллекции для определенного вида доступа определяет интерфейс оставшихся двух последовательностей — list и deque.

Список (list) оптимизирован для удаления и вставки элементов в произвольную позицию. Индексирование элементов в списке не реализовано как раз по соображениям эффективности. За исключением индексирования, список реализует все остальные методы вектора, которые мы упоминали.

Дополнительно к вектору, у списка есть методы сортировки и слияния:

template <class T, class A = allocator<T> >
class std::list
{
public:
   // ...
   // подсоединить элементы списка x к текущему списку
   // после позиции pos
   void splice(iterator pos, list& x);
   void splice(iterator pos, list& x, iterator p);
   void splice(iterator pos, list& x,
               iterator first, iterator lst);
   // отсортировать текущий список
   void sort();

   // ...
};
Класс deque представляет дек или двунаправленную очередь. Интерфейс deque объединяет интерфейс вектора и интерфейс списка.
Последовательности, которые реализованы с помощью адаптеров (стек, очередь и очередь с приоритетами) в качестве дополнительного аргумента шаблона позволяют задавать тип основной последовательности, на которой они основаны. Например, определение стека выглядит следующим образом:

```cpp
template <class T, class C = deque<T> >
class std::stack
{
protected:
    C c;
public:
    // переопределение псевдонимов типов
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type; // тип основной коллекции

    explicit stack(const C& a = C()) : c(a) {}
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    // стандартные операции для стека
    // реализованы через методы основной коллекции
    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};
```

По умолчанию стек использует двунаправленную очередь для хранения элементов (параметр c). Однако в принципе возможно использование и других видов коллекций для внутреннего представления стека.

11.5.3. Строки

Стандартная библиотека Си++ содержит класс string, предназначенный для манипулирования со строками символов. Операции, которые этот класс поддерживает, включают сравнение, слияние, поиск в строках, копирование строк, выделение символов из строки.
Манипулирование строками постоянно часто встречается в программах, что необходимость стандартных средств для её поддержки не вызывает сомнений. Однако удовлетворить все возможные потребности
одним классом вряд ли возможно. Поэтому практически все библиотеки классов общего назначения включают в себя свои собственные классы строк. Естественно, что такой класс включает и стандартная библиотека Си++. Кроме того, если в каком-то конкретном случае программисту необходимы дополнительные средства обработки строк, у него имеется возможность использовать имеющийся в библиотеке класс в качестве базового и вывести из него собственный, дополненный класс строк.

Достаточно часто используемым для стандартной библиотеки Си++ является прием определения базовых классов-шаблонов для задания общих свойств семейства классов. Поскольку большинство программистов использует весьма ограниченный круг возможностей, в библиотеке для их удобства определяются несколько наиболее применяемых классов как обычные классы – не шаблоны. Это делается чаще всего с помощью определения псевдонимов через оператор typedef.

В случае строк в стандартной библиотеке Си++ определен класс-шаблон basic_string.

```cpp
template <class Ch, class Tr = char_traits<Ch>,
    class A = allocator<Ch> >
class std::basic_string
{
    // определение класса ...
};
```

Строка – это последовательность символов. Поэтому интерфейс класса basic_string совпадает с интерфейсом последовательностей (см. предыдущий раздел).

Вообще говоря, класс basic_string определяет строки не только обычных символов, но и любых элементов с подходящими операциями копирования и сравнения. Свойства символов задаются с помощью специализации шаблона char_traits. Для обычных символов и для символов в коде Unicode (2-х байтовых символов), определены два класса:

```cpp
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

В большинстве случаев программисту будет достаточно использовать эти два класса. Если объявляется переменная или атрибут класса, лучше использовать класс string (или wstring для двухбайтовых строк). Если программисту нужно расширить возможности имеющихся в стандартной библиотеке Си++ строк и построить собственный класс, то в качестве базового лучше указывать класс basic_string с соответствующими параметрами.

**11.5.4. Стандартная библиотека ввода-вывода потоков**

Библиотека iostream была включена в стандартную библиотеку Си++ не только потому, что существует большой багаж программного
обеспечения, написанный с её использованием. Несмотря на то, что для ввода-вывода на терминал в современных графических средах используются совсем другие механизмы, потоки не потеряли своего значения как средство ввода-вывода файлов, средство передачи данных через коммуникационные протоколы, да и как средство вывода простой отладочной информации.

Первое изменение, которое претерпели классы iostream — это то, что они стали шаблонами. Вместо класса ostream в библиотеке имеется класс

```
template<class charT, class Traits = char_traits<charT> >
class basic_ostream
```

В качестве параметров шаблона используются два класса: первый, charT, — это тип элементов потока, и второй, Traits — параметр-штрих, определяющий свойства первого типа. В большинстве случаев в качестве элементов потока будет использоваться один из встроенных типов Си++ char или wchar_t, т.е. класс будет использоваться как поток байтов (каким и был старый класс ostream). Однако программист может определить поток, состоящий из элементов произвольного типа.

Параметр Traits описывает поведение элементов потока. В частности, он должен описывать какое значение используется для признака конца потока с помощью метода eof(), каким образом сравнивать элементы потока с помощью метода compare().

Подходящее поведение встроенных типов char и wchar_t описывает класс-шаблон char_traits<charT>, который и используется в качестве значения параметра по умолчанию. Заметим, что по умолчанию будут использоваться тот же самый класс Traits, что и для параметра-штриха в классе строк basic_string.

Традиционные имена классов потоков определены через псевдонимы типов:

```
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_iosstream<char> iostream;
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;
```

Таким образом, все возможности потоков ввода-вывода, о которых мы рассказывали в главе 9, определены и в новой стандартной библиотеке языка Си++.

Глобальные объекты cin, cout, cerr также определены, так что имеющиеся программы не требуют переделки при использовании новой библиотеки потоков.

Все классы потоков наследуют двум базовым классам. Один из них, ios_base содержит код, независимый от конкретного типа элементов потока. Второй класс, basic_ios<class charT, class Traits>, содержит
жив код, зависящий от конкретного типа параметра шаблона. Такое построение позволяет минимизировать объем кода, порождаемого при генерации конкретных классов потоков. В частности управление форматированием вывода реализовано без помощи шаблонов.

Поток может сообщать об ошибках двумя способами: с помощью индикаторов состояния и с помощью исключительных ситуаций. Поток может находиться в одном из нескольких состояний: нормальное, конец файла, ошибка ввода-вывода и разрушенное состояние. Соответствующие методы класса ios_base дают возможность проверять эти состояния. При необходимости можно задать, что поток будет возбуждать исключительные ситуации при определенных состояниях. такая схема дает возможность гибко управлять обработкой ошибок при вводе-выводе.

Строковые потокиistrstream заменены классом stringstream, который позволяет ввод-вывод со строками, составленными из объектов тех же типов, что и классыbasic_string. Кроме того, в новом классе улучшено распределение памяти и обращение к внутреннему буферу с помощью методаstr() не замораживает буфер (см. п.9.2).

Подробное описание стандартной библиотеки языка Си++ содержится в третьем издании книги Страуспула [13].
12. Краткий справочник по языку Си++

В данном разделе полностью приводятся все конструкции языка и кратко объясняется их смысл. Для записи синтаксиса используется слегка модифицированная запись Бэкуса-Наура. Квадратные скобки означают необязательный элемент. Разные варианты отделяются друг от друга вертикальной чертой. Несколько вариантов, заключенных в фигурные скобки, означает выбор одного из них. Повторение конструкции обозначается многоточием. Для того, чтобы отделить квадратные скобки, фигурные скобки и вертикальную черту, используемые для записи синтаксических правил, от тех же символов, используемых в самом языке, символы языка выделяются ширным шрифтом. Синтаксические правила в основном следует описанию в книге [13].

В каждом разделе этой главы дается ссылка на разделы в первой части книги, в которых соответствующие конструкции языка рассматриваются более подробно.

12.1 Лексические элементы

12.1.1 Алфавит

Каждый файл — это текст. Для записи программ используются знаки в кодировке ASCII. Русские буквы можно использовать в комментариях и литералах.

(§3.3.1)

12.1.2 Комментарии

Любой текст, начинающая с двух знаков деления // и до конца строки является комментарием, никак не анализируется компьютером и служит лишь для пояснений. Кроме того, любой текст, заключенный между символами /* и */ также является комментарием.

(§3.3.1)

12.1.3 Идентификаторы

Имя или идентификатор в языке Си++ — это последовательность знаков, начинающаяся с буквы или знака подчеркивания. В идентификаторах можно использовать заглавные и строчные латинские буквы, цифры и знак подчеркивания. Длина идентификаторов произвольна.

(§3.3.2)

12.1.4 Константы
Целые константы могут быть записаны в десятичной, восьмеричной или шестнадцатеричной системе счисления. Система счисления определяется по началу числа. Число в восьмеричной системе счисления начинается с ноля. Число в шестнадцатеричной системе начинается с 0x или 0X. Число в десятичной системе начинается с ненулевой десятичной цифры.

Заканчиваться число может суффиксом типа: u или U означает беззнаковый тип, l или L означает длинное целое. Перед числом может стоять необязательный знак плюс или минус. По умолчанию целая константа относится к типу int.

Целая константа = [знак] [префикс_счисления] ненулевая цифра последовательность_цифр [суффикс_целого_типа]
знак = + | -
префикс_счисления = 0 | 0x | 0X
суффикс_целого_типа = [беззнаковый_суффикс] [суффикс_длины]
беззнаковый_суффикс = u | U
суффикс_длины = l | L
восьмеричные_цифры = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
шестнадцатеричные_цифры = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F

(§4.1.1)

Вещественные константы

Вещественные константы служат для представления вещественных чисел одинарной, двойной или повышенной точности. По умолчанию вещественная константа относится к типу double.

Вещественная константа = мантисса [экспонента] [суффикс_вещ_числа]
мантисса = последовательность_цифр . [последовательность_цифр]
экспонента = {e | E} [знак] последовательность_цифр
суффикс_вещ_числа = e | E | i | l

(§4.1.2)

12.1.4.3 Логические константы

Имеются две логические константы true (истина) и false (лжь).
(§4.1.3)

12.1.4.4 Символьные константы

Символьная константа — это символ, заключенный в апострофы. Вместо символа может быть записан его код в восьмеричном или шестна-
дцатеричном виде или экранированная последовательность.

символьная_константа = 'символ'
| \a | \b | \f | \n | \r | \t
| \v | \v | \v | \v | \v | \?

(§4.1.4)

12.1.4.5 Строковые литералы

Последовательность символов, заключенная в двойные кавычки — это строковая константа или строковый литерал. Компилятор автоматиче-
ски добавляет нулевой байт в конец литерала.
(§5.5)

12.1.5 Ключевые слова

Ряд слов в языке Си++ имеет особое значение и не может использо-
ваться в качестве идентификаторов. Такие зарезервированные слова
называются ключевыми. Ниже приведен полный список ключевых слов:

asm auto bad_cast bad_typeof bool
break case catch char class
const const_cast continue default delete
do double dynamic_cast else enum
extern float for friend goto
if inline int long namespace
new operator private protected public
register reinterpret_cast return short signed
12.2 Выражения

Выражения — это последовательность операндов, соединенных знаками операций, которая вычисляет некоторую величину, называемую значением выражения.

(§3.3.4.2)

12.2.1. Простейшие операнды

К простейшим operandам относятся константы, имена и выражения, заключенные в круглые скобки. Имена могут быть уточнены (к какому классу или контексту они относятся) с помощью операции ::.

первичное_выражение = константа
  | this
  | :: имя
  | :: код_функции_операции
  | :: сложное_имя
  | ( выражение )
  | выражение_имя

Константа — это одна из описанных выше числовых, символьных, логических или строковых констант (литералов). Ключевое слово this означает указатель на текущий объект и естественно может использоваться только в нестатических методах классов. Имя (идентификатор) с предшествующими двумя двоеточиями означает глобальное имя, т.е. имя, определенное в наиболее внешнем контексте программы. Это имя может определять либо имя переменной, либо имя константы, либо имя функции. Сложное имя — это идентификатор, квалифицированное именами контекстов или классов, т.е. имя относящееся к некоторому контексту (A::B::C).

выражение_имя = простое_имя
  | сложное_имя

простое_имя = имя
  | код_функции_операции
  | код_функции_преобразования
  | ~ имя_класса
  | код_шаблона

сложное_имя = уточненное_имя [ template ] простое_имя

уточненное_имя = имя_класса_или_контекста :: [ уточненное_имя ]
имя_класса_или_контекста = имя_класса
    | имя_контекста

12.2.2. Порядок составления выражений

Выражения составляются из operandов (или первичных выражений) применением к ним унарных, бинарных или трехарных операций. Синтаксические правила для бинарных выражений упрощены и не отражают приоритета операций.

выражение = присваивание
    | выражение , присваивание

Выражение может состоять из нескольких выражений присваивания, разделенных запятыми. Запятая — это операція "последовательность", т.е. в таком списке присваивания выполняются последовательно.

константное_выражение = условное_выражение

присваивание = условное_выражение
    | бинарное_выражение операция_присваивания присваивание
    | выражение_throw

Константное выражение не изменяется во время выполнение и оно не может содержать присваивания.

Слева от операции присваивания может стоять так называемое l-value — выражение, результат которого сводится к переменной или адресу, по которому можно запомнить значение выражения справа от операции присваивания. Выражение throw описано ниже, в пункте 12.3.3. Оно состоит из оператора throw вместе со стоящим после него выражением.

операция_присваивания =
    = | += | -= | *= | /= | %= | >>= | <<= | &= | |= | ^= |

условное_выражение = бинарное_выражение
    | выражение ? выражение : присваивание

Условное выражение — либо выражение, не включающее в себя присваивания, либо выражение составленное из трех выражений с помощью трехарной условной операции ?:

§3.3.4.2.

бинарное_выражение = указатель_элемент_выражение
    | бинарное_выражение бинарная_операция указатель_элемент_выражение

бианарная_операция =
    * | / | + | - | % | << | >> | < | > | <= | >= | == | != | & | | ^ | | | && | |
Бинарные выражения составляются из обращений по указателю на элемент класса (§ 6.12), преобразований к указанному типу (§ 6.10) и унарных выражений. Унарное выражение — это результат применения унарной (однооперандной) операции к постфиксному выражению.

К унарным операциям относятся операции, знаки которых ставятся перед operandом: плюс (+), минус (-), отрицание (!), битовой дополнение (~), взятие по адресу (*), нахождение адреса (&), префиксное автоувеличение (++) и автоуменьшение (--).

Операция sizeof возвращает количество байтов, необходимое для хранения в памяти объекта указанного типа или значения выражения. Операции new и delete описаны в следующем параграфе.

указатель_элемент_выражение = преобразование_типа
  | указатель_элемент_выражение .* преобразование_типа
  | указатель_элемент_выражение ->* преобразование_типа

преобразование_типа = унарное_выражение
  | ( код_типа ) преобразование_типа

унарное_выражение = постфиксное_выражение
  | ++ преобразование_типа
  | -- преобразование_типа
  | унарная_операция преобразование_типа
  | sizeof унарное_выражение
  | sizeof ( код_типа )
  | выражение_new
  | выражение_delete

унарная_операция = + | - | * | & | ! | ~

Из простейших первичных выражений составляются постфиксные выражения применением операций, которые записываются после (пост) операнда. К ним относятся:
вызовы функций, в которых список аргументов записан в круглых скобках после имени функции;
операция индексирования ([ ]);
операция обращения к элементу класса (., или ->);
постфиксные автоувеличение (++) и автоуменьшение (--).
Операции dynamic_cast, static_cast, reinterpret_cast и const_cast преобразуют указанное в круглых скобках выражение к типу, стоящему после них в угловых скобках (§6.10, §6.11).
Операция typeid возвращает ссылку на объект класса type_info, описывающий тип выражения (§ 6.11).

постфиксное_выражение = первичное_выражение
  | постфиксное_выражение [ выражение ]
  | постфиксное_выражение [ [ список_выражений ] ]
  | простой_тип [ [ список_выражений ] ]
  | постфиксное_выражение . [ template ] [ :: ] выражение_имя
  | постфиксное_выражение -> [ [ template ] [ :: ] выражение_имя
  | постфиксное_выражение . имя_псевдо_деструктора
12.2.3. Операции new и delete

Операции new и delete являются составными частями выражений. Операция new создает новый объект указанного типа и возвращает его адрес. Операция delete удаляет объект и освобождает занимаемую им память. Описание см. §5.6 и §6.9.

С помощью операции new можно создать как один объект, так и массив объектов одного класса. В случае создания массива его размерности указываются после названия типа. Если операция new создала массив, то удалён этот массив должен быть с помощью операции delete [].

Создание объекта с помощью операции new вызывает также выполнение конструктора для этого объекта. Если в new не указан список инициализации (инициализатор_new), либо он пуст (состоит только из открывающейся и закрывающейся скобок), то выполняется стандартный конструктор. Если имеется непустой список инициализации, то выполняется тот конструктор, для которого этот список инициализации является списком аргументов.

При создании массивов, для каждого элемента массива выполняется стандартный конструктор.

При удалении объекта операцией delete, вначале вызывается деструктор этого объекта, а потом освобождается память. При удалении массива с помощью операции delete [] деструктор вызывается для каждого элемента массива.

Операция new может переопределенна для класса. Более того, можно определить несколько операций new с дополнительными аргументами. При вызове эти аргументы указываются сразу после ключевого слова new (расположение). Компилятор автоматически добавляет еще один аргумент операции new – размер необходимой памяти.

выражение_new =
    [ :: ] new [ расположение ] код_типа_для_new [ инициализатор_new ]
    | [ :: ] new [ расположение ] ( код_типа ) [ инициализатор_new ]
расположение = ( список_выражений )
код_типа_для_new = последовательность_спецификаторов_типа декларатор_new
декларатор_new = оператор_указатель декларатор_new
размерности_new = [ выражение ]
размерности_new = [ константное_выражение ]
инициализатор_new = ( [ список_выражений ] )
выражение_delete = [ :: ] delete преобразование_типа
выражение_delete = [ :: ] delete [ ] преобразование_типа

12.3. Программы

Программа состоит из исходных файлов, называемых иногда единицами компиляции. Каждый исходный файл — это последовательность объявления переменных, типов и функций.

объявление = блок_объявлений
объявление = определение_функции
объявление = объявление_шаблона
объявление = явная_генерация_шаблона
объявление = явная_специализация
объявление = спецификация_компактовки
объявление = определение_конк斯特а

В процессе компиляции каждый исходный файл по-отдельности преобразуется в промежуточное представление — объектный файл. Затем все объектные файлы объединяются в одну выполняемую программу с помощью процедуры компоновки. (§7.1)

12.3.1. Функции

Функция может быть объявлена или определена. Объявление функции — это лишь объявление ее имени и типа. Объявлений одной и той же функции может быть сколько угодно в одной программе (разумеется при условии, что все они одинаковы). Функция определяется ровно один раз на всю программу. Определение функции состоит из ее объявления и тела функции — описания действий, которые она выполняет при вызове. Функция может быть методом какого-то класса или принадлежать к какому-либо контексту. Тогда ее имя уточняется именами контекста или класса, стоящими перед ним через два двоеточия.

Если функция — конструктор, после списка аргументов может идти список инициализации. Поскольку инициализация в конструкторе потенци-
ально может возбудить исключительную ситуацию, можно записать try перед инициализацией:

```cpp
Klass::Klass()
    try : Base(), attrib(0)
    { /* тело конструктора */ }
    catch (exception& e)
    { /* обработка исключительной ситуации. */ }
```

определение_функции =
последовательность_описателей описатель [ инициализатор_конструктора ]
    тело_функции
| последовательность_описателей описатель блок_try_функции

tело_функции = составной_оператор
(§3.3.5)

12.3.2. Операторы

Операторы — это единицы действия программы. Каждый оператор либо объявляет тип, переменную или функцию, либо выполняет какие-либо операции. Операторы управления описаны в разделе 3.3.4, объявления — в разделах 3.3.3 и главе 5.

оператор = помеченный_оператор
    | оператор_выражение
    | составной_оператор
    | оператор_выбора
    | оператор_цикла
    | оператор_перехода
    | оператор_объявление
    | блок_try

помеченный_оператор = идентификатор : оператор
    | case константное_выражение : оператор
    | default : оператор

Оператор можно пометить либо простой меткой — именем, либо меткой case, после которой стоит константное выражение, либо меткой default. Две последние употребляются только внутри оператора switch. На помеченный оператор можно передать управление с помощью оператора goto (на простую метку) или оператора switch (на метки case и default).

оператор_выражение = { выражение };

Выражение со стоящей после него точкой с запятой является оператором. Такой оператор просто вычисляет выражение.

составной_оператор = { последовательность_операторов }
последовательность_операторов = оператор 
                                      | последовательность_операторов оператор

Везде, где можно использовать один оператор, можно использовать последовательность операторов, заключенную в фигурные скобки.

оператор_выбора = if ( условие ) оператор
                                      | if ( условие ) оператор else оператор
                                      | switch ( условие ) оператор

условие = выражение
                                      | последовательность_спецификаторов_типа описатель = выражение_присваивания

Оператор выбора позволяет в зависимости от значения условия выполнять один из операторов. Оператор if преобразует значение условия к логическому значению и, если значение истина, выполняет оператор, стоящий после условия, если же значение ложь — оператор, стоящий после else. Часть else необязательна и тогда в случае ложного условия управление переходит на следующий оператор.

Оператор switch в зависимости от значения условия передает управление на один из помеченных меткой case или default операторов. Эти помеченные операторы должны располагаться внутри оператора switch (естественно, в подавляющем большинстве случаев <оператор> оператора switch — это последовательность операторов в фигурных скобках).

Условие может быть либо просто выражением, либо в нем можно определить переменную, действительную на протяжении данного оператора выбора.

оператор_цикла = while ( условие ) оператор
                                      | do оператор while ( выражение )
                                      | for( начало_цикла [ условие ] ; [ выражение ] ) оператор

Операторы цикла позволяют многократно выполнять оператор. Конструкция while выполняет <оператор> до тех пор, пока <условие> истинно. При этом вначале вычисляется условие, а затем, если оно истино, выполняется тело цикла. Конструкция do while вначале выполняет оператор — тело цикла, а затем вычисляет <выражение> и, если значение выражения истинно, повторяет итерацию. Цикл for повторяет выполнение оператора то тех пор, пока условие истинно. При этом перед первой итерацией выполняется <начало_цикла>, которое представляет из себя выражение или объявление переменной, а после каждой очередной итерации выполняется <выражение> — третий элемент в заголовке цикла.

начало_цикла = оператор_выражение
                                      | простое_объявление

оператор_перехода = break ;
                                      | continue ;
operator_объявление = блок_объявлений

Операторы перехода прерывают последовательное выполнение операторов. Оператор goto передает управление на оператор, помеченный данным именем. Оператор break заканчивает выполнение текущего оператора цикла или оператора switch. Оператор continue заканчивает выполнение данной итерации текущего цикла и вызывает переход к следующей итерации. Оператор return вызывает выход из функции и передачу значения выражение в качестве её результата.

12.3.3. Исключительные ситуации

Исключительные ситуации возбуждаются с помощью оператора throw, а обрабатывается с помощью блока try (§8.3, §8.4).

блок_try = try составной_оператор последовательность_обработчиков

блок_try_функции = try [ инициализатор_конструктора ] тело_функции
последовательность_обработчиков =
  обработчик
  | последовательность_обработчиков обработчик
обработчик =
  catch ( объявление_исключительной_ситуации ) составной_оператор

объявление_исключительной_ситуации =
  последовательность_спецификаторов_типа описание
  | последовательность_спецификаторов_типа абстрактный_описатель
  | последовательность_спецификаторов_типа
  | ...

Блок try состоит из составного оператора, после которого записываются несколько операторов catch. Каждый оператор catch определяет какой тип исключительной ситуации он будет обрабатывать (тот, который подходит по типу) и действия, производимые программой в случае возникновения исключительной ситуации – составной оператор. Специальное объявление ... означает, что данный обработчик будет обрабатывать все исключительные ситуации, до него дошедшие.

Можно определить обработку исключительных ситуаций для функции. В этом случае блок try записывается в виде тела функции:

```c
void foo
{
  // тело функции
```
catch (SomeException& e) {
    // обработка исключительной ситуации
}

Подобные конструкции удобно использовать в конструкторах класса в тех случаях, когда инициализатор может возбудить исключительную ситуацию:

class A
{
public:
    A(const B& b);
private:
    B x;
};

A::A(const B& b)
try : x(b)  // возможна исключительная ситуация
{
    // тело конструктора
} catch (SomeException& e) {
    // обработка исключительной ситуации
}

выражение_throw = throw [выражение_присваивания]

Выражение throw — это собственно выражение, которое и возбуждает исключительную ситуацию.

описание_исключительных_ситуаций =
    throw ([ список_кодов_типов ]

список_кодов_типов = код_типа
    | список_кодов_типов код_типа

При определении функции или метода класса желательно задавать, какие исключительные ситуации она может возбудить (и может ли вообще). Для этого в определении функции или метода после списка аргументов может указываться список исключительных ситуаций:

void func() throw (DatabaseException, OverflowException);

Такое описание гарантирует, что функция func может возбудить исключительные прерывания типов DatabaseException и OverflowException, но больше никаких типов исключительных ситуаций не будет возбуждать. Функция, которая не возбуждает никаких исключительных ситуаций, объявляется как
void func() throw();

При отсутствии спецификаций исключительных ситуаций подразумевается, что функция или метод могут возбудить любую исключительную ситуацию.

12.4. Объявления

Объявления определяют имена и связывают их с определенным типом. Объявления состоят из четырех частей: необязательного спецификатора, базового типа, описателя и возможного инициализатора. За исключением определений функций и контекстов, объявления заканчиваются точкой с запятой.

Часто объявления являются и определениями, например, объявление переменной

    char* months[] = {"january", "february", "march"};

является определением, поскольку не только определяет имя months, но и выделяет память под него. В данном объявлении char — это базовый тип, *months[] — описатель, и {"january", "february", "march"} — инициализатор. В объявлении имени

    extern long x;

спецификатором является extern. Данное объявление не является определением, поскольку оно не выделяет память под переменную x.

Все объявления свободных переменных, т.е. переменных не являющихся атрибутами класса, являются определениями за исключением тех, где в качестве спецификатора памяти задан extern. Все атрибуты класса (в том числе статические), являются объявлами, поскольку для них память не выделяется. Для статических атрибутов класса необходимо явно задать их определение вовне объявления класса.

Объявление функции или метода, содержащее тело функции, является ее определением.

12.4.1. Структура объявлений

Объявления могут объявлять функции, переменные, новые типы (классы и перечисления), контексты и ассемблерные вставки.

При объявления переменных допустимо указывать, каким образом под них выделяется память (см §7.3). Имеется 5 спецификаций памяти: auto, register, static, extern и mutable. Ключевое слово auto говорит о том, что переменная — автоматическая. Указание register перед переменной говорит компилятору, что скорость обращения к переменной критична
и её значение лучше держать в быстрой памяти (на регистрах процессора). Описание mutable говорит, что переменную или атрибут класса всегда можно изменять, даже если атрибут является частью неизменяемого объекта. Описатель static определяет статическую переменную, т.е. переменную создающуюся в самом начале выполнения программы (перед выполнением main) и уничтожающуюся только по завершению всей программы. Ключевое слово extern предваряет описание глобальной статической переменной, т.е. определяет имя, но не выделяет память под переменную.

Если ни один из описателей памяти не задан, то способ выделения памяти определяется местоположением определения переменной. Если переменная определена внутри класса, составного оператора, метода или функции, то она будет автоматической, если вовне — глобальной, для которой память выделяется в начале выполнения программы.

При объявлении функций или методов класса перед ними можно употреблять один из трех спецификаторов функции: inline, virtual и explicit.

Ключевое слово explicit перед определением конструктора класса запрещает его использование компилятором для неявного преобразования типов (§6.10.4). Ыными словами, такой конструктор можно вызывать только явно. Спецификатор virtual можно употреблять для нестатических методов классов и деструкторов, конструкторы не могут быть виртуальными. Если метод определен как virtual, то при вызове этого метода через указатель или ссылку на объект, конкретный вызываемый метод определяется только во время выполнения. Виртуальные методы реализуют полиморфизм в языке Си++.

Вызовы методов и функций, объявленных как inline, преобразуются в текстуальную подстановку тела функции на место вызова.

Если в текущем контексте существует неоднозначность между именем типа и другим именем, её можно разрешить с помощью расширения имени типа. Поставив вначале вид типа (class, struct, union, enum или typename) а затем имя типа (для шаблона — с аргументами) мы говорим, что данное имя относится к типу.

блок_объявлений = простое_объявление
    | директива_asm
    | определение_пseudонима_kontekста
    | объявление_using
    | директива_using

простое_объявление =
последовательность_спецификаций { список_инициализированных_описателей }

последовательность_спецификаций =
    спецификация
    | последовательность_спецификаций спецификация

спецификация = спецификация_памяти
    | спецификация_типа
    | спецификация_функции
12.4.2. Простые типы

Простые типы — это встроенные типы языка C++, определенные в программе классы, перечисления и псевдонимы типа.

простой_тип = [ :: ] [ уточненное_имя ] имя_типа  
| char  
| wchar_t  
| bool  
| short  
| int  
| long  
| signed  
| unsigned  
| float  
| double  
| void

имя_типа = имя_класса  
| имя_перечисления  
| имя_псевдонима_типа

имя_перечисления = имя

Таким образом простой тип — это лишь имя типа (быть может уточненное именами классов или контекстов, в которых это имя определено,
например Math::Number). Объявление перечисления состоит из ключевого слова enum, имени перечисления и, в фигурных скобках, списка всех элементов перечисления. Каждый элемент перечисления представляет из себя идентификатор либо идентификатор, которому присвоено значение — целое число.

объявление_перечисления = enum [ имя ] { [ список_элементов_перечисления ] }

список_элементов_перечисления = определение_элемента_перечисления
  | список_элементов_перечисления , определение_элемента_перечисления

определение_элемента_перечисления =
  элемент_перечисления
  | элемент_перечисления = константное_выражение

элемент_перечисления = имя

(§4.1)

12.4.3. Сложные типы

Задав простой тип, можно непосредственно с его помощью определить переменную или функцию, а можно его модифицировать, создав массив (§5.1), указатель на переменную или функцию (§5.4) или указатель на элемент (§6.12). Кроме того, создаваемый объект можно явно инициализировать начальным значением.

Ключевое слово volatile дает указание компилятору, что данный объект может изменять свое значение вне зависимости от работы программы. Например, два последовательных обращения к этой переменной могут дать разные значения. Так, описать переменную, которая соответствует ячейке памяти, в которой компьютер хранит текущее время, можно так:

    extern const volatile clock;

Ключевое слово const предшествует описанию константы, т.е. неизменяемой на протяжении времени жизни данного имени величины.

список_инициализированных_описателей =
  инициализированный_описатель
  | список_инициализированных_описателей инициализированный_описатель

инициализированный_описатель = описатель [ инициализатор ]

описатель = непосредственный_описатель
  | операция_указатель описатель

непосредственный_описатель =
  код_описателя
  | непосредственный_описатель ( объявление_параметров )
определение_указатель =
  * [ последовательность_св_описателей ]
  & | [ :: ] уточненное_имя * [ последовательность_св_описателей ]
последовательность_св_описателей =
  св-описатель
  | [ последовательность_св_описателей ] св-описатель
св-описатель = const
  | volatile
код_описателя = [ :: ] выражение_имя
  | [ :: ] [ уточненное_имя ] имя_типа
код_типа = последовательность_спецификаций_типа [ абстрактный_описатель ]
последовательность_спецификаций_типа =
  спецификация_типа
  | спецификация_типа последовательность_спецификаций_типа
абстрактный_описатель = определение_указатель [ абстрактный_описатель ]
  | непосредственный_абстрактный_описатель
непосредственный_абстрактный_описатель =
  | непосредственный_абстрактный_описатель ( объявление_параметров )
  | [ последовательность_св_описателей ]
  | [ описание_исключительных_ситуаций ]
  | непосредственный_абстрактный_описатель [ константное_выражение ]
  | (абстрактный_описатель )
объявление_параметров = [ список_объявлений_параметров ][ ... ]
  | список_объявлений_параметров , [ ... ]
список_объявлений_параметров =
  объявление_параметра
  | список_объявлений_параметров , объявление_параметра
объявление_параметра =
  последовательность_описателей описатель
  | последовательность_описателей описатель = выражение_присваивания
  | последовательность_описателей [ абстрактный_описатель ]
  | последовательность_описателей
    [ абстрактный_описатель ] = выражение_присваивания
инициализатор = = инициализация
  | ( список_выражений )
инициализация = выражение_присваивания
  | { список_инициализации [ , ]}
12.4.4. Контексты имен

Контексты имен определяют область видимости переменных. Оператор namespace присваивает имя области видимости имен.

определение_konteksta =  
начальное_определение_konteksta  
| расширение_определения_konteksta  
| определение_безымянного_konteksta

начальное_определение_konteksta = namespace имя { тело_konteksta }

расширение_определения_konteksta = 
namespace первоначальное_имя_konteksta { тело_konteksta }

первоначальное_имя_konteksta = имя

определение_безымянного_konteksta = namespace { тело_konteksta }

тело_konteksta = последовательность_объявлений

Определение контекста можно расширить, т.е. добавить в имеющийся контекст дополнительные переменные, функции или классы, используя оператор namespace с именем одного из предыдущих контекстов, например:

namespace A {
    int x;
}

namespace B {

}

namespace A {
    // расширение контекста A
}

Определение безымянного контекста, т.е. контекста без имени, служит для того, чтобы обезопасить все имена в этом контексте от возможных конфликтов с другими именами. В частности, безымянный контекст недоступен из другого исходного файла. Фактически, безымянные контексты служат заменой объявления глобальных переменных как static.

Обращение к именам, определенным в контексте, происходит с помощью уточненного имени, т.е. имени, перед которым стоят имена необ-
ходимых контекстов через два двоеточия. Начальные :: означают "глобальный" контекст.

Для имени контекста можно ввести псевдоним, т.е. альтернативное имя, например:

```cpp
namespace current = A::B::C;
```

определение_ псевдонима_ контекста =
namespace имя = уточненное_ имя_ контекста

уточненное_ имя_ контекста = [ :: ] [ уточненное_ имя ] имя_ контекста

Если какой-либо контекст (или определенные имена из него) постоянно используются в фрагменте программы, его можно объявить текущим с помощью оператора using. У оператора две формы: можно объявить текущим весь контекст с помощью оператора using namespace, а можно использовать только определенные имена из контекста с помощью оператора using имя. Ключевое слово typename перед именем говорит о том, что используемое имя — это имя типа (класса, типа, определенного через typedef или enum).

объявление_using = using [ typename ] [ :: ] уточненное_ имя простое_ имя ;
| using :: простое_ имя ;

директива_using = using namespace [ :: ] [ уточненное_ имя ] имя_ контекста ;

§7.5

12.4.5. Директива asm

Директива asm вставляет в программу ассемблерные (машинные) инструкции. Точное значение этой директивы может быть различно в разных компиляторах, однако везде смысл заключается в том, что строка — аргумент директивы, представляет из себя ассемблерный код, который подставляется компилятором на место директивы asm.

dиректива_asm = asm ( строковый_ литерал ) ;

Пример

asm ("mov r0, r1");

12.4.6. Спецификация компоновки

В случае компоновки программы на языке Си++ с модулями, написанными на других языках программирования, возникает необходимость задать специальные инструкции компоновщику. Такие инструкции задаются с помощью спецификации компоновки в форме:
Например, задать тот факт, что функция `strcat` написана на языке Си и, соответственно, при её вызове надо пользоваться соглашениями языка Си, можно с помощью описания:

```c
extern "C" char* strcat(char* dst, const char* src);
```

Можно заключить сразу несколько объявлений в один блок:

```c
extern "C" {
    char* strcat(char* dst, const char* src);
    int var1;
    extern long var2;
}
```

или даже

```c
extern "C" {
    #include <stdlib.h>
}
```

Отметим разницу между следующими конструкциями:

```c
extern "C" int x1;
extern "C" {
    int x2;
}
extern "C" {
    extern int x3;
}
```

Первое описание — это объявление внешней переменной `x1`, память под нее предполагается выделенной где-то в другом месте. Второе объявление — это определение переменной `x2`, под нее выделяется память. Третье объявление эквивалентно первому — определяется внешняя переменная `x3`, память под которую выделена где-то в другом месте. И все три переменные объявляются как имеющие компоновку языка Си.

### 12.5. Классы

Классы — это новые типы, создаваемые программистом (см §4.2 и главу 6). Структуры и объединения — частные виды классов. Для совместимости с языком Си, одно и то же имя может использоваться в качестве
имени класса и не-класса (например функции). В таком случае, чтобы отличить имя класса, перед ним нужно записать ключевое слово class.

class Complex { . . . }; // определение класса Complex
double Complex(. . .); // функция Complex

class Complex c; // переменная типа Complex

имя_класса = имя
    | код_шаблона

объявление_класса = заголовок_класса { [ определение_элементов ] }

заголовок_класса = вид_класса [ имя ] [: список_наследования ]
    | вид_класса уточненное_имя имя [: список_наследования ]

вид_класса = class
    | struct
    | union

определение_элементов = объявление_элемента [ определение_элементов ]
    | описание_доступа [: определение_элементов ]

объявление_элемента =
    описатель [ = 0 ]
    описатель [ = константное_выражение ]
    [ имя ] : константное_выражение

12.5.1. Производные классы

Класс может наследовать свои свойства от одного или нескольких базовых классов (§6.5). Наследование может отличаться по правам доступа (public, protected или private) и может быть виртуальным (§6.8).

При внешнем наследовании (public) внешняя часть базового класса становится внешней частью порожденного класса, защищенная часть базового класса становится доступной из методов порожденного класса, в внутренняя часть базового класса недоступна для порожденного. В случае внутреннего наследования (private) внешняя и защищенная части базового класса становятся внутренней частью производного класса. Внутренняя часть базового класса остается недоступной для производного класса. При защищенном наследовании (protected) внешняя и защищенная части базового класса становятся защищенной частью производного класса. Внутренняя часть базового класса остается недоступной для производного класса.

Если наследование описано как виртуальное с помощью ключевого слова virtual, то элементы базового класса будут включены в порожденный класс только один раз, независимо от структуры иерархии наследования.
12.5.2. Определение операций

Для класса можно переопределить целый ряд операций, кроме того можно определить операции преобразования к другому типу (§6.4 и §6.10.4). Операции определяются с помощью методов с именем:

```cpp
operator <символ_операции>
```

После того, как операция, например +, определена, её можно записывать как в общепринятом виде k + m или в форме k.operator+(m).

Кроме символов операций, методы в форме operator <тип> определяют преобразования величин данного класса к указанному типу.

функция_преобразования = operator тип_преобразования

tип_преобразования =
    последовательность_спецификаций_типа [ описатель_преобразования ]

описатель_преобразования = операція_указатель [ описатель_преобразования ]

инициализатор_конструктора = : список_инициализации_элементов

список_инициализации_элементов =
    инициализация_элемента
    | инициализация_элемента , список_инициализации_элементов

инициализация_элемента = имя_инициализатора ([ список_выражений ])

имя_инициализатора = [: ::] [ уточненное_имя ] имя_класса
    | имя

функция_операция = operator операция

операция = new | delete | new [ ] | delete [ ]
    | + | - | * | / | % | ^ | & | | | ! | = | < | >
    | += | -= | *= | /= | %= | ^= | &= | |= | <<= | >> | << | >>= | == | ==
    | ! | <= | >= | || | ++ | -- | - | - | -> | -> | ( ) | [ ]
12.6. Шаблоны

Шаблоны служат для задания семейства классов или функций (глава 10). В качестве параметров шаблона могут указываться типы или значения. (Ключевое слово typename эквивалентно слову class в описании параметров шаблона).

Аналогично аргументам функций, для параметров шаблона можно задавать значения по умолчанию, которые будут использоваться компилятором если соответствующий параметр явно не задан.

Допускается задавать в виде шаблона отдельный атрибут или метод класса.

template <class T> class vector
{
    public:

        template<class Cmp> sort(void);
    
};

Конкретный класс или функция генерируется из шаблона автоматически компилятором, по мере того, как шаблон используется для определение объектов классов либо для вызова функций с конкретными аргументами.

vector<int> vec;

Иногда желательно явно сгенерировать функцию или класс из шаблона, тогда записывается объявление шаблона с нужными типами:

template vector<int>;

объявление_шаблона =
    [ export ] template < список_параметров_шаблона > объявление

список_параметров_шаблона = параметр_шаблона
    | список_параметров_шаблона , параметр_шаблона

параметр_шаблона = параметр_тип
    | объявление_параметра

параметр_тип =
    class [ имя ]
    | class [ имя ] = код_типа
    | typename [ имя ]
    | typename [ имя ] = код_типа
    | template < список_параметров_шаблона > class [ имя ]
    | template < список_параметров_шаблона > class [ имя ] = имя_шаблона

код_шаблона = имя_шаблона < список_аргументов_шаблона >
имя_шаблона = имя

список_аргументов_шаблона = аргумент_шаблона
    | список_аргументов_шаблона , аргумент_шаблона

аргумент_шаблона = выражение_присваивания
    | код_типа
    | имя_шаблона

явная_генерация_шаблона = объявление_шаблона

явная_специализация = template <> объявление

12.6.1. Специализация

Специализация шаблонов заключается в возможности уточнять шаблон другими шаблонами для особых видов классов. С помощью специализации мы можем запрограммировать следующее поведение: "Если аргумент шаблона — указатель на байт, использовать один вариант реализации, если аргумент — указатель на любой другой тип, использовать второй вариант реализации, во всех остальных случаях использовать третий вариант реализации".

В качестве примера рассмотрим функцию \texttt{max}. Её вид (определение 1):

\begin{verbatim}
template <class T> T max(T a, T b)
    { return a < b ? b : a; }
\end{verbatim}

не подходит для указателей на байт, где мы бы хотели получать максимум не из адресов строк, а из их значений. Но тем не менее желательно использовать то же самое имя для функции и не заводить "специальную" функцию для строк.

Добиться подобного эффекта можно с помощью специализации шаблона. Для указателей на байт функция \texttt{max} записывается так (определение 2):

\begin{verbatim}
template<> const char* max <const char*>(const char* a, const char* b)
    { return strcmp(a,b) < 0 ? b : a; }
\end{verbatim}

Две подряд открывающиеся и закрывающиеся угловые скобки говорят о том, что данный шаблон — явная специализация, а тип в скобках после названия функции — что для указателя на байт.

Функция \texttt{max} для всех остальных указателей тоже нуждается в уточнении (определение 3):

\begin{verbatim}
template<class T> T max <T*>
    (T a, T b)
    { return *a < *b ? b : a; }
\end{verbatim}
Итак, для всех аргументов шаблона, являющимися указателями на произвольный тип, сравниваются значения, на которые эти указатели указывают. Итак, если функция применяется с аргументами типа `const char*`, используется определение 2, если с другими указателями — определение 3 и с не указателями — определение 1.

Разумеется специализацию можно задавать и для классов, пользуясь тем же синтаксисом:

```cpp
template <class T> class A { ... };
    // общее определение

template <class T> class A<T*> { ... };
    // специализация для указателей

template <> class A<void*> { ... };
    // явная специализация для void*
```

12.7. Препроцессор

Как мы уже говорили, исходный файл вначале обрабатывается препроцессором. Иногда препроцессор называют макро-процессором, поскольку в нем определяются макросы. Директивы препроцессора начинаются со знака `#`, который должен быть первым символом в строке после пробелов.

директива_препроцессора = #define имя определение
    | #define имя ( список_имен ) определение
    | #undef имя
    | #include имя_файла
    | условная_компиляция
    | #pragma строка
    | #line строка
    | #error строка

условная_компиляция = условие текст [ список_альтернатив ] endif

условие = #if константное_выражение
    | #ifdef имя
    | #ifndef имя

список_альтернатив = альтернатива текст
    | список_альтернатив альтернатива текст

альтернатива = #elif константное_выражение
    | #else
12.7.1. Определение макросов

Первая форма директивы define определяет макро-имя. Везде, где в исходном файле встречается это имя, оно будет заменено на его определение. Например, текст:

```c
#define NAME "database"
Connect (NAME);
```

после препроцессора будет заменен на

```c
Connect ("database");
```

По умолчанию имя определяется как пустая строка, т.е. после директивы

```c
#define XYZ
```

макро-имя XYZ считается определенным со значением — пустой строкой.

Вторая форма define определяет макрос — текстовую подстановку с аргументами

```c
#define max (X, Y) ((X > Y) ? X : Y)
```

Текст `max(5, a)` будет заменен на

```c
((5 > a) ? 5 : a)
```

В большинстве случаев использование макросов (как с аргументами так и без) в языке Си++ является признаком непродуманного дизайна. В языке Си макросы были действительно важны и без них тяжело было обойтись. В Си++ при наличие констант и шаблонов макросы не нужны. Макросы осуществляют текстовую подстановку, поэтому они в принципе не могут осуществлять никакого контроля за типами, которая имеется в шаблонах. Кроме того, возможности текстовой подстановки существенно меньше, чем возможности генерации шаблонов.

Директива `#undef` отменяет определение имени, после неё имя перестаёт быть определённым.

У препроцессора есть несколько макро-имен, которые он определяет сам, их называют предопределёнными именами. У разных компиляторов набор этих имен различен, но два определены всегда: `__FILE__` и `__LINE__`. Значением макро-имени `__FILE__` является имя текущего исходного файла, заключённое в кавычки. Значением `__LINE__` — номер текущей строки в файле. Эти макро-имена часто используют для печати отладочной информации.
12.7.2. Условная компиляция

Исходный файл можно компилировать не целиком, а частями, используя директивы условной компиляции:

```c
#ifLEVEL > 3
text1
#elifLEVEL > 1
text2
#else
text3
#endif
```

Предполагается, что LEVEL - это макро-имя, поэтому выражение в директивах #if и #elif можно вычислить во время обработки исходного текста препроцессором.

Итак, если LEVEL больше 3, то компилироваться будет текст1, если LEVEL больше 1, то компилироваться будет текст2, в противном случае компилируется текст3. Блок условной компиляции должен завершаться директивой #endif.

В каком-то смысле директива #if похожа на условный оператор if. Однако в отличие от него, условие - это константа, которая вычисляется на стадии предпроцессора и куски текста, не удовлетворяющие условию, вообще игнорируются.

Директива #elif может быть несколько (либо вообще ни одной), директива #else также может быть опущена.

Директива #ifdef - модификация условия компиляции. Условие считается выполненным, если указанное после её макро-имя определено. Соответственно, для директивы #ifndef условие выполнено, если имя не определено.

12.7.3. Дополнительные директивы препроцессора

Директива #pragma используется для выдачи дополнительных указаний компилятору. Например, не выдавать предупреждений при компиляции, или вставить дополнительную информацию для отладчика. Конкретные возможности директивы #pragma разные у разных компиляторов.

Директива #error выдает сообщение и завершает компиляцию.

```c
#ifndef unix
#error "Программу можно компилировать только для Unix!"
#endif
```

выдаст сообщение и не даст откомпилировать исходный файл, если макро-имя unix не определено.

Директива #line изменяет номер строки и имя файла, которые хранятся в предопределенных макро-именах __LINE__ и __FILE__.

Кроме директив, у препроцессора есть одна операція ###, которая соединяет строки, например `A ### B`.
Список литературы

1. Кнут Д. Искусство программирования для ЭВМ. т.1, Основные алгоритмы. Пер. с англ. М., Мир, 1976.
   Классическое описание основных структур данных и алгоритмов, используемых в программировании. Обязательна для прочтения каждым программистом. Автор в 1997 году выпустил дополненное и переработанное издание книги, которое пока не переведено.

   Классическое описание множества алгоритмов случайных чисел и арифметических вычислений. Автор в 1997 году выпустил дополненное и переработанное издание книги, которое пока не переведено.

   Достаточно подробное описание синтаксиса и семантики языка (исключая шаблоны и некоторые другие сравнительно новые конструкции). Описание ведётся больше с точки зрения структурного программирования, без широкого использования понятий объектно-ориентированного программирования.

   Для тех, кто уже знает основы Си++ книга предлагает методы решения сложных, но тем не менее часто встречающихся задач на языке Си++.

   До появления стандарта ANSI, эта книга была стандартом. При ссылках её часто называют ARM. Дает полное описание синтаксиса и семантики языка, подходит для тех, кто изучает способы построения компилятора Си++.

   Хорошее изложение основ объектно-ориентированного подхода к разработке программного обеспечения.

   Без этой книги невозможно изучать образцы программирования.

   Описание "внутренностей" языка, подробностей его реализации.
   Один из наиболее широко используемых учебников по Си++. Эта книга более проста для восприятия, чем “The C++ Programming Language”.


   Прекрасно изложенные рекомендации по правильному использованию возможностей языка.

   Подробное и обстоятельное описание стандартной библиотеки шаблонов.

   Классическое описание языка, написанное его создателем. Третье издание соответствует принятому стандарту языка. Серьезная книга для серьезных программистов.
   (Русский перевод первого издания этой книги был издан в 1991 году в издательстве "Радио и связь". Перевод второго издания выпущен в 1993 году в Киеве издательством “Диасофт”.)

   Страуструп рассказывает как развивался язык, почему были приняты те или иные решения добавлять (или не добавлять) новые возможности в Си++.

   Пожалуй единственное подробное описание библиотеки потоков ввода-вывода, которое рассказывает не только о том, как ее использовать, но и как ее модифицировать. Несколько устарела, поэтому за описанием современной версии можно обратиться к [12].

Содержание

1. ВВЕДЕНИЕ ................................................................. 3
   1.1. История языка Си++ .............................................. 3
   1.2. Назначение Си++ и области его использования .............. 4
   1.3. Стандарт языка .................................................. 6

2. ОБЪЕКТНАЯ МОДЕЛЬ ЯЗЫКА СИ++ ...................................... 8
   2.1. Понятие объекта ................................................. 10
   2.2. Что такое объектно-ориентированное программирование? .... 12
   2.3. Класс, тип и объект в языке Си++ ............................. 14

3. НАЧАЛЬНЫЕ СВЕДЕНИЯ ОБ ЯЗЫКЕ ...................................... 14
   3.1. Простейшая программа ........................................... 15
   3.2. Компиляция и выполнение программы .......................... 15
   3.2.1. Компиляция и выполнение программ в среде Windows ... 16
   3.2.2. Компиляция и выполнение программ в среде Unix ....... 17
   3.3. Структура программы на языке Си++ .......................... 17
   3.3.1. Запись программ ............................................. 18
   3.3.2. Имена ......................................................... 18
   3.3.3. Переменные и константы .................................. 20
   3.3.4. Операторы .................................................. 33
   3.3.5. Функции ..................................................... 40
   3.4. Стиль записи программ ........................................ 43

4. ТИПЫ ДАННЫХ, КЛАССЫ И ОБЪЕКТЫ ............................... 43
   4.1. Встроенные типы данных ..................................... 43
   4.1.1. Целые числа ................................................ 47
   4.1.2. вещественные числа ....................................... 48
   4.1.3. Логические величины ...................................... 49
   4.1.4. Символы и байты .......................................... 51
   4.1.5. Наборы переменных значений .............................. 52
   4.2. Классы и объекты ............................................... 55
   4.2.1. Подписи методов и необязательные аргументы .......... 58
   4.2.2. Запись классов ............................................. 57

5. ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ ........................................ 57
   5.1. Массивы ......................................................... 59
   5.2. Структуры ...................................................... 60
   5.2.1. Битовая поля ............................................... 60
   5.3. Объединения .................................................... 62
   5.4. Указатели ....................................................... 65
   5.4.1. Адресная арифметика ...................................... 65
   5.4.2. Связь между массивами и указателями ...................... 68
   5.4.3. Бестиповой указатель ...................................... 68
   5.4.4. Нулевой указатель ......................................... 69
   5.5. Строки и литералы ............................................. 72
   5.6. Распределение памяти ........................................ 76
   5.7. Ссылки .......................................................... 77
   5.7.1. Распределение памяти при передаче аргументов функции 77
   5.8. Использование описателя сопа ................................ 81
   5.9. Переменные типов ............................................. 83

6. КЛАССЫ ................................................................. 84
   6.1. Контроль доступа к объекту .................................. 84
   6.1.1. Доступ по чтению и по записи ............................. 87
   6.2. Оптимизация записи простых методов ........................ 88
   6.3. Конструкторы и деструкторы ................................ 89
   6.4. Перепределение операций ..................................... 95
   6.5. Производные классы, наследование ......................... 101
   6.5.1. Виртуальные методы ...................................... 105
   6.5.2. Преобразование базового и производного классов ...... 109
   6.5.3 Виртуальное и защищенное наследование .................. 110
   6.6 Абстрактные классы ........................................... 110
   6.7 Инициализация объектов ......................................... 112
   6.8 Множественное наследование .................................. 117
   6.8.1 Виртуальное наследование ................................ 120
   6.9 Операции new и delete ........................................ 123
   6.10 Преобразования типов ........................................ 124
   6.10.1 Явные преобразования типов ................................ 124
   6.10.2 Стандартные преобразования типов ....................... 125
   6.10.3 Преобразования указателей и ссылок ...................... 126
   6.10.4. Преобразования типов, определенных в программе ... 127
   6.11. Динамическое определение типов .......................... 128
Имеются в продаже:

Нейронные сети. STATISTICA Neural Networks: Пер. с англ.

В книге популярно изложены методы анализа данных, основанные на построении нейросетевых моделей. Рассматриваются различные типы нейронных сетей, алгоритмы их обучения и примеры решения конкретных задач с использованием пакета STATISTICA Neural Networks.

Заключительная глава содержит краткое руководство для пользователей STATISTICA Neural Networks.

Книга представляет интерес в первую очередь для тех, кто начинает изучать нейросетевые методы и хочет научиться грамотно решать практические задачи, используя современные подходы к обработке данных.

Круглов В. В., Борисов В. В. Искусственные нейронные сети. Теория и практика.

Книга посвящена одному из современных направлений в области информатики и вычислительной техники — нейрокомпьютерным технологиям. Достоинством книги является то, что в ней рассмотрены не только вопросы теории искусственных нейронных сетей, но и большое внимание уделено современным программным оболочкам-имитаторам нейронных сетей, а также решению с их помощью практических задач распознавания образов, кластеризации, прогнозирования, оптимизации, построения и использования нейросетевых экспертных систем. Книга содержит обширный справочный материал.

Для научных и инженерно-технических работников в области информатики и вычислительной техники, занимающихся созданием и использованием интеллектуальных систем, а также аспирантов и студентов разных специальностей в области компьютерных технологий.

Роберт Лоренс Бейбер
Программное обеспечение без ошибок. Приемы и секреты создания правильных программ.

Приводятся различные способы проверки правильности программного обеспечения. Все содержащиеся выкладки основываются на строгом математическом аппарате. Однако автор, профессор Университета им. Гете, Германия, акцентирует внимание на практических аспектах решения данной проблемы, избавляя создателей программного обеспечения от долгих и скурупелных доказательств, благодаря которым нужный результат может быть достигнут относительно простыми средствами. Для программистов.

По вопросам приобретения книг обращайтесь

по адресу в Интернет radios@cityline.ru.
а так же через интернет-магазины

www.dessy.ru
www.mistral.ru

Один из наиболее широко используемых учебников по Си++. Эта книга более проста для восприятия, чем "The C++ Programming Language".


Прекрасно изложенные рекомендации по правильному использованию возможностей языка.


Подробное и обстоятельное описание стандартной библиотеки шаблонов.


Классическое описание языка, написанное его создателем. Третье издание соответствует принятому стандарту языка. Серьезная книга для серьезных программистов.

(Русский перевод первого издания этой книги был издан в 1991 году в издательстве "Радио и связь". Перевод второго издания выпущен в 1993 году в Киеве издательством "Диасофт".)


Страуструп рассказывает как развивался язык, почему были приняты те или иные решения добавлять (или не добавлять) новые возможности в Си++.


Пожалуй единственное подробное описание библиотеки потоков ввода-вывода, которое рассказывает не только о том, как ее использовать, но и как ее модифицировать. Несколько устарела, поэтому за описанием современной версии можно обратиться к [12].

Содержание

1. ВВЕДЕНИЕ ............................................................................................................. 3
   1.1. История языка Си++ .................................................................................... 3
   1.2. Назначение Си++ и области его использования ............................................. 4
   1.3. Стандарт языка .............................................................................................. 6

2. ОБЪЕКТНАЯ МОДЕЛЬ ЯЗЫКА СИ++ ..................................................................... 8
   2.1. Понятие объекта ............................................................................................. 8
   2.2. Что такое объектно-ориентированное программирование? ......................... 10
   2.3. Класс, тип и объект в языке Си++ ................................................................. 12

3. НАЧАЛЬНЫЕ СВЕДЕНИЯ ОБ ЯЗЫКЕ .................................................................. 14
   3.1. Простейшая программа ................................................................................. 14
   3.2. Компиляция и выполнение программы ....................................................... 15
      3.2.1. Компиляция и выполнение программы в среде Windows ...................... 15
      3.2.2. Компиляция и выполнение программы в среде Unix ............................... 16
   3.3. Структура программы на языке Си++ ............................................................. 17
      3.3.1. Запись программ ....................................................................................... 17
      3.3.2. Имена ....................................................................................................... 18
      3.3.3. Переменные и константы ...................................................................... 18
      3.3.4. Операторы ............................................................................................... 20
      3.3.5. Функции .................................................................................................. 33
   3.4. Стиль записи программ ................................................................................. 40

4.ТИПЫ ДАННЫХ, КЛАССЫ И ОБЪЕКТЫ ............................................................. 43
   4.1. Встроенные типы данных .............................................................................. 43
      4.1.1. Целые числа ............................................................................................. 43
      4.1.2. вещественные числа ................................................................................. 47
      4.1.3. Логические величины .............................................................................. 48
      4.1.4. Символы и байты .................................................................................... 49
      4.1.5. Наборы перечисляемых значений ............................................................ 51
   4.2. Классы и объекты .......................................................................................... 52
      4.2.1. Подписи методов и необязательные аргументы ..................................... 55
      4.2.2. Запись классов ......................................................................................... 56

5. ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ ..................................................................... 57
   5.1. Массивы ......................................................................................................... 57
   5.2. Структуры ....................................................................................................... 59
      5.2.1. Битовые поля ........................................................................................... 60
   5.3. Объединения ................................................................................................... 60
   5.4. Указатели ......................................................................................................... 62
      5.4.1. Адресная арифметика ............................................................................. 65
      5.4.2. Связь между массивами и указателями ................................................. 67
      5.4.3. Бестиповый указатель ........................................................................... 68
      5.4.4. Нулевой указатель .................................................................................. 68
   5.5. Строки и литералы .......................................................................................... 69
   5.6. Распределение памяти ..................................................................................... 72
   5.7. Ссылки ............................................................................................................. 76
      5.7.1. Распределение памяти при передаче аргументов функции ...................... 77
      5.7.2. Использование указателя ссылки ............................................................ 81
      5.7.3. Конечные типы ......................................................................................... 83

6. КЛАССЫ ............................................................................................................. 84
   6.1. Контроль доступа к объекту ........................................................................... 84
      6.1.1. Доступ по чтению и по записи ................................................................ 87
   6.2. Оптимизация записей простых методов ....................................................... 88
   6.3. Конструкторы и деструкторы ...................................................................... 89
   6.4. Перепрерапдение операций .......................................................................... 95
   6.5. Производные классы, наследование ............................................................. 101
      6.5.1. Виртуальные методы .............................................................................. 105
      6.5.2. Преобразование базового и производного классов ............................... 109
      6.5.3. Внутреннее и защищенное наследование ............................................. 110
   6.6. Абстрактные классы ..................................................................................... 110
   6.7. Инициализация объектов .............................................................................. 112
   6.8. Множественное наследование .................................................................... 117
      6.8.1 Виртуальных наследование ................................................................. 120
   6.9. Операции new и delete ............................................................................... 121
   6.10. Преобразования типов ................................................................................ 123
      6.10.1 Явные преобразования типов ................................................................. 124
      6.10.2 Стандартные преобразования типов ..................................................... 125
      6.10.3 Преобразования указателей и ссылок .................................................... 126
      6.10.4. Преобразования типов, определенных в программе ......................... 127
   6.11. Динамическое определение типов ............................................................. 128