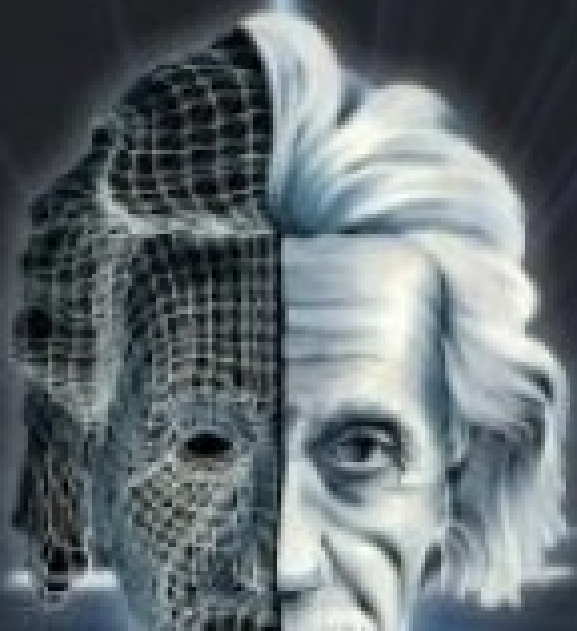
Ю.В. МАРАПУЛЕЦ

Язык C++

ОСНОВЫ ПРОГРАММИРОВАНИЯ

УЧЕБНОЕ ПОСОБИЕ



Министерство науки и высшего образования Российской Федерации
Камчатский государственный университет имени Витуса Беринга

Ю. В. МАРАПУЛЕЦ

ЯЗЫК С++.
ОСНОВЫ
ПРОГРАММИРОВАНИЯ

Издание второе, исправленное и дополненное

Рекомендовано
Дальневосточным региональным учебно-методическим центром
(ДВ РУМЦ) в качестве учебного пособия для студентов направления подготовки
бакалавров 01.03.02 "Прикладная математика и информатика" вузов региона

Петропавловск-Камчатский
2019

УДК 681.306
ББК 32.973-018.2
М25

Рецензент
Г. М. Водинчар,
кандидат физико-математических наук,
доцент кафедры математики и физики КамГУ им. Витуса Беринга

Марапулец Ю. В.

М25 Язык С++. Основы программирования. Издание второе, исправленное и дополненное / Ю. В. Марапулец. — Петропавловск-Камчатский: КамГУ им. Витуса Беринга, 2019. — 158 с.

ISBN 978-5-7968-0675-3

Целью предложенного пособия является систематизированное изложение принципов и приемов программирования в соответствии с рабочими программами дисциплин «Программирование С++» и «Объектно-ориентированное программирование» для студентов направления подготовки бакалавров 01.03.02 «Прикладная математика и информатика». В качестве базового языка использован язык программирования высокого уровня *С++*. В книге подробно рассмотрен синтаксис языка *С++* стандарта *ANSI C++*. Особое внимание уделено технологии построения объектно-ориентированных программных продуктов, потоковым классам, описанию функций основных библиотек. Приведен материал для дополнительного углубленного изучения программирования на языке *С++*. Учебное пособие предназначено для студентов, а также для тех, кто самостоятельно изучает язык *С++*. Во втором издании приведены дополнительные знания по синтаксису языка С++, исправлены ошибки и опечатки.

Издание второе, исправленное и дополненное рекомендовано учебно-методическим советом ФГБОУ ВО «Камчатский государственный университет имени Витуса Беринга» в качестве учебного пособия для студентов, обучающихся по направлению подготовки «Прикладная математика и информатика».

Рекомендовано Дальневосточным региональным учебно-методическим центром (ДВ РУМЦ) в качестве учебного пособия для студентов направления подготовки бакалавров 01.03.02 «Прикладная математика и информатика» вузов региона.

УДК 681.306
ББК 32.973-018.2

ISBN 978-5-7968-0675-3

Введение

Необходимость человечества в вычислениях возникла очень давно — наука о вычислениях является одной из самых древнейших. Потребность в вычислениях всегда была неразрывно связана с практической деятельностью человека. Наряду с непрерывным ростом потребностей в вычислениях и с развитием новых методов математики развивалась вычислительная техника, т. е. совокупность средств, предназначенных для снижения трудоемкости, для ускорения и автоматизации вычислительных процессов. Особенно бурными темпами она стала развиваться в середине XX столетия, что обусловливалось прежде всего проблемами освоения космоса и потребностями важнейших современных отраслей науки и техники, таких как ядерная физика, реактивная и ракетная техника и т. д. Появилась настоятельная необходимость создания вычислительных машин, способных за сравнительно небольшие промежутки времени выполнять большой объем вычислительной работы. Практическое построение таких машин стало возможным благодаря достаточно высокому уровню развития, достигнутому в других отраслях науки и техники, например в радиоэлектронике.

Появление современных вычислительных машин оказало глубокое обратное воздействие на многие отрасли науки и техники, открыв для них новые потенциалы для исследований. Возможность быстрого проведения большого количества вычислений позволяет ставить и решать задачи во всей их полноте, отказавшись от многих допущений и упрощений, и тем самым глубже вникать в сущность изучаемых явлений и объектов. Это же обстоятельство в ряде случаев привело к возникновению совершенно новых, более эффективных методов исследований. Для примера можно привести такой метод, как моделирование процессов. Применение моделирования в ряде случаев позволяет отказаться от дорогостоящих, требующих больших затрат времени экспериментов, которые к тому же не всегда приводят к положительным результатам.

Таким образом, наличие современных вычислительных машин часто позволяет совершенно по новому подойти и к постановке задачи, и к выбору метода ее решения. Это обстоятельство, в частности, привело к широкой компьютеризации даже таких отраслей науки и техники, которые раньше считались весьма далекими от вычислительной математики — экономики, биологии, лингвистики, музыки и т. д.

Вычислительная система в целом представляет собой сложный комплекс, состоящий из разнообразных технических средств, с одной стороны, и соответствующего программного обеспечения — с другой. При этом как технические, так и программные средства имеют обычно модульную структуру, позволяющую наращивать их в зависимости от назначения и условий эксплуатации системы.

Программное обеспечение персональных вычислительных машин (ПЭВМ) содержит необходимые программы и документацию на них, предназначенные для обеспечения эффективного использования вычислительных средств, а также облегчения их эксплуатации и снижения трудоемкости подготовительной работы при решении различных задач. При этом обеспечивается минимальное вмешательство программиста в вычислительный процесс.

Все программное обеспечение ПЭВМ можно разделить на три составные части:

1. ***Системные программы*** (операционная система, драйверы, программы-оболочки, вспомогательные программы-утилиты, программы управления сетью и т. д.).

2. ***Инструментальные системы*** (среды разработки для программистов).

3. ***Прикладные программы*** (графические и текстовые редакторы, системы управления базами данных, обработки табличных данных, системы автоматизированного проектирования, системы деловой и научной графики, бухгалтерские программы и т. д.).

В предлагаемом учебном пособии рассматриваются принципы программирования на языке высокого уровня *C++*, с помощью которого можно создавать прикладные и системные программы различной сложности. Язык *C++* выбран не случайно. В настоящее время он стал одним из основных языков программирования, используемых профессиональными программистами. Все операционные системы семейства *UNIX* и программное обеспечение к ним написаны на этом языке. Да это и естественно, ведь язык Си (предок языка *C++*) был специально создан для разработки *UNIX*. Все современные системные и прикладные программные продукты (включая *Windows* любой версии), разработанные компанией *Microsoft* — мирового лидера в изготовлении программной продукции, также разработаны на языках Си и *C++*.

В процессе изучения языка студенты должны научиться создавать алгоритмы решения сложных математических и прикладных задач, изучить общие принципы структурного и объектно-ориентированного программирования на языке *C++*, научиться разрабатывать и отлаживать программы различной сложности в средах разработки *Borland C/C++* и *Microsoft Visual C++*.

Глава 1
Общий синтаксис языка *C++*

§ 1.1. Этапы разработки программы на ПЭВМ

Процесс разработки программы на ПЭВМ состоит из следующих основных этапов:
- формулировка и постановка задачи;
- выбор метода решения задачи;
- разработка алгоритма решения задачи;
- составление программы;
- ввод программы;
- трансляция;
- исправление синтаксических ошибок;
- выполнение программы;
- анализ и документирование полученных результатов.

На стадии формулировки и постановки задачи определяются состав и характер исходных данных; выбирается общий подход к решению задачи; осуществляется разбиение задачи на составные элементы; определяется последовательность решения элементов задачи; выбирается метод решения задачи.

При выборе метода решения задачи следует учитывать множество факторов, влияющих на эффективность его применения. Наиболее важным среди них является получение требуемой точности решения задачи, а также необходимого времени для ее решения. Кроме того, нужно предусмотреть возможность реализации метода в конструктивных особенностях применяемой ПЭВМ (объем памяти, быстродействие, тип и объем памяти видеокарты и др.), а также его универсальность (возможность применения на других ПЭВМ).

Суммарное время решения задачи на ПЭВМ складывается из времени на подготовку задачи к решению и машинного времени (времени выполнения задачи на ПЭВМ). Если задача должна быть решена в кратчайший срок, то ради сокращения общего расхода времени можно пойти на некоторое увеличение машинного времени, т. е. использовать менее оптимальный метод, требующий большего количества вычислительных операций, но зато не требующий длительного выделения времени на программирование. Если же решение задачи не является срочным, выбирается метод, наиболее оптимально расходующий машинное время, но требующий наибольшего времени программирования. Для сокращения машинного времени решения задачи необходимо, чтобы выбранный метод обеспечивал выполнение наименьшего числа операций, что уменьшает число команд в программе и ускоряет ее выполнение [5].

За математической постановкой задачи и выбором метода ее решения следует реализация выбранного метода на ПЭВМ. При этом возникает необходимость в полном и однозначном описании требуемого вычислительного процесса. Такое описание задается ***алгоритмом*** решения задачи. Поиск, разработка и описание алгоритма решения задачи называется ***алгоритмизацией***.

Алгоритм — строгая и четкая конечная система правил, которая определяет последовательность над некоторыми объектами и после конечного числа шагов приводит к достижению поставленной цели**.**

Любому алгоритму присущи следующие основные свойства:

1. ***Финитность*** (конечность) — алгоритм всегда должен заканчиваться после конечного числа шагов. Если у процедуры есть все признаки алгоритма, кроме финитности, то это алгоритмический метод.
2. ***Определенность*** — каждый шаг должен быть строго определен.
3. ***Ввод*** — каждый алгоритм имеет некоторое, может быть равное 0, число входных данных.

4. ***Вывод*** — каждый алгоритм имеет одну или несколько выводных величин, т. е. величин, имеющих определенное отношение к входным данным.

5. ***Эффективность*** — все операции, которые необходимо произвести в алгоритме, должны быть достаточно простыми, чтобы их в принципе можно было выполнить точно и за конечный момент времени.

6. ***Массовость*** — алгоритм должен быть применим к любым допустимым значениям начальных данных.

Существуют три формы записи алгоритма:

1) словесная;

2) графическая;

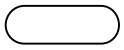
3) на алгоритмическом языке.

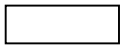
Словесная форма использует для описания алгоритма естественный язык, например язык математики. Запись алгоритма на таком языке должна быть однозначно воспринята человеком в виде последовательности действий, приводящих к получению искомого результата. В качестве примера рассмотрим описание на естественном языке алгоритма Евклида [4]. Сущность задачи заключается в следующем. Даны два положительных числа m и n. Требуется найти их наибольший общий делитель. Алгоритм сводится к последовательному выполнению следующих действий:

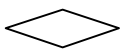
1. Сравнить числа m и n. Если $m \geq n$, то $x = m$, $y = n$. В противном случае $x = n$, $y = m$.
2. Разделить x на y, принять остаток от деления равным r.
3. Если $r = 0$, то принять наибольший общий делитель $d = y$ и закончить вычисления; в противном случае выполнить действие 4.
4. Принять $x = y$, $y = r$; выполнить действие 2.

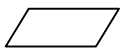
Данный алгоритм выполняется многократно до нахождения наибольшего общего делителя заданных чисел m и n.

Словесное описание алгоритма неприемлемо для ввода в ПЭВМ. Для этого необходимо изложить алгоритм на машинном языке таким образом, чтобы с его помощью происходило автоматическое управление работой ПЭВМ в процессе решения данной задачи. ***Алгоритм, записанный в форме, воспринимаемой ПЭВМ, представляет собой*** **программу** ***решения задачи.*** Однако непосредственный перевод словесной формы алгоритма в программу несет в себе определенные сложности. При таком переходе теряется связь между отдельными частями алгоритма, что может привести к ошибкам в программировании. Поэтому, как промежуточный этап между словесной формой алгоритма и программой, разработана графическая форма алгоритма. Этапы решения задачи при графической форме представляются в виде структурной схемы с отдельными блоками, которые изображаются соответствующими символами. Основные элементы:


Начало (конец), останов, вход и выход в подпрограммах


Выполнение операций, в результате которых изменяется значение данных

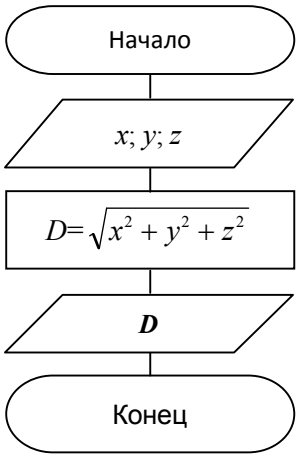

Выбор направления выполнения алгоритма в зависимости от некоторых условий


Ввод, вывод данных, отображение результатов

Существуют три основные комбинации базовых структур алгоритма. Рассмотрим их:

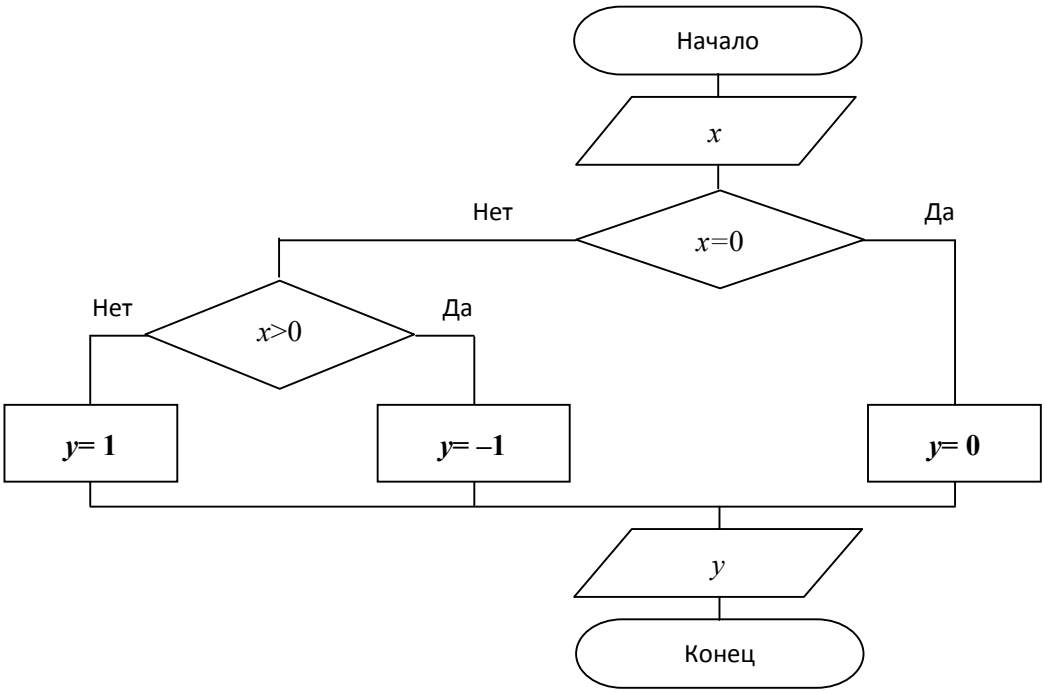
1. ***Линейная*** — простая последовательность действий, которые выполняются только один раз в порядке их следования. Например [5], необходимо вычислить выражение:

$$D = \sqrt{x^2 + y^2 + z^2}\ .$$



2. ***Ветвящаяся*** — содержащая блок проверки некоторого логического условия. Например [5], необходимо вычислить следующее выражение:

$$y = \begin{cases} 1; & x < 0, \\ 0; & x = 0, \\ -1; & x > 0. \end{cases}$$



3. ***Циклическая*** — содержащая некоторую последовательность действий, выполняемых многократно. Разделяется на итерационную и с заданным числом шагов. С левой стороны листа приводится пример с итерацией, в котором необходимо вычислить приближенным методом выражение

$$x = \sqrt{a} \ .$$

Справа — пример с заданным числом шагов, в котором необходимо вычислить выражение

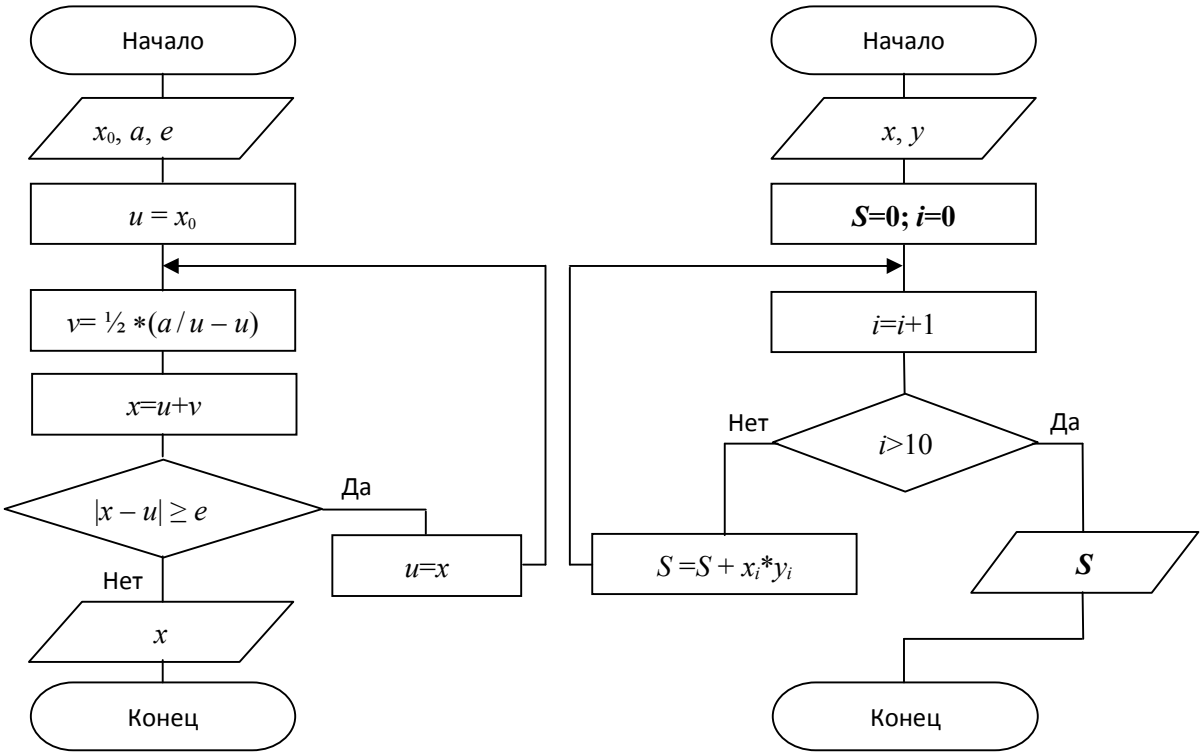
$$S = \sum_{i=1}^{10} x_i * y_i \ .$$

Итерация. Вычислим приближенным методом выражение $x=\sqrt{a}$;

$x_n = x_{n-1} + 1/2 * (a / x_{n-1} - x_{n-1})$,

где n = 1, …, e; $e > (x_n - x_{n-1})$.

Заданное число шагов. *Вычислим* $S = \sum_{i=1}^{10} x_i * y_i$



Следующим этапом подготовки задач к решению на ПЭВМ является программирование. На этом этапе составляется и вводится в память ПЭВМ программа, состоящая из отдельных команд и определяющая последовательность выполняемых операций. Система команд, используемая в данной ПЭВМ, вместе с принятыми в ней способами кодирования и адресации образуют ***машинный*** язык данной ПЭВМ.

Вопросы программирования для ПЭВМ будут подробно освещены в последующих главах. В этой главе будут отмечены лишь наиболее характерные особенности данного этапа подготовки задач к решению на ПЭВМ.

Хочется отметить, что программы, составленные на машинном языке, не наглядны, требуют большой и утомительной работы для их создания, затрудняют обнаружение и устранение неизбежных ошибок, допущенных в процессе написания программ. Поэтому в настоящее время машинные языки практически не применяются при программировании задач, за ними сохраняются лишь функции внутреннего языка ПЭВМ. Для целей программирования используются различные формальные языки, не зависящие от конкретной ПЭВМ, а целиком ориентированные на особенности решаемых задач. Такие языки отражают в большей степени алгоритм решения задачи, поэтому их принято называть ***алгоритмическими языками***.

Алгоритмический язык существенно отличается от внутреннего машинного языка ПЭВМ, поэтому программа, составленная на алгоритмическом языке, не может быть непосредственно воспринята ПЭВМ. Для перевода программы с алгоритмического языка на машинный разработаны специальные программы — ***трансляторы***.

Для характеристики степени близости языка программирования к машинному языку, используется понятие ***уровня языка***. За начало отсчета принимается машинный язык, уровень которого равен 0. Естественный язык человека рассматривается как язык наивысшего уровня. Остальные языки, применяемые в настоящее время, занимают промежуточные уровни.

Обычно процессы трансляции и исполнения программы разделены во времени. Сначала вся программа транслируется, затем исполняется. Трансляторы, работающие в данном режиме, называются ***компиляторами***.

Трансляторы, в которых преобразование и выполнение каждого оператора исходного языка осуществляется последовательно, называют ***интерпретаторами***. В данных трансляторах отдельные участки исходной программы выполняются сразу после трансляции. Недостатком интерпретаторов является неэффективное использование машинного времени.

Важным этапом в процессе подготовки и решения задачи на ПЭВМ является ***отладка программы***, в ходе которой обнаруживаются и устраняются ошибки, допущенные в процессе программирования. Отладка программы состоит из ***синтаксического контроля***, выполняемого в процессе трансляции, ***автономной отладки*** и в конце — ***комплексной отладки программы*** на машинном языке.

В процессе синтаксического контроля устраняются формальные ошибки, допущенные при записи алгоритма на языке программирования. Автономная отладка программы в современных языках программирования осуществляется специальными отладочными программами, которые дают программисту детальную информацию о работе отлаживаемой программы. После устранения ошибок, обнаруженных при автономной отладке, проводят комплексную отладку всей программы. При этом выполняют для нескольких примеров контрольные расчеты, при которых все части отлаживаемой программы работают совместно. В процессе комплексной отладки не только окончательно устраняются допущенные ошибки, но совершенствуется программа в целом.

На завершающем этапе производится непосредственное ***выполнение программы*** на ПЭВМ. После выполнения программы производится ***анализ и документирование полученных результатов***. Полученные результаты могут выводиться на дисплей, записываться в файл, печататься на принтере. Форма представления результатов также может быть самая различная: как в виде цифровых данных, таблиц, строк символов и т. п., так и в графическом виде.

§ 1.2. Базовые элементы языка *C++*

При использовании первых вычислительных машин программисты были вынуждены писать свои команды в двоичном коде на машинном языке конкретной вычислительной машины. По мере развития вычислительной техники появились языки программирования высокого уровня, такие как Алгол, Си, Фортран, Бейсик, Паскаль и т. д., существенно облегчившие работу программистов. Представляет интерес история появления языка Си как наиболее популярного в тот период времени. Этот язык был разработан Б. Керниганом и Д. Ритчи [2] в начале 70-х гг. как инструментальное средство для реализации операционной системы *UNIX* на ЭВМ *PDP*-11, однако его популярность быстро переросла рамки конкретной ЭВМ, конкретной операционной системы и конкретных задач системного программирования. В некотором смысле язык Си можно назвать самым универсальным языком программирования, так как в нем наряду с набором средств, присущих языкам программирования высокого уровня (структурность, модульность, определяемые типы данных), включены также средства программирования почти на уровне ассемблера (использование указателей, побитовые операции, операции сдвига). Другим несомненным достоинством языка Си являются подключаемые библиотеки, благодаря которым данный язык программирования можно использовать практически в любых, даже самых новейших операционных системах. Благодаря этому язык Си и в настоящее время остался основным языком программирования для специализированных сигнальных процессоров управления фирм *Motorola*, *Texas Instruments* и т. д. Языки высокого уровня позволяли создавать программы, универсальные для различного типа ЭВМ. Однако требования к программам росли, времени для их написания отводилось все меньше, программистам необходимо было сосредоточиться на сложных алгоритмах, их эффективной реализации, не отвлекаясь на внутреннюю структуру компьютера. Новый подход к программированию появился в виде объектно-ориентированного программирования.

Язык *Simula*, использовавшийся в 70–80-е гг. в норвежских вооруженных силах, явился одним из первых языков, основанных на понятии «класс». Примерно в это же время окончательно утвердился стандарт языка программирования Си. В 1978 г. Б. Страуструпом, работающим в лаборатории

Bell были сделаны первые попытки объединить достоинства языков *Simula* и Си: так появился язык «*C* с классами» [11]. Лишь спустя несколько лет он стал языком *C++*, используемым в настоящее время. Следует отметить, что язык *C++* не требует обязательного применения объектов в программах. Это позволяет модернизировать ранее написанные и создавать новые программы, пользуясь синтаксисом языка *C*.

Подготовка исполняемой программы. Программа на языке *C++* проходит следующие этапы подготовки к исполнению на ПЭВМ. Первоначально исходный текст программы на языке *C++* подготавливается в файле с расширением *.CPP*. Далее после устранения синтаксических ошибок происходит компиляция. Создается объектный файл с расширением *.OBJ*. На следующем этапе происходит компоновка (дополнение объектного файла необходимыми библиотечными функциями). В результате создается исполняемый модуль программы с расширением *.EXE*. Структурная схема подготовки исполняемой программы приведена на рис.1.1 [6].

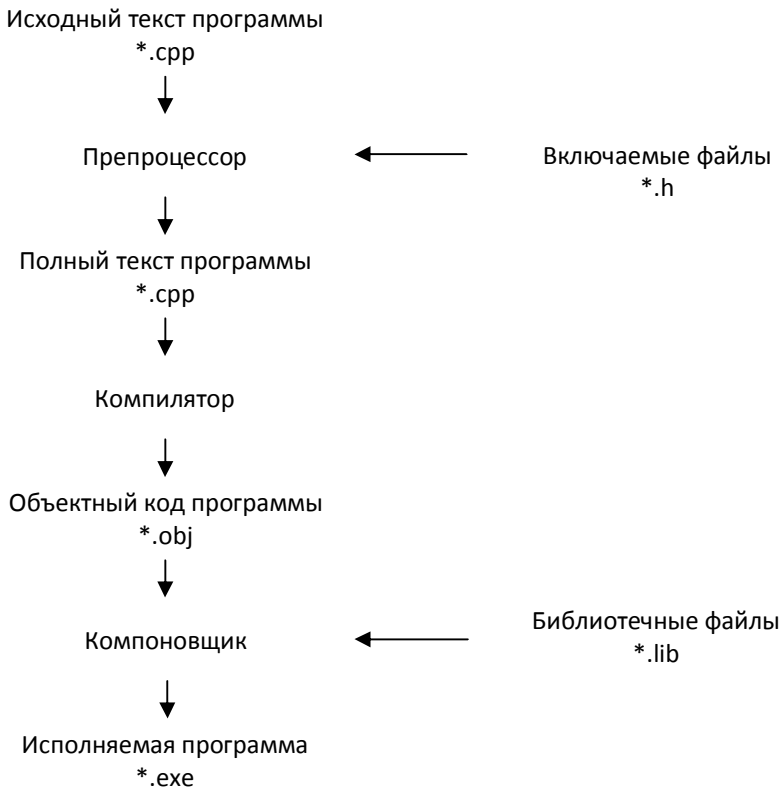


Рис. 1.1. Структурная схема подготовки исполняемой программы

Разделители и комментарии. В качестве ***разделителей*** в языке *C++* используются пробелы, символы табуляции, символы перевода на новую строку и перевода страницы. Строки заключаются в двойные кавычки ". В случае перехода на новую строку внутри текста, заключенного в кавычки используется символ \.

Пример: char test[ ] = "Начало \ Конец"; есть то же, что и char test[ ] ="Начало Конец";

Часть текста программы, начиная с символов /* и до */ включительно, рассматривается как ***комментарий*** и игнорируется. Синтаксис языка *C++*, поддерживающий стандарт *ANSI C++*, по умолчанию не разрешает употребление вложенных комментариев. *C++* позволяет задавать комментарии, распространяющиеся на одну строку. В данном случае часть строки, начиная от символов // и до конца строки, также рассматривается как комментарий.

```
Пример:     int i;          /*длинный
                            комментарий*/
            char c;         //короткий комментарий
```

Идентификаторы. Идентификатор представляет собой последовательность букв и цифр, первой в которой является буква. Символ _ (подчеркивание) рассматривается как буква. По умолчанию значащими являются первые 32 символа. Маленькие и большие буквы различаются. Таким образом, идентификаторы *name* и *Name* являются различными. Идентификаторы типа *pascal* всегда переводятся на верхний регистр. В качестве идентификаторов не могут использоваться следующие служебные слова языка *C++* [6]:

- ***asm*** — используется для встраивания команд на языке ассемблера в программу на *C++*;
- ***auto*** — класс памяти, используемый для объявления локальных переменных;
- ***break*** — оператор, применяемый для выхода из циклов *for* и *while*, а также переключателя *switch*;
- ***case*** — метка, используемая в операторе *switch* для определения одного из выбираемых вариантов;
- ***catch*** — является частью механизма обработки исключений *try*;
- ***char*** — спецификатор типа;
- ***class***— определяет имя класса объекта;
- ***const*** — определяет переменную, значение которой не должно изменяться;
- ***continue*** — оператор, используемый для игнорирования оставшихся команд в теле цикла и перехода к очередной проверке условия выполнения тела цикла;
- ***default*** — метка, используемая в операторе *switch* для указания инструкций, если проверяемое значение не совпадает со значениями в метках *case*;
- ***delete*** — используется для удаления динамических объектов, память под которые была предварительно выделена с помощью оператора *new*;
- ***do*** — оператор цикла;
- ***double*** — спецификатор типа;
- ***else*** — оператор выбора в конструкции *if-else*;
- ***enum*** — спецификатор перечисляемых типов данных;
- ***explicit*** — позволяет предотвратить автоматическое создание объекта определенного класса конструктором;
- ***extern*** — модификатор, описывающий внешнюю переменную;
- ***far*** — указатель, описываемый как «дальний», используется только для 16-битных приложений;
- ***float*** — спецификатор типа;
- ***for*** — оператор цикла;
- ***friend*** — используется для объявления функций или классов, дружественных объявленному классу;
- ***goto*** — оператор перехода в конкретную точку программы;
- ***huge*** — указатель, описываемый как «огромный», используется только для 16-битных приложений;
- ***if***— оператор проверки условия;
- ***inline*** — используется для описания встраиваемых функций;
- ***int*** — спецификатор типа;
- ***long*** — спецификатор типа;
- ***mutable*** — применяется для обеспечения возможности изменения отдельных данных-членов класса в том случае, если этот класс объявлен как *const*;
- ***namespace*** — определяет пространство имен функций, классов и переменных, находящихся в отдельной области видимости;
- ***near*** — указатель, описываемый как «ближний», используется только для 16-битных приложений;
- ***new*** — позволяет выделить память под объект, создаваемый в процессе выполнения программы;

• ***operator*** — используется для описания функций операторов, переопределяющих встроенные операторы языка (+, –, * и т. д.);

• ***pascal*** — используется при необходимости;

• ***private*** — используется для объявления закрытых членов класса;

• ***protected*** — используется для объявления защищенных членов класса;

• ***public*** — используется для объявления открытых членов класса;

• ***register*** — класс памяти, используемый для объявления локальных переменных, размещаемых в наиболее быстродействующей регистровой памяти;

• ***return*** — оператор, возвращающий управление из тела функции;

• ***short*** — спецификатор типа;

• ***signed*** — спецификатор типа;

• ***sizeof*** — знак операции фазы трансляции;

• ***static*** — класс памяти;

• ***struct*** — спецификатор, объявляющий объект, подобный классу, тип доступа к элементам которого по умолчанию — *public*;

• ***switch*** — оператор-переключатель;

• ***template*** — используется для описания шаблонов или параметризованных функций и классов;

• ***this*** — задает указатель на объект, вызвавший функцию-член класса;

• ***throw*** — применяется для обработки исключений путем их передачи следующему обработчику исключений;

• ***typedef***— оператор, используемый для определения новых типов данных;

• ***typeid*** — позволяет определять тип операнда в процессе выполнения программы, обычно применяется к указателям на объекты производных классов в виртуальных функциях;

• ***try*** — используется для обработки исключений;

• ***union*** — спецификатор, объявляющий объект, подобный классу, который позволяет содержать водной и той же области данных взаимоисключающие переменные разных типов;

• ***unsigned*** — спецификатор типа;

• ***using*** — позволяет упростить доступ к членам пространства имен;

• ***virtual*** — используется для объявления виртуальных функций, реализующих свойство полиморфизма родственных классов или виртуальных базовых классов;

• ***void*** — спецификатор типа, использующийся для объявления функций, которые не возвращают значений;

• ***volatile*** — используется в объявлении переменной для того, чтобы указать оператору, что ее значение может изменяться извне, например системными часами или операционной системой;

• ***while*** — оператор цикла.

Константы. *C++* поддерживает четыре типа констант: целые, вещественные, символьные (строковые) и перечислимые.

Примеры:
```
char x;
int y;
double z;
```

Целые константы

1. *Десятичные*. Цифры 0–9, при этом первой цифрой не может быть 0. Если значение превышает наибольшее машинное целое со знаком, то оно представляется как длинное целое.

Примеры: 2, 123, 1945.

2. *Восьмеричные*. Цифры 0–7, начинаются с нуля.

Примеры: $011=9_{10}$, $0113=75_{10}$.

3. *Шестнадцатеричные*. Цифры 0–9, буквы *A–F* (*a–f*) для значений 10–15, начинаются с 0X (0x).

Примеры: $0x10=16_{10}$, $0XF=15_{10}$.

Длинные целые константы явно определяются латинской буквой *l* или *L*, стоящей после константы. Суффикс *U* (или *u*) соответствует типу *unsigned* (беззнаковый).

Примеры:

- длинная десятичная $764l = 764_{10}$, $342L = 342_{10}$;
- длинная восьмеричная $011L = 9_{10}$, $0113l = 75_{10}$;
- длинная шестнадцатеричная $0x10l = 16_{10}$, $0XFL = 15_{10}$.

Вещественные константы представляются как числа с фиксированной или плавающей точкой. По умолчанию имеют тип *double*. Суффикс *f*(*F*) задает тип *float*, суффикс *l*(*L*) — тип *long double*. Вещественные константы состоят из следующих частей:

– целой части — последовательности цифр;
– десятичной точки;
– дробной части — последовательности цифр или символа экспоненты — *e*(*E*);
– экспоненты в виде целой константы (может быть со знаком).

Любая часть (но не обе сразу) из следующих пар может быть опущена:

– целая или дробная часть;
– десятичная точка или символ *e* (*E*) и экспонента в виде целой константы.

Примеры: $4.125 = 4.125_{10}$;
$0.34 = 0.34_{10}$;
$456 = 456_{10}$;
$1.5e2 = 150_{10}$;
$12E{-}3 = 0.012_{10}$.

Символьные константы состоят из одного символа, заключенного в апострофы. Символьные константы считаются данными типа *char*.

Примеры:'D', '2', '&', 'g'.

Символ \ (*backslash*) используется для определения *ESC*-последовательностей, которые служат для визуального представления неграфических символов. Этот же символ с восьмеричным или шестнадцатеричным числом представляет собой *ASCII*-символ или управляющий код с соответствующим значением.

Пример: '\03' для Ctrl-C.

Основные *ESC*-последовательности представлены в табл. 1.1.

Таблица 1.1

Наименование константы	Обозначение	Код
Новая строка (перевод строки)	HL (LF)	'\n'
Горизонтальная табуляция	HT	'\t'
Вертикальная табуляция	VT	'\v'
Возврат на шаг	BS	'\b'
Возврат каретки	CR	'\r'
Перевод формата	FF	'\f'
Обратная косая	\	'\\'
Апостроф	'	'\' '
Кавычки	"	'\"'
Нулевой символ (пусто)	NULL	'\0'

Стандарт *ANSI C++* также поддерживает двухсимвольные константы: 'ab', '\n\t' и т. д. Такие константы представляются как 16-битное целое, где младший байт — первый символ, старший байт — второй символ (это представление непереносимо для *C*-компиляторов).

Строковые константы представляются последовательностью символов, заключенных в двойные кавычки "". Строковые константы — это массивы символов типа *char*.

Пример: "Строковая константа", "D".

Размер основных типов данных приведен в табл. 1.2 [1].

Таблица 1.2

Тип	Описание	Область измерения
bool	Булево (логическое) значение	*true* или *false*
char	1-байтовое целое, используемое для хранения символа	От −128 до 127 или от 0 до 255
unsigned char	1-байтовое беззнаковое целое	От 0 до 255
signed char	1-байтовое целое	От −128 до 127
int	Целое (2 или 4 байта)	*short* или *long*
unsigned int	Беззнаковое целое	От 0 до 65 535
short	2-байтовое целое	От −32 768 до 32 767
unsigned short	2-байтовое беззнаковое целое	От 0 до 65 535
long	4-байтовое целое	От −2 147 483 648 до 2 147 483 647
unsigned long	4-байтовое беззнаковое целое	От 0 до 4 294 967 295
float	Действительное число (4 байта)	От $-3{,}4*10^{+38}$ до $-3{,}4*10^{-38}$ и от $3{,}4*10^{-38}$ до $3{,}4*10^{+38}$
double	Действительное число с двойной точностью (8 байт)	От $-1{,}7*10^{+308}$ до $-1{,}7*10^{-308}$ и от $1{,}7*10^{-308}$ до $1{,}7*10^{+308}$
long double	Длинное действительное число с двойной точностью (10 байт)	От $-3{,}4*10^{+4932}$ до $-3{,}4*10^{-4932}$ и от $3{,}4*10^{-4932}$ до $3{,}4*10^{+4932}$
wchar_t	2-байтовый символ, используемый в международной кодировке (Unicode)	От 0 до 65 535

Перечислимые константы. При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть неважны). Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант.

Формат: enum [имя_типа] { список_констант };

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается значение на единицу большее, чем предыдущей:

```
enum Err {ERR_READ, ERR_WRITE, ERR_CONVERT};
Err error;
...
switch (error){
    case ERR_READ:      /* операторы */break;
    case ERR_WRITE:     /* операторы */break;
    case ERR_CONVERT:   /*операторы */ break;}
```

Константам *ERR_READ*, *ERR_WRITE*, *ERR_CONVERT* присваиваются значения 0, 1 и 2 соответственно.

Пример: enum {two = 2, three, four, ten = 10, eleven, fifty = ten + 40};

Константам *three* и *four* присваиваются значения 3 и 4; константе *eleven* — 11.

Имена перечисляемых констант должны быть уникальными, а их значения могут совпадать. Преимущество применения перечисления перед описанием именованных констант и директивой # *define* состоит в том, что связанные константы нагляднее; кроме того, компилятор при инициализации констант может выполнять проверку типов.

При выполнении арифметических операций перечисления преобразуются в целые. Поскольку перечисления являются типами, определяемыми пользователем, то для них можно вводить собственные операции.

Переименование типов. Для того чтобы сделать программу более ясной, можно задать типу новое имя с помощью ключевого слова *typedef*:

typedef тип новое_имя [размерность];

В данном случае квадратные скобки являются элементом синтаксиса. Размерность может отсутствовать.

Примеры:
```
typedef long int LI;
typedef unsigned int UINT;
typedef char Msg[100];
```

Введенное таким образом имя можно использовать таким же образом, как и имена стандартных типов:

```
LI a,b;          // Две переменных типа long int
UINT i, j;       // Две переменных типа unsigned int
Msg str[10];     // Массив из 10 строк по 100 символов
```

Введенный с помощью *typedef* идентификатор не может использоваться с другими спецификаторами типов. Следующая запись неверна:

unsigned LI pay;

Кроме задания типам с длинными описаниями более коротких псевдонимов *typedef* используется для облегчения переносимости программ: если машинно-зависимые типы объявить с помощью операторов *typedef*, то при переносе программы потребуется внести изменения только в эти операторы.

Определение переменных. Любая переменная при определении может быть ***инициализирована***. В качестве начального значения может использоваться любое константное выражение.

Примеры:
```
int f=2;
float x=4.56;
```

Переменные бывают ***локальные*** и ***глобальные***. Локальные переменные подразделяются на [5]:

• ***автоматические***. Значения автоматических переменных теряются при выходе из блока. Областью определения является блок, в котором эта переменная определена.

Пример: int x;

• ***регистровые***. Доступ к регистровым переменным более быстрый. В регистрах можно сохранять любые переменные, если размер занимаемой ими памяти не превышает разрядности регистра. Если компилятор не может сохранить переменные в регистрах, он трактует их как автоматические. Областью определения является блок, в котором эта переменная определена. Значения регистровых переменных теряются при выходе из блока.

Пример: register int x;

• ***статические***. Значения статических переменных не теряются при выходе из блока и инициализируются нулем, если явно не заданы начальные значения. Областью действия является блок.

Пример: static int x.

Глобальные переменные не локальны ни в каком блоке, постоянные инициализируются нулем, если явно не задано другое начальное значение. Областью действия является вся программа. Должны быть описаны во всех файлах программы, в которых к ним есть обращения. Должны быть описаны в файле до первого использования.

Пример: int Global_x;

Глобальные переменные могут быть статическими.

Пример: static int File_x;

Кроме того, переменные могут быть внешними. В данном случае они определяются в другом компоненте программы (файле), где должны быть явно описаны. При описании внешнего объекта используется ключевое слово *extern*.

*Примеры:*extern int x;extern int Global_x;

Препроцессор работает до трансляции и обеспечивает две важные возможности. Он позволяет определять текст с помощью макроподстановок и включать в текст файлы. Командная строка препроцессора начинается с символа # и заканчивается символом перевода на новую строку. Если непосредственно перед концом строки поставить символ обратной косой черты \ , то командная строка будет продолжена на следующую строку программы. Обычно препроцессор в программе включается до текста самой программы.

Описание: # define идентификатор строка
Пример: # define DEF 125

Заменяет каждое вхождение идентификатора *DEF* программы на 125. Отмена макроподстановки:

undef идентификатор

Пример: # undef DEF

Отменяет предыдущее определение для идентификатора *DEF*.
Как уже было указано, препроцессор позволяет включать файлы в текст программы.

Описание: # include <*имя_файла*>
Пример: # include <math.h>

В данном случае содержимое строки заменяется файлом *math.h*. Угловые скобки означают, что файл будет взят из некоторого стандартного каталога. Текущий каталог не просматривается. При необходимости включения файла из текущего каталога вместо угловых скобок ставятся кавычки.

Описание: # include "*имя_файла*"
Пример: # include "df.h"

В данном случае поиск файла *df.h* будет осуществляться в текущем каталоге. Если в текущем каталоге данного файла нет, то поиск производится в каталогах, определенных именем пути в опции –1 препроцессора. Если и там файла нет, то просматривается стандартный каталог.
Командные строки препроцессора используются для условной компиляции различных частей исходного текста в зависимости от внешних условий:

```
# if константное_выражение
# ifdef идентификатор
# ifndef идентификатор
# else
   …
# endif
```

 # elif (*else if*) — вставляет альтернативное условие как продолжение блока *# if …# endif*. Эта директива определяет, выполняется ли условие компиляции таким же образом, как директива *# if*.

Описание: #elif константное_выражение
Примеры: # if DFR+5

«Истина», если константное выражение *DFR*+5 не равно нулю.

```
# ifdef DFR
```

«Истина», если идентификатор *DFR* определен ранее командой # *define*.

```
# ifndef DFR
```

«Истина», если идентификатор *DFR* не определен в настоящий момент.

Если предшествующие проверки # *if*, # *ifdef*, # *ifndef* дают значение «истина», то строки от # *else* до # *endif* игнорируются при компиляции. Если эти проверки дают ложь, то строки от проверки до # *else* (а при отсутствии # *else* — до # *endif*) игнорируются. Команда # *endif* обозначает конец условной компиляции.

error — останавливает компиляцию и выводит на экран сообщение об ошибке.

Описание: #error сообщение
Пример:
```
#ifdef ERROR
#error Внимание ошибка
#endif
```

В данном случае останавливается компиляция и выводится на экран сообщение, если символ *ERROR* определен. Когда компилятор выполняет директиву # *error*, он останавливается и печатается сообщение «Внимание ошибка». Как правило, директива используется, если дальнейшая компиляция программы не имеет смысла.

line — вставляет новый номер строки и новое имя исходного файла, что отражается макросами *__FILE__* и *__LINE__*, которые будут рассмотрены далее. Данная директива используется крайне редко и не оказывает существенного влияния на исходный файл.

Описание: #line *номер* ["имя_файла"]

После выполнения компилятором данной строки текущему номеру следующей строки присваивается номер *номер*, нумерация следующих строк продолжается с нового номера. Если файл *filename* определен, он становится новым значением *__FILE__*.

#*pragma* — позволяет использовать специфичные для конкретных реализаций директивы.

Описание: #pragma аргументы

Аргумент *аргументы* содержит имя директивы. В данном случае появляется возможность определить любые желаемые директивы, не обращаясь к другим, поддерживающим их компиляторам. С помощью данной директивы можно, например, задавать программе функцию, которая должна вызываться либо при загрузке перед вызовом функции *main*, либо непосредственно перед выходом из программы.

Примеры:
```
#pragma startup имя_функции
#pragma exit имя_функции
```

В этом случае аргумент *имя_функции* содержит имя ранее объявленной функции.

Макросы. Язык *C++* определяет набор символических имен, которые служат для определения «состояния дел во время компиляции», которые также называют макросами. Рассмотрим их подробней:

__cplusplus — определяется, если исходный файл был скомпилирован как файл *C++*. Данный макрос также задает возможность использования функций *C++*.

Пример:
```
#ifdef __cplusplus
int I = get_random ( );
#endif
```

В данном случае в программе на языке *C* появляется возможность использовать функцию *get_random* ( ), из набора функций языка *C++*.

__DATE__ — во время компиляции данный макрос преобразуется в строку, ограниченную кавычками, которая содержит текущую дату компиляции, в виде *месяц день год*.

__FILE__ — во время компиляции транслируется в строку, заключенную в кавычки, которая содержит имя текущего исходного файла.

__LINE__ — во время компиляции транслируется в номер текущей строки компилируемого исходного файла и может изменяться директивой # *line*.

__TIME__ — компилируется в строку, заключенную в кавычки и содержащую текущее время компиляции в формате *часы: минуты: секунды*.

§ 1.3. Основные виды операций

Выражение состоит из одного или большего числа операндов и символов операций. Рассмотрим основные виды операций.

Арифметические операции

1. + (сумма).

Пример. Сумма значений *x* и *y*:
```
z=x+y;
```

2. − (разность).

Пример. Разность значений *m* и *n*:
```
i=m−n;
```

3. − (унарный минус).

Пример. Изменение знака *fr*:
```
fr=−fr;
```

4. * (произведение).

Пример. Произведение значений *x* и *y*:
```
z=x*y;
```

5. / (частное).

Пример. Частное от деления *m* на *n*:
```
i=m/n;
```

6. **%** (деление по модулю).

Пример. Деление по модулю *f* на 30:
```
g=f%30;
```

7. ++ (увеличение на 1).

Пример. Увеличить *x* на 1:
```
y=x++;
```

8. −− (уменьшение на 1).

Пример. Уменьшить *x* на 1:
```
y=x−−;
```

Операции присваивания

Значением выражения, в которое входит операция присваивания, является значение левого операнда после присваивания. Существуют следующие виды операций присваивания:

9. = (присваивание значения переменной).

Пример. Присваивание переменной *x* значение *y*:
```
x=y;
```

10. += (увеличение).

Пример. Увеличение переменной *y* на 2:
```
y+=2;
```

11. −= (уменьшение).

Пример. Уменьшение переменной*y* на *a*:
y−=a;

12. *= (умножение).

Пример. Умножение переменной *gt* на *re*:
gt*=re;

13. /= (деление).

Пример. Деление переменной *d* на 3:
d/=3;

14. %= (значение по модулю).

Пример. Значение *x* по модулю 2.
x%=2;

15. >>= (сдвиг двоичного представления вправо на количество бит).

Пример. Сдвинуть двоичное представление *x* вправо на 3 бит:
x>>=3;

16. <<= (сдвиг двоичного представления влево на количество бит).

Пример. Сдвинуть двоичное представление *y* влево на 4 бит:
y<<=4;

17. &= (побитовая операция «И»).

Пример. Побитовая операция «И» двоичных представлений *x* и *y*:
x&=y;

18. ^= (побитовая операция «исключающее ИЛИ»).

Пример. Побитовая операция «исключающее ИЛИ» двоичных представлений *x* и *y*.
x^=y;

19. |= (побитовая операция ИЛИ).

Пример. Побитовая операция «ИЛИ» двоичных представлений *x* и *y*:
x|=y;

Операции отношения

20. == («истина», если переменные равны, иначе — «ложь»).

Пример:
```
if (i==0)
break;
```

21. != («истина», если переменные не равны, иначе — «ложь»).

Пример:
```
while (i!=0)
I=x;
```

22. < («истина», если одна переменная меньше другой).

Пример:
```
if(x<0)
x=y;
```

23. <= («истина», если одна переменная меньше или равна другой).

Пример:
```
if(x<=0)
x=y;
```

24. > («истина», если одна переменная больше другой).

Пример:
```
if(x>0)
x=y;
```

25. **>=** («истина», если одна переменная больше или равна другой).

Пример:
```
if(x>=0)
x=y;
```

Логические операции

26. **!** («истина», если выражение ложно, логическое «НЕ»).

Пример:
```
if(!well)
printf("not well");
```

27. **||** (логическая операция «ИЛИ»).

Пример:
```
if(x<m||x>n)
y=x;
```

28. **&&** (логическая операция «И»).

Пример:
```
f(x<m&&y>n)
z++;
```

Побитовые операции

29. **~** (дополнение до единицы, побитовое «НЕ»). Значение выражения *~x* содержит 1 во всех разрядах, где *x* содержит 0, и 0 во всех разрядах, где *x* содержит 1.

Пример:
```
opposite=~mask;
```

30. **>>** (сдвиг вправо в двоичном представлении на количество разрядов). Освобождающиеся слева разряды заполняются нулями.

Пример:
```
x=x>>3;
```

31. **<<** (сдвиг влево в двоичном представлении на количество разрядов). Освобождающиеся справа разряды заполняются нулями.

Пример:
```
x=x<<3;
```

32. **&** (побитовая операция «И» в двоичном представлении). Значение выражения содержит 1 во всех разрядах, в которых оба числа содержат 1, 0 — в остальных разрядах.

Пример:
```
flag=(x&mask)!=0;
```

33. **|** (побитовая операция «ИЛИ» в двоичном представлении). Значение выражения содержит 1 во всех разрядах, в которых одно из чисел содержит 1, 0 — в остальных разрядах.

Пример:
```
swert=sde|tre;
```

34. **^** (побитовая операция «исключающее ИЛИ» в двоичном представлении). Значение выражения содержит 1 в тех разрядах, в которых числа имеют разные двоичные значения, 0 — в остальных разрядах.

Пример:
```
dewr=m^n;
```

Адресные операции

35. **&x** (адрес *x*).

Пример:
```
itre=&x;
```

36. ***pe** (указание на переменную). Значением является переменная, адресуемая указателем *pe*.

Пример:
```
*rty=x;
```

37. ***fpe** (указание на функцию). Значением является функция, адресуемая указателем *fpe*.

Пример:
```
fpe=function;
(*fpe) (arg1, arg2);
```

Операции над массивами

38. **[ ]** (элемент массива). Значением выражения является переменная, отстоящая на *ie* переменных от начального адреса массива *pe* (pe[ie]=*(pe+ie)).

Пример. Присвоить *i*-му элементу массива *arr* значение 3:
arr[i]=3;

Операции над классами, структурами и объединениями

39. **.**(элемент структуры или объединения).

Пример. Присвоить значение 25 элементу *ht_gtyu* структуры *str*:
str.ht_gtyu=25;

40. **−>**(элемент структуры или объединения, на которую она указывает).

Пример. Присвоить значение *n* элементу *ht_gtyu*, на которую указывает *str*:
str−>ht_gtyu=n;

41. **::**(определение области видимости). Позволяет получить доступ к полям и методам класса за пределами объявления класса.

Пример. Для класса под названием *My* и его методом *void MyFunction* (*void*):
void My::MyFunction(void)

Другие операции

42. **?:**(если истинно выражение до вопросительного знака, то выполняется 1 элемент, иначе — второй).

Пример. Если *i* меньше 0, то *x* присвоить значение *–i*, иначе *x = i*:
x = (i< 0)? – i:i;

43. **,**(сначала выполняется выражение до запятой, далее — после запятой).

Пример: for (i = x, j = y; i < j; i++, j−−) z[i] = p[j];

44. ***sizeof***(число байт, требуемых для размещения данных).

Пример. Присвоить *x* количество элементов массива целых чисел *array*, определяемое как отношение общего числа байт массива к числу байт, занимаемых одним элементом:
x=sizeof(array)/sizeof(int);

45. **(*mun*)**(преобразование в другой тип данных).

Пример. Преобразовать целое значение *x* в число с плавающей точкой двойной точности перед умножением на 5:
y=(double)x*5;

46. **()** (вызов функции с аргументами).

Пример: y=sin(x);

Операции выделения и очистки памяти

В синтаксис языка Си не были включены операции для динамического выделения памяти. Для этой цели использовались функции библиотеки *stdlib.h* — *calloc*(), *malloc*() и *realloc*() для выделения памяти и *free*( ) — для ее освобождения. Для упрощения процесса работы с памятью в язык *C++* добавлены специальные операции для динамического выделения и освобождения памяти.

47. ***new*** — выделение динамической памяти.

48. ***delete*** — освобождение памяти, ранее выделенной операцией *new*.

Пример. Пусть *ptr* — любой тип, например *int*. Операция *new ptr* выдает указатель на объект типа *ptr* или *NULL*, если память невозможно выделить. В случае успеха выделяется *sizeof*(*ptr*) байт памяти. Операция *new*, в отличие от *malloc*, сама вычисляет необходимое число байт. Операция *delete* позволяет освободить ранее выделенную память.

```
int *ptr;
if (!(ptr=new int))          // Запрос памяти
{
    printf ("Нет памяти в системе\n"); exit (1);
}
delete ptr,                  // Освобождение памяти
```

Если *ptr* — массив, то операция new возвращает указатель на первый элемент массива. В этом случае операцию *delete* обязательно нужно использовать с квадратными скобками: *delete* [ ] *ptr*. Когда создается многомерный массив, то все размерности должны быть известны:
arr_ptr=new int[3][5][7]. Возможно использование операции *new* совместно с инициализацией [9]:

```
int *i;
i=new int(3); //*i=3
```

Приоритеты и порядок выполнения операций представлены в табл. 1.3. Для каждого элемента группы операций приоритет одинаковый. Чем выше приоритет, тем выше группа расположена в таблице.

Таблица 1.3

Приоритет	Операция	Название операций	№ в тексте	Порядок выполнения
1	*2*	*3*	*4*	*5*
1	()	Вызов функции	46	Слева направо
	[ ]	Выделение элемента массива по индексу	38	
	.	Выделение элемента класса (структуры, объединения)	39	
	–>	Выделение элемента класса (структуры, объединения), адресуемого указателем	40	
	::	Определение области видимости	41	
2	!	Логическое «НЕ»	26	Справа налево
	~	Побитовое «НЕ»	29	
	–	Унарный минус	3	
	++	Увеличение на единицу	7	
	––	Уменьшение на единицу	8	
	&	Определение адреса	35	
	*	Обращение по адресу	36, 37	
	(*mun*)	Преобразование типа	45	
	sizeof	Определение размера в байтах	44	
	new	Выделение памяти для данных	47	
	delete	Освобождение выделенной памяти	48	
3	*	Произведение	4	Слева направо
	/	Частное	5	
	%	Деление по модулю	6	
4	+	Сумма	1	«
	–	Разность	2	
5	<<	Сдвиг влево	31	«
	>>	Сдвиг вправо	30	

Окончание табл. 1.3

1	*2*	*3*	*4*	*5*
6	< <= > >=	Меньше Меньше или равно Больше Больше или равно	22 23 24 25	«
7	== !=	Равно Не равно	20 21	«
8	&	Побитовая операция «И»	32	«
9	^	Побитовая операция «исключающее ИЛИ»	34	«
10	|	Побитовая операция «ИЛИ»	33	«
11	&&	Логическая операция «И»	28	«
12	||	Логическая операция «ИЛИ»	27	«
13	? :	Условная операция	42	«
14	=	Все операции присваивания (= *= /= %= += −= <<= >>= &= ^= |=)	9–19	Справа налево
15	,	Операция «запятая»	43	Слева направо

Преобразование типов операндов. Для выражений, включающих одну бинарную операцию (т. е. операцию с двумя операндами разного типа), тип результата определяется по типам операндов. При этом действует следующее правило: значение типов *char* и *short* преобразуется в *int*, *int* в *long*, *a float* преобразуется в *double*. Если в результате преобразования любой из операндов оказывается типа *double*, то второй также преобразуется в *double*. Если один из операторов — *long*, то другой преобразуется к *long* и результат — *long* и т. д. Поэтому можно говорить, что неявные преобразования всегда идут от объектов, занимающих меньший объем памяти, к большим.

§ 1.4. Виды операторов

Выражение, заканчивающееся точкой с запятой, является ***оператором***. Один оператор может занимать одну или более строк. Два или большее количество операторов может быть расположено на одной строке. Одни операторы могут быть вложены в другие. Перед любым оператором для перехода к нему с помощью оператора *goto* может стоять метка. Метка состоит из идентификатора, за которым стоит двоеточие (:). Областью определения метки является вся функция.

Пример: def: x=3;

- **Оператор-выражение.** Как было указано ранее, любое выражение, заканчивающееся (;), является оператором.

Примеры:
```
m=n;
y=cos(x);
```

- **Составной оператор**. Состоит из одного или большего числа операторов любого типа, заключенных в фигурные скобки {}.

Пример: { x=1; y=2; z=3;}

- **Оператор перехода *goto.***

Описание: goto *метка*;

Управление передается на оператор с указанной меткой. Используется для выхода из вложенных управляющих операторов. Область действия ограничена текущей функцией.

Пример: goto def;

- **Оператор возврата *return*.**

Описание: return *выражение*;

Прекращает выполнение текущей функции и возвращает управление вызывавшей программы с передачей значения выражения.

Пример: return n+m;

- **Условные операторы *if*, *if — else*.**

*Описание оператора **if**:*

```
if(выражение)
        оператор;
```

Если выражение истинно, то выполняется оператор.

Пример:
```
if(x<=0)
    y=sin(x);
```

*Описание оператора **if — else**:*

```
if(выражение)
        оператор 1;
else
        оператор 2;
```

Если выражение истинно, то выполняется оператор 1, иначе — оператор 2.

Пример:
```
if(y!=5)
{
    x=43;
    z=4*x;
}
else
    z=25;
```

- **Оператор цикла *for*.**

Описание: for (*выражение 1*; *выражение 2*; *выражение 3*)
оператор;

Выражение 1 описывает инициализацию цикла, выражение 2 — проверка условия завершения цикла. Если оно истинно, то выполняется оператор тела цикла *for*, выполняется выражение 3, все повторяется, пока выражение 2 не станет ложным. Если оно ложно, то цикл заканчивается и управление передается следующему оператору. Выражение 3 вычисляется после каждой итерации.

Пример:
```
for(i=0; i<=8;i++)
    x=x*3;
```

- **Оператор цикла *while*.**

Описание:
```
while(выражение)
   оператор;
```

Если выражение истинно, то оператор выполняется до тех пор, пока выражение не станет ложным. Если выражение ложно, то управление передается следующему оператору.

Пример:
```
while(x<n)
{
   y=y*x;
   x++;
}
```

- **Оператор цикла *do — while*.**

Описание:
```
do оператор;
while (выражение);
```

Оператор выполняется. Если выражение истинно, то оператор выполняется вновь и вычисляется значение выражения; это повторяется до тех пор, пока выражение не станет ложным. Если выражение ложно, то управление передается следующему оператору. В отличие от операторов цикла *while* и *for*значение выражения в данном случае определяется после выполнения оператора, поэтому оператор выполняется хотя бы один раз.

Пример:
```
do
{
    x=x+i;
    i++;
}
while (i<=5);
```

- **Оператор завершения *break*.**

Описание:
```
break;
```

Прекращает выполнение ближайшего вложенного внешнего оператора *while*, *do*, *for*, *switch*. Управление передается оператору, следующему за заканчиваемым. Одно из назначений этого оператора — закончить выполнение цикла при присваивании некоторой переменной определенного значения.

Пример:
```
for (i=0; i<k; i++)
    if(a[i]==n)
        break;
```

- **Оператор продолжения *continue*.**

Описание:
```
continue;
```

Передает управление в начало ближайшего внешнего оператора цикла *while*, *do* или *for*, вызывая начало следующей итерации. Этот оператор по действию противоположен оператору *break*.

Пример:
```
for (i=0; i<k; i++)
{
    if(a[i]!=0)
        continue;
    a[i]=b[i];
}
```

- **Оператор-переключатель *switch*.**

Описание:
```
switch (выражение)
{
case константа: операторы;
case константа: операторы;
.
.
.
default: операторы;
}
```

Оператор сравнивает значение выражения с константами во всех вариантах *case*и передает управление оператору, который соответствует значению выражения. Каждый вариант *case* может быть помечен целой или символьной константой либо константным выражением. Константное выражение не может включать переменные или вызовы функций. Операторы, связанные с *default*, выполняются, если ни одна из констант в операторах *case* не равна значению выражения. Данный оператор может отсутствовать в командной строке. Ключевое слово *case* вместе с константой служат просто метками, и если будут выполняться операторы для некоторого варианта *case*, то далее будут

выполняться операторы всех последующих вариантов до тех пор, пока не встретится оператор *break*. Это позволяет связывать одну последовательность операторов с несколькими вариантами. Никакие две константы в одном операторе-переключателе не могут иметь одинаковые значения.

Пример:
```
switch (x)
{
case 1:
    y=n;
    break;
case 2:
case 3:
    y=n+x;
    break;
default:
    y=n – x;
    break;
}
```

• **Модификатор *const***. Модификатор const позволяет определять постоянные объекты определенного типа. Значения констант не могут быть изменены. Отметим при этом, что постоянный указатель не может быть изменен, тогда как объект, на который ссылается указатель, может. Обычный путь задания констант в языке *C* состоял в использовании директивы препроцессора # *define*. Введение типизированных констант позволяет обнаруживать больше ошибок на этапе трансляции.

Пример:
```
const float pi=3.14159;
char *const str="Hello, world";          /* Постоянный указатель */
char const *str1="Hello, world";         /* Указатель на постоянную строку */
```

Использование описания *const* без указания типа эквивалентно *const int*.

§ 1.5. Указатели

Специальными объектами в программах на языке *C++* являются ***указатели***. Применение указателей является одной из основных особенностей программирования на языке Си и *C++*. Значениями указателей служат адреса участков памяти, выделенных для объектов конкретных типов. Именно поэтому в определении и описании указателя всегда присутствует обозначение соответствующего ему типа. Эта информация позволяет в последующем с помощью указателя получить доступ ко всему сохраняемому объекту в целом. В простейшем случае определение и описание указателя на некоторый объект имеет вид:

type **имя_указателя*;

где *type* — обозначение типа; *имя_указателя* — идентификатор; * — унарная операция обращения по адресу. Признаком указателя служит символ *.

Пример:
```
int *1pi, *2pi, n;
```

В данном примере определяются два указателя на объекты типа *int* и переменная *n* типа *int*. При определении указателя возможно выполнять его инициализацию. В данном случае командная строка будет иметь вид:

type**имя_указателя*=инициализатор;

В качестве инициализатора используется константное выражение, частными случаями которого являются:

– явно заданный адрес участка памяти;
– указатель, уже имеющий значение;
– выражение, позволяющее получить адрес объекта с помощью операции &.

Примеры:
```
int x=5;                          /* Переменная x типа int, равная 5 */
int *p=&x;                        /* Инициализированный указатель на объект
                                     типа int */
char *cr = (char *)0xF000FFFE;    /* Указатель cr при инициализации получает
                                     значение адреса байта, содержащего сведения
                                     о типе компьютера (только для IBM PC) */
char *t(NULL);                    /* Нулевой указатель на объект типа char*/
```

Операция & называется ссылкой, которая представляет собой *синоним имени,* указанного при инициализации. Формат объявления ссылки:

тип &имя;

где *тип* — это тип величины, на которую указывает ссылка; & — оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа.

Пример:
```
int kol;
int &pal = kol;          // Ссылка pal— альтернативное имя для kol
const char &CR = '\n';   // Ссылка на константу
```

Следует отличать адресные операции * и &. Значением выражения **p* является переменная, адресуемая указателем *p*. Значением выражения *&v* является адрес переменной *v*.

Основными операциями над указателями являются:

- доступ по адресу;
- приведение типов;
- присваивание;
- получение адреса (&);
- сложение и вычитание;
- автоувеличение(++) и автоуменьшение(−−);
- сравнение.

Первые четыре операции уже были рассмотрены. Использование операции сложения указателя с целочисленным значением позволяет получить доступ к переменной, адрес которой больше на величину этого целочисленного значения. Суммировать два указателя синтаксис языка *C++* не позволяет. Операция вычитания указателя и целочисленного значения позволяет соответственно получить доступ к переменной, адрес которой меньше на величину этого целочисленного значения. Разность двух указателей позволяет определять объем (в байтах) между двумя участками памяти, а также количество переменных определенного типа в данном объеме. Увеличение или уменьшение указателя на единицу позволяет переместить его к соседнему элементу в памяти. Синтаксис языка *C++* позволяет определять указатель на указатель.

Примеры:
```
int x=23;
int *p=&x;
int *pp=&p;
int *ppp=&pp;
```

§ 1.6. Массивы

Массив — совокупность данных, имеющих одинаковые характеристики и размещающихся по порядку в выделенной области памяти. Язык *C++* не накладывает ограничений на размерность массива. Нумерация элементов массива всегда начинается с нуля. В отличие от других алгоритмических языков программирования после выделения памяти для массива его имя воспринимается как указатель того типа, к которому отнесены элементы массива. Значением имени массива является адрес его первого элемента.

Пример. Определение одномерного массива типа *type*:
type *имя_массива* [*константное_выражение*];

где *имя_массива* — идентификатор; *константное_выражение* определяет количество элементов в массиве. Возможно описание массива без указания количества его элементов (без *константного_выражения*). В данном случае определение массива происходит в другой части программы, где ему выделяется память. При определении массива может выполняться его инициализация, т. е. присваивание конкретных значений его элементам. Список значений элементов массива должен быть заключен в фигурные скобки. Явная инициализация элементов массива разрешена только при его определении и возможна либо с указанием размера массива в квадратных скобках, либо без явного указания.

Пример:
```
char s[ ]={'h','y','u','p','o'};
int x[4]={2,5,8,12};
```

Если задан размер массива, то значения, не заданные явно, равны 0. Если размер массива опущен, то он определяется по числу начальных значений.

При использовании отдельных элементов массива по каждому измерению отдельно указывается индекс в прямоугольных скобках:

имя_массива [индекс];

Так как имя массива после выделения памяти является указателем, то к нему применимы все правила адресной математики, связанной с указателями. Таким образом, записи

имя_массива[индекс] и *(имя_массива + индекс)

эквивалентны. В целом следует отметить, что индекс определяет не номер элемента массива, а его смещение относительно начала (поэтому номер первого элемента массива всегда равен 0).

Инициализация символьных массивов может быть выполнена с помощью строк и с помощью символов.

Пример:
```
char h[ ] ="hello";
```

Данная инициализация эквивалентна:

```
char h[ ] ={'h','e','l','l','o','\0'};
```

Массивы могут быть многомерными. Многомерный массив есть массив, элементами которого служат массивы. Командная строка

type имя_массива[a_1][a_2]…[a_N];

где *type* — допустимый тип; *имя_массива* — идентификатор; N — размерность массива; a_1 — количество в массиве элементов размерности $N - 1$ каждый и т. д.

Пример:
```
int A[5][34];
```

Переменная *A* — массив из пяти символов, каждый из которых состоит из 34 символов.

Начальные значения элементов многомерных массивов в списке инициализации располагаются с учетом порядка расположения в памяти.

Пример:
```
int A[4][3][5] = {0, 1, 2, 3, 4, 5, 6, 7};
```

В данном случае начальную инициализацию получили только первые 8 элементов массива:

```
A[0][0][0]=0;
A[0][0][1]=1;
A[0][0][2]=2;
A[0][0][3]=3;
A[0][0][4]=4;
A[0][1][0]=5;
A[0][1][1]=6;
A[0][1][2]=7;
```

Если необходимо инициализировать только часть элементов многомерного массива, но они размещены не подряд, то вводятся дополнительные фигурные скобки, каждая пара которых выделяет последовательность значений, относящихся к одной размерности.

Пример [9]: int B[4][4][5] = { { {0} }, { {50}, {55,60} }, { {245}, {123}, {342, 234, 127} } };

Данный пример задает инициализацию следующим элементам массива:

```
B[0][0][0] =0;
B[1][0][0] =50; B[1][1][0] =55; B[1][1][1] =60;
B[2][0][0] =245; B[2][1][0] =123;
B[2][2][0] =342; B[2][2][1] =234; B[2][2][2] =127;
```

Остальные элементы массива явно не инициализируются.

Как и в случае одномерных массивов, доступ к элементам многомерных массивов возможен с помощью индексированных переменных и с помощью указателей. В общем случае для трехмерного массива, например, индексированный элемент *d* [*i*] [*j*] [*k*] соответствует выражению *(*(*(*d* + + *i*)+*j*)+*k*). Допустимо в одном выражении комбинировать обе формы доступа к элементам многомерного массива *(*d*[*i*] [*j*]+*k*).

Элементами массива могут быть указатели. Например, массив *s*, состоящий из 5 указателей на символы:

```
char *s[5];
```

Использование указателей в качестве элементов массива позволяет в ряде случаев экономить объем памяти, выделяемой под массив. Особенно это актуально, если для каждого элемента массива требуется выделять различный объем памяти. В данном случае при использовании стандартных массивов выделяется объем памяти, необходимый для наибольшего элемента. Свободные ячейки памяти элементов меньшей размерности заполняются нулями. При использовании указателей память располагается более рационально.

Пример. Необходимо создать массив, элементами которого будет список фамилий. Количество символов в фамилиях нам неизвестно, поэтому под каждый элемент необходимо выделять максимально возможное их число, например 15. Рассмотрим массив из 3 фамилий:
char s[ ][15] = {"Иванов", "Петров", "Сидоров"};

Нерациональное использование памяти налицо. При использовании в данном случае массива указателей

```
char *p[] = {"Иванов", "Петров", "Сидоров"};
```

для каждого указателя массива *p* выделяется 3**sizeof*(*char**) байт.

§ 1.7. Сортировка и поиск в массиве

Задача поиска элементов в массиве является одной из главных задач при работе с массивами. И в простейшем случае поиск можно осуществить простым перебором элементов массива:

```
for (i=0; i<n; i++)
    if (A[i] == B) break;
if (i != n) ...элемент найден, i — его порядковый номер...
```

Предлагаемый алгоритм называется ***линейным* (*последовательным*) *поиском***, потому что он просматривает по очереди все элементы, сравнивая их с искомым. Когда данных немного, последовательный поиск работает достаточно быстро. Однако время его работы прямо пропорционально количеству данных, которые нужно просмотреть: удвоение количества элементов приведет к удвоению времени на поиск. Поэтому если размерность массива — сотни и тысячи элементов, то линейный поиск занимает продолжительное время, тем более что в реальных программах «элементами массива» являются, конечно, не простые переменные, а более сложные образования (например,

структурированные переменные). Та часть элемента данных, которая идентифицирует его и используется для поиска, называется ***ключом***. Остальная часть несет в себе содержательную информацию, которая извлекается и используется из найденного элемента данных.

Ключ — часть элемента данных, которая используется для его идентификации и поиска среди множества других таких элементов. Приведенный ранее фрагмент программы обеспечивает в неупорядоченном массиве последовательный, или линейный, поиск, а среднее количество просмотренных элементов для массива размерности N будет равно $N/2$.

Можно использовать более оптимальные алгоритмы поиска, которые приведут к уменьшению количества просмотренных элементов при поиске. Например, если элементы массива первоначально упорядочить, то для поиска элемента достаточно разбить массив на две половины, выбрать из них ту, в которой находится искомый элемент, разбить ее еще на две части и т. д. Такой алгоритм поиска называется ***двоичным поиском***. Однако для его реализации необходима предварительная проверка на упорядоченность массива. Функция проверки упорядоченности массива служит живой иллюстрацией теоремы: массив упорядочен, если упорядочена любая пара соседних элементов:

```
int check_order (int a[ ], int n)
{
    for (int i=0; i<n – 1; i++)
            if (a[i]>a[i+1])
                    return 0;
    return 1;
}
```

Таким образом, если функция возвращает 1 — массив упорядочен, если 0 — нет.

Алгоритм ***двоичного поиска*** выражается следующей последовательностью шагов:

– искомый интервал поиска делится пополам, и по значению элемента массива в точке деления определяется, в какой части следует искать значение на следующем шаге цикла;

– для выбранного интервала поиск повторяется;

– при «сжатии» интервала в 0 поиск прекращается;

– в качестве начального интервала выбирается весь массив.

Следующий пример исходного кода демонстрирует алгоритм двоичного поиска в упорядоченном массиве:

```
int binary_search (int A[ ], int n, int value)
{
int low;                                  // Левая граница
int high;                                 // Правая граница
int middle;                               // Середина
for(low = 0, high = n – 1; low <= high;)  //while (low<= high)
{
    middle = (low + high) / 2;            // Середина интервала
    if (A[middle] == value)               // Значение найдено
        return middle;                    // Вернуть индекс элемента
    if (A[middle] >value)
        high = middle–1;                  // Выбрать левую половину
    else low = middle+1;                  // Выбрать правую половину
}
return –1;                                // Значение не найдено
}
```

Функция возвращает индекс найденного элемента или значение –1, если элемент не найден; *A* [ ] — исходный массив; *n* — количество элементов; *value* — искомое значение.

С небольшими изменениями данный алгоритм может использоваться для определения места включения нового значения в упорядоченный массив. Для этого необходимо ограничить деление интервала до получения единственного элемента (*low* == *high*), после чего дополнительно проверить, куда следует производить включение:

```
int find_place (int A[], int n, int value)
{
```

```
int low;                                  // Левая граница
int high;                                 // Правая граница
int middle;                               // Середина
for(low = 0, high = n – 1; low < high;)   //while (low<high)
{
    middle = (low+ high)/ 2;              // Середина интервала
    if (A[middle] == value)               // Значение найдено
        return middle;                    // Вернуть индекс элемента
    if (A[middle] >value)
        high = middle – 1;                // Выбрать левую половину
else low = middle+1;                      // Выбрать правую половину
}                                         // Выход по low == high
if (value > A[low])
    return low+1;                         // Включить на следующую позицию
return low;                               // Включить на текущую позицию
}
```

При двоичном поиске после первого сравнения интервал уменьшается в 2 раза, после второго — в 4 раза и т. д. Таким образом, количество сравнений K будет не больше соответствующей степени 2, дающей размерность массива N, т. е. $2^K = N$, $K = \log_2(N)$. Поэтому для массива из $N = 1000$ элементов количество сравнений $K = 10$ при $N = 1\,000\,000$ — $K = 20$ и т. д. Величина K называется ***трудоемкостью*** алгоритма. ***Трудоемкость*** — зависимость числа базовых операций алгоритма от размерности входных данных. Трудоемкость показывает не абсолютные затраты времени в секундах или минутах, что зависит от конкретных особенностей компьютера, а в какой зависимости растет время выполнения программы при увеличении объемов обрабатываемых данных. На рис. 1.2 приведены графики трудоемкости известных алгоритмов [10]:

– трудоемкость линейного поиска — $N/2$ — линейная зависимость;

– трудоемкость двоичного поиска — зависимость логарифмическая $\log_2 N$;

– для сортировки обычно используется цикл в цикле. Отсюда видно, что трудоемкость даже самой плохой сортировки не может быть больше $N * N$ — зависимость квадратичная. За счет оптимизации она может быть снижена до $N * \log(N)$;

– алгоритмы рекурсивного поиска, основанные на полном переборе вариантов (будут рассмотрены далее), имеют обычно показательную зависимость трудоемкости от размерности входных данных (m^N).

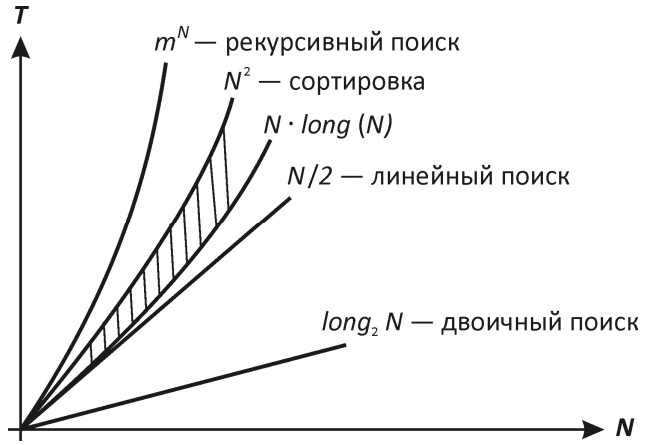


Рис. 1.2. Трудоемкость алгоритмов сортировки и поиска

Алгоритм двоичного поиска работает только в том случае, если элементы отсортированы. Если по одному и тому же набору данных планируется неоднократный повторный поиск, то выгоднее один раз отсортировать данные, а затем использовать двоичный поиск. Если набор данных известен заранее, то он может быть отсортирован при написании программы и проинициализирован во время компиляции. В противном случае необходимо производить сортировку во время выполнения программы.

Алгоритмы сортировки можно классифицировать по нескольким признакам [10].

Вид сортировки **по размещению элементов**: *внутренняя* — в памяти, *внешняя* — в файле данных.

Вид сортировки **по виду структуры данных,** ***содержащей сортируемые элементы***: сортировка *массивов*, *массивов указателей*, *списков* и *других структур данных*.

Существует достаточно большое количество алгоритмов сортировки. Рассмотрим основные виды. Прежде всего выделим сортировки, в которых в процессе работы создается упорядоченная часть — размер ее увеличивается на 1 за каждый шаг внешнего цикла. Сюда относятся две группы сортировок:

- ***сортировка выбором***: выбирается очередной минимальный элемент и помещается в конец последовательности;
- ***сортировка вставками***: очередной элемент помещается по месту своего расположения в выходную последовательность (массив).

Две другие группы используют разделения на части, но по различным принципам и с различной целью:

- ***сортировка разделением***: последовательность (массив) разделяется на две частично упорядоченные части по принципу «больше — меньше», которые затем могут быть отсортированы независимо (в том числе тем же самым алгоритмом);
- ***сортировка слиянием***: последовательность регулярно распределяется в несколько независимых частей, которые затем объединяются (слияние).

Сортировки этих групп отличаются от «банальных сортировок» тем, что процесс упорядочения в них в явном виде не просматривается.

Отдельная группа ***обменных сортировок*** с многочисленными оптимизациями основана на идее регулярного обмена соседних элементов.

Особняком стоит ***сортировка подсчетом***. В ней определяется количество элементов, больших или меньших данного, определяется его местоположение в выходном массиве.

Рассмотрим программные решения для перечисленных групп алгоритмов сортировок более подробно.

Сортировка выбором. Простейший алгоритм сортировки выбором будет выглядеть следующим образом:

– ищется минимальный элемент массива и записывается на первое место. Для этого в качестве минимального назначается первый элемент, с ним сравнивается второй; если второй меньше первого, то они меняются местами. Далее второй сравнивается с третьим и т. д. Функция, осуществляющая поиск минимального элемента, приведена ниже:

```
void min(int A[ ], int n)
{
int min=0;                          //Предположим, что первый элемент минимальный
                                    //Сравним оставшиеся элементы с минимальным
for (int i=1; i<n; i++)
    if(A[i]<A[min])
          min=i;
}
```

– повторяется операция поиска минимального элемента и записи в конец отсортированной последовательности, но уже без учета отсортированных элементов.

Исходный текст функции, осуществляющей сортировку выбором, приведен ниже:

```
void sort_choice(int A[ ], int n)
{
for (int i=0; i < n – 1; i++)
{
    for (int j=i+1, min=i; j<n; j++)
          if (A[j] < A[min])
                min=j;
    int buf=A[i];
```

```
    A[i] = A[min];
    A[min] =buf;
}
}
```

Трудоемкость такого алгоритма — $K = N*N/2$.

Сортировка вставками. Основная идея алгоритма: имеется упорядоченная часть, в которую очередной элемент помещается так, что упорядоченность сохраняется (включение с сохранением порядка). Можно проводить линейный поиск от начала упорядоченной части до первого элемента, больше данного, с конца — до первого элемента, меньше данного (трудоемкость алгоритма по операциям сравнения — $N*N/4$); использовать двоичный поиск места в упорядоченной части (трудоемкость алгоритма — $N*\log(N)$). Сама процедура вставки включает в себя перемещение элементов массива (не учтенное в приведенной трудоемкости). В следующем примере последовательность действий по вставке очередного элемента в упорядоченную часть «разложена по полочкам» в виде последовательности четырех действий:

```
void sort_insert1 (int A[ ], int n)
{
for (int i=1; i < n; i++)
{
    int buf=A[i];                  // Действие 1: сохранить очередной элемент
    for (int k=0; k<i; k++)        // Действие 2: поиск места вставки
        if(A[k]>buf) break;        // перед первым элементом, бóльшим buf
    for(int j=i – 1; j>=k; j – –)  // Действие 3: сдвиг на 1 элемент вправо
        A[j+1]=A[j];
    A[k]=buf;                      //Действие 4: вставка очередного элемента
}                                  // на место первого, который больше его
}
```

Вставка погружением. Очередной элемент «погружается» путем ряда обменов с предыдущим до требуемой позиции в уже упорядоченную часть массива, пока «не достигнет дна» либо пока не встретит элемент меньше себя:

```
void sort_insert2(int A[ ], int n)
{
for (int i=1; i<n; i++)                    // Пока не достигли «дна» или элемента меньше себя
    for (int k=i; k !=0 && A[k] < A[k – 1]; k– –)
    {
        int buf=A[k],
        A[k]=A[k – 1];
        A[k – 1]=buf;
    }
}
```

Сортировка Шелла. Существенными в сортировках вставками являются затраты на обмены или сдвиги элементов. Для их уменьшения желательно сначала производить погружение с бóльшим шагом, сразу определяя элемент «по месту», а затем делать точную «подгонку». Так реализовано в сортировке Шелла: исходный массив разбивается на m частей, в каждую из которых попадают элементы с шагом m, начиная от 0, 1,..., m–1 соответственно, т. е.

```
0 ,     m ,     2m ,     3m ,...
1 ,     m+1,    2m+1,    3m+1,...
2 ,     m+2,    2m+2,    3m+2,...
```

Каждая часть сортируется отдельно с использованием алгоритма вставок или обмена. Затем выбирается меньший шаг и алгоритм повторяется. Шаг удобно выбрать равным степеням 2, например: 64, 32, 16, 8, 4, 2, 1. Последняя сортировка выполняется с шагом 1. Несмотря на увеличение числа циклов, суммарное число перестановок будет меньшим. Принцип сортировки Шелла можно применить и во всех обменных сортировках. Следует отметить, что сортировка Шелла требует четырех вложенных циклов:

– 1-й — по шагу сортировки (по уменьшающимся степеням 2 – m = 64, 32, 16, ...);

– 2-й — по группам (по индексу первого элемента в диапазонеk = 0, ..., m – 1);

– еще два цикла обычной сортировки — погружением для элементов группы, начинающейся с k шагом m.

Для двух последних циклов нужно взять базовый алгоритм, заменив шаг 1 на m и поменяв границы сортировки.

Обменная сортировка «пузырьком». Суть данной сортировки заключается в следующем: производятся попарное сравнение соседних элементов 0–1, 1–2, ... и перестановка, если пара расположена не в порядке возрастания. Просмотр повторяется до тех пор, пока при пробегании массива от начала до конца перестановок больше не будет:

```
void sort_bubble (int A[ ], int n)
{
int i, k;                          // Количество сравнений
do                                 // Повторять просмотр...
{
    k = 0;
    for (i=0; i<n – 1; i++)
        if (A[i] >A[i+1])          // Сравнить соседей
        {
            int buf = A[i];        // Переставить соседей
            A[i]=A[i+1];
            A[i+1]=buf;
            K++;
        }
}
while(k != 0);                     //Пока есть перестановки
}
```

Оценить трудоемкость алгоритма можно через среднее количество сравнений, которое равно $(N*N – N)/2$. Обменные сортировки имеют ряд особенностей. Прежде всего они чувствительны к степени исходной упорядоченности массива. Полностью упорядоченный массив будет просмотрен ими только один раз.

Шейкер-сортировка учитывает тот факт, что от последней перестановки до конца массива будут находиться уже упорядоченные данные. Исходя из этого утверждения, имеет смысл делать просмотр не до конца массива, а до последней перестановки, выполненной на предыдущем просмотре. Для этой цели в программе обменной сортировки необходимо запоминать индекс переставляемой пары, который по завершении внутреннего цикла просмотра и будет индексом последней перестановки. Кроме того, необходима переменная — граница упорядоченной части, которая должна при переходе к следующему шагу получать значение индекса последней перестановки. Условие окончания — граница сместится к началу массива:

```
void sort_sheyker(intA[ ], int n)
{                                  // b— граница отсортированной части
int i, b, b1;                      // b1 — место последней перестановки
for (b=n – 1; b!=0; b=b1)          // Пока граница не сместится к правому краю
{
    b1=0;
    for (i=0; i<b; i++)            // Просмотр массива
        if (A[i] >A[i+1])          // Перестановка с запоминанием места
        {
            int buf = A[i];
            A[i]=A[i+1];
            A[i+1]=buf;
            b1=i;
}   }   }
```

Если просмотр делать попеременно в двух направлениях и фиксировать нижнюю и верхнюю границы неупорядоченной части, то в результате получим классическую Шейкер-сортировку.

Сортировка подсчетом — сортировка, требующая обязательного выходного массива, поскольку элементы в нем размещаются не подряд. Идея алгоритма: число элементов, меньше текущего, определяет его позицию (индекс) в выходном массиве. Наличие переменной-счетчика и использование его в качестве индекса в выходном массиве являются хорошо заметными программными контекстами. Трудоемкость алгоритма — $N*N/2$:

```
void sort_calc (int A[ ],int B[ ], int n)
{
int i, j, k;                          // k — счетчик элементов, бóльших текущего
for (i=0; i< n; i++)
{
    for (k=0, j=0; j<n; j++)
        if (A[j] <A[i])
            k++;
        B[k]=A[i];                    // Место элемента в выходном массиве
}
}
```

Этот фрагмент некорректно работает, если в массиве имеются равные элементы.

Сортировка слиянием. Первоначально рассмотрим алгоритм слияния упорядоченных последовательностей. Каждый шаг слияния включает выбор минимального из двух очередных элементов и перенос его в выходную последовательность. Каждый массив имеет собственный индекс, но только индекс выходного массива меняется линейно, поскольку за один шаг производится одно перемещение. Переход к следующему элементу во входной последовательности происходит только в одной из них (где выбран минимальный элемент), поэтому индексы изменяются неравномерно внутри условной конструкции. Кроме этого, любая последовательность может закончиться раньше, чем противоположная, что необходимо отслеживать:

```
void sleave(int B[ ], intA1[ ], int A2[ ])
{
int i, j, k;
for(i=j=k=0; i<2*n; i++)
{
    if(k==n)
        B[i]=A1[j++];                 //Второй кончился, сливать только 1
    else if(j==n)
        B[i]=A2[k++];                 //Первый кончился, сливать только 2
    else if(A1[j]<A2[k])              //Сливать меньший элемент
        B[i]=A1[j++];
    else B[i]=A2[k++];
}
}
```

На практике слияние эффективно при работе с данными большого объема в последовательных файлах, где принцип слияния последовательно читаемых данных без просмотра вперед выглядит естественно.

Простое однократное слияние базируется на других алгоритмах сортировки. Массив разбивается на *n* частей, каждая из них сортируется независимо, а затем отсортированные части объединяются слиянием. Реально такое слияние используется, если массив целиком не помещается в памяти. В данной простой модели одномерный массив разделяется на 10 частей — используется двумерный массив из 10 строк по 10 элементов. Затем каждая строка сортируется отдельно. Алгоритм слияния использует стандартные контексты: выбирается строка, в которой первый элемент минимальный (минимальный из очередных) — он и сливается в выходную последовательность. Исключение его производится сдвигом содержимого строки к началу, причем в конец добавляется «очень большое число», играющее роль «пробки» при окончании этой последовательности:

```
void sort(int a[ ], int n);                 // Любая сортировка одномерного массива
#define N 4                                 // Количество массивов
void sort_sleave (int A[ ], int n)
{
int B[N][10];                               // Размерность массивов
int i, j, m = n / N;
for (i=0; i<n; i++)                         // Распределение
    B[i / m][i%m]=A[i];
for (i=0; i<N; i++)                         // Сортировка частей
    sort(B[i],10);
for (i=0; i<n; i++)                         // Слияние
{
    for (int k=0, j=0; j<N; j++)            // Индекс строки с минимальным
        if (B[j][0] < B[k][0]) k=j;         // B[k][0]
            A[i] = B[k][0];                 // Слияние элемента
    for (j=1; j<m; j++)                     // Сдвиг сливаемой строки
        B[k][j – 1]=B[k][j];
    B[k][m – 1]=10000;                      // Запись ограничителя
}
}
```

Циклическое слияние. Оригинальный алгоритм «сортировки без сортировки» базируется на том факте, что при слиянии двух упорядоченных последовательностей длиной *s* длина результирующей в два раза больше. Главный цикл включает в себя разделение последовательности на две части и их обратное слияние в одну. Первоначально они неупорядочены, тем не менее можно считать, что в них имеются группы упорядоченных элементов длиной $s = 1$. Каждое слияние увеличивает размер группы вдвое, т. е. размер группы меняется: $s = 2, 4, 8,...$ Поэтому идея заключена в способе слияния: оно не может выйти за пределы очередной группы, пока обе сливаемые группы не закончились. Это значит, что переход к следующей паре осуществляется «скачком».

В приведенной программе размерность массива для простоты должна быть равна степени 2, чтобы группы были всегда полными. Внешний цикл организован формально: переменная *s* принимает значения степени 2. В теле цикла сначала производится разделение массива на две части, а затем их слияние. Для успешного проектирования слияния важно правильно выбрать индексы с учетом независимости и относительности «движений» по отдельным массивам. Поэтому их здесь целых четыре на три массива. Индекс *i* в выходном массиве увеличивается в заголовке цикла. Это значит, что за один шаг цикла один элемент из входных последовательностей переносится в выходную. Движение по группам разложено на две составляющие: *k* — общий индекс начала обеих групп, *i*1, *i*2 — относительные индексы внутри групп. Здесь же отрабатывается «скачок» к следующей паре групп: при условии, что обе группы закончились (*i*1==*s*&&*i*2==*s*), обнуляются относительные индексы в группах, а индекс начала увеличивается на длину группы. В процессе слияния отрабатываются четыре возможные ситуации: завершение первой или второй группы и (в противном случае) выбор минимального из пары очередных элементов групп:

```
void sort_cycle (int A[ ], int n)
{
int B1[100], B2[100];
int i, i1 ,i2, s, a1 ,a2, a, k;
    for (s=1; s!=n; s*=2)                   // Размер группы кратен 2
    {
    for (i=0; i<n / 2; i++)                 // Разделить пополам
    {
        B1[i]=A[i];
        B2[i]=A[i+n / 2];
    }
    i1=i2=0;
```

```
    for (i=0, k=0; i<n; i++)                // Слияние с переходом «скачком»
    {
        if (i1==s&&i2==s)                   // при достижении границ
            k+=s, i1=0, i2=0;               //обеих групп
    // 4 условия слияния по окончании групп и по сравнению
        if (i1==s)
            A[i]=B2[k+i2++];
        else if (i2==s)
            A[i]=B1[k+i1++];
        else if (B1[k+i1] < B2[k+i2])
            A[i]=B1 [k+i1 ++];
        else
            A[i]=B2[k+i2++];
    }
}
}
```

Сортировки рекурсивным разделением. Сортировки разделяют массив на две части относительно некоторого значения, называемого медианой. Медианой может быть выбрано любое «среднее» значение, например среднее арифметическое. Сами части не упорядочены, но обладают таким свойством, что в левой части элементы меньше медианы, а в правой — больше. Благодаря такому свойству эти части можно сортировать независимо друг от друга. Для этого нужно вызвать ту же самую функцию сортировки, но уже по отношению не к массиву, а к его частям. Функции, вызывающие сами себя, называются рекурсивными и будут рассмотрены далее. Рекурсивный вызов продолжается до тех пор, пока очередная часть массива не станет содержать единственный элемент:

```
void sort_recurs(int A[ ], int a, int b)
{
int i;
if (a>=b)
    return;
                                        // Разделить массив в интервале a…b на две части a…i – 1
                                        // и i…b относительно значения V по принципу <V, >=V
sort_recurs (A, a, i – 1);
sort_recurs (A, i, b);
}
```

Необходимо следить, чтобы разделяемые части содержали хотя бы один элемент. Разделение лучше всего производить в отдельном массиве, после чего разделенные части перенести обратно. Далее приведен алгоритм разделения массива. Разделение легко сделать, заполняя выходной массив с двух концов, слева и справа от медианы:

```
int division(int A[ ], int B[ ], int n, int middle)
{                                       //A[ ] — входной, B[ ] — выходной
int i, j, k;
for (i=0, j=0, k=n – 1; i<n; i++)
{
    if(A[i]<middle)                     // В левую часть
        B[j++]=A[i];
    else B[k– –]=A[i];                  //В правую часть
}
return j;                               // Вернуть точку разделения
}
```

«Быстрая» сортировка осуществляет разделение в одном массиве с использованием оригинального алгоритма на основе обмена. Сравнение элементов производится с концов массива ($i = a$, $j = b$) к середине (i++ или j—), причем «укорочение» происходит только с одной из сторон. После каждой перестановки меняется тот конец, с которого выполняется «укорочение». В результате этого массив разделяется на две части относительно значения первого элемента $A[a]$, который и становится медианой:

```
void sort_quick (int A[ ], int a, int b)
{
int i, j, mode;
if (a>=b) return;                                          // Размер части = 0
for (i=a, j=b, mode=1; i < j; mode > 0 ? j– – : i++)
    if (A[i] >A[j])
    {
        int buf = A[i];                    // Перестановка концевой пары
        A[i] = A[j];
        A[j]=buf;
        mode =– mode;                      // со сменой сокращаемого конца
    }
sort_quick (A, a, i – 1);
sort_quick (A, i+1, b);
}
```

Очевидно, что медиана делит массив на две неравные части. Алгоритм разделения можно выполнить итерационно, применяя его к той части массива, которая содержит его середину (по аналогии с двоичным поиском). Тогда в каждом шаге итерации медиана будет сдвигаться к середине массива.

Глава 2
Структурное и модульное программирование

§ 2.1. Функции

Понятие функции есть центральный элемент при программировании на *C++*. Вводной точкой для любой программы является функция *main* (*WinMain* для *Windows*).

По умолчанию функция имеет тип external, т. е. является доступной из любых файлов программы. Если для функции указан тип *static*, то функция доступна только внутри данного файла. Различают прототипы (предварительные описания, форварды) функций и их определения. В *C++* прототипы обязательны.

Определение функции имеет следующий формат:

type *имя_функции* (*спецификация_формальных_параметров*) {*тело_функции*},

где *type* — тип возвращаемого функцией значения, в том числе *void*, если функция не возвращает значения; *имя_функции* — идентификатор. Спецификация формальных параметров — это либо пусто, либо *void*, либо список спецификаций отдельных параметров. Для каждого параметра может быть задано значение. *Тело_функции* — блок или составной оператор (последовательность операторов и описаний), заключенный в фигурные скобки. Важнейшим оператором тела функции является возврат в точку вызова *return*.

Примеры:

```
float f (float x, float y, float z)
{return (x*y+z);}
void s( )
{
    printf ("Функция, не возвращающая значение");
}
```

Следует отметить, что язык *C++* — язык со строгой типизацией, из чего следует, что если в языке *Cu* описание *int f*( )обозначает функцию типа *int*, то в языке *C++* — функцию типа *int* без параметров. Более предпочтительной формой является описание *int f* (*void*).

Пример:

```
int func(void);                    // Функция без параметров
int func(T1 t1, T2 t2, T3 t3=l);   //Три параметра. Для третьего задано
                                   // значение по умолчанию
```

Параметры, которые задаются по умолчанию, должны быть последними в списке параметров. Значение (...) в описании соответствует случаю, когда функция может быть вызвана с переменным числом параметров.

Пример:

```
int printf (char *format, ...);
```

При трансляции вызова функции компилятор преобразует тип фактических параметров в соответствии с описанием прототипа. Если прототип не указан, то преобразования параметров осуществляются в соответствии с некоторыми стандартными соглашениями. Правила преобразования указаны в табл. 2.1.

Таблица 2.1

Исходный тип	Преобразованный тип	Метод
char	int	Знаковое расширение или 0 в старшем байте
unsigned char	int	0 в старшем байте
signed char	int	Знаковое расширение
short	int	«
unsigned short	int	«
enum	int	«

Вызов функции может осуществляться двумя способами:

имя_функции (x_1, x_2, …, x_n)
(**указатель_на_функцию*) (x_1, x_2, …, x_n)

Указатель_на_функцию — переменная, содержащая адрес функции. Аргументы передаются по значению, т. е. каждое выражение x_1, …, x_n вычисляется и значение передается функции. Вызов функции — выражение, значением которого является возвращаемое функцией значение.

§ 2.2. Структуры и объединения

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе простых типов данных, массивов и указателей. Язык *C++* позволяет программисту определять свои типы данных и правила работы с ними. Рассмотрим основные структурированные типы данных.

2.2.1. Структуры

Структура — определенный тип, представляющий собой поименованную совокупность компонент (элементов, членов, полей структуры). В отличие от массива, все элементы которого однотипны, структура может содержать элементы разных типов.

Дополнительно элементы структуры могут быть битовыми полями, которые по-другому недоступны.

В *C++* структуры рассматриваются как класс. Это означает, что для компонент структуры могут быть определены спецификации доступа: *public* (по умолчанию), *private* и *protected*. Структуры описываются с помощью ключевого слова *struct*:

```
struct [имя_типа]
{
    тип_1 элемент_1;
    тип_2 элемент_2;
    ...
    тип_n элемент_n;
} [список_описателей];
```

Элементы структуры называются **полями структуры** и могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него. ***Если отсутствует имя типа, то должен быть указан список описателей переменных, указателей или массивов.*** В этом случае описание структуры служит определением элементов этого списка:

```
struct
{
    char Surname [25]:
    int date, code;
    double money;
}mas[200], *ps;
```

Если список отсутствует, описание структуры определяет новый тип, имя которого можно использовать в дальнейшем наряду со стандартными типами. Например:

```
struct Man
{                                   //Описание нового типа Man
    char Surname [25];
    int date, code;
    double money;
};                                  // Описание заканчивается точкой с запятой
    // Определение массива типа Man и указателя на тип Man:
. . .
Man mas[200], *ps;
```

Имя структуры можно использовать сразу после его объявления (определение можно дать позднее) в тех случаях, когда компилятору не требуется знать размер структуры. Например:

```
struct Struct1;                                  // Объявление структуры Struct1
struct Struct2
{
    Struct1 *p;                                  // Указатель на структуру Struct1
    Struct2 *prev, *succ;                        // Указатели на структуру Struct2
};
struct Struct1 { /* определение структуры Struct1 */};
```

Это позволяет создавать связные списки структур. Для инициализации структуры значения ее элементов перечисляют в фигурных скобках в порядке их описания:

```
struct
{
    char Surname [25];
    int date, code;
    double money;
} Man = {"Иванов", 26, 119, 9500.55};
```

При инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива (учитывая, что многомерный массив — это массив массивов):

```
struct complex
{
    float Real, Im;
} compl [2][3] = {
    {{1, 1}, {1, 1}, {1, 1}},          // Строка 1, т.е.массив compl [0]
    {{2, 2}, {2, 2}, {2, 2}}           // Строка 2, т.е. массив compl [1]
};
```

Для переменных одного и того же структурного типа определена **операция присваивания**, при этом происходит поэлементное копирование. Структуру можно передавать в функцию и возвращать в качестве значения функции. Другие операции со структурами могут быть определены пользователем. Размер структуры не обязательно равен сумме размеров ее элементов, поскольку они могут быть выровнены по границам слова.

Доступ к полям структуры выполняется с помощью операций выбора **.** (точка) при обращении к полю через имя структуры и **–>** при обращении через указатель. Например:

```
Man men, mas[200], *ps;
men.Surname = "Иванов";
mas[8].code = 119;
ps–>money = 9500.55;
```

Если элементом структуры является другая структура, то доступ к ее элементам выполняется через две операции выбора:

```
struct A {int a; double x;};
struct B {A a; double x;} x[2];
x[0].a.a = 1;
x[1].x = 0.1;
```

Как видно из примера, поля разных структур могут иметь одинаковые имена, поскольку у них разная область видимости. Более того, можно объявлять в одной области видимости структуру и другой объект (например, переменную или массив) с одинаковыми именами, если при определении структурной переменной использовать слово *struct*.

Память под элементы структуры выделяется последовательно, элемент за элементом, слева направо, от младших к старшим адресам. В примере

```
struct mystruct
{
    int i;
```

```
    char str[21];
    double d;
} s;
```

память выделяется следующим образом: 2 байта на целое, 21 байт на строку, 8 байт на *double*. Формат расположения в памяти определяется опцией «выравнивание слов». Если эта опция выключена (по умолчанию) — выделяется непрерывный участок памяти в 31 байт. Если эта опция включена, то выполняются следующие правила:

1. Структура начинается с границы слова.
2. Несимвольные элементы размещаются с четным смещением относительно начала структуры.
3. Если необходимо, то в конце добавляется еще один байт, так чтобы структура занимала четное число байт.

Элементы структуры можно определить как знаковые или беззнаковые **битовые поля** шириной от 1 до 16 бит. Битовые поля — это особый вид полей структуры. Они используются для плотной упаковки данных, например флажков типа «да / нет». Общая форма описания для битового поля следующая:

тип идентификатор: ширина,

где *тип — char*, *unsigned char, int* или *unsigned int*; *ширина* — целое от 0 до 16.

Если идентификатор поля пропущен, то указанное число бит размещается в памяти, но не доступно. Например [7],

```
struct mystruct
{
    int i:2;
    unsigned j:5;
    int:4;
    int k:1;
    unsigned m:4;
} a, b, c;
```

имеет следующее размещение в памяти:

```
15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
|     m     | k |не используется|        j        | i |
```

Для знаковых полей старший левый бит интерпретируется как знак. Например, если в поле шириной 2 бита записано двоичное 11, то это интерпретируется как 3 для типа *unsigned*, но как 1 для типа *int*. Битовые поля могут быть определены только в структурах, объединениях и классах. Доступ к ним осуществляется с помощью тех же операторов **.** и **–>**.

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать — такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом — по имени. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры. Следует учитывать, что операции с отдельными битами реализуются гораздо менее эффективно, чем с байтами и словами, так как компилятор должен генерировать специальные коды и экономия памяти под переменные оборачивается увеличением объема кода программы. Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

В заключение изучения структур рассмотрим пример, демонстрирующий их возможности. В примере создается структура *Name*, полями которой являются фамилия и имя. Далее создаются функции ввода и вывода для полей структуры *Name*. В исполняемом коде создаются два объекта (*first*, *second*) структуры *Name* и осуществляется ввод фамилий и имен двух человек в созданные объекты структуры и дальнейший их вывод на экран. В результате выполнения программы на экран будет выведен следующий текст:

```
Программа, использующая структуры
Введите фамилию:    Иванов
Введите имя:        Иван
Введите фамилию:    Петров
Введите имя:        Петр
```

Иванов Иван
Петров Петр
Для завершения нажмите любую клавишу

```
#include<stdio.h>
#include <conio.h>
struct Name
{
    char fam[10];
    char name[10];
};
void NamePrint(struct Name a)
{
    printf("%s\t%s\n", a.fam, a.name);
}
void NameInput(struct Name *a)
{
    printf("Введите фамилию:\t");
    scanf("%10s", a–>fam);
    printf("Введите имя:\t");
    scanf("%10s", a–>name);
}
void main( )
{
    struct Name first, second;
    printf("Программа использующая структуры\n");
    NameInput(&first);
    NameInput(&second);
    NamePrint(first);
    NamePrint(second);
    printf("\nДля завершения нажмите любую клавишу");
    getch( );
}
```

2.2.2. Объединения

Объединение (*union*) представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания у него такой же, как у структуры, только вместо ключевого слова ***struct*** используется слово ***union***:

```
union [имя_типа]
{
    тип_1 элемент_1;
    ...
} [список_описателей];
```

Длина объединения равна наибольшей из длин его полей. В каждый момент времени в переменной типа объединения хранится только одно значение, и ответственность за его правильное использование лежит на программисте. Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно не требуется [8]:

```
include <iostream>
using namespace std;
int main( )
{
    enum paytype {CARD, CHECK};
    paytype ptype;
    union payment {
        char card[25];
        long check;
} info;                                    // Присваивание значений info и ptype
    switch (ptype)
```

```
    {
        case CARD: cout << "Оплата по карте:" << info.card; break;
        case CHECK: cout << "Оплата чеком:"<< info.check; break;
    }
    return 0;
}
```

Объединение часто используют в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Имя объединения можно не указывать, что позволяет обращаться к его полям непосредственно:

```
#include <iostream>
using namespace std;
int main()
{
    enum paytype {CARD, CHECK};
    struct{
        paytype ptype;
        union{
            char card[25];
            long check;
        };
    } info;
    ... /* присваивание значения info */
    switch (info.ptype)
    {
        case CARD: cout << "Оплата по карте:" << info.card; break;
        case CHECK: cout << "Оплата чеком:" << info.check; break;
    }
    return 0;
}
```

Объединения применяются также для разной интерпретации одного и того же битового представления (но, как правило, в этом случае лучше использовать явные операции преобразования типов). В качестве примера рассмотрим работу со структурой, содержащей битовые поля:

```
struct Options
{
    bool centerX:1;
    bool centerY:1;
    unsigned int shadow:2;
    unsigned int palette:4;
};
union
{
    unsigned char ch;
    Options bit;
}option = {0xC4};
cout << option.bit.palette;
option.ch&= 0xF0;                    // Наложение маски
```

По сравнению со структурами на объединения налагаются некоторые ограничения. Смысл некоторых из них станет понятен позже, при изучении объектно-ориентированного программирования [8]:

– объединение может инициализироваться только значением его первого элемента;

– объединение не может содержать битовые поля;

– объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания;

– объединение не может входить в иерархию классов.

§ 2.3. Динамические структуры данных

Любая программа включает в себя единство алгоритма (процедур, функций) и обрабатываемых данных. При этом единицами описания данных и манипулирования ими в любом языке программирования являются переменные. Формы их представления — типы данных — могут быть и заранее определенными (базовыми), и сконструированными в программе (производными). Но так или иначе переменные — это «непосредственно представленные в языке» данные. Между переменными в программе существуют неявные, непосредственно ненаблюдаемые ***логические связи***. Они могут заключаться в том, что несколько переменных используются алгоритмом для достижения определенной цели, решения частной задачи, при этом значения этих переменных будут взаимозависимы. Кроме того, между переменными существуют и ***физические связи***, которые устанавливаются либо через физическую память (связыванием переменных через указатели), либо путем включения их друг в друга. Сказанное демонстрируется на рис. 2.1 [10].

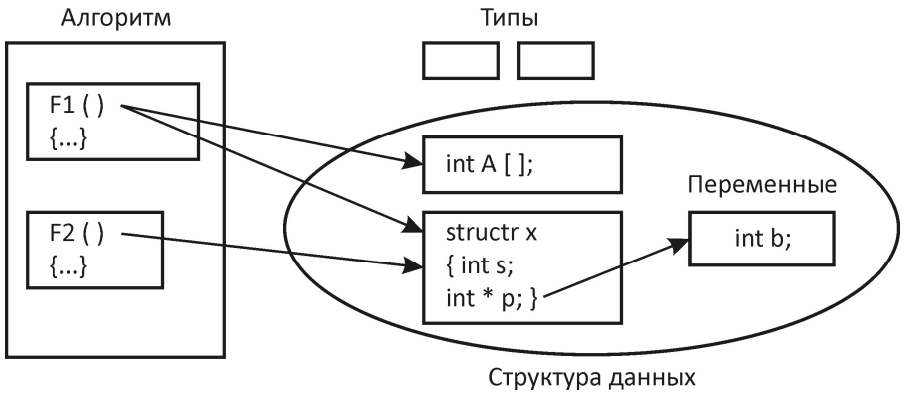


Рис. 2.1. Обобщенная структура взаимосвязи между переменными

Таким образом, ***структура данных*** — совокупность физически (типы данных) и логически (алгоритм, функции) взаимосвязанных переменных и их значений. Поскольку главной задачей любой программы является обработка данных (от способа организации данных зависят алгоритмы работы), выбор структур данных должен предшествовать созданию алгоритмов. Наиболее часто в программах используются массивы, структуры и их сочетания, например массивы структур, полями которых являются массивы и структуры. Память под данные выделяется либо на этапе компиляции (в этом случае необходимый объем должен быть известен до начала выполнения программы, т. е. задан в виде константы), либо во время выполнения программы с помощью операции *new* или функции *malloc* (необходимый объем должен быть известен до распределения памяти). В обоих случаях выделяется непрерывный участок памяти.

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, то память выделяется по мере необходимости отдельными блоками, связанными друг с другом с помощью указателей. Такой способ организации данных называется ***динамическими структурами данных***, так как их размер изменяется во время выполнения программы. Из динамических структур в программах чаще всего используются ***динамически расширяемые массивы, линейные списки, стеки, очереди*** и ***бинарные деревья***. Они различаются способами связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Динамические структуры широко применяют и для более эффективной работы сданными, размер которых известен, особенно для решения задач сортировки, поскольку упорядочивание динамических структур не требует перестановки элементов, а сводится к изменению указателей на эти элементы. Например, если в процессе выполнения программы требуется многократно упорядочивать большой массив данных, имеет смысл организовать его в виде ***линейного списка***. При решении задач поиска элемента в тех случаях, когда важна скорость, данные лучше всего представить в виде ***бинарного дерева***.

Элемент любой динамической структуры данных представляет собой структуру (*struct*), содержащую по крайней мере два поля: для хранения данных и для указателя. Полей данных и указателей может быть несколько. Поля данных могут быть любого типа: основного, составного или типа указателя. Описание простейшего элемента (компоненты, узла) выглядит следующим образом:

```
structX
{
    Data Value;                    // Тип данных Data должен быть определен ранее
    X *p;
};
```

Далее рассмотрим основные виды динамических структур данных более подробно.

2.3.1. Динамически расширяемые массивы

Все массивы, рассматриваемые ранее, были статическими, их размер и содержимое задавались во время компиляции. Однако часто возникает необходимость использования массивов с переменным количеством параметров. Для решения этой задачи наиболее целесообразно использовать массив в качестве поля структуры. В данном случае новые элементы добавляются в конец, в результате чего размер массива удлиняется по мере необходимости. Структура помимо самого массива должна иметь дополнительные поля, в которых содержится информация о его размере. Понятно, что в этом случае структура будет иметь только один описатель, который целесообразно объявить непосредственно при ее описании. Приведем пример подобной структуры:

```
struct MASSIV
{
    int nElement;          //Текущее количество элементов в массиве
    int maxElement;        //Количество элементов, для которых выделена память
    int *mass;             //Массив данных
} massiv;
```

Инициализировать непустой массив во время компиляции трудно, поэтому он создается динамически. Для начала необходимо создать первый элемент — выделить под него память, а далее добавлять последующие элементы массива. Рассмотрим это на примере функции *AddElement*( ), которой в качестве аргумента передается значение очередного элемента массива:

```
int AddElement(int NewKey)
{
    int *n;                                          //Промежуточная переменная
    if(massiv.mass==NULL)                  //Ввод первого элемента
    {
        massiv.mass=(int *)malloc(sizeof(int));
        if(massiv.mass==NULL)
            return –1;
        massiv.maxElement=1;
        massiv.nElement=0;
    }
    else                                             //Ввод последующих элементов
    {
        massiv.maxElement++;
        n=(int *)realloc(massiv.mass, (massiv.maxElement)*sizeof(int));
        if (n==NULL)
            return –1;
        massiv.mass=n;
    }
    massiv.mass[massiv.nElement]=NewKey;
    return massiv.nElement++;
}
```

Функция возвращает текущее количество элементов в массиве. Доступ к каждому элементу производится по его индексу. Промежуточная переменная *n* используется в качестве защиты. Если по какой-либо причине функция *realloc*( ) выдаст ошибку, ранее введенный массив не будет поте-

рян, так как память в этом случае выделяется для *n*, а не для самого массива. Рассмотрим пример использования динамического массива. В программе последовательно создается массив, состоящий из 10 элементов, значение которых равно удвоенному произведению их порядковых номеров. Далее элементы массива выводятся через пробел. В заголовочном файле *function.h* приведен исходный код структуры и функции *AddElement*( ). В результате выполнения программы на экран будут выведены значения:

0 2 4 6 8 10 12 14 16 18

```
#include<stdlib.h>
#include<stdio.h>
#include "function.h"

void main( )
{
    int i;
    for (i=0; i<10; i++)
            AddElement(i*2);
    for (i=0; i<massiv.nElement; i++)
            printf("%d", massiv.mass[i]);

}
```

2.3.2. Линейные списки

По своей встречаемости в типичных программах списки занимают второе место после массивов. Многие языки программирования имеют встроенные типы списков, некоторые (например, *Lisp*) даже построены на них. К сожалению, в языке Си необходимо конструировать их самостоятельно (в *C++* работа со списками поддерживается стандартными библиотеками, но и в этом случае необходимо знать их возможности и типичные применения). Самый простой способ связать множество элементов — сделать так, чтобы каждый элемент содержал ссылку на следующий. Такой список называется «*single-linked list*» — ***однонаправленным* (*односвязным, цепным*)**. Головой списка является указатель на первый элемент, а конец помечен нулевым указателем. На рис. 2.2 показан пример списка, состоящего из четырех элементов.

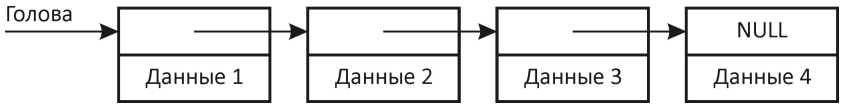


Рис. 2.2. Список из четырех элементов

Если добавить в каждый элемент вторую ссылку — на предыдущий элемент, то получится ***двунаправленный список* (*двусвязный*)**; если последний элемент связать указателем с первым — получится ***кольцевой список***. Каждый элемент списка содержит ***ключ***, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных. Например, если создается линейный список из записей, содержащих фамилию, год рождения, стаж работы и пол, любая часть записи может выступать в качестве ключа: при упорядочении списка по алфавиту ключом будет фамилия, а при поиске, к примеру, ветеранов труда ключом будет стаж. Ключи разных элементов списка могут совпадать. Над списками можно выполнять следующие ***операции:***

- начальное формирование списка (создание первого элемента);
- добавление элемента в конец списка;
- чтение элемента с заданным ключом;
- вставка элемента в заданное место списка (до или после элемента с заданным ключом);
- удаление элемента с заданным ключом;
- упорядочение списка по ключу.

Рассмотрим пример двунаправленного линейного списка. Для формирования списка и работы с ним требуется иметь по крайней мере один указатель — на начало списка. Удобно завести еще один указатель — на конец списка. Для простоты допустим, что список состоит из целых чисел *Value*, т. е. описание элемента списка выглядит следующим образом:

```
struct X{
    int Value;
    X *Next;
    X *Prev;
};
```

Инициализировать непустой список во время компиляции трудно, поэтому списки создаются динамически, аналогично динамическим массивам, рассмотренным ранее. Для начала необходимо создать первый элемент. Наиболее простой подход — выделить под него память. Для этого напишем новый код функции *FirstElement* ( ), которая создает первый элемент, получая его значение *A* в качестве параметра и выделяя под него память:

```
X *FirstElement(int A)
{
    X *FirstX;
    FirstX = (X *)malloc(sizeof(X));
    FirstX–>Value = A;
    FirstX–>Next = 0;
    FirstX–>Prev = 0;
    return FirstX;
}
```

Простейший и самый быстрый способ собрать список — это добавлять новые элементы в его начало, но это приводит к определенным неудобствам, поскольку список получается собранным наоборот: первым является последний добавленный элемент. Но в нашем случае, поскольку кроме указателя на начало имеется указатель на конец списка, нет больших сложностей в добавлении элемента в конец списка. Продемонстрируем эту возможность функцией *AddElement* ( ), которая вставляет элемент *A* в список (в данном примере **EndX* — указатель на конец списка):

```
void AddElement(X **EndX, int A)
{
    X *NewX;
    NewX = (X *)malloc(sizeof(X));          //Выделяем память
    NewX–>Value = A;
    NewX–>Next = 0;                         //Корректируем указатель на конец списка
    NewX–>Prev = *EndX;
    (*EndX)–>Next = NewX;
    *EndX = NewX;
}
```

Поиск элемента с заданным ключом (значением) осуществляется достаточно просто: перебирается весь список от начала до конца на предмет поиска определенного значения. Это демонстрируется в функции *FindElement* ( ), где ключом является значение переменной *A* (в данном примере **BeginX* — указатель на начало списка):

```
X * FindElement(X * const BeginX, int A)
{                                                   //Начинаем с указателя на начало списка
    X *Element = BeginX;
    while (Element)
    {
        if(Element–>Value == A)
            break;
        Element = Element–>Next;
    }
    return Element;
}
```

Более сложно решается задача вставки элемента в заданное место списка (до или после элемента с заданным ключом). После вставки в этом случае необходимо обновить все связи между элементами в списке. Для выполнения данной операции удобно первоначально воспользоваться функцией поиска, представленной ранее. Рассмотрим алгоритм решения этой задачи на примере функции *InsertElement* ( ), которая вставляет элемент *A* после заданного элемента *Key*:

```
X *InsertElement(X *const BeginX, X **EndX, int Key, int A)
{
    if(X *KeyX = FindElement(BeginX, Key))        //Если Key найден
    {
        X *Element;
        Element = (X *)malloc(sizeof(X));         //Выделяем память
        Element–>Value = A;                       //Присваиваем значение A
                                                  // Связываем новый элемент с последующим
        Element–>Next = KeyX–>Next;               // Связываем новый элемент с предыдущим
        Element–>Prev = KeyX;                     // Связываем предыдущий элемент с новым
        KeyX–>Next = Element;                     // Связываем последующий элемент с новым
        if (KeyX != *EndX)
            (Element–>Next)–>Prev = Element;      // Или обновляем указатель на конец списка,
                                                  // если вставка в конец
        else
            *EndX = Element;
        return Element;
    }
    return 0;
}
```

Задача удаления заданного элемента близка к задаче вставки. В этом случае так же необходимо первоначально найти элемент, затем его удалить и обновить связи. Следует учесть, что технология удаления зависит от местоположения элемента в списке. Исходный код такой задачи демонстрируется функцией *RemoveElement* ( ):

```
bool RemoveElement(X **BeginX, X **EndX, int Key)
{
    if(X *KeyX = FindElement(*BeginX, Key))       //Если Key найден
    {
        if (KeyX == *BeginX)                      //Если элемент в начале списка,
        {                                         //то корректировка указателя на начало списка
            *BeginX = (*BeginX)–>Next;
            (*BeginX)–>Prev =0;
        }
        else if (KeyX == *EndX)                   //Если элемент в конце списка,
        {                                         //то корректировка указателя на конец списка
            *EndX = (*EndX)–>Prev;
            (*EndX)–>Next =0;
        }
        else
        {                                         //Если элемент в середине списка,
                                                  //то корректировка указателей на предыдущий
                                                  // и последующий элементы
            (KeyX–>Prev)–>Next = KeyX–>Next;
            (KeyX–>Next)–>Prev = KeyX–>Prev;
        }
        free(KeyX);                               // Удаляем элемент
        return true;
    }
    return false;
}
```

Сортировка связанного списка заключается в изменении связей между элементами. Алгоритм состоит в том, что исходный список просматривается и каждый новый элемент вставляется в список на место, определяемое значением его ключа. Решение такой задачи демонстрируется функцией *SortElement* ( ):

```
void SortElement(X **BeginX, X **EndX, int A)
{
    X *NewElement;
    NewElement = (X *)malloc(sizeof(X));                    // Выделяем память
    NewElement–>Value = A;
    X * Element = *BeginX;                                  // Встаем на начало списка
    while (Element)                                         // Просматриваем
    {
        if (A<Element–>Value)                               //Если значение меньше текущего,
        {                                                   // заносим перед текущим элементом
            NewElement–>Next = Element;
            if (Element == *BeginX)
            {                                               // в начало списка
                NewElement–>Prev =0;
                *BeginX = NewElement;
            }
            else
            {                                               // или в середину списка
                (Element–>Prev)–>Next = NewElement;
                NewElement–>Prev = Element–>Prev;
            }
            Element–>Prev = NewElement;
            return;
        }
        Element = Element–>Next;
    }
    NewElement–>Next =0;                                    //Заносим в конец списка
    NewElement–>Prev = *EndX;
    (*EndX)–>Next = NewElement;
    *EndX = NewElement;
}
```

В заключение рассмотрения линейных списков приведем пример использования функций. В этом примере первоначально создается один элемент со значением 0; далее добавляются в конец списка элементы со значениями от 3 до 7; после этого вставляется элемент со значением 1 после элемента со значением 0; удаляется элемент со значением 6 и затем вставляется элемент со значением 2 согласно сортировке по возрастанию. В заголовочном файле function.h приведен исходный код всех перечисленных функций. В результате выполнения программы на экран будут выведены значения

0 1 2 3 4 5 7

```
#include<stdio.h>
#include <stdlib.h>
#include "function.h"
struct X
{
    int Value;
    X *Next;
    X *Prev;
};
int main( )
{
    X *first = FirstElement(0);          // Формирование первого элемента списка
    X *end = first;                      //Конец списка на его начале
                                         // Добавление в конец списка четырех элементов 3, 4, 5, 6 и 7:
```

```
    for (int i = 3; i<8; i++)
        AddElement(&end, i);                    // Вставка элемента 1 после элемента 0:
    InsertElement(first, &end, 0, 1);           // Удаление элемента 6:
    if(!RemoveElement (&first, &end, 6))
        printf("Don't find Element\n");         // Вставка элемента 2 по сортировке
    SortElement(&first, &end, 2);               // Вывод списка на экран
    X *element = first;
    while (element)
    {
        printf("%d", element–>Value);
        element = element–>Next;
    }
return 0;
}
```

2.3.3. Стеки

Операции вставки и извлечения элементов из линейного списка адресные — они используют номер элемента (индекс). Если ограничить возможности изменения последовательности только ее концами, то получим структуры данных, называемые ***стеком*** и ***очередью***. **Стек**—последовательность элементов, включение элементов в которую и исключение из которой производятся только с одного конца. Начало последовательности называется дном стека, конец последовательности, в который добавляются элементы и из которых они исключаются, — вершиной стека. Операция добавления нового элемента (запись в стек) имеет общепринятое название *Push* (погрузить), операция исключения — *Pop* (звук выстрела). Операции *Push* и *Pop* безадресные: для их выполнения никакой дополнительной информации о месте размещения элементов не требуется. Таким образом, стек — это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца. Стек обычно представляется списком с одной дополнительной переменной, которая указывает на последний элемент последовательности в вершине стека — указатель стека (рис. 2.3) [10].

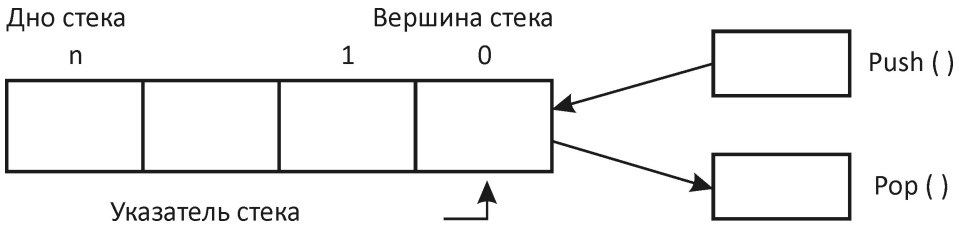


Рис. 2.3. Структура стека

Исключительная популярность стека в программировании объясняется тем, что при заданной последовательности записи элементов в стек (например, *A–B–C*) извлечение их происходит в обратном порядке (*C–B–A*). А именно эта последовательность действий соответствует таким понятиям, как вложенность вызовов функций, вложенность определений конструкций языка и т. д. Следовательно, везде, где речь идет о вложенности процессов, структур, определений, механизмом реализации такой вложенности является стек. Для способа хранения данных в стеке имеется общепринятый термин — ***LIFO*** (*last in — first out*, «последний пришел — первый ушел»). Другое важное свойство стека — относительная адресация его элементов. На самом деле для элемента, сохраненного в стеке, важно не его абсолютное положение в последовательности, а положение относительно вершины стека или его указателя, которое отражает «историю» его заполнения. Поэтому адресация элементов стека происходит относительно текущего значения указателя стека. В архитектуре практически всех компьютеров используется аппаратный стек. Он представляет собой обычную область внутренней (оперативной) памяти компьютера, с которой работает специальный регистр — указатель стека. С его помощью процессор выполняет операции *Push* и *Pop* по сохранению и восстановлению из стека байтов и машинных слов различной размерности.

Рассмотрим пример стека. Для его формирования и работы достаточно иметь указатель на вершину стека. Как и ранее, допустим, что стек состоит из целых чисел *Value*, т. е. описание элемента стека выглядит следующим образом:

```
struct X
{
    int Value;
    X *p;
};
```

Первоначально создадим первый элемент стека с помощью функции *FirstElement* ( ), исходный код которой практически аналогичен приведенному ранее при инициализации списка. Функция создает первый элемент, получая его значение *A* в качестве параметра и выделяя под него память:

```
X * FirstElement(int A)
{
    X *FirstX;
    FirstX = (X *) malloc (sizeof (X));
    FirstX –> Value = A;
    FirstX –> p = 0;
    return FirstX;
}
```

Следующая функция *Push* ( )заносит последующие элементы *A* в стек. В ней **Top* — указатель на вершину стека:

```
void Push(X **Top, int A)
{

    X *ElementX;
    ElementX = (X *) malloc (sizeof (X));
    ElementX–>Value = A;
    ElementX–>p = *Top;
    *Top = ElementX;
}
```

Функция *Pop* ( ) осуществляет выборку элементов из стека:

```
int Pop(X **Top)
{
    int n = (*Top)–>Value;
    X *ElementX = *Top;
    *Top = (*Top) –> p;
    free (ElementX);
    return n;
}
```

Рассмотрим пример использования функций. Первоначально создается один элемент со значением 0; далее в стек помещаются элементы со значениями от 1 до 7; после этого элементы последовательно извлекаются из стека и выводятся на экран. В заголовочном файле *function.h* приведен исходный код всех перечисленных функций. В результате выполнения программы на экран будут выведены значения

7 6 5 4 3 2 1 0

```
#include<stdio.h>
#include <stdlib.h>
#include "function.h"
struct X
{
    int Value;
    X *p;
};
```

```
int main()
{
    X *top = FirstElement(0);
    for (int i = 1; i<8; i++)
            Push(&top, i);
    while (top)
            printf("%d", Pop(&top));
    return 0;
}
```

2.3.4. Очереди

Очередь — частный случай однонаправленного списка, добавление в который производится с одного конца, а исключение — с другого. Для способа хранения данных в очереди есть общепринятый термин — ***FIFO*** (*first in—first out*, «первый пришел — первый ушел»).

Простейший способ представления очереди последовательностью, размещенной от начала списка, не совсем удобен, поскольку при извлечении из очереди первого элемента все последующие придется постоянно передвигать к началу. Альтернатива: у очереди должно быть два указателя — на ее начало в списке и на ее конец. По мере постановки элементов в очередь ее конец будет продвигаться к концу списка, то же самое будет происходить с началом при исключении элементов. Выход из создавшегося положения — «зациклить» очередь, т. е. считать, что за последним элементом списка следует опять первый. Подобный способ организации очереди иногда называют циклическим буфером (рис. 2.4).

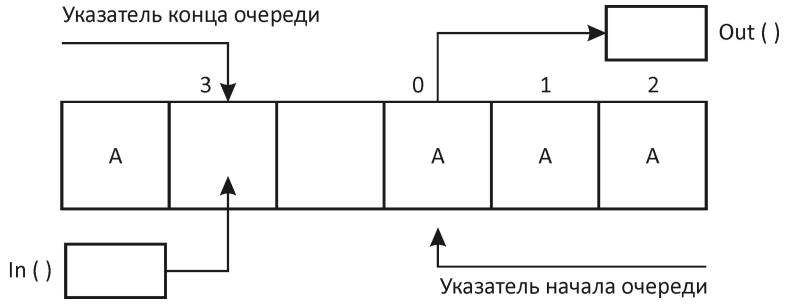


Рис. 2.4. Структура очереди

В отличие от стека указатель на конец очереди ссылается не на последний занятый, а на первый свободный элемент списка. В программировании очереди наиболее часто применяются в моделировании, диспетчеризации задач, буферизированном вводе / выводе.

Рассмотрим пример очереди. Как и ранее, допустим, что очередь состоит из целых чисел *Value*, т. е. описание элемента очереди выглядит следующим образом:

```
struct X
{
    int Value;
    X *p;
};
```

Первоначально создадим первый элемент очереди с помощью функции *FirstElement* ( ), исходный код которой аналогичен приведенному ранее при инициализации стека. Функция создает первый элемент, получая его значение *A* в качестве параметра и выделяя под него память:

```
X * FirstElement(int A)
{
    X *FirstX;
    FirstX = (X *)malloc(sizeof(X));
    FirstX–>Value = A;
    FirstX–>p = 0;
    return FirstX;
}
```

Следующая функция *In*( ) заносит последующие элементы A в конец очереди. В ней **EndX* — указатель на конец очереди:

```
void In(X **EndX, int A)
{
    X *ElementX;
    ElementX = (X *)malloc(sizeof(X));
    ElementX–>Value = A;
    ElementX–>p = 0;
    (*EndX)–>p = ElementX;
    *EndX = ElementX;
}
```

Функция *Out*( ) осуществляет выборку элементов из начала очереди. В ней **BeginX* — указатель на начало очереди:

```
int Out(X **BeginX)
{
    int n = (*BeginX)–>Value;
    X *ElementX = *BeginX;
    *BeginX = (*BeginX)–>p;
    free (ElementX);
    return n;
}
```

Рассмотрим пример использования функций. Первоначально создается один элемент со значением 0; далее в очередь помещаются элементы со значениями от 1 до 7; после этого элементы последовательно извлекаются из начала очереди и выводятся на экран. В заголовочном файле *function.h* приведен исходный код всех перечисленных функций. В результате выполнения программы на экран будут выведены значения

0 1 2 3 4 5 6 7

```
#include<stdio.h>
#include <stdlib.h>
#include "function.h"
struct X
{
    int Value;
    X *p;
};

int main( )
{
    X *first = FirstElement(0);
    X *end = first;
    for (int i = 1; i<8; i++)
            In(&end, i);
    while (first)
            printf("%d", Out(&first));
    return 0;
}
```

2.3.5. Бинарные деревья

Бинарное дерево — это иерархическая динамическая структура данных, состоящая из узлов (вершин), каждый из которых содержит, кроме данных, не более двух ссылок на другие бинарные деревья. На каждый узел имеется только одна ссылка. Начальный узел называется ***корнем*** дерева. Узел, не имеющий поддеревьев, называется ***листом***, исходящие узлы — ***предками***, восходящие — ***потомками***. ***Высота дерева*** определяется количеством уровней, на которых располагаются его узлы.

Есть много типов деревьев, которые отражают различные сложные структуры, например ***деревья синтаксического разбора*** (*parse trees*), хранящие синтаксис предложения или программы, либо ***генеалогические деревья***, описывающие родственные связи. Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется ***деревом поиска***. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму количества узлов. На рис.2.5 [8] приведен пример ***бинарного дерева поиска.***

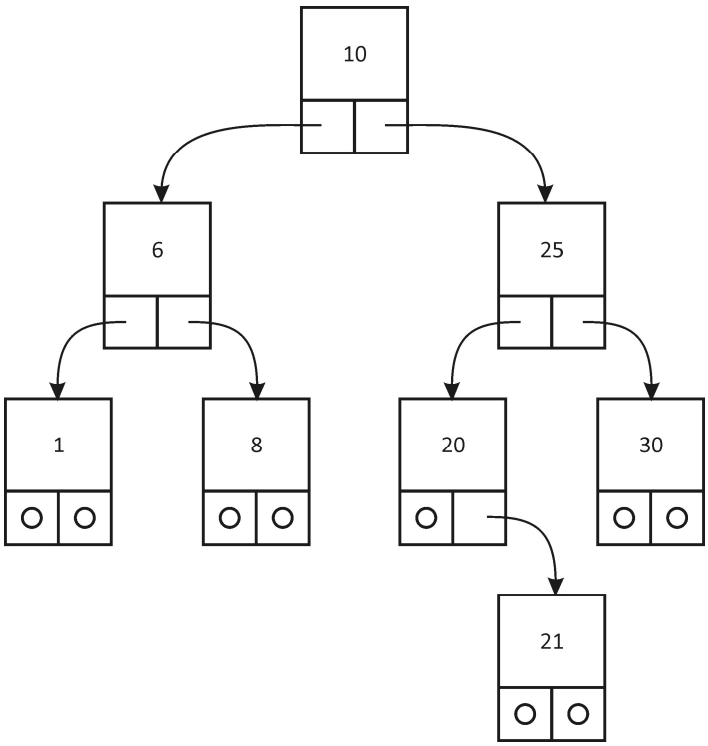


Рис. 2.5. Бинарное дерево

Для бинарных деревьев определены операции [8]:

- включения узла в дерево;
- поиска по дереву;
- обхода дерева;
- удаления узла.

С бинарными деревьями удобнее всего работать с помощью рекурсивных алгоритмов (понятие рекурсии будет рассмотрено далее). Но для каждого рекурсивного алгоритма можно создать его нерекурсивный эквивалент, который, правда, несколько усложняет написание программного кода.

Рассмотрим пример исходного кода, реализующего бинарное дерево, представленное на рис. 2.5. Для формирования дерева и работы с ним удобно иметь два указателя — на левое и правое поддерево. Как и ранее, значения состоят из целых чисел *Value*. Таким образом, получаем следующее описание элемента дерева:

```
struct X
{
    int Value;
    X *Left;
    X *Right;
};
```

Первоначально создадим первый элемент дерева с помощью функции *FirstElement*( ), исходный код которой практически аналогичен приведенному ранее при инициализации списка. Функция создает первый элемент, получая его значение *A* в качестве параметра и выделяя под него память:

```
X * FirstElement(int A)
{
    X *FirstX;
    FirstX = (X *)malloc(sizeof(X));
    FirstX–>Value = A;
    FirstX–>Left = 0;
    FirstX–>Right = 0;
    return FirstX;
}
```

Далее рассмотрим исходный код функции, которая осуществляет поиск элемента с заданным значением. Если искомого элемента нет — функция включает элемент в соответствующее место дерева. Возвращаемым значением является указатель на элемент. Для включения элемента необходимо помнить пройденный по дереву путь на один шаг назад и знать, выполняется ли включение нового элемента в левое или правое поддерево его предка:

```
X * SearchAndInsertElement(X *Root, int A)
{
    X *ElementX = Root;
    X *Prev;
    bool Found = false;
    while (ElementX &&!Found)
    {                                          // Поиск заданного элемента
        Prev = ElementX;
        if (A == ElementX–>Value)
            Found = true;
        else if (A< ElementX–>Value)
            ElementX = ElementX–>Left;
        else
            ElementX = ElementX–>Right;
    }
    if (Found)
        return ElementX;
    X *NewX;                                   // Создание нового узла
    NewX = (X *)malloc(sizeof(X));
    NewX–>Value = A;
    NewX–>Left = 0;
    NewX–>Right = 0;
    if (A<Prev–>Value)                         // Присоединение к левому поддереву
        Prev–>Left = NewX;
    else                                       // Иначе присоединение к правому поддереву
        Prev–>Right = NewX;
    return NewX;
}
```

Для обхода дерева удобнее всего воспользоваться рекурсивной функцией. Исходный код такой функции будет приведен при изучении соответствующей темы. В данном примере для вывода воспользуемся обычной функцией. В этом случае все передвижение по дереву приходится делать вручную, от его корня:

```
void PrintTree(X *Root)
{
    X *p=Root;                                 //Корень дерева
    printf("Main Root\t%d\n", p–>Value);       //Левое поддерево
    p=p–>Left;
    printf("Left Root\t%d\n", p–>Value);
    p=p–>Left;
    printf("Left Tree:\t%d", p–>Value);
```

```
    p=Root;
    p=p–>Left;
    p=p–>Right;
    printf("%d\n", p–>Value);                    //Правое поддерево
    p=Root;
    p=p–>Right;
    printf("Right Root\t%d\n", p–>Value);
    p=p–>Left;
    printf("Right Tree:\t%d", p–>Value);
    p=p–>Right;
    printf("%d", p–>Value);
    p=Root;
    p=p–>Right;
    p=p–>Right;
    printf("%d\n", p–>Value);
}
```

Рассмотрим пример использования функций. Первоначально создается массив *A* со значениями, указанными на рис. 2.5. Первый элемент массива равен значению главного корня дерева, остальные расположены в случайном порядке. Первоначально создается корень, далее расставляются остальные значения согласно правилу: все ключи левого поддерева меньше ключа текущего узла, а все ключи правого поддерева — больше. Далее ключи дерева выводятся на экран. В заголовочном файле *function.h* приведен исходный код всех перечисленных функций. В результате выполнения программы на экран будут выведены значения:

```
MainRoot  10
Left Root   6
Left Tree   1  8
Right Root 25
Right Tree  20  21  30
```

```
int main( )
{
    int A [ ] = {10, 25, 20, 6, 21, 8, 1, 30};
    X *root = FirstElement(A[0]);
    for (int i = 1; i<8; i++)
            SearchAndInsertElement(root, A[i]);
    PrintTree(root);
    return 0;
}
```

§ 2.4. Рекурсивные функции

Функция называется ***рекурсивной,*** если во время ее обработки возникает ее повторный вызов либо непосредственно, либо косвенно, путем цепочки вызовов других функций. Таким образом, **рекурсия** — это такой способ организации процесса вычислений, при котором функция в ходе выполнения ее операторов обращается сама к себе, прямо или косвенно. Рекурсия бывает ***линейной***, когда определение объекта включает в себя единственный аналогичный объект, или ***ветвящейся***, когда таких включаемых объектов много.

Рекурсивная форма организации алгоритма обычно дает более компактный текст программы, чем другие способы решения этой же задачи, но требует дополнительных затрат оперативной памяти для размещения данных и времени для организации рекурсивных вызовов функции. Таким образом, благодаря рекурсии уменьшается исходный код программы, но при этом нет выигрыша ни в использовании памяти, ни в быстродействии. Рассмотрим трудоемкость рекурсивных алгоритмов — зависимость времени их выполнения от размерности входных данных. В рекурсивных функциях размерность входных данных определяет глубину рекурсии. Если имеется ветвящаяся

рекурсия — цикл из m повторений, то при глубине рекурсии N общее количество рекурсивных вызовов будет порядка m^N, поскольку с каждым шагом рекурсии оно увеличивается в m раз. Таким образом, трудоемкость рекурсивных алгоритмов значительно превышает трудоемкость рассмотренных ранее алгоритмов сортировки и поиска.

При выполнении правильно организованной рекурсивной функции осуществляется ее многократный последовательный вызов. При этом система:

1) сохраняет в стеке значения всех локальных переменных функции и ее параметры для всех предыдущих вызовов;

2) выделяет место в оперативной памяти для локальных переменных очередного вызова функции.

Для внутренних переменных, объявленных с классом памяти ***extern*** или ***static***, новые области памяти при каждом рекурсивном вызове не выделяются, выделенная им память сохраняется в течение всего времени выполнения программы. Сохраненные в стеке значения могут использоваться после получения результата от последующего вызова функции. Вызов рекурсивной функции производится до тех пор, пока не будет получено ***тривиальное решение*** задачи, т. е. будет получен результат в виде конкретного значения без вызова рекурсивной функции. Правильно организованная рекурсивная функция обязательно должна иметь этот вариант решения для завершения рекурсивных вызовов. По завершении рекурсивных вызовов функции и после получения результатов ее выполнения управление передается следующему за рекурсивной функцией оператору.

Компилятор не ограничивает количество рекурсивных вызовов одной функции, при этом для каждого вызова функции в памяти выделяется место для формальных параметров и локальных переменных ***auto*** и ***register***. Перечисленные переменные образуют группу (фрейм стека). Стек«помнит историю» рекурсивных вызовов в виде последовательности (цепочки) таких фреймов. Программа в каждый конкретный момент работает с последним вызовом и с последним фреймом. При завершении рекурсии программа возвращается к предыдущей версии рекурсивной функции и к предыдущему фрейму в стеке. Некоторые операционные системы накладывают ограничения на количество вызовов, так как это может вызвать переполнение стека.

Принцип программирования рекурсивных функций имеет много общего с методом математической индукции: рекурсивная функция представляет собой переход из n-го в $(n + 1)$-ecостояние некоторого процесса. Если этот переход корректен, т. е. соблюдение некоторых условий на входе функции приводит к их соблюдению на выходе (в рекурсивном вызове), то эти условия будут соблюдаться во всей цепочке состояний. Таким образом, самое важное в определении рекурсии — выделить те условия (инварианты), которые соблюдаются (сохраняются) во всех точках процесса, и обеспечить их справедливость от входа в рекурсивную функцию до ее рекурсивного вызова. При этом не рекомендуется рассматривать следующий или предыдущий шаг рекурсии.

С помощью рекурсии можно решать различные задачи. В некоторых случаях (вычисление факториала; определение сумм и произведений членов ряда, вычисляемых до получения заданной точности вычислений; формирование и использование рекурсивно-определяемых связанных структур данных, например очередей, списков, деревьев) рекурсивное построение алгоритма является наиболее естественным и экономичным путем решения задачи. Рассмотрим примеры использования рекурсивных функций [3].

1. Функция, которая выводит некоторую фразу несколько раз.

Принцип работы функции *PrintString*( ) состоит в том, что при $k > 0$ она выводит фразу из строки *str* и вызывает функцию *PrintString*( ) для значения фактического параметра $(k – 1)$. Так, например, для $k = 5$ при втором вызове *PrintString*( ) фактический параметр $k = 4$, при следующем $k = 3$ и т. д. до $k = 0$. При вызове функции с $k = 0$ вывод не производится и последовательно осуществляется возврат обратно во все вызванные функции *PrintString*( ) и выход в главную функцию:

```
void PrintString(char *str, int k )
{
    if (k> 0)                          //Проверка на тривиальность решения
    {
        printf ("%s", str);            // Вывод текста
        PrintString (str, k – 1);      // Рекурсивный вызов функции
    }
}
```

Конечно, можно реализовать программный код и без рекурсии. В этом случае функция *PrintString*( ) будет выглядеть следующим образом:

```
void PrintString(char *str, int k)
{
    for(int i=0; i<k; i++)
            printf ("%s", str);
}
```

В обоих случаях для исполнения кода достаточно вызвать функцию *PrintString* ( ), передав ей некоторую строку, например «*I love C++*!*n*», и количество раз для вывода текста. В результате выполнения программы для *k* = 5 на экран будет выведено:

```
I love C++!
I love C++!
I love C++!
I love C++!
I love C++!
```

```
#include<stdio.h>
void main( )
{
    PrintString("I love C++!\n", 5);
}
```

2. Функция, которая считает факториал.

Рекурсивная функция *Fact* ( ) вычисляет факториал своего аргумента. При *n* = 0 или *n* = 1 решение тривиально (результат = 1). Оно используется для начала обратного хода рекурсивных вычислений:

```
long Fact(int n)
{
    if ( n < 0 )
    {
        printf ("Значение %d ошибочно \n", n);
        exit(1);
    }
    else if (n == 0 || n == 1)
        return 1;                          //Проверка на тривиальность решения
    else
        return ( n * Fact(n– 1));          // Рекурсивный вызов функции
}
```

В этом случае также легко реализовать программный код без рекурсии. Исходный код нерекурсивной функции *Fact*( ) будет выглядеть следующим образом:

```
long Fact(int n)
{
    for (int f = 1; n!=0; n– –)
        f *= n;
    return f;
}
```

В обоих случаях для исполнения кода достаточно вызвать функцию *Fact* ( ), передав ей число для вычисления факториала. В результате выполнения программы для факториала 4 на экран будет выведено:

```
input N
4
4 != 24
```

```
#include <stdio.h>
void main ( )
{
    int N;
    printf ("Input N:\n");
    scanf ("%d", &N);
    printf("%d ! = %ld\n", N, Fact(N));
}
```

3. Функция, которая считает сумму элементов массива.

Рекурсивная функция *Sum* ( ) вычисляет сумму элементов массива *A* [ ] для индексов, изменяющихся от *n*0 до *nMax*. При *n*0 < *nMax* возвращаемое значение определяется в виде суммы *A* [*nMax*] — элемента массива и результата вызова функции *Sum* ( ) с фактическим параметром — номером последнего суммируемого элемента, равным *nMax* – 1. Рекурсивный вызов функции производится в операторе *return* в виде

```
return (A[nMax] + Sum(A, n0, nMax– 1));
```

При *n*0 = *nMax* решение тривиально, возвращаемый результат =*A* [*n*0], начинается обратный ход рекурсивных вычислений: возврат из вызванных функций результатов суммирования. Для исполнения кода в данном примере динамически создается массив, состоящий из *N* элементов, который заполняется цифрами от 1 до *N*. Память выделяется функцией *malloc* ( ). Вызов рекурсивной функции производится в функции *printf* ( ). В результате выполнения программы для массива, состоящего из четырех элементов, на экран будет выведено:

```
input N
4
Sum = 10
```

```
#include <stdio.h>
#include<stdlib.h>

        // Рекурсивная функция для вычисления суммы элементов массива:
int Sum (int A [ ], int n0, int nMax)
{
    if (n0 == nMax )
        return (A[n0]);                                  // Тривиальное решение
                                                         // Рекурсивный вызов
    else
        return (A[nMax] + Sum(A, n0, nMax – 1));
}
void main ( )
{
    int *mass;
    int N;
    printf ("Input N:\n");
    scanf ("%d", &N);                                    // Ввод количества элементов массива
    mass=(int *)malloc(N*sizeof(int));
    for (int i=0; i<N; i++)
        mass[i]=i+1;
    printf ("Sum =%d\n", Sum (mass, 0, N – 1));          // Вызов функции
    free(mass);
}
```

4. Функция, которая производит обход бинарного дерева.

Ранее при рассмотрении темы «Бинарные деревья» было создано дерево бинарного поиска, обход которого производился обычной функцией *PrintTree* ( ). Использование в данном случае рекурсивной функции существенно упростит программный код, так как дерево само является рекурсивной структурой данных (каждое поддерево также является деревом). В общем виде функцию обхода всех узлов дерева можно описать следующим образом [8]:

```
function NodeWay (дерево)
{
    NodeWay (левое поддерево)
    посещение корня
    NodeWay (правое поддерево)
}
```

Можно обходить дерево и в другом порядке, например: сначала корень, потом поддеревья. Но приведенная функция позволяет получить на выходе отсортированную последовательность ключей, поскольку сначала посещаются вершины с меньшими ключами, расположенные в левом поддереве. Таким образом, деревья поиска можно применять для сортировки значений. При обходе дерева узлы не удаляются.

Рассмотрим пример исходного кода рекурсивной функции обхода дерева, пример которого приведен в разделе «Бинарные деревья». Для этого создадим рекурсивную функцию *PrintTreeRecurs* (). Вторым параметром в нее передается целая переменная, определяющая, на каком уровне находится узел. Корень находится на уровне 0. Следует отметить, что функция обхода дерева длиной всего в несколько строк может напечатать дерево любого размера — важно только следить, чтобы рекурсивные вызовы не переполнили стек. В результате выполнения программы (для выполнения просто замените в приведенном коде функцию *PrintTree* на *PrintTreeRecurs*) дерево печатается по горизонтали так, что корень находится слева. Если повернуть ответ на 90° вправо и зеркально перевернуть, то получится схема, аналогичная дереву, приведенному на рис. 2.5. Следует обратить внимание, что все элементы выводятся в порядке возрастания. Перед значением узла для имитации структуры дерева выводятся символы табуляции в количестве, пропорциональном уровню узла:

```
        1
    6
        8
10
        20
            21
    25
        30

void PrintTreeRecurs(X *p, int Level)
{
    if (p)
    {
        PrintTreeRecurs(p–>Left, Level+ 1);          // Вывод левого поддерева
        for (int i = 0; i<Level; i++)
            printf("\t");
        printf("%d\n",p–>Value);                     // Вывод корня
        PrintTreeRecurs(p–>Right, Level +1);         // Вывод правого поддерева
    }
}
```

Глава 3
Объектно-ориентированное программирование

§ 3.1. Классы и объекты

С увеличением размера программы удерживать в памяти архитектуру и детали проекта становится довольно сложно. Возникает настоятельная необходимость структурировать информацию, выделять главное и отбрасывать несущественное (повышать степень абстракции).

Первым шагом к повышению абстракции было ***использование функций***, позволяющее после написания и отладки функции отвлечься от деталей ее реализации, поскольку для вызова функции требуется знать только ее интерфейс. Если глобальные переменные не используются, интерфейс полностью определяется заголовком функции.

Следующий шаг — ***описание собственных типов данных,*** позволяющих структурировать и группировать информацию, представляя ее в более естественном виде. Например, можно представить с помощью одной структуры все разнородные сведения, относящиеся к одному виду товара на складе.

Для работы с собственными типами данных требуются специальные функции. Естественно, следует сгруппировать их с описанием этих типов данных в одном месте программы, а также по возможности отделить от остальных ее частей. При этом для использования этих типов и функций не требуется полного знания того, как именно они написаны — необходимы только описания интерфейсов. Таким образом, дальнейшим развитием структуризации программы явилось ***объединение в модули*** описаний типов данных и функций, предназначенных для работы с ними.

Все три описанных метода повышения абстракции созданы для упрощения структуры программы, что позволяет управлять большим объемом информации и, следовательно, успешно отлаживать более сложные проекты. Введение понятия ***класса*** является естественным развитием идей модульности. В классе структуры данных и функции их обработки объединяются. Класс используется только через его интерфейс — детали реализации для пользователя класса несущественны.

Класс является типом данных, определяемым программистом. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы.

Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса за интерфейсом. Интерфейсом класса являются заголовки его методов. Таким образом, класс как модель объекта реального мира является черным ящиком, замкнутым по отношению к внешнему миру.

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются ***полями*** (по аналогии с полями структуры), а функции класса — ***методами***. Поля и методы называются ***элементами класса***. В общем случае описание класса представляется в следующей форме:

```
class имя [:базовый список]
{
    [private:]
        описание скрытых элементов
    protected:
        описание защищенных элементов
    public:
        описание доступных элементов
};
```

Скобки [и], как и выше по тексту, обозначают необязательные элементы. *Имя* класса определяет уникальный идентификатор внутри данного блока. *Базовый список* определяет классы, из которых данный класс наследует элементы и методы. В базовом списке также могут

определяться спецификаторы доступа к наследуемым элементам. При описании класса создается уникальный тип, идентифицируемый именем класса. Спецификаторы доступа *private*, *protected* и *public* управляют видимостью элементов класса. Элементы, описанные после служебного слова private, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. *Protected* практически аналогичен *private*, но по-другому реализуется при наследовании; элементы, объявленные как *public*, открыты извне класса. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций *private* и *public* — порядок их следования значения не имеет.

Элементы типа «класс» могут использоваться как аргументы функций и как возвращаемые функциями значения. *C++* позволяет переопределять (*overloading*) стандартные функции и операции, когда они используются с элементами определенного класса. *C++* поддерживает механизмы, при которых функция или оператор с одним и тем же именем может вызываться в разных задачах в зависимости от типа и числа аргументов.

Список членов класса есть последовательность элементов любых типов, включая перечисления, битовые поля и другие классы, а также определения функций. Для элементов могут задаваться классы памяти и спецификации доступа. Классы памяти *auto*, *extern* и*register* недоступны. Члены класса могут быть определены со спецификацией *static*.

Конкретные переменные типа «класс» называются ***объектами***. Время жизни и видимость объектов зависит от вида и места их описания и подчиняется общим правилам *C++*, например:

```
class X {...};
…
X x;                    // Элемент класса X с параметрами по умолчанию
X rt(200, 300);         // Объект с явной инициализацией
X mas[100];             // Массив объектов с параметрами по умолчанию
X &xr;                  // Ссылка на объект класса X
X *xptr;                // Указатель на объект класса X
```

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор (деструкторы будут описаны далее).

Функции типа *inline*.Если тело метода определено внутри класса, то он является встроенным (*inline*). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, то сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости (::). В общем случае описание метода представляется в следующей форме:

```
возвр_тип имя_класса::имя_метода(параметры)
{
   тело метода
}
```

Пример:

```
void X::draw(int x, int y, int scale, int position)
{
    /*тело метода*/
}
```

Альтернативным является случай, когда метод определяется внутри класса. В этом случае метод имеет тип *inline* и его вызов заменяется компилятором на непосредственную подстановку тела метода.

Пример:

```
class X
{
public:
    char func (void)
    {return i;}
    char *i;
};
```

Метод *func* в этом случае по умолчанию имеет тип *inline*. Метод, описываемый вне класса, также может быть определен как *inline* с помощью соответствующего ключевого слова.

Пример:
```
inline void X::func(void)
{
    return i;
}
```

Данная запись эквивалентна предыдущему определению. Следует отметить, что выполнение непосредственной подстановки текста функции может оказаться не всегда возможным. Поэтому описание *inline* в общем случае необходимо рассматривать как указание компилятору по возможности осуществить текстовую подстановку.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операция **.** (точка) при обращении к элементу через имя объекта и операция –> при обращении через указатель.

Пример:
```
X x, mas[200], *ps;
…
x.Surname = "Петров";
mas[133].date = 119;
ps–> code = 543;
```

Обратиться таким образом можно только к элементам со спецификатором *public*. Получить или изменить значения элементов со спецификатором *private* или *protected* можно только через обращение к соответствующим методам.

К элементам классов можно обращаться с помощью ***указателей***. Для этого определены операции **.*** и **–>***. Указатели на поля и методы класса определяются по-разному. Формат указателя на метод класса:

возвр_тип (*имя_класса*::**имя_указателя*)(*параметры*);

Перечислим правила использования указателей на методы классов:

– указателю на метод можно присваивать только адреса методов, имеющих соответствующий заголовок;

– нельзя определить указатель на статический метод класса;

– нельзя преобразовать указатель на метод в указатель на обычную функцию, не являющуюся элементом класса.

Как и указатели на обычные функции, указатели на методы используются в том случае, когда возникает необходимость вызвать метод, имя которого неизвестно. Однако в отличие указателя на переменную или обычную функцию указатель на метод не ссылается на определенный адрес памяти. Он больше похож на индекс в массиве, поскольку задает смещение. Конкретный адрес в памяти получается путем сочетания указателя на метод с указателем на определенный объект. Формат указателя на поле класса:

тип_данных(*имя_класса*::**имя_указателя*);

В определение указателя можно включить его инициализацию в форме:

&имя_класса::**имя_поля*; // Поле должно быть *public*

Следует обратить внимание на то, что указатели на поля классов не являются обычными указателями. Ведь при присваивании им значений они не ссылаются на конкретный адрес памяти, поскольку память выделяется не под классы, а под объекты классов.

Можно создать ***константный объект,*** значения полей которого изменять запрещается. К нему должны применяться только ***константные методы***.

Пример:

```
class X
{
    ...
    const int get_boy ( ) {return boy};
};

const X Dn(0, 0);                    // Константный объект
cout << Dn.get_boy( );
```

Свойства константного метода:

– объявляется с ключевым словом const после списка параметров;

– не может изменять значения полей класса;

– может вызывать только константные методы;

– может вызываться для любых (не только константных) объектов.

Рекомендуется описывать как константные те методы, которые предназначены для получения значений полей.

Классы могут быть ***глобальными*** (объявленными вне любого блока) и ***локальными*** (объявленными внутри блока, например функции или другого класса). Рассмотрим некоторые особенности локального класса:

– внутри локального класса можно использовать типы, статические (*static*) и внешние (*extern*) переменные, внешние функции и элементы перечислений из области, в которой он описан; запрещается использовать автоматические переменные из этой области;

– локальный класс не может иметь статических элементов;

– методы этого класса могут быть описаны только внутри класса;

– если один класс вложен в другой, то они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

Ключевое слово *this.* Нестатические функции-члены класса оперируют с тем объектом типа класса, из которого они вызваны. Например, пусть *x* есть объект класса *X*. Тогда функция *f* — член класса *X*, вызываемая *x.f*( ),оперирует с объектом *x*. Возникает вопрос: каким образом в теле функции *f* определить тот объект класса, который вызвал функцию *f*? Ключевое слово *this* обозначает специальную локальную переменную, доступную в теле любой функции-члена класса, описанной без спецификации *static*. Переменная *this* не требует описания и всегда содержит указатель соответствующего объекта. Например, при вызове *x.f*( ) переменная *this* устанавливается в & *x*.

Статические элементы класса. С помощью модификатора *static* можно описать статические поля и методы класса. Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

Статические поля применяются для хранения данных, общих для всех объектов класса, например количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, т. е. не дублируются.

Пример:

```
class A
{
    public:
        static int count;        // Объявление в классе
};
```

Статические поля обладают следующими особенностями:

– память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора (определение должно быть записано вне функций);

– статические поля доступны как через имя класса, так и через имя объекта;

– на статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как *private*, нельзя изменить с помощью операции доступа к области действия, как уже было описано. Это можно сделать только с помощью статических методов (см. далее);

– память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции *sizeof*.

Статические методы предназначены для обращения к статическим полям класса. Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель *this*. Обращение к статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

Пример:

```
class A
{
        static int count;      // Поле count — скрытое
    public:
        static void inc_count ( )
        {
            count++;
        }
    ...
};
    ...
int A::count = 0;              // Определение в глобальной области
void main( )
{
    A a;
    // a.count++ — нельзя, поле count скрытое
    // Изменение поля с помощью статического метода:
    a.inc_count( );            //или A::inc_count( );
}
```

Статические методы не могут быть ***константными*** (*const*) и ***виртуальными*** (*virtual*). Недопустимо описывать статические и нестатические методы с теми же самыми именами и типами аргументов. В отличие от методов типа *inline* статические методы глобальных классов подключаются при внешнем редактировании связей. Статические члены класса, описанного локальным для какого-либо метода, не могут быть инициализированы при внешнем редактировании связей. Статические члены глобальных классов инициализируются как обычные глобальные объекты, но только внутри данного файла.

Основное использование статических членов класса есть сохранение общих для всех объектов данных: число созданных объектов, выделенные ресурсы и т. п. Статические элементы также сокращают число видимых глобальных имен.

Дружественные функции и классы. Иногда желательно иметь непосредственный доступ извне к скрытым полям класса, т. е. расширить интерфейс класса. Для этого служат дружественные функции и дружественные классы. Дружественной функцией класса называется функция, которая сама не является членом класса, но имеет полные права на доступ к элементам класса типа *private* и *protected*. Во всех остальных аспектах это есть нормальная функция. Поскольку такая функция не является членом класса, то она не может быть выбрана с помощью операторов **.** и **–>**. Дружественная функция класса вызывается обычным образом. Для описания дружественной функции используют ключевое слово ***friend***.

Дружественные функции применяются для доступа к скрытым полям класса и представляют собой альтернативу методам. Метод, как правило, описывает свойство объекта, а в виде дружественных функций оформляются действия, не являющиеся свойствами класса, но концептуально входящие в его интерфейс и нуждающиеся в доступе к его скрытым полям. Например:

```
class X
{
        int i;                                  // private
        friend void friend_func(X *xptr, int);
    public:
        void member_func (int);
};
                                                // Определение функций
void friend_func(X *xptr, int a)
```

```
{                                                    //Доступ к private-элементам через указатель
    xptr–>i=a;
}
void X::member_func(int a)
{
    i=a;
}                                                     // Прямой доступ к private-элементам
…
X xobj;                                              // Объект класса X
friend_func(&xobj, 6);                               // Вызов дружественной функции
xobj.member_func(6);                                 // Вызов члена класса
```

С помощью описания ***friend*** *имя_класса* можно объявить все функции класса дружественными для другого класса:

```
class Y;                                             // Неполное описание
class X {friend Y; int i; void member_func(void);};
class Y {void friend_xl(X&); void friend_x2(X*);};
```

В данном случае все функции, описанные в *Y*, есть дружественные в *X*. Возможно также объявить отдельные функции-члены класса дружественными для другого класса:

```
class X
{
    void member_funcX(void);
};
class Y
{
    int i;
    friend void X::member_funcX(void);
};
```

Отношение «дружественности» не является транзитивным: *X* дружественно *Y*, *Y* дружественно *Z*, но не обязательно *X* дружественно *Z*. Вместе с тем отношение дружественности наследуется. Использования дружественных функций нужно по возможности избегать, поскольку они нарушают принцип инкапсуляции и таким образом затрудняют отладку и модификацию программы.

Если все методы какого-либо класса должны иметь доступ к скрытым полям другого, то весь класс объявляется дружественным с помощью ключевого слова *friend*. В приведенном далее примере класс *A* объявляется дружественным классу *B*:

```
class B
{
    friend class A;
}
class A
{
    void f1( ):
    void f2( );
}
```

Функции *f*1 и *f*2 являются дружественными по отношению к классу *B* (хотя и описаны без ключевого слова *friend*) и имеют доступ ко всем его полям.

Объявление *friend* не является спецификатором доступа и не наследуется.

Рассмотрим итоговый пример, демонстрирующий возможности классов. Данный пример аналогичен приведенному ранее при рассмотрении структур, но реализован посредством классов. Создается класс *Name*, полями которого являются фамилия и имя, методами — функции ввода и вывода для полей класса. В исполняемом коде создаются два объекта (*first*, *second*) класса *Name*, осуществляется ввод фамилий и имен двух человек в созданные объекты и дальнейший их вывод на экран. В результате выполнения программы на экран будет выведен следующий текст:

Программа, использующая классы
Введите фамилию: Иванов
Введите имя: Иван
Введите фамилию: Петров
Введите имя: Петр

Иванов Иван
Петров Петр
Для завершения нажмите любую клавишу

```
#include<iostream>
#include <conio.h>
using namespace std;
class Name
{
    char fam[10];
    char name[10];
    public:
        void NamePrint( );
        void NameInput( );
};
void Name::NamePrint( )
{
    cout<<fam<<”\t”<<name<<endl;
}
void Name::NameInput( )
{
    cout<<"Введите фамилию:\t";
    cin>>fam;
    cout<<"Введите имя:\t";
    cin>>name;
}
void main( )
{
    Name first, second;
    cout<<"Программа, использующая классы\n";
    first.NameInput( );
    second.NameInput( );
    first.NamePrint( );
    second.NamePrint( );
    cout<<"\nДля завершения нажмите любую клавишу";
    getch( );
}
```

§ 3.2. Конструкторы и деструкторы

В *C++* существуют специальные функции-члены класса, которые определяют, как объекты класса создаются, инициализируются, копируются и уничтожаются. Важнейшие из них — конструкторы и деструкторы. Им свойственны многие черты методов, но существуют и свои особенности:

– конструкторы и деструкторы не могут описываться как функции, возвращающие значение (даже как *void*);

– конструкторы и деструкторы не наследуются (механизм наследования далее будет подробно рассмотрен), однако из порождаемого класса можно вызывать конструкторы и деструкторы базового класса;

– конструкторы, подобно большинству функций языка *C++*, могут иметь аргументы. В качестве параметров могут быть элементы, получающие значения по умолчанию;

– деструкторы могут быть виртуальными (далее виртуальные функции будут подробно рассмотрены), конструкторы — нет;

– имя конструктора совпадает с именем класса. Имя деструктора — имя класса, которому предшествует символ ~;

– нельзя получить адрес конструктора или деструктора;

– если они явно не описаны, то конструкторы и деструкторы автоматически создаются компилятором. Тип доступа для создаваемых функций — *public*;

– транслятор автоматически вставляет вызовы конструктора и деструктора при описании и уничтожении объекта;

– конструкторы и деструкторы могут использовать операторы *new* и *delete*.

Имя ***конструктора*** совпадает с именем класса. Конструктор вызывается при создании или копировании объектов класса. Конструкторы для глобальных переменных вызываются до исполнения стартовой функции программы. Конструктор класса *X* не может иметь *X* в качестве аргумента. В качестве параметра конструктор может иметь ссылку на свой собственный класс. Конструктор, который не имеет параметров, называется конструктором по умолчанию.

Если класс *X* имеет один или более конструктор, то любой из них может быть использован для определения объекта *x* класса *X*. Конструктор создает объект *x* и инициализирует его. Деструктор выполняет обратные действия. Кроме того, конструктор используется, когда создается локальный или временный объект класса, деструктор — когда этот объект удаляется из памяти.

По умолчанию конструктор для класса *X* есть одна функция без параметров *X* :: *X* ( ). Подобно всем функциям конструкторы могут иметь значения аргументов по умолчанию. Например, конструктор *X* :: *X* (*int*, *int* = 0) может иметь один или два аргумента. В частности, конструктор по умолчанию *X* :: *X* ( )можно представлять как *X* :: *X* (*int* = 0). В этой связи возможна, например, следующая неоднозначность:

```
class X
{
    public:
        X( );
        X(int i=0);
}
void main ( )
{
    X one(10);                  // Допустимо, используется X::X(int)
    X two;                      // Недопустимо. Не ясно X::X( )или X::X(int=0)
}
```

Конструкторы для элементов массивов вызываются в порядке увеличения индекса. Остановимся несколько подробнее на инициализации объектов класса. Если члены класса имеют тип доступа к элементам *public* и нет конструктора и базовых классов, то объекты такого класса могут инициализироваться с помощью списка начальных значений. Типичный пример — структуры.

Другие объекты инициализируются с помощью задания параметров для конструкторов. Рассмотрим следующий пример (тела конструкторов опущены для краткости):

```
class X
{
        int i;
    public:
        X( );
        X(int x);
        X(const X&);
};
void main ( )
{
    X one;                      // Конструктор по умолчанию
    X two(l);                   //Конструктор X::X(int)
    X three=l;                  // Вызов X::X(int)
```

```
    X four=one;                    // Использование X::X(const X&)
    X five(two);                   // Вызов X::X(const X&)
}
```

Конструктор может присвоить начальные значения членам класса:

```
class X
{
        int a, b;
    public:
        X(int i, int j)
        {
            a=i;
            b=j;
        }
};
```

В этом случае запись *X obj* (1, 2) вызывает присваивание: *a* — значения 1, *b* — значения 2.

Синтаксис языка C++ включает еще один вид конструктора — конструктор копирования, получающий в качестве единственного параметра указатель на объект этого же класса:

```
T::T(const T&) { ... /* Тело конструктора */}
```

где *T* — имя класса.

Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего:

- при описании нового объекта с инициализацией другим объектом;
- при передаче объекта в функцию по значению;
- при возврате объекта из функции.

Если программист не указал ни одного конструктора копирования, то компилятор создает его автоматически. Такой конструктор выполняет поэлементное копирование полей. Если класс содержит указатели или ссылки, это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

Деструктор — это специальная функция-член класса, которая используется для освобождения членов или объектов класса до их уничтожения. Имя деструктора — имя класса, которому предшествует символ ~:

```
class X
{
    public:
        ~X( );                     // Деструктор класса X
};
```

Деструктор не имеет параметров. Для него не указывается тип результата. Если деструктор не описывается явно, то он создается компилятором. Деструктор вызывается, когда соответствующая переменная удаляется из памяти. Для локальных переменных деструктор вызывается в конце соответствующего блока, для глобальных переменных — как часть процедуры *exit* после *main*.

Когда удаляется указатель на объект, то деструктор автоматически не вызывается. Это означает, что для уничтожения объекта должна использоваться операция *delete*. Деструкторы вызываются в порядке, обратном порядку вызова конструкторов. Когда внутри программы используется функция *exit*, то деструкторы для локальных переменных не вызываются. Глобальные переменные освобождаются в обычном порядке. Если внутри программы вызывается функция *abort*, то не исполняются никакие деструкторы — ни для глобальных, ни для локальных переменных.

Деструктор может быть вызван одним из двух способов: косвенно, посредством операции *delete* и непосредственно, путем указания полного имени деструктора. Операция *delete* используется для уничтожения объектов, созданных с помощью операции *new*. Если объект создан каким-либо другим образом, то используется явный вызов деструктора. Деструктор может быть объявлен как виртуальный. Это позволяет, используя указатель объекта базового класса, вызвать правильный деструктор, если в текущий момент деструктор ссылается на объект порожденного класса.

Реализуем пример класса *Name*, рассмотренный выше, с использованием конструкторов и деструкторов. Для этого будем в конструкторе класса выделять память под все поля, а в деструкторе — освобождать:

```
#include<iostream>
#include <conio.h>
using namespace std;
class Name
{
    char *fam;
    char *name;
    public:
        void NamePrint( );
        void NameInput( );
        Name( );
        ~Name( );
};
Name::Name( )
{
    fam=new char[10];
    name=new char[10];
}
Name::~Name( )
{
    delete [ ] fam;
    delete [ ] name;
}
void Name::NamePrint( )
{
    cout<<fam<<"\t"<<name<<endl;
}

void Name::NameInput( )
{
    cout<<"Введите фамилию:\t";
    cin>>fam;
    cout<<"Введите имя:\t";
    cin>>name;
}
void main( )
{
    Name first, second;
    first.NameInput( );
    second.NameInput( );
    first.NamePrint( );
    second.NamePrint( );
    cout<<"\nДля завершения нажмите любую клавишу";
    getch( );
}
```

§ 3.3. Основы механизма наследования

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства. При большом количестве никак не связанных классов управлять ими становится невозможно. Наследование позволяет справиться с этой проблемой путем упорядочения и ранжирования классов, т. е. объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

Как указывалось ранее, при описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью ключей доступа *private*, *protected* и *public*:

class *имя*: [private | protected | public] *базовый_класс*
{*тело класса*};

Если базовых классов несколько, то они перечисляются через запятую. Ключ доступа может стоять перед каждым классом.

Пример:
```
class A { тело класса };
class B { тело класса };
class C { тело класса };
class D: A, protected B, public C
{ тело класса };
```

При этом *D* наследует все элементы базовых классов. По умолчанию наследуются только элементы базовых классов, имеющие тип *public* и *protected*. Для изменения этого умолчания при описании базовых классов могут указываться спецификаторы доступа *public* или *private*. Эти спецификаторы не меняют спецификаций доступа для элементов базовых классов, а только определяют доступ к этим элементам из порождаемых классов. При описании классов для базовых элементов по умолчанию берется спецификатор *private*, для структур — *public*. Объединения не участвуют в иерархии классов. В общем случае для определения возможности доступа к элементу руководствуются табл. 3.1.

Таблица 3.1

Доступ в базовом классе	Ключ доступа	Доступ в порожденном классе
public	public	public
private	public	недоступен
protected	public	protected
public	private	private
private	private	недоступен
protected	private	private
public	protected	protected
private	protected	private
protected	protected	protected

Как видно из табл. 3.1, *private* элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним может осуществляться только через методы базового класса.

Элементы *protected* при наследовании с ключом *private* становятся в производном классе *private*, в остальных случаях права доступа к ним не изменяются. Доступ к элементам *public* при наследовании становится соответствующим ключу доступа.

Данные, приведенные в таблице, иллюстрируются следующими примерами:

```
class X: A {...}          // Класс X порождается из класса A доступ — private
class Y: B, public C {...} // Класс Y порождается (множественное наследование)
                          //Из B и C. B — private, C — public
struct S: D {...}         // Структура S порождается из D, доступ — public
```

Определение доступа к наследуемым элементам может быть проведено и явно, путем указания наследуемых элементов в описаниях порождаемого класса.

Пример:

```
class B
{
        int a;                          // private по умолчанию
    public: int b, c;
        int Bfunc (void);
};
class X: private B                      // b, c, Bfunc — private для X
{
        int d;                          // private по умолчанию
                                        // a недоступно для X
    public: B::c;              // Теперь c — public
        int e;
        int Xfunc (void);
};
int Efunc(X &x);                        // Внешняя функция для X и B
```

В данном случае функция *Efunc*( ) может использовать из класса *X* только *c*, *e* и *Xfunc* (*public*-элементы). Функция *Xfunc* в *X* имеет доступ к:

- *c* (наследуется из *B* как *public*);
- *b*, *Bfunc* (наследуются из *B* как *private*);
- *d*, *e*, *Xfunc* (собственные элементы *X*).

Функция *Xfunc* не имеет доступа к элементу *B*::*a*.

Если базовый класс наследуется с ключом *private*, можно выборочно сделать некоторые его элементы доступными в производном классе, объявив их в секции *public* производного класса с помощью операции доступа к области видимости:

```
class A
{
    ...
    public:
        void f( );
};
class B: private A
{
    ...
    public:
        A::void f( );
};
```

Наследование бывает простым и множественным. Простым называется наследование, при котором производный класс имеет одного родителя. Множественное наследование означает, что класс имеет несколько базовых классов.

Рассмотрим пример с классом *Name*, использующим механизм наследования. Для этого создадим дочерний класс *Man* с полем — возраст, методами — ввод и вывод для всех полей обоих классов:

```
#include<iostream>
#include <conio.h>
using namespace std;
class Name
{
    public:
        char fam[10];
        char name[10];
};
class Man: Name
{
    int age;
    public:
```

```
        void ManPrint( );
        void ManInput( );
};
void Man::ManPrint( )
{
    cout<<fam<<"\t"<<name<<"\t"<<age<<endl;
}
void Man::ManInput( )
{
    cout<<"Введите фамилию:\t";
    cin>>fam;
    cout<<"Введите имя:\t";
    cin>>name;
    cout<<"Введите возраст:\t";
    cin>>age;
}
void main( )
{
    Man first, second;
    cout<<"Программа, использующая классы\n";
    first. ManInput( );
    second. ManInput( );
    first. ManPrint( );
    second. ManPrint( );
    cout<<"\nДля завершения нажмите любую клавишу";
    getch( );
}
```

Рассмотрим правила наследования различных методов [8].

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется следующими правилами:

– если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (т. е. тот, который можно вызвать без параметров);

– для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса;

– в случае нескольких базовых классов их конструкторы вызываются в порядке объявления.

Не наследуется и ***операция присваивания*****,** поэтому ее также требуется явно определить в порожденном классе. Вызов функций базового класса предпочтительнее копирования фрагментов кода из функций базового класса в функции производного. Кроме сокращения объема кода этим достигается упрощение модификации программы: изменения требуется вносить только в одну точку программы, что сокращает количество возможных ошибок.

Перечислим ***правила наследования деструкторов***:

– деструкторы не наследуются, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов;

– в отличие от конструкторов при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически;

– для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем — деструкторы элементов класса, а потом деструктор базового класса.

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса.

Пример:

```
// Описывается указатель на базовый класс:
A *p;// Указатель ссылается на объект производного класса:
p = new B;
```

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается. Поэтому при выполнении оператора будет вызван метод класса *A*, а не класса *B*, поскольку ссылки на методы разрешаются во время компоновки программы.

Пример: p–>func ( );

Этот процесс называется ***ранним связыванием***. Чтобы вызвать метод класса *B*, можно использовать явное преобразование типа указателя:

((B *p))–>func( );

Это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс. Другой пример — связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

В *C++* реализован механизм ***позднего связывания***, когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов и будет рассмотрен далее.

Виртуальные методы. Для определения виртуального метода используется спецификатор ***virtual***.

Пример: virtual void f(int x, int y);

Рассмотрим правила описания и использования виртуальных методов:

– если в базовом классе метод определен как виртуальный, то метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а с отличающимся набором параметров — обычным;

– виртуальные методы наследуются, т. е. переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя;

– если виртуальный метод переопределен в производном классе, то объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости;

– виртуальный метод не может объявляться с модификатором *static*, но может быть объявлен как дружественный;

– если в классе вводится описание виртуального метода, то он должен быть определен хотя бы как ***чисто виртуальный***.

Чисто виртуальный метод содержит признак — 0 вместо тела.

Пример: virtual void f(int) = 0;

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Если определить метод *func* в классе *A* как виртуальный, то решение о том, метод какого класса вызвать, будет приниматься в зависимости от типа объекта, на который ссылается указатель:

```
A *r, *p;
r = new A;                  // Создается объект класса
A p = new B;                // Создается объект класса B
r–>func(1, 1);              // Вызывается метод A:: func
p–>func(1, 1);              // Вызывается метод B::func
p–>A::func(1, 1);           //Обход механизма виртуальных методов
```

Итак, виртуальным называется метод, ссылка на который разрешается на этапе выполнения программы (ссылка разрешается по факту вызова).

Для каждого класса, содержащего хотя бы один виртуальный метод, компилятор создает таблицу виртуальных методов (*vtbl*), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в *vtbl* одно и то же смещение для каждого класса в пределах иерархии.

Каждый объект содержит скрытое дополнительное поле ссылки на *vtbl*, называемое *vptr*. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).

На этапе компиляции ссылки на виртуальные методы заменяются на обращения к *vtbl* через *vptr* объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода в отличие от обычных методов и функций выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы.

Рекомендуется делать деструкторы виртуальными, для того чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта. Деструктор передает операции *delete* размер объекта, имеющий тип *size_t*. Если удаляемый объект является производным и в нем не определен виртуальный деструктор, то передаваемый размер объекта может оказаться неправильным.

Деструктор класса, порожденного из класса с виртуальным деструктором, сам является виртуальным. Например:

```
class color
{
    public:
        virtual ~color( );                  // Виртуальный деструктор
};
class red: public color
{
    public:
        ~red( );
                                            // Деструктор также виртуальный,
                                            // так как ~color ( ) — виртуальный деструктор
};
class brightred: public red
{
    public:
        ~brightred ( );                     // Также виртуальный деструктор
};
…
color *palette [3];
palette [0]=new red;
palette [1]=new brightred;
palette [2]=new color;
…
                                            // Вызов деструкторов
delete palette [0];                         // Вызов ~color ( ) и ~red ( )
delete palette [1];                         // Вызов ~color( ); ~red( ); ~brightred( );
delete palette [2];                         // Вызов ~co1or( )
```

Если бы деструкторы не были описаны как виртуальные, то каждая операция *delete* вызывала бы только деструктор для класса *color*.

Четкого правила, по которому метод следует делать виртуальным, не существует. Можно только дать рекомендацию объявлять виртуальными методы, для которых есть вероятность, что они будут переопределены в производных классах. Методы, которые во всей иерархии останутся неизменными, или те, которыми производные классы пользоваться не будут, делать виртуальными нет смысла.

Виртуальный механизм работает только при использовании указателей или ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется ***полиморфным***. В данном случае полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

Абстрактные классы. Класс, содержащий хотя бы один чисто виртуальный метод, называется ***абстрактным***. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для других классов — объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении.

При определении абстрактного класса необходимо выполнять следующие правила:

– абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;

– допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;

– если класс, производный от абстрактного, не определяет все чисто виртуальные функции, то он также является абстрактным.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать ***полиморфные функции***, работающие с объектом любого типа в пределах одной иерархии.

Виртуальные классы. При множественном наследовании базовый класс не может быть специфицирован более одного раза в описании порожденного:

```
class B {...};
class D: B, B {...};                    //Недопустимо
```

Однако подобного эффекта можно добиться косвенным описанием:

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

В данном случае каждый объект класса *Z* имеет два подобъекта класса *B*. Для разрешения этой проблемы в описании базовых классов может быть добавлено ключевое слово *virtual*:

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

В этом случае класс *Z* имеет только один подобъект виртуального класса *B*.

Множественное наследование применяется для того, чтобы обеспечить производный класс свойствами двух или более базовых. Чаще всего один из этих классов является основным, а другие обеспечивают некоторые дополнительные свойства, поэтому они называются классами подмешивания. По возможности классы подмешивания должны быть виртуальными и создаваться с помощью конструкторов без параметров, что позволяет избежать многих проблем, возникающих при ромбовидном наследовании (когда у базовых классов есть общий предок).

§ 3.4. Шаблоны классов

Использование шаблонов классов позволяет отделить алгоритм от конкретных типов данных, с которыми класс работает, передавая тип в качестве параметра. В данном случае создаются ***параметризованные классы***.

Параметризованный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемому в качестве параметра. Наиболее широкое применение шаблоны находят при создании ***контейнерных классов***. Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними. Стандартная библиотека *C++* содержит множество контейнерных классов для организации структур данных различного вида.

Преимущество использования шаблонов состоит в том, что, как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода. Рассмотрим синтаксис описания шаблона:

template <*описание_параметров_шаблона*> *определение_класса*;

Параметры шаблона перечисляются через запятую. В качестве параметров могут использоваться типы, шаблоны и переменные.

Типы могут быть как стандартными, так и определенными пользователем. Для их описания используется ключевое слово *class*. Внутри шаблона параметр типа может применяться в любом месте, где допустимо использовать спецификацию типа. Например:

```
template <class Data> class A
{
    class B
    {
        public:
            Data d;
            B *next;
            B *prev;
            B(Data dat = 0)
            {
                d = dat;
                next = 0;
                prev =0;
            }
    };
    ...
};
```

Класс *Data* можно рассматривать как формальный параметр, на место которого при компиляции будет подставлен конкретный тип данных.

Для любых параметров шаблона могут быть заданы значения по умолчанию.:

Пример:

```
template<class T> class array { /* ... */ };
…
template<class K, class V, template<class T> class C = array> class Map
{
    C<K> key;
    C<V> value;
    ...
};
```

Область действия параметра шаблона — от точки описания до конца шаблона, поэтому параметр можно использовать при описании следующих за ним.

Пример: template <class T, T*p, class U = T> class X {/*...*/};

Методы шаблона класса автоматически становятся шаблонами функций. Если метод описывается вне шаблона, то его заголовок должен иметь следующие элементы:

template<*описание_параметров_шаблона*>
возвр_тип имя_класса<*параметры_шаблона*>::
имя_функции (*список_параметров_функции*)

Описание параметров шаблона в заголовке функции должно соответствовать шаблону класса, при этом имена параметров могут не совпадать. Проще рассмотреть синтаксис описания методов шаблона на следующем примере:

```
template<class Data>void A<Data>::print ( )
{/* тело функции */}
```

Здесь <*class Data*> — описание параметра шаблона; *void* — тип возвращаемого функцией значения; *A* — имя класса; <*Data*> — параметр шаблона; *print* — имя функции без параметров.

В случае нескольких параметров порядок их следования в *описании_параметров* и *параметрах_шаблона* должен быть один и тот же.

Пример:

```
template<class T1, class T2> struct A
{
      void f1 ( );
};
template<class T2, class T1> void A<T2, T1>::f1( ) {...}
```

Перечислим правила описания шаблонов:

– локальные классы не могут содержать шаблоны в качестве своих элементов;

– шаблоны методов не могут быть виртуальными;

– шаблоны классов могут содержать статические элементы, дружественные функции и классы;

– шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов;

– внутри шаблона нельзя определять *friend*-шаблоны.

При определении синтаксиса шаблона было сказано, что в него, кроме типов и шаблонов, могут передаваться ***переменные***. Они могут быть целого или перечисляемого типа, а также указателями или ссылками на объект или функцию. В теле шаблона они могут применяться в любом месте, где допустимо использовать константное выражение. В качестве примера создадим шаблон класса, содержащего блок памяти определенной длины и типа:

```
template <class Type, int kol> class B
{
    public:
        B( ) {p = new Type [kol];}
        ~B( ) {delete [ ] p;}
        operator Type *( );
    protected:
        Type *p;
};
template <class Type, int kol> B <Type, kol>:: operator Type *( )
{
    return p;
}
```

После создания и отладки шаблоны классов удобно помещать в заголовочные файлы.

Чтобы создать при помощи шаблона конкретный объект конкретного класса (инстанцирование), при описании объекта после имени шаблона в угловых скобках необходимо перечислить его аргументы:

имя_шаблона <аргументы> имя_объекта [(*параметры_конструктора*)];

Аргументы должны соответствовать параметрам шаблона. Имя шаблона вместе с аргументами можно воспринимать как уточненное имя класса. Приведем примеры создания объектов по шаблонам, описанным ранее:

```
A <int> A_int;
A <double> A_double;
A <day> A_day;
B <char, 128> but;
B <a, 100> st;
```

При использовании параметров шаблона по умолчанию список аргументов может оказаться пустым, при этом угловые скобки опускать нельзя:

```
Template<class T = char> class Str;
Str <> *p;
```

Если параметром шаблона является шаблон, имеющий специализацию, то она учитывается при инстанцировании:

```
template<class T> class A
{                                                    // Исходный шаблон
    int x;
};
template<class T> class A<T*>
{                                                    // Специализация шаблона
    long x;
};
template <template <class U> class V> class C
{
    V<int> y;
    V<int*> z;
};
...
C<A> c;
```

В этом примере *V <int>* внутри *C <A>* использует исходный шаблон, поэтому *c.y.x* имеет тип *int*, а *V<int*>* использует специализацию шаблона, поэтому *c.z.x* имеет тип *long*.

На месте формальных параметров, являющихся переменными целого типа, должны стоять константные выражения. После создания объектов с помощью шаблона с ними можно работать так же, как с объектами обычных классов.

Пример:

```
for (int i = 1; i<10; i++)
    A_double.add(i * 0.08);
A_double.print( );
for (int i = 1; i<10; i++)
    A_day.add(i);
A_day.print( );
strcpy(buf, "Сообщение");
cout<<buf<<endl;
```

Для упрощения использования шаблонов классов можно применить переименование типов с помощью *typedef*:

```
typedef A <double> Adbl;
Adbl A double;
```

Каждая версия класса или функции, создаваемая по шаблону, содержит одинаковый базовый код; изменяется только то, что связано с параметрами шаблона. При этом эффективность работы версий, создаваемых для различных типов данных, может сильно различаться.

Если для какого-либо типа данных существует более эффективный код, можно либо предусмотреть для этого типа специальную реализацию отдельных методов, либо полностью переопределить (специализировать) шаблон класса.

Для специализации метода требуется определить вариант его кода, указав в заголовке конкретный тип данных. Например, если заголовок обобщенного метода *print* шаблона *A* имеет вид

```
template <class Data> void A <Data>::print( );
```

то специализированный метод для вывода списка символов будет выглядеть следующим образом:

```
void A <char>::print ( )
{
    ...                                  // Тело специализированного варианта метода print
}
```

Если в программе создать экземпляр шаблона *A* типа *char*, то соответствующий вариант метода будет вызван автоматически.

При специализации целого класса после описания обобщенного варианта класса помещается полное описание специализированного класса, при этом требуется заново определить все его методы. Допустим, требуется специализировать шаблон *B*, описанный в предыдущем разделе, для хранения 100 целых величин:

```
class B <int, 100>
{
    public:
        B ( ) {p = new int [100];}
        ~B ( ) {delete [ ] p;}
        operator int *( );
    protected:
    int *p;
};
B <int, 100>:: operator int *( )
{
    return p;
}
```

При определении экземпляров шаблона *B* с параметрами *int* и 100 будет задействован специализированный вариант.

Шаблоны представляют собой мощное и эффективное средство обращения с различными типами данных, которое можно назвать параметрическим полиморфизмом, а также, в отличие от макросов препроцессора, обеспечивают безопасное использование типов. Однако следует иметь в виду, что программа, использующая шаблоны, содержит полный код для каждого порожденного типа, что может увеличить размер исполняемого файла. Кроме того, с некоторыми типами данных шаблоны могут работать не так эффективно, как с другими. В этом случае имеет смысл использовать специализацию шаблона.

В заключение рассмотрим пример, демонстрирующий возможности шаблонов классов. Этот пример, как и ранее, использует класс *Name* с полями — фамилия и имя, методами — функции ввода и вывода для всех полей. В данном случае создается шаблон этого класса:

```
#include<stdio.h>
#include <conio.h>
template <class Data> class Name
{
    Data fam[10];
    Data name[10];
    public:
            void NamePrint( );
            void NameInput( );
};
template <class Data> void Name<Data>::NamePrint( )
{
    cout<<fam<<"\t"<<name<<endl;
}
template <class Data> void Name<Data>::NameInput( )
{
    cout<<"Введите фамилию:\t";
    cin>>fam;
    cout<<"Введите имя:\t";
    cin>>name;
}
void main( )
{
    Name <char> a;
    cout<<"Программа, использующая классы\n";
    a.NameInput( );
    a.NamePrint( );
    cout<<"\nДля завершения нажмите любую клавишу";
    getch( );
}
```

§ 3.5. Обработка исключительных ситуаций

Исключительная ситуация, или исключение — это возникновение непредвиденного или аварийного события, которое может порождаться некорректным использованием аппаратуры. Например, это деление на ноль или обращение по несуществующему адресу памяти. Обычно такие события приводят к завершению программы с системным сообщением об ошибке. *C++* дает программисту возможность восстанавливать программу и продолжать ее выполнение. Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработка.

Синтаксис обработки исключений следующий. Место, в котором может произойти ошибка, должно входить в контролируемый блок — составной оператор, перед которым записано ключевое слово *try*.

Рассмотрим, каким образом реализуется обработка исключительных ситуаций:

– обработка исключения начинается с появления ошибки. Функция, в которой она возникла, генерирует исключение. Для этого используется ключевое слово *throw* с параметром, определяющим вид исключения. Параметр может быть константой, переменной или объектом и используется для передачи информации об исключении его обработчику;

– отыскивается соответствующий обработчик исключения, которому передается управление;

– если обработчик исключения не найден, вызывается стандартная функция *terminate*, которая вызывает функцию *abort*, аварийно завершающую текущий процесс. Можно установить собственную функцию завершения процесса.

Ключевое слово *try* служит для обозначения контролируемого блока-кода, в котором может генерироваться исключение. Блок заключается в фигурные скобки:

```
try
{
...
}
```

Все функции, прямо или косвенно вызываемые из *try*-блока, также считаются ему принадлежащими.

Генерация (порождение) исключения происходит по ключевому слову *throw*, которое употребляется либо с параметром, либо без него:

```
throw [выражение];
```

Тип выражения, стоящего после *throw*, определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, происходит поиск соответствующего обработчика и передача ему управления. Как правило, исключение генерируется не непосредственно в *try*-блоке, а в функциях, прямо или косвенно в него вложенных.

Не всегда исключение, возникшее во внутреннем блоке, может быть сразу правильно обработано. В этом случае используются вложенные контролируемые блоки, и исключение передается на более высокий уровень с помощью ключевого слова *throw* без параметров.

Обработчики исключений начинаются с ключевого слова catch, за которым в скобках следует тип обрабатываемого исключения. Они должны располагаться непосредственно за *try*-блоком. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Синтаксис обработчиков напоминает определение функции с одним параметром — типом исключения. Существует три формы записи:

```
catch(тип имя){ ... /* тело обработчика */ }
catch(тип){ ... /* тело обработчика */ }
catch(...){ ... /* тело обработчика */ }
```

Первая форма применяется тогда, когда имя параметра используется в теле обработчика для выполнения каких-либо действий, например вывода информации об исключении. Вторая форма не предполагает использования информации об исключении, играет роль только его тип. Многоточие вместо параметра обозначает, что обработчик перехватывает все исключения. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа следует помещать после всех остальных. Например:

```
catch(int i)
{
...                                  // Обработка исключений типа int
}
catch(const char *)
{
...                                  // Обработка исключений типа const char*
}
catch(Overflow)
{
...                                  // Обработка исключений класса Overflow
}
catch(j)
{
...                                  // Обработка всех необслуженных исключений
}
```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений. Туда же, минуя код всех обработчиков, передается управление, если исключение в *try*-блоке не было сгенерировано.

Когда с помощью *throw* генерируется исключение, функции исполнительной библиотеки *C++* выполняют следующие действия:

1) создают копию параметра throw в виде статического объекта, который существует до тех пор, пока исключение не будет обработано;

2) в поисках подходящего обработчика раскручивают стек, вызывая деструкторы локальных объектов, выходящих из области действия;

3) передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

Обработчик считается найденным, если тип объекта, указанного после *throw*:

– тот же, что и указанный в параметре *catch* (параметр может быть записан в форме *T*, *const T*, *T&* или *const T&*, где *T* — тип исключения);

– является производным от указанного в параметре *catch* (если наследование производилось с ключом доступа *public*);

– является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре *catch*.

Из сказанного следует, что обработчики производных классов следует размещать до обработчиков базовых, поскольку в противном случае им никогда не будет передано управление. Обработчик указателя типа *void* автоматически скрывает указатель любого другого типа, поэтому его также следует размещать после обработчиков указателей конкретного типа.

Рассмотрим пример на использование обработчиков исключений. В данном примере используется класс *Name*, который рассматривался в примерах и ранее. Для срабатывания механизма исключений — «ошибка выделения памяти» — попытаемся выделить память для полей класса очень большого размера. В результате исполнения на экран будет выведен следующий текст:

```
Программа использующая классы
Входим в try-блок
Вызван обработчик исключений,
исключение — Ошибка выделения памяти для поля Фамилия
```

```
#include<iostream.h>
                        //Запрашиваем очень много памяти, чтобы ее невозможно было выделить
#define size1 1000000000000
#define size2 1000000000000
class Name
{
    char *fam;
    char *name;
    public:
        void NamePrint( );
        void NameInput( );
        Name( );
        ~Name( );
};
Name::Name()
{
    fam=new char[size1];
    if(!fam)
        throw "Ошибка выделения памяти для поля Фамилия";
    name=new char[size2];
    if(!name)
        throw "Ошибка выделения памяти для поля Имя";
    cout<<"Данные инициализированы\n";
}
Name::~Name( )
{
    delete [ ] fam;
    delete [ ] name;
    cout<<"\nДанные очищены";
}
void Name::NameInput( )
{
    cout<<"Введите Фамилию:\t";
    cin>>fam;
    cout<<"Введите Имя:\t";
    cin>>name;
}
void Name::NamePrint( )
{
    cout<<fam<<"\t"<<name<<"\n";
}
int main( )
{
    cout<<"Программа использующая классы\n";
    try
    {
        Name a;
        a.NameInput( );
        a.NamePrint( );
    }
    catch (char *msg)
    {
        cerr<<"Вызван обработчик исключений, \nисключение —"
            <<msg<<endl;
        return –1;
    }
    return 0;
}
```

§ 3.6. Переопределение операций

C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Это дает возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в *C++*, за исключением

. .* ?: :: # ## sizeof

Перегрузка операций осуществляется с помощью методов специального вида *(функций-операций)* и подчиняется следующим правилам:

– при перегрузке операций сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
– для стандартных типов данных переопределять операции нельзя;
– функции-операции не могут иметь аргументов по умолчанию;
– функции-операции наследуются (за исключением =);
– функции-операции не могут определяться как *static*.

Функцию-операцию можно определить тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс.

Функция-операция содержит ключевое слово *operator*, за которым следует знак переопределяемой операции:

тип operator *операция* (*список параметров*) { *тело функции* }

Для примера введем операцию + для класса *complex*:

```
class complex
{
        double real, imag;
    public:
        complex ( ) {real=imag=0;}
        complex (double r, double i=0)
        { real=r; imag=i;}
}
```

Можно легко реализовать функцию

```
complex Add (complex c1,complex c2),
```

которая реализует сложение двух комплексных чисел. Но более естественной была бы запись

```
complex c1 (0, 1), c2(1, 0), c3;
c3=c1+c2;
```

чем *c*3 = *Add*(*c*1,*c*2). Это может быть реализовано переопределением операции +:

```
friend complex operator + (complex c1, complex c2)
{
    return complex (c1.real+c2.real, c1.imag+c2.imag);
}
```

Следует заметить, что введенная таким образом функция для операции сложения может быть вызвана и явно:

```
c3 = operator + (c1,c2);
```

За исключением операций *new* и *delete*, для которых существуют собственные правила (они будут рассмотрены позже), функции-операции должны быть нестатическими функциями-членами класса или иметь как минимум один аргумент типа класса. Функции-операции = ( ) [ ] и –> должны быть нестатическими членами класса.

Переопределение унарных операций. Префиксная или постфиксная унарная операция может быть переопределена либо путем задания нестатической функции-члена класса без аргументов, либо путем определения функции, не являющейся членом класса и имеющей один аргумент. Если @ есть унарная операция, то *@x* и *x@* интерпретируются как *x.operator@*( ) или *operator@*(*x*) в зависимости от типа описания.

Отсюда следует, что при переопределении теряется разница между префиксными и постфиксными унарными операциями.

Пример:

```
class X
{
      X operator ++();
};
…
X x, y;
y=++x;                          // То же, что и y=x++ в этом контексте
```

Переопределение бинарных операций. Бинарная операция может быть переопределена заданием нестатической функции-члена класса, имеющей один аргумент, или определением функции, не являющейся членом класса (обычно дружественной), имеющей два аргумента. Если @ есть некоторая бинарная операция, то *x@y* может быть интерпретировано как *x.operator@*(*y*) или*operator@*(*x*, *y*) в зависимости от типа описания.

Переопределение операции присваивания. Операция присваивания может быть переопределена путем задания нестатической функции-члена класса.

Пример:

```
class String
{
      String &operator=(String &str);
      String (String&);
      ~String();
}
```

Переопределение операции вызова ( ). Вызов функции

выражение (<*список выражений*>)

рассматривается как бинарная операция с операндами «*выражение*» и «*список выражений*» (возможно, пустой). Соответствующая функция есть *operator* ( ). Эта функция должна быть нестатическим членом класса. Вызов *x* (*arg*1, *arg*2), где *x* есть объект класса *X*, интерпретируется как *x.operator*( )(*arg*1, *arg*2).

Переопределение операции индексации. Операция индексации

*выражение*1 [*выражение*2]

рассматривается как бинарная операция с операндами «*выражение*1» и «*выражение*2». Соответствующая функция есть *operator* ( ). Эта функция должна быть нестатическим членом класса. Выражение *x* [*y*], где *x* есть объект класса *X*, интерпретируется как *x.operator*[ ](*y*).

Переопределение операций *new*и*delete.* Чтобы обеспечить альтернативные варианты управления памятью, можно определять собственные варианты операций *new* и *new*[ ] для выделения динамической памяти соответственно под объект и массив объектов, а также операции *delete* и *delete* [ ] для ее освобождения.

Эти функции-операции должны соответствовать следующим правилам:

– им не требуется передавать параметр типа класса;

– первым параметром функциям new и *new* [ ] должен передаваться размер объекта типа *size_t* (этот тип, возвращаемый операцией *sizeof*, определяется в заголовочном файле <stddef.h>); при вызове он передается в функцию неявным образом;

– они должны определяться с типом возвращаемого значения *void**, даже если *return* возвращает указатель на другие типы (чаще всего на класс);

– операция *delete* должна иметь тип возврата *void* и первый аргумент типа *void**;

– операции выделения и освобождения памяти являются статическими элементами класса.

Поведение переопределенных операций должно соответствовать действиям, выполняемым ими по умолчанию. Для операции *new* это означает, что она должна возвращать правильное значение, корректно обрабатывать запрос на выделение памяти нулевого размера и порождать исключение при невозможности выполнения запроса. Для операции *delete* следует соблюдать условие: удаление нулевого указателя должно быть безопасным. Поэтому внутри операции необходима проверка указателя на нуль и отсутствие каких-либо действий в случае равенства.

Стандартные операции выделения и освобождения памяти могут использоваться в области действия класса наряду с переопределенными (с помощью операции доступа к области видимости**::** для объектов этого класса и непосредственно для любых других).

Переопределение операции выделения памяти применяется для экономии памяти, повышения быстродействия программы или для размещения данных в некоторой конкретной области.

Переопределение операции приведения типа. Можно определить функции-операции, которые будут осуществлять преобразование объекта класса к другому типу.

Формат: operator *имя_нового_типа* ( );

Указывать тип возвращаемого значения и параметры не требуется. Можно определять виртуальные функции преобразования типа.

Глава 4
Библиотеки функций

§ 4.1. Библиотеки потоковых классов ввода/вывода

4.1.1. Описание потоковых классов

Объектно-ориентированный подход привел к изменению организации ввода / вывода в языке *C++*. В языке Си для ввода / вывода использовались две основные библиотеки — *stdio.h* и *conio.h* (основные функции данных библиотек будут рассмотрены далее). Синтаксис языка *C++* наряду с библиотеками языка Си позволяет использовать специально организованные потоковые классы.

Поток в целом можно воспринять как абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специальную область оперативной памяти — ***буфер***. Фактическая передача данных выполняется при выводе после заполнения буфера, а при вводе — если буфер исчерпан.

По направлению обмена потоки можно разделить на ***входные*** (данные вводятся в память), ***выходные***(данные выводятся из памяти) и ***двунаправленные***(допускающие как извлечение, так и включение).

По виду устройств, с которыми работают потоки, их можно разделить на ***стандартные***, ***файловые*** и ***строковые***.

Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея, файловые потоки — для обмена информацией с файлами на внешних носителях данных (например, на магнитном диске), строковые потоки — для работы с массивами символов в оперативной памяти.

Для поддержки потоков библиотека *C++* содержит иерархию классов, построенную на основе двух базовых классов — *ios* и *streambuf*. Класс *ios* содержит общие для ввода и вывода поля и методы, класс *streambuf* обеспечивает буферизацию потоков и их взаимодействие с физическими устройствами. От этих классов наследуется класс *istream* для входных потоков и *ostream* — для выходных. Два последних класса являются базовыми для класса *iostream*, реализующего двунаправленные потоки. Ниже в иерархии классов располагаются файловые и строковые потоки.

Описания потоковых классов находятся в следующих заголовочных файлах:

<ios> — базовый класс потоков ввода / вывода;

<iosfwd> — предварительные объявления средств ввода / вывода;

<istream> — шаблон потока ввода;

<ostream> — шаблон потока вывода;

<iostream> — стандартные объекты и операции с потоками ввода / вывода;

<fstream> — потоки ввода / вывода в файлы;

<strstrea> — потоки ввода / вывода в строки;

<streambuf> — буферизация потоков ввода / вывода.

Перечисленные заголовочные файлы подключаются стандартным образом через директиву *include* препроцессора. Подключение к программе файлов <*fstream*> и <*sstream*> автоматически подключает и файл <*iostream*>, так как он является для них базовым.

Основным преимуществом использования потоковых классов по сравнению с функциями ввода / вывода, унаследованными из библиотеки Си, является контроль типов, а также расширяемость, т. е. возможность работать с типами, определенными пользователем. Кроме того, потоки могут работать с расширенным набором символов *wchar_t*. Для этого используются классы *wistream*, *wostream*, *wofstream* и т. д. К недостаткам использования потоковых классов можно отнести снижение быстродействия программ.

Рассмотрим основные классы и объекты ввода / вывода:

ios — базовый класс потоков ввода / вывода;

istream — класс входных потоков. Является производным от базового класса *iostream* и наследует его члены. В свою очередь является базовым для классов *ifstream* и *istrstream*. Для использования функций класса *istream* в текст программы необходимо включить заголовочный файл *istream.h*:

```
#include<istream.h>
```

ostream — класс выходных потоков. Является производным от базового класса *iostream* и наследует его члены. В свою очередь, является базовым для классов *ofstream* и *ostrstream*. Для использования функций класса *ostream* в текст программы необходимо включить заголовочный файл *ostream.h*:

```
#include<ostream.h>
```

iostream — класс двунаправленных потоков. Является производным от абстрактного базового класса *ios* и наследует все его члены. В свою очередь является базовым для классов *istream* и *ostream*. Для использования функций класса *iostream* в текст программы необходимо включить заголовочный файл *iostream.h*:

```
#include<iostream.h>
```

istrstream — класс входных строковых потоков. Позволяет читать данные из строк в памяти. Является производным от класса *strstream*. Для использования функций класса *istrstream* в текст программы необходимо включить заголовочный файл *strstrea.h*:

```
#include<strstrea.h>
```

ostrstream — класс выходных строковых потоков. Позволяет писать строки в память. Является производным от класса *strstream*. Для использования функций класса *ostrstream* в текст программы необходимо включить заголовочный файл *strstrea.h*:

```
#include<strstrea.h>
```

strstream — класс двунаправленных строковых потоков. Позволяет читать и писать строки в память. Является базовым для классов *istrstream* и *ostrstream*. Для использования функций класса *ostrstream* в текст программы необходимо включить заголовочный файл *strstrea.h*:

```
#include<strstrea.h>
```

ifstream — класс входных файловых потоков. Класс *ifstream* позволяет открывать файлы аналогично потокам. Является производным от класса *fstream*. Для использования функций класса *ifstream* в текст программы необходимо включить заголовочный файл *fstream.h*:

```
#include<fstream.h>
```

ofstream — класс выходных файловых потоков. Позволяет открывать файлы для записи аналогично потокам вывода. Является производным от класса *fstream*. Для использования функций класса *ofstream* в текст программы необходимо включить заголовочный файл *fstream.h*:

```
#include<fstream.h>
```

fstream — класс двунаправленных файловых потоков. Поддерживает потоки с возможностью как ввода, так и вывода информации. Является базовым для классов *ifstream* и *ofstream*. Для использования функций класса fstream в текст программы необходимо включить заголовочный файл *fstream.h*:

```
#include<fstream.h>
```

cin — объект, предоставляющий последовательный доступ к стандартному устройству ввода информации. Обычно поток связан с клавиатурой. В операционных системах типа *UNIX*, *DOS* или *Windows* предусмотрена возможность перенаправлять чтение информации с клавиатуры на чтение из файла. Объект *cin* является членом класса *istream*.

cout — объект, предоставляющий последовательный строковый доступ к устройству стандартного вывода. В операционных системах типа *UNIX*, *DOS* или *Windows* предусмотрена возможность перенаправлять запись информации в файл. В данном случае объект *cout* функционирует как файл. Объект *cout* является членом класса *ostream*.

cerr — объект (консоль ошибок), предоставляющий последовательный доступ к стандартному устройству вывода ошибок. Данный поток связан с монитором и обычно идентичен стандартному потоку *cout*. Основное различие заключается в том, что даже если стандартный вывод направлен в файл, то все, что направлено в поток *cerr*, появляется на экране монитора. Объект *cerr* является членом класса *ostream*.

clog — объект, предоставляющий последовательный строковый доступ к монитору аналогично объектам *cout* и *cerr*. В отличие от *cerr* поток *clog* использует буферизированный вывод. Оба потока (*cerr* и *clog*) являются предпочтительными устройствами для вывода сообщений об ошибках, так как они продолжают выводить сообщения на монитор, даже если стандартный вывод *cout* перенаправлен. Объект *clog* является членом класса *ostream*.

4.1.2. Стандартные потоки ввода/вывода

Для ***вывода*** используется переопределенная операция <<.

Описание: *операнд* 1 <<*операнд* 2;

где *операнд* 1 — объект класса *ostream*, *операнд* 2 — выводимое значение.

Пример:
```
cout<<"Hello!\n";          // Запись текста с переводом строки
```

Операция << автоматически переопределяется в зависимости от типа выводимого значения и возвращает ссылку на объект класса *ostream*, с которым она оперирует. Это позволяет использовать каскадные последовательности выводов.

Пример:
```
cout<<"i="<<I<<", d="<<d<<"\n";
```

По умолчанию поддерживается вывод объектов следующих типов: *char* (*signed*, *unsigned*), *short* (*signed*, *unsigned*), *int* (*signed*, *unsigned*), *long* (*signed*, *unsigned*), *char**, *float*, *double*, *long double* и *void**. Для вывода используются правила-умолчания из функции *printf* языка *C* (это может быть изменено установкой различных флагов в *ios*). Например, если есть описания

```
int i; long l;
```

то следующие операторы эквивалентны:

```
cout<<i<<l;                    // Потоки языка C++
printf("%d %ld", i, l);        // Стандартная функция языка Си
```

Вещественные значения выводятся по аналогии со спецификацией %*g*, указатели — как шестнадцатеричные значения.

Для ввода используется переопределяемая операция >>.

Описание: *операнд* 1 >>*операнд* 2;

где *операнд* 1 — объект класса *istream*, *операнд* 2 — вводимое значение.

Пример:
```
cin >> d;
```

Операция >> автоматически переопределяется в зависимости от типа выводимого значения и возвращает ссылку на объект класса *istream*, с которым она оперирует. Все правила, перечисленные для потоков вывода, соответствуют и вводу. Рассмотрим пример на одновременное использование потоков ввода и вывода:

```
#include <iostream.h>
int main( )
{
    int i;
    cout<< "Введите значение переменной i:";
    cin>>i;
    return 0;
}
```

Как и для других перегруженных операций, для вставки и извлечения невозможно изменить приоритеты, поэтому в необходимых случаях используются скобки:

```
cout<<i + j;          // Скобки не требуются — приоритет сложения больше, чем <<
cout<< (i<j);         // Скобки необходимы — приоритет операции отношения меньше, чем <<
```

Величины при вводе должны разделяться пробельными символами (пробелами, знаками табуляции или перевода строки). Извлечение прекращается, если очередной символ оказался недопустимым.

Поскольку ввод буферизован, то помещение в буфер ввода происходит после нажатия клавиши перевода строки, после чего из буфера выполняется операция извлечения из потока. Это дает возможность исправлять введенные символы до того, как нажата клавиша *Enter*.

При вводе строк извлечение происходит до ближайшего пробела (вместо него в строку заносится нуль-символ):

```
char str1[l00], str2[100];
cin >> str1 >> str2;
```

Таким образом, если с клавиатуры вводится строка «раз два три четыре пять», то переменные *str*1 и *str*2 примут соответственно значения «раз» и «два», а остаток строки воспринят не будет. При необходимости ввести из входного потока строку целиком (до символа '\n') следует использовать методы *get* или *getline* (будут рассмотрены далее).

Значения указателей выводятся в шестнадцатеричной системе счисления. Под любую величину при выводе отводится столько позиций, сколько требуется для ее представления. Чтобы отделить одну величину от другой, используются пробелы:

```
cout<<i<< '   ' <<d<< "   " <<j;
```

Изменение формата вывода, используемого по умолчанию, выполняется тремя способами: с помощью ***флагов***, ***манипуляторов*** и форматирующих ***методов***.

Флаги форматирования, определяемые в *ios* как биты числа типа *long int*, действуют только для операции <<. Рассмотрим эти флаги:

```
enum {
    skipws = 0x0001,          // Пропуск пробелов при вводе
    left = 0x0002,            // Левое выравнивание при выводе
    right = 0x0004,           // Правое выравнивание при выводе
    internal = 0x0008,        // Заполнитель после знака или базы системы счисления
    dec = 0x0010,             // Десятичное преобразование
    oct = 0x0020,             // Восьмеричное преобразование
    hex = 0x0040,             // Шестнадцатеричное преобразование
    showbase = 0x0080,        // Показ базы системы счисления
    showpoint = 0x0100,       // Показ десятичной точки
    uppercase = 0x0200,       // Шестнадцатеричные символы на верхнем регистре
    showpos = 0x0400,         //Выводить + для положительных чисел
    scientific = 0x0800,      // Вывод с плавающей точкой (E и степень)
    fixed = 0x1000,           // Вывод с фиксированной точкой
    unitbuf = 0x2000,         // Сброс всех потоков
    stdio = 0x4000            // Сброс stdout
};
```

Для управления флагами в классе *ios* описаны методы *flags* ( ), *setf*( ) и *unsetf* ( ), которые представлены в табл. 4.1.

Таблица 4.1

Функция	Действие
long ios::flags ( );	Возвращает текущие флаги потока
long ios::flags (long);	Присваивает флагам значение параметра
long ios::setf (long, long);	Присваивает флагам, биты которых установлены в первом параметре, значение соответствующих битов второго параметра
long ios::setf(long);	Устанавливает флаги, биты которых установлены в параметре
long ios::unsetf (long);	Сбрасывает флаги, биты которых установлены в параметре

Флаги (*left*, *right* и *internal*), (*dec*, *oct* и *hex*), а также (*scientific* и *fixed*) взаимно исключают друг друга, т. е. в каждый момент может быть установлен только один флаг из каждой группы. Для сброса предыдущего флага удобно использовать в качестве второго параметра метода *setf*( )перечисленные далее статические константы класса *ios*:

```
adjustfield (left | right | internal)
basefield (dec | oct | hex)
floatfield (scientific | fixed)
```

Помимо флагов, для форматирования используются ***специальные поля*** класса *ios*, представленные в табл. 4.2.

Таблица 4.2

Флаг	Значение
int x_width	Минимальная ширина поля вывода
int x_precision	Количество цифр в дробной части
int x_fill	Символ заполнения поля вывода

Для управления перечисленными полями применяются методы *width* ( ), *precision*( )и *fill* ( ). Описание методов представлено в табл. 4.3.

Таблица 4.3

Функция	Действие
int ios:: width ( )	Возвращает значение ширины поля вывода
int ios:: width (int)	Устанавливает ширину поля вывода в соответствии со значением параметра
int ios::precision( )	Возвращает значение точности представления при выводе вещественных чисел
int ios::precision(int)	Устанавливает значение точности представления при выводе вещественных чисел, возвращает старое значение точности
char fill( )	Возвращает текущий символ заполнения
char fill(char)	Устанавливает значение текущего символа заполнения, возвращает старое значение символа

Рассмотрим пример использования перечисленных методов:

```
#include <iostream.h>
int main( )
{
    int i = 123;
    cout.width(7);
    cout<<i;
```

Результат:….123

В данном примере при выводе результата использован для наглядности вместо пробела символ (.). По умолчанию символом заполнения является пробел. Изменить данное умолчание на любой другой символ позволяет метод *fill* ( ):

```
int i = 123;
cout.width(7);
cout.fill ('*');
cout<<i;
```

*Результат:*****123

По умолчанию установлено выравнивание по правому краю. При помощи функций *setf* и *unsetf* можно изменить флаг форматирования:

```
int i = 123;
cout.width(7);
cout.fill ('*');
cout.setf(ios::left, ios::adjustfield);
cout<<i;
```

*Результат:*123****

Если используется вывод каскадом, то ширина поля устанавливается по умолчанию после каждого вывода. Простой способ установки форматных переменных состоит в использовании манипуляторов. ***Манипуляторами*** называются функции, которые можно включать в цепочку операций помещения и извлечения для форматирования данных. Пользоваться манипуляторами более удобно, чем методами установки флагов форматирования. Манипуляторы делятся на простые, не требующие указания аргументов, и параметризованные. Манипулятор получает ссылку на поток в качестве аргумента и возвращает ссылку на тот же поток. Это позволяет использовать манипуляторы в каскадах.

Пример: cout<<setw(4) <<i<<setw(6) <<j;

эквивалентно:

```
cout.width(4);
cout<<i;
cout.width (6);
cout<<j;
```

В данном примере *setw*( ) есть параметризованный манипулятор, описанный в файле *iomanip.h.* Стандартные манипуляторы представлены в табл. 4.4.

Рассмотрим пример использования флагов, манипуляторов и методов для форматирования вывода:

```
#include <iostream.h>
#include <iomanip.h>

int main( )
{
    long a = 1000, b = 077;
    cout.width(7);
    cout.setf(ios:: hex | ios::showbase | ios::uppercase);
    cout << a;
    cout.width(8);
    cout << b << endl;
    double d = 0.12, c = 1.3e – 4;
    cout.setf(ios::left);
    cout << d << endl;
    cout << c; return 0;
}
```

Таблица 4.4

Манипулятор	Синтаксис	Действие
dec	cout<<dec, cin>>dec	Установка десятичного преобразования
hex	cout<<hex, cin>>hex	Установка шестнадцатеричного преобразования
oct	cout<<oct, cin>>oct	Установка восьмеричного преобразования
ws	cin>>ws	Удаление пробелов
endl	cout<<endl	Перевод строки и сброс потока
ends	cout<<ends	Вставка 0-символа в конце строки
flush	cout<<flush	Сброс потока
setbase(int)	cout<<setbase(n)	Установка базы системы счисления *n* = (0, 8, 10, 16) Ноль — значение по умолчанию (10-я)
resetiosflags (long)	cin>>resetiosflags(l) cout<<resetiosflags(1)	Сброс форматных битов в аргументе 1
etiosflags(long)	cin>>etiosflags(1) cout<<etiosflags(1)	Установка битов форматирования по аргументу l
setfill(int)	cin>>setfill(n) cout<<setfill(n)	Установка символа заполнения в *n*
setprecision(int)	cin>>setprecision(n) cout<<setprecision(n)	Установка точности представления чисел с плавающей точкой в *n* цифр
setw(int)	cin>>setw(n) cout<<setw(n)	Установка ширины поля в *n* позиций

В результате работы программы в первой строке будут прописными буквами выведены переменные *a* и *b* в шестнадцатеричном представлении, под них отводится по 7 и 8 позиций соответственно. Значения переменных *c* и *d* прижаты к левому краю поля:

```
..0X3E8….0X3F
0.12
0.00013
```

В потоковых классах наряду с операциями извлечения >> и включения << определены методы для неформатированного чтения и записи в поток (при этом преобразования данных не выполняются). Методы неформатированного чтения, определенные в классе *istream*, приведены в табл. 4.5, а методы неформатированной записи, определенные в классе *ostream*, приведены в табл. 4.6.

Рассмотрим пример на использование методов неформатированного ввода/вывода.
Программа считывает строки из входного потока в символьный массив.

```
#include<iostream.h>
#include <conio.h>

int main( )
{
    const int N=20, Len=100;
    char str[Len][N];
    int i = 0;
    while (cin.getline(str[i], Len, '\n') && i<N)
    {
        i++;
    }
    cout<<"Для завершения нажмите любую клавишу";
    getch( );
    return 0;
}
```

Таблица 4.5

Функция	Действие
gcount( )	Возвращает количество символов, считанных с помощью последней функции неформатированного ввода
get ( )	Возвращает код извлеченного из потока символа или *EOF*
get (c)	Возвращает ссылку на поток, из которого выполнялось чтение, и записывает извлеченный символ в *c*
get(buf, num, lim='\n')	Считывает *num* – 1 символов (или пока не встретится символ *lim*) и копирует их в символьную строку *buf*. Вместо символа *lim* в строку записывается признак конца строки ('\0'). Символ *lim* остается в потоке. Возвращает ссылку на текущий поток
getline(buf, num, lim='\n')	Аналогична функции *get*, но копирует в *buf* и символ *lim*
ignore(num=1, lim=EOF)	Считывает и пропускает символы до тех пор, пока не будет прочитано *num* символов или не встретится разделитель, заданный параметром *lim*. Возвращает ссылку на текущий поток
peek ( )	Возвращает следующий символ без удаления его из потока или *EOF*, если достигнут конец файла
putback(c)	Помещает в поток символ *c*, который становится текущим при извлечении из потока
read(buf, num)	Считывает *num* символов (или все символы до конца файла, если их меньше *num*) в символьный массив *buf* и возвращает ссылку на текущий поток
readsome(buf, num)	Считывает *num* символов (или все символы до конца файла, если их меньше *num*) в символьный массив *buf* и возвращает количество считанных символов
seekg(pos)	Устанавливает текущую позицию чтения в значение *pos*
seekg(offs, org)	Перемещает текущую позицию чтения на *offs*байт, считая от одной из трех позиций, определяемых параметром *org*: *ios*::*beg* (от начала файла), *ios*::*cur* (от текущей позиции) или *ios*::*end* (от конца файла)
tellg()	Возвращает текущую позицию чтения потока
unget()	Помещает последний прочитанный символ в поток и возвращает ссылку на текущий поток

Таблица 4.6

Функция	Действие
flush( )	Записывает содержимое потока вывода на физическое устройство
put (c)	Выводит в поток символ *c* и возвращает ссылку на поток
seekg(pos)	Устанавливает текущую позицию записи в значение *pos*
seekg (offs, org)	Перемещает текущую позицию записи на *offs* байт, считая от одной из трех позиций, определяемых параметром *org*: *ios*::*beg* (от начала файла), *ios*::*cur* (от текущей позиции) или *ios*::*end* (от конца файла)
tellg( )	Возвращает текущую позицию записи потока
write(buf, num)	Записывает в поток *num* символов из массива *but*и возвращает ссылку на поток

4.1.3. Файловые потоки ввода/вывода

Как было указано ранее, существуют специальные классы, описанные в *fstream.h*, позволяющие производить ввод / вывод информации в файл:

ifstream — класс входных файловых потоков;

ofstream — класс выходных файловых потоков;

fstream — класс двунаправленных файловых потоков.

Эти классы являются производными соответственно от классов *istream*, *ostream* и*iostream*, поэтому они наследуют перегруженные операции << и >>, флаги форматирования, манипуляторы, методы, состояние потоков и т. д.

Использование файлов в программе предполагает следующие операции:

– создание потока;

– открытие потока и связывание его с файлом;

– обмен (ввод / вывод);

– уничтожение потока;

– закрытие файла.

Каждый класс файловых потоков содержит конструкторы, с помощью которых можно создавать объекты этих классов различными способами:

– конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом:

```
ifstream( );
ofstream( );
fstream( );
```

– конструкторы с параметрами создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

```
ifstream(const char *name, int mode = ios::in);
ofstream(const char *name, int mode = ios::out | ios::trunc);
fstream(const char *name. int mode = ios::in | ios::out);
```

Вторым параметром конструктора является режим открытия файла. Если установленное по умолчанию значение не устраивает программиста, то можно указать другое, составив его из битовых масок, определенных в классе *ios*:

```
enum open_mode
{
    in = 0x01,              // Открыть для чтения
    out = 0x02,             // Открыть для записи
    ate = 0x04,             // Установить указатель на конец файла
    app = 0x08,             // Открыть для добавления в конец
    trunc = 0x10,           // Если файл существует — удалить
    nocreate = 0x20,        // Если файл не существует — выдать ошибку
    noreplace = 0x40.       // Если файл существует — выдать ошибку
    binary= 0x80,           // Открыть в двоичном режиме
};
```

Производить операции ввода / вывода с файлами можно как с помощью конструкторов, так и с помощью методов классов *ifstream* и *ofstream*, которые представлены в табл. 4.7.

Открыть файл в программе можно с использованием либо конструкторов, либо метода *open*, имеющего такие же параметры, как и в соответствующем конструкторе. Например:

```
ifstream def ("input.txt");                    // Использование конструктора
if (!def)
{
    cout<< "Невозможно открыть файл для чтения";
    return 1;
}
ofstream cef;
```

```
cef.open ("output.txt", ios::out);                    // Использование метода open
if(!cef)
{
    cout<< "Невозможно открыть файл для записи";
    return 1;
}
```

Таблица 4.7

Функция	Действие
close( )	Закрывает поток
gcount( )	Возвращает количество введенных последний раз символов
open(fname, mode)	Открывает поток для ввода или вывода, используя те же параметры, что и соответствующий конструктор
read(buf, num)	Считывает *num* символов и возвращает ссылку на текущий поток ввода, *buf* — массив символов
seekg(offset, org)	Перемещает указатель ввода на *offset* байт, считая от одной из трех позиций, определяемых параметром *org*: *ios*::*beg*, *ios*::*cur*, *ios*::*end*
seekg(position)	Перемещает указатель ввода в положение *position*
seekp(offset, org)	Сдвигает указатель на*offset* байт, считая от одной из трех позиций, определяемых параметром *org*: *ios*::*beg*, *ios*::*cur*, *ios*::*end*
seekp(position)	Перемещает указатель в положение *position*
tellg( )	Возвращает позицию указателя ввода файла, имеющую тип *streampos*, определенный в файле *iostream.h*
tellp( )	Возвращает позицию указателя, имеющую тип *streampos*, определенный в файле *iostream.h*
write(buf, num)	Записывает *num*байт в массив символов *buf*

Чтение и запись выполняются либо с помощью операций чтения и извлечения, аналогичных потоковым классам, либо с помощью методов классов. Для примера рассмотрим программу, которая выводит на экран содержимое файла:

```
#include <fstream.h>
int main( )
{
    char text[81], buf[81];
    cout <<"Введите имя файла:";
    cin>> text;
    ifstream cef(text);
    if (!cef)
    {
        cout<< "Ошибка открытия файла";
        return 1;
    }
    while (!cef.eof( ))
    {
        f.getline(buf, 81);
        cout<<buf<<endl;
    }
    return 0;
}
```

Для закрытия потока определен метод *close*( ), но поскольку он неявно выполняется деструктором, явный вызов данного метода необходим только тогда, когда требуется закрыть поток раньше конца его области видимости.

4.1.4. Строковые потоки ввода/вывода

Как было указано ранее, существуют специальные классы, описанные в *strstrea.h*, позволяющие производить ввод / вывод информации в файл:

istrstream — входные строковые потоки;

ostrstream — выходные строковые потоки;

strstream — двунаправленные строковые потоки.

Эти классы являются производными соответственно от классов *istream*, *ostream* и *iostream*, поэтому они наследуют перегруженные операции << и >>, флаги форматирования, манипуляторы, методы, состояние потоков и т. д.

Участки памяти, с которыми выполняются операции чтения и извлечения, по стандарту определяются как строки *C++* (класс *string*), которые позже будут рассмотрены более подробно. Строковые потоки создаются и связываются с этими участками памяти с помощью конструкторов:

```
explicit istrstream(int mode = ios::in);
explicit istrstream(const strings name, int mode = ios::in);
explicit ostrstream(int mode = ios::out);
explicit ostrstream(const strings name, int mode = ios::out);
explicit ststream(int mode = ios::in | ios::out);
explicit strstream(const strings name, int mode = ios::in |ios ::out);
```

Строковые потоки являются некоторым аналогом функций *sscanf* и *sprintf* библиотеки C (далее будут рассмотрены более подробно) и могут применяться для преобразования данных, когда они заносятся в некоторый участок памяти, а затем считываются в величины требуемых типов. Эти потоки могут применяться также для обмена информацией между модулями программы.

В строковых потоках описан метод *str*, возвращающий копию строки или устанавливающий ее значение:

```
string str ( ) const;
void str(const string &s);
```

Проверять строковый поток на переполнение не требуется, поскольку размер строки изменяется динамически.

В приводимом далее примере строковый поток используется для формирования сообщения, включающего текущее время и передаваемый в качестве параметра номер:

```
#include <strstrea.h>
#include <string.h>
#include <iostream.h>
#include <time.h>
using namespace std;
string message(int i)
{
    ostrstream os;
    time_t t;
    time(&t);
    os << "time:" << ctime(&t) << "number:"<< i << endl;
    return os.str( );
}
int main( )
{
    cout<<message (22);
    return 0;
}
```

4.1.5. Ошибочные состояния потоков ввода/вывода.

Каждый поток имеет связанные с ним ошибочные состояния. В базовом классе *ios* определено поле *state*, которое представляет собой состояние потока в виде совокупности битов:

```
enum io_state
{
    goodbit= 0x00,                  //Нет ошибок
    eofbit = 0x01,                  // Достигнут конец файла
    failbit = 0x02,                 // Ошибка форматирования или преобразования
    badbit = 0x04,                  // Серьезная ошибка, после которой
                                    // пользоваться потоком невозможно
    hardfail= 0x08                  // Неисправность оборудования
};
```

Состоянием потока можно управлять с помощью методов и операций, представленных в табл. 4.8.

Таблица 4.8

Операция	Значение
int rdstate( )	Возвращает текущее состояние потока
int eof( )	Возвращает ненулевое значение, если установлен флаг *eofbit*
int fail( )	Возвращает ненулевое значение, если установлен один из флагов *failbit*, *badbit* или *hardfail*
int bad ( )	Возвращает ненулевое значение, если установлен один из флагов *badbit* или *hardfail*
int good ( )	Возвращает ненулевое значение, если сброшены все флаги ошибок
void clear(int = 0)	Параметр принимается в качестве состояния ошибки; при отсутствии параметра состояние ошибки устанавливается 0
operator void *( )	Возвращает нулевой указатель, если установлен хотя бы один бит ошибки
operator ! ( )	Возвращает ненулевой указатель, если установлен хотя бы один бит ошибки

Наиболее часто используются следующие операции с флагами состояния потока:

```
                                    //Проверить, установлен ли флаг
flag: if (stream_obj.rdstate ( ) &ios ::flag)
                                    // Сбросить флаг flag:
stream_obj.clear(rdstate ( ) &~ios::flag)
                                    // Установить флаг flag:
stream_obj.clear(rdstate ( ) | ios::flag)
                                    //Установить флаг flag и сбросить все остальные:
stream_obj.clear( ios::flag)
                                    //Сбросить все флаги:
stream_obj.clear( )
```

Можно также проверять ошибки в потоках, используя логические выражения:

```
if (cin>>x)                         // Ввод успешный
{
…
}
if (!cout)
    cerr<<"Ошибка!";
```

Данный способ возможен потому, что класс *ios* переопределяет следующие операции:

```
int operator !( );
operator void *( );
```

Операция *void* *( ) преобразует поток в указатель, который будет нулевым, если установлены *failbit*, *badbit* или *hardfail*.

Операция (!) возвращает ненулевое значение, если в потоке установлены биты *failbit*, *badbit* или *hardfail*.

§ 4.2. Библиотеки ввода/вывода

4.2.1. Основная библиотека ввода/вывода

Одной из базовых динамических подключаемых библиотек является библиотека ввода / вывода. Описание функций библиотеки содержится в файле *stdio.h*. В данном файле определяется несколько типов и макросов, а также объявляются многочисленные функции, полезные при работе с файлами и устройствами ввода / вывода. Рассмотрим состав *stdio.h*. и его функции:

clearer( ) — «сбрасывает» (очищает) индикаторы достижения конца файла и обнаружения ошибки для файла, на который указывает поток (*stream*).

Описание: void clearer (FILE *stream);

fclose( ) — вызывает очистку потока, адресуемого указателем *stream*, и последующее закрытие соответствующего файла. При успешном закрытии потока возвращаемое значение функции *fclose*( ) равно нулю, если поток уже был закрыт или ошибка — возвращаемое значение *EOF*. Допускается закрытие файлов, на которые указывают *stdin*, *stdout*, *stderr*.

Описание: int fclose (FILE *stream);

feof( ) — проверяет состояние индикатора достижения конца файла для файла, на который указывает *stream*. Если индикатор «очищен», то значение, возвращаемое функцией, равно нулю, в противном случае оно отлично от нуля.

Описание: int feof (FILE *stream);

ferror ( ) — проверяет индикатор ошибок для файла, на который указывает *stream*. Если индикатор «очищен», то значение, возвращаемое функцией, равно нулю, в противном случае оно отлично от нуля.

Описание: int ferror (FILE *stream);

fflush( ) — очищает буфер ввода / вывода открытого потока. Если осуществлялась операция записи в поток, то любые незаписанные данные, находящиеся в буфере вывода, записываются. Функцию *fflush*( ) следует использовать только с потоками, открытыми для вывода или обновления и находящимися в текущий момент в режиме вывода. При успешном выполнении операции функция возвращает нулевое значение; в случае обнаружения ошибки возвращаемым значением является *EOF*.

Описание: int fflush (FILE *stream);

fgetc( ) — берет очередной символ из файла, на который указывает *stream*. Символ читается как данное типа *unsigned char*, а возвращается как данное типа *int*. При обнаружении признака конца файла *fgetc* ( )возвращает *EOF* и для данного потока устанавливается индикатор достижения конца файла. При появлении ошибки чтения возвращаемое значение равно *EOF* и устанавливается индикатор ошибки для данного потока. Для проверки состояния этого индикатора можно использовать функцию *ferror* ( ).

Описание: int fgetc (FILE *stream);

fgetpos ( ) — записывает текущее значение индикатора позиции в файле в объект, на который указывает *pos*. Функция предназначена для управления очень большими файлами, для которых индикатор текущей позиции в файле не может быть представлен как данное типа *long int*. Объект,

адресуемый указателем *pos*, пригоден для использования функцией *fsetpos*( )с целью восстановления в файле предшествующей позиции. При успешном завершении операций значением, которое возвращает *fgetpos*, является нуль.

Описание: int fgetpos (FILE *stream, fpos_t *pos);

fgets( ) — читает не более (*n* – 1) символов из файла, на который указывает *stream*, в массив, адресуемый указателем *s*. После прочтения последнего символа и его записи в массив дописывается символ '\0'. Если встречается «новая строка» или регистрируется факт достижения конца файла, то чтение символов прекращается. Будучи обнаруженным, символ «новая строка» включается в файл. После успешного выполнения операций функция *fgets* ( ) возвращает *s*. Если имеет место достижение конца файла при условии, что еще ни один символ не прочитан, возвращаемым значением является *NULL*, а содержимое массива, на который указывает *s*, остается неизменным.

Описание: char *fgets (char *s, int n, FILE *stream);

fopen( ) — открывает файл, на имя которого указывает *filename* в режиме, задаваемом *mod*e. В результате успешного выполнения функция *fopen*( ) возвращает *FILE* — указатель на открытый поток; в случае неудачи — *NULL*.

Описание: FILE *fopen (const char *filename, const char *mode);

где *mode* указывает на строку символов, содержимым которой должна быть одна из следующих последовательностей символов: *r*, *w*, *a*, *rb*, *wb*, *ab*, *r*+, *w*+, *a*+, *rb*+, *wb*+, *ab*+, *r* + *b*, *w* + *b*, *a* + *b*. Режим *r*означает чтение (*read*), *w* — запись или создание файла (*write*), *a* — добавление (*append*). Если режим *b* не указан, то предполагается, что файл — текстовый поток; при наличии *b* — бинарный поток. Наличие символа + означает, что файл открыт для обновления. Вид *rb* + аналогичен *r* + *b*;

fprintf( ) — помещает форматированные данные в файл, задаваемый посредством *stream*, согласно формату, задаваемому с помощью *format*. Значение, которое возвращает *fprintf*( ) — количество переданных на вывод символов. В случае возникновения ошибки при выводе функция *fprintf* ( )возвращает отрицательное значение.

Описание: int fprintf (FILE *stream, const char *format, …);

Обобщенный формат представляется в следующем виде:

%[*флажки*][*ширина*][.*точность*][*модификатор*]*спецификатор*

Флажки (необязательный параметр):

−	выравнивание по левой границе;
+	наличие ведущего знака;
пробел	наличие ведущего пробела;
#	альтернативная выходная форма;
0	наличие ведущих нулей.

Ширина (необязательный параметр) устанавливает минимальную ширину поля для выводимого значения. Ширина задается одним из двух способов:

– непосредственно, с помощью строки десятичных чисел;

– косвенно, с помощью *.

Точность (необязательный параметр) всегда начинается с точки (**.**), отделяющей ее от соответствующей спецификации ширины. Далее спецификация точности задается аналогично ширине либо непосредственно с помощью десятичных чисел, либо с помощью символа *.

Модификаторы (необязательный параметр):

h — short int;

l — long int;

L — long double.

Спецификаторы:

c — символ;
d — десятичное число со знаком;
e — строчный символ экспоненты;
E — прописной символ экспоненты;
f — число с дробной частью (6 десятичных разрядов);
g — короткое число из двух: *e* или *f*;
G — короткое число из двух: *E* или *F*;
i — десятичное число со знаком;
n — запись счетчика записанных символов в *int*;
0 — восьмеричное число со знаком;
p — указатель на *void*;
s — строка символов;
u — десятичное число без знака;
x — шестнадцатеричное число без знака строчными буквами;
X — шестнадцатеричное число без знака прописными буквами;
% — вывод символа %;

fputc ( ) — помещает символ, задаваемый посредством *c* (преобразуемое в данное типа *unsigned char*), в файл, на который указывает *stream*. При успешном выполнении функции возвращаемым значением является символ, записываемый в файл. При возникновении ошибки записи функция возвращает *EOF*, при этом для заданного потока устанавливается индикатор ошибки. Для проверки состояния данного индикатора можно использовать функцию *ferror* ( ).

Описание: int fputc (int c, FILE *stream);

fputs ( ) — записывает строку символов, адресуемую посредством *s*, в файл, на который указывает *stream*. Символ '\0', завершающий заданную строку символов, записи в файл не подлежит. В случае ошибки функция возвращает *EOF*.

Описание: int fputs (const char *s, FILE *stream);

fread ( ) — читает до *nmemb* элементов, каждый из которых размером *size*, в массив, адресуемый указателем *ptr*, из файла, на который указывает *stream*. При успешном выполнении функция возвращает количество считанных элементов.

Описание: size_t fread (void *ptr, size_t size, size_t nmemb, FILE *stream);

freopen ( ) — практически идентична функции *fopen*, за исключением того, что используется заново существующий *FILE*-указатель, который адресуется к текущему открытому файлу. Назначение параметров функции также аналогично *fopen*.

Описание: FILE *freopen (const char *filename, const char *mode, FILE *stream);

fscanf ( ) — читает форматированные входные данные из файла, на который указывает *stream*, согласно формату, задаваемому посредством *format*. Возвращаемым значением функции является число элементов, получивших значение при вводе. Если ошибка, то значение, которое возвращает функция — *EOF*.

Описание: int fscanf (FILE *stream, const char *format, …)

Обобщенный формат представляется в следующем виде:

%[*][*ширина*][*модификатор*]*спецификатор*

Символ * — символ подавления (необязательный параметр), который обеспечивает пропуск элементов входных данных. Значение *ширины*, *модификатора* и *спецификатора* аналогичны функции *fprintf* ( );

fseek ( ) — присваивает параметру *offset* значение индикатора текущей позиции в файле, исходя из значения *whence*. В случае успешного выполнения функция очищает индикатор конца файла, отбрасывая любые символы, которые могли быть «вытолкнуты» в поток *stream*, и возвращает нуль. Для очень больших файлов, в которых значение индикатора текущей позиции не может быть представлено как данное типа *long int*, следует использовать функцию *fsetpos* ( ).

Описание: int fseek (FILE *stream, long int offset, int whence);

В качестве параметра *whence* можно использовать одно из трех значений:

- SEEK_SET (начало файла);
- SEEK_CUR (текущая позиция);
- SEEK_END (конец файла);

fsetpos ( ) — присваивает значение индикатора текущей позиции в файле, на который указывает *stream*, объекту, адресуемому указателем *pos*. В случае успешного выполнения *fsetpos* ( ) возвращает нуль, очищает индикатор конца файла и отбрасывает символы, которые могли быть «вытолкнуты» обратно в поток *stream* посредством функции *ungetc* ( ).

Описание: int fsetpos(FILE *stream, const fpos_t *pos);

ftell ( ) — возвращает значение индикатора текущей позиции в файле, на который указывает *stream*. В случае неудачи значение, возвращаемое функцией, равно –1*L*, а *errno* присваивается положительное значение. Для очень больших файлов, в которых значение индикатора текущего положения не может быть представлено как данное типа *long int*, следует применять функцию *fgetpos* ( ).

Описание: long int ftell (FILE *stream);

fwrite ( ) — помещает до *nmemb* элементов, каждый из которых размером *size*, из массива, адресуемого указателем *ptr*, в файл, на который указывает *stream*. Возвращаемое значение — число успешно прочитанных элементов. Если ошибка, то значение индикатора позиции в файле не определено.

Описание: size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *stream);

getc ( ) — извлекает очередной символ из файла, на который указывает *stream*. Функция *getc* ( ) эквивалентна *fgetc* ( ) за исключением того, что *getc* ( ) допускает существование одноименного макроса.

Описание: int getc (FILE *stream);

getchar ( ) — извлекает очередной символ из стандартного потока *stdin*. Функция эквивалентна функции *getc* (*stdin*).

Описание: int getchar (void);

gets ( ) — читает символы из *stdin* в массив, на который указывает *s*, до тех пор, пока не встретится «новая строка» или не будет достигнут конец файла. После записи последнего прочитанного символа в массив добавляется '\0'. Если встречается «новая строка», то она отбрасывается (в отличие от *fgets* ( ), удерживающей «новую строку»). Если выполнение *gets* ( ) успешно, то возвращается *s*.

Описание: char *gets (char *s);

perror ( ) — записывает в *stderr* сообщение, соответствующее текущему значению *errno*. Сообщение включает определяемую пользователем строку символов, адресуемую указателем *s*. Выходному сообщению предшествует адресуемая указателем *s* строка символов, за которой следуют двоеточие и пробел, при условии, что значение *s* не является *NULL* и не указывает на пустую строку.

Описание: void perror (const char *s);

printf ( ) — пишет форматированные выходные данные в *stdout* согласно строке форматирования *format*. Вызов функции *printf* ( ) эквивалентен вызову *fprintf* ( ), если в последнем случае в качестве выходного потока используется *stdout*. Функция возвращает количество переданных на вывод символов. Обобщенный формат аналогичен функции *fprintf* ( ).

Описание: int printf (const char *format, …);

putc ( ) — записывает символ, задаваемый параметром *c* (преобразуемым в *unsigned char*), в файл, на который указывает *stream*. Функция *putc* эквивалентна *fputc* ( ), за исключением того, что *putc* ( ) может являться макросом.

Описание: int putc (int c, FILE *stream);

putchar ( ) — записывает символ, задаваемый параметром *c*, в *stdout*. Функция эквивалентна функции *putc* ( ), записывающей символ в *stdout*.

Описание: int putchar (int c);

puts( ) — записывает адресуемую указателем *s* строку символов, за которой следует «новая строка», в *stdout*. Завершающий строку символ '\0' не записывается. При ошибке *puts* возвращает *EOF*. В отличие от *fputs* ( )функция *puts* ( ) дополняет запись «новой строкой».

Описание: int puts (const char *s);

remove ( ) — делает недоступным файл, имя которого адресуется указателем *filename*. В случае успешного выполнения функция возвращает нулевое значение.

Описание: int remove (const char *filename);

rename ( ) — меняет старое имя файла, на которое указывает *old*, на новое имя, адресуемое указателем *new*. В случае успешного выполнения функция возвращает нуль.

Описание: int rename (const char *old, const char *new);

rewind ( )— устанавливает индикатор положения в файле на начало файла. Вызов функции идентичен вызову *fseek* ( ) со смещением *offset*, равным 0*L*, и точкой отсчета *SEEK_SET*. Однако *rewind*( ) выполняет также очистку индикатора ошибок.

Описание: void rewind (FILE *stream);

scanf ( ) — читает форматированные входные данные из *stdin* согласно строке форматирования, на которую указывает *format*. Возвращаемым значением функции является число элементов, получивших значение при вводе. Если ошибка, то значение, которое возвращает функция — *EOF*. Все параметры, передаваемые функции, должны быть адресами.

Описание: int scanf (const char *format, …)

Обобщенный формат представляется в следующем виде:

%[*][*ширина*][*модификатор*]с*пецификатор*

Значения *, *ширины*, *модификатора* и *спецификатора* определяются аналогично функции *fscanf* ( );

setvbuf ( ) — позволяет изменить тип буферизации для заново открываемого файла. Она также дает пользователю возможность предоставить файлу назначаемый программистом буфер. Функция должна вызываться прежде, чем будут выполнены какие-либо операции чтения или записи для вновь открытого потока. В результате успешного выполнения функция возвращает значение «нуль».

Описание: int setvbuf (FILE *stream, char *buf, int mode, size_t size);

Параметр *mode* может принимать одно из следующих значений:

_IOFBF — полная буферизация;
_IOLBF — буферизация строк;
_IONBF — без буферизации.

Если значением параметра *buf* является *NULL*, то функция использует свой внутренний буфер, иначе используется буфер, на который указывает *buf*; и в данном случае значение параметра *size* должно быть не меньше размера массива, на который указывает *buf*.

setbuf( ) — позволяет изменить тип буферизации для заново открываемого файла. Функция эквивалентна функции *setvbuf* ( ), если последняя вызывается либо с параметром *mode*, равным _IOFBF, и параметром *size*, равным *BUFSIZ*, либо с параметром *mode*, равным *_IONBF*, и параметром *buf*, равным *NULL*. Функция не возвращает никакого значения.

Описание: void setbuf (FILE *stream, char *buf);

sprint ( ) — записывает форматированные выходные данные в строку символов, на которую указывает *s*, в формате, задаваемом строкой форматирования, на которую указывает *format*. Функция возвращает количество символов, записанных в строку. Это число не включает символ '\0', добавляемый автоматически в конец строки. Значение параметра формат определяется аналогично функции *fprintf* ( ).

Описание: int sprintf (char *s, const char *format, …);

sscanf ( ) — читает форматированные входные данные из строки символов, на которую указывает *s*, используя строку форматирования, на которую указывает *format*. Функция возвращает число элементов ввода, при ошибке — *EOF*. Все параметры, подлежащие передаче данной функции, должны быть адресами. Подробное описание строки *forma*t приводится в функции *fscanf* ( ).

Описание: int sscanf (const char *s, const char *format, …);

tmpfile ( ) — создает временный бинарный файл, удаляемый после его закрытия или нормального завершения выполнения программы. Функция открывает файл для обновления, как будто он открыт посредством функции *fopen*( ) с заданием режима "*wb+*". Если временный файл не может быть создан, то возвращаемым значением функции является *NULL*, иначе возвращается указатель на *FILE*.

Описание: FILE *tmpfile(void);

tpmnam ( ) — создает и возвращает в качестве результата имя файла. Далее созданный файл может быть открыт посредством *fopen* ( ). У функции нет способа передачи информации об ошибке.

Описание: char *tmpnam (char *s);

ungetc ( ) — «выталкивает» символ *c* (после преобразования его в *unsigned char*) обратно во входной поток, на который указывает *stream*. При успешном выполнении функция возвращает *c* и индикатор признака конца файла для данного потока очищается, в противном случае возвращаемое значение — *EOF*.

Описание: int ungetc (int c, FILE *stream);

vfprintf ( ) — функция эквивалентна *fprintf* ( ), за исключением того, что список параметров заменен списком с переменным числом параметров (*arg*), который инициализируется макросом *va_start*. При успешном выполнении функция возвращает число переданных символов.

Описание: int vfprintf (FILE *stream, const char *format, va_list arg);

vprintf ( ) — функция эквивалентна *printf* ( ), за исключением того, что список параметров заменен списком с переменным числом параметров (*arg*), который инициализируется макросом *va_start*. При успешном выполнении функция возвращает число переданных символов.

Описание: int vfprintf (const char *format, va_list arg);

vsprintf ( ) — функция эквивалентна *sprint* ( ), за исключением того, что список параметров заменен списком с переменным числом параметров (*arg*), который инициализируется макросом *va_start*. При успешном выполнении функция возвращает число символов, записанных в строку *s* (исключая символ '\0').

Описание: int vfprintf (char *s, const char *format, va_list arg);

Типы и макросы

BUFSIZ — макрос, являющийся целочисленным константным выражением, задающим размер буфера, используемого функцией *setbuf*( ). Значение должно быть не менее 256;

EOF — макрос, являющийся целочисленным константным выражением, принимающим отрицательное значение, при достижении функциями конца файла;

FILE — объект, позволяющий хранить в себе «текущий контекст» открытого файла. Информация включает детальные сведения о буферизации, флажки индикации ошибок и достижения конца файла, индикатор текущей позиции в файле;

FILENAME_MAX — макрос, являющийся целочисленным константным выражением, значение которого — максимально допустимая длина строки символов, являющейся именем файла;

FOPEN_MAX — макрос, являющийся целочисленным константным выражением, значение которого — максимальное число файлов, которое можно открыть одновременно;

L_tmpnam — макрос, являющийся целочисленным константным выражением, значение которого — размер массива символов, достаточно большой для размещения имени временного файла, создаваемого функцией *tmpnam* ( );

NULL — макрос, расширяемый до определяемой реализацией системы константы в виде *null*-указателя;

TMP_MAX — макрос, являющийся целочисленным константным выражением, представляющим максимальное количество уникальных имен файлов, которые могут быть созданы функцией *tmpnam* ();

fpos_t — объект типа, используемый функциями *fgetpos*( ) и *fsetpos* ( ). Данные этого типа отображаются в данные типа бóльшего размера, достаточного для размещения самого большего возможного значения индикатора текущей позиции в файле для конкретной реализации системы программирования;

size_t — тип результата, порождаемый операцией вычисления размера объекта (*sizeof*);

stderr — макрос, являющийся выражением типа *FILE**, значение которого — указатель на объект типа *FILE*, соответствующий «стандартному устройству приема ошибок». По умолчанию связывается с терминалом;

stdin — макрос, являющийся выражением типа *FILE**, значение которого — указатель на объект типа *FILE*, соответствующий стандартному устройству ввода. По умолчанию связывается с терминалом;

stdout — макрос, являющийся выражением типа *FILE**, значение которого — указатель на объект типа *FILE*, соответствующий стандартному устройству вывода. По умолчанию связывается с терминалом.

4.2.2. Библиотека нестандартных операций ввода / вывода

Помимо *stdio.h* в языке C++ применяется еще одна библиотека, в состав которой входят средства поддержки разнообразных нестандартных операций ввода / вывода. Функции данной библиотеки описываются в файле *conio.h*. Рассмотрим основные функции *conio.h*:

cgets( ) — считывает символьную строку с консоли и сохраняет ее в буфере, указываемом параметром *str*. До вызова функции аргумент *str*[0] должен быть установлен на максимальную длину считываемой строки. При возврате *str*[1] содержит количество считанных символов. Символы хранятся, начиная с *str*[2], и заканчиваются нулевым символом. При успешном завершении функция возвращает указатель на *str*[2].

Описание: char *cgets (char *str);

clreol( ) — удаляет все символы от позиции курсора до конца строки в текущем текстовом окне без перемещения курсора.

Описание: void clreol (void);

clrscr( ) — очищает окно в текстовом режиме и перемещает курсор в верхний левый угол экрана (в позицию 1, 1).

Описание: void clrscr (void);

cprintf( ) — осуществляет форматированный вывод на экран. Функция возвращает количество выведенных символов. Спецификация строки *format* аналогична функции *fprintf* ( ).

Описание: int cprintf (const char *format, …);

cputs( ) — выводит строку *str*, заканчивающуюся нулем, в текущее текстовое окно экрана. Функция возвращает последний выведенный символ.

Описание: int cputs (const char *str);

cscanf( ) — считывает с консоли и просматривает набор вводимых полей по одному символу. Далее каждое поле форматируется в соответствии со спецификацией формата, передаваемого функцией в строке *format*. Далее функция помещает отформатированный ввод по адресам и отображает введенные символы на экране. Спецификация строки *format* приведена в функции *fscanf* ( ). Функция возвращает число успешно введенных, преобразованных и сохраненных полей. Если ни одно поле не было сохранено, возвращается 0.

Описание: int cscanf (char *format, …);

delline( ) — удаляет строку, в котором находится курсор, в текстовом окне и сдвигает все строки ниже удаленной на одну вверх.

Описание: void delline (void);

farheapfillfree( ) — заполняет блоки глобального *heap* константным значением. При успешном завершении функция возвращает значение больше 0.

Описание: int farheapfillfree (unsigned int fillvalue);

getch( ) — читает один символ с консоли без вывода на экран. Функция возвращает введенный с клавиатуры символ.

Описание: int getch (void);

getche( ) — читает символ с консоли и одновременно отображает его в текущем текстовом окне на экране. Функция возвращает введенный с клавиатуры символ.

Описание: int getche (void);

getpass— считывает пароль с системной консоли после выдачи на экран специального сообщения — символьной строки с нулевым окончанием (*promt*) — и отменяет отображение пароля на экране. Возвращаемое значение — указатель на строку символов типа *static*, которая перезаписывается при каждом вызове.

Описание: char *getpass (char *promt);

gettextinfo( ) — получает информацию о текстовом режиме. Функция заполняет структуру типа *text_info*, на которую указывает параметр *r*, информацией о текущем текстовом режиме.

Описание: void gettextinfo (struct text_info *r);

gotoxy( ) — перемещает курсор в текстовом окне в указанную позицию. Если координаты указаны неверно, то вызов функции игнорируется.

Описание: void gotoxy (int x, int y);

insline( ) — вставляет в текущем текстовом окне пустую строку.

Описание: void insline (void);

kbhit( ) — проверяет, была ли нажата какая-либо клавиша клавиатуры. Если клавиша была нажата, то функция возвращает ненулевое значение, иначе возвращает нуль.

Описание: int kbhit (void);

putch ( ) — выводит символ *c* в текущее текстовое окно. При успешном завершении функция выведенный символ.

Описание: int putch (int c);

ungetch( ) — помещает символ *ch* назад, в буфер клавиатуры. Функция приводит к тому, что *ch* становится следующим вводимым символом. Функция возвращает *ch* в случае успешного выполнения.

Описание: int ungetch (int ch);

wherex( ) — возвращает координату *x* текущей позиции курсора.

Описание: int wherex (void);

wherey( ) — возвращает координату *y* текущей позиции курсора.

Описание. int wherey (void);

Рассмотрим примеры на основные функции библиотек ввода/вывода:

1. Необходимо вывести на экран монитора текст «Здравствуй, мир» посредством функции *printf* ( ). Текст программы:

```
#include <stdio.h>
int main(void)
{
    /* Вывод данных на экран*/
    printf ("Здравствуй, мир");
    return 0;
}
```

2. Выполнить задачу, аналогичную предыдущему примеру, но осуществить посимвольный вывод посредством функции *putc* (). Текст программы:

```
#include <stdio.h>
int main(void)
{
    /* Инициализация символьного массива */
    char msg [ ] = "Здравствуй, мир\n";
    int i = 0;
    while (msg[i])                          // Цикл по выводу символов массива на экран
        putc(msg[i++], stdout);
    return 0;
}
```

3. Необходимо написать программу вычисления площади прямоугольника, в которой размеры сторон прямоугольника вводятся с клавиатуры. При этом перед каждым вводом данных программа выводит текст с просьбой ввести конкретный параметр. Текст программы:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    float l,w,s;
/* Предварительная очистка экрана */
/* Вывод заголовка программы на экран */
    printf("Программа вычисления площади прямоугольника\n");
    printf("Введите исходные данные:\n");
    printf("Длина (см):");
/* Ввод первого параметра — длины */
    scanf("%f", &l);
    printf("Ширина (см):");
/* Ввод второго параметра — ширины */
    scanf("%f", &w);
```

```
/* Вычисление площади прямоугольника */
    s=l*w;
/* Вывод величины площади на экран */
    printf("Площадь прямоугольника:%10.2fкв.см\n", s);
    printf("\n Для завершения нажмите любую клавишу");
/* Программа в ожидании ввода очередного символа.
В этот момент на экране можно рассмотреть всю введенную и выведенную информацию */
    getch( );
}
```

4. Необходимо написать программу, которая производит запись в кодах *ASCII* трех переменных типа *int*, *float* и *char* в файл на «жесткий» диск. Текст программы:

```
#include<stdio.h>
int main(void)
{
    FILE *stream;
    int i = 100;
    char c = 'C';
    float f = 1.234;
/* Открытие файла для записи */
    stream = fopen("File1.txt", "w+");
/* Запись данных в файл */
    fprintf(stream, "%d %c %f", i, c, f);
/* Закрытие файла */
    fclose(stream);
    return 0;
}
```

5. Необходимо написать программу, которая производит считывание из файла и вывод на экран символов, записанных в предыдущем примере. Текст программы:

```
#include<stdio.h>
#include <conio.h>
int main(void)
{
    FILE *stream;
    int i;
    char c;
    float f;
/* Открытие файла для считывания данных */
    stream = fopen("File1.txt", "r+");
/* Чтение данных из файла */
    fscanf(stream, "%d %c %f", &i, &c, &f);
/* Очистка экрана */
    clrscr( );
/* Вывод данных на экран*/
    printf ("i=%d c=%c f=%f", i, c, f);
    getch( );
/* Закрытие файла */
    fclose(stream);
    return 0;
}
```

6. Необходимо написать программу, которая производит запись в бинарном виде символьного массива на диск, считывание данных с диска в другой массив и вывод этого массива на экран. Текст программы:

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
```

```
    FILE *stream;
    char msg [ ] = "Это проверочный тест";
    char buf[25];
    clrscr( );
    if ((stream = fopen("FILE2.TXT", "w+"))== NULL)
    {
            fprintf(stderr, "Невозможно открыть файл.\n");
            return 1;
    }
/* Запись данных в файл */
    fwrite(msg, strlen(msg)+1, 1, stream);
/* Установка указателя в начало файла */
    fseek(stream, SEEK_SET, 0);
/* Чтение данных в массив и вывод на экран */
    fread(buf, strlen(msg)+1, 1, stream);
    printf("%s\n", buf);
    getch( );
/* Закрытие файла */
    fclose(stream);
    return 0;
}
```

§ 4.3. Библиотеки математических функций и макросов, характеризующих свойства целых чисел и чисел с плавающей точкой

4.3.1. Библиотека математических функций

Библиотека включает средства поддержки математических функций. Описание функций содержится в заголовочном файле *math.h*. Основные функции *math.h*:

acos( ) — вычисляет арккосинус своего аргумента *x* и возвращает в качестве результата значение (в радианах) в диапазоне [0, π].

Описание: double acos (double x);

asin( ) — вычисляет значение арксинуса для заданного аргумента *x*; возвращаемое значение измеряется в радианах и принадлежит диапазону $[-\pi/2, +\pi/2]$.

Описание: double asin (double x);

atan( ) — вычисляет значение арктангенса для заданного аргумента *x*; возвращаемое значение измеряется в радианах и принадлежит диапазону $[-\pi/2, +\pi/2]$.

Описание: double atan (double x);

atan2( ) — вычисляет значение арктангенса x/y; возвращаемое значение измеряется в радианах и принадлежит диапазону $[-\pi, +\pi]$.

Описание: double atan2 (double x, double y);

ceil( ) — вычисляет наименьшую целую часть, которая не меньше, чем аргумент *x*.

Описание: double ceil (double x);

cos( ) — вычисляет косинус своего аргумента *x*, значение которого измеряется в радианах.

Описание: double cos (double x);

cosh( ) — вычисляет гиперболический косинус своего аргумента *x*, значение которого измеряется в радианах.

Описание: double cosh (double x);

exp( ) — вычисляет экспоненциальное значение своего аргумента *x*. Если величина аргумента слишком велика, возникает ошибка выхода за диапазон допустимых значений.

Описание: double exp (double x);

fabs( ) — вычисляет абсолютное значение аргумента *x*, представляющего собой число с плавающей точкой.

Описание: double fabs (double x);

floor( ) — вычисляет наибольшую целую часть аргумента *x*, значение которой не превосходит значения *x*.

Описание: double floor (double x);

fmod( ) — вычисляет остаток от деления *x*/*y* согласно арифметике чисел с плавающей точкой. Значение, возвращаемое функцией, равно *x* – *i* * *y*, где *i* — такое целое число, что при ненулевом значении *y* знак результата совпадает со знаком *x*, а величина значения результата меньше, чем величина значения*y*.

Описание: double fmod (double x, double y);

frexp( ) — разбивает число с плавающей точкой *value* на нормализованную дробную часть как степень числа 2. Целочисленная степень записывается в области памяти, на которую указывает *exp*, а дробная часть используется как значение, которое возвращает функция.

Описание: double frexp (double value, int *exp);

ldexp( ) — умножает число с плавающей точкой *x* на 2 в целочисленной степени *exp*.

Описание: double ldexp (double x, int exp);

log( ) — вычисляет натуральный логарифм своего аргумента *x*.

Описание: double log (double x);

log10( ) — вычисляет десятичный логарифм своего аргумента *x*.

Описание: double log10 (double x);

modf( ) — разделяет число с плавающей точкой на целую и дробную части. Каждая часть имеет тот же знак, что и исходное число. Целая часть записывается в области памяти, на которую указывает *iptr*, а дробная часть возвращается функцией.

Описание: double modf (double value, double *iptr);

pow( ) — возводит *x* в степень *y*.

Описание: double pow (double x, double y);

sin( ) — вычисляет синус своего аргумента *x*, значение которого измеряется в радианах.

Описание: double sin (double x);

sinh ( ) — вычисляет гиперболический синус своего аргумента *x*, значение которого измеряется в радианах.

Описание: double sinh (double x);

sqrt( ) — вычисляет неотрицательный квадратный корень своего аргумента *x*.

Описание: double sqrt (double x);

tan( ) — вычисляет тангенс своего аргумента *x*, измеряемого в радианах.

Описание: double tan (double x);

tanh( ) — вычисляет гиперболический тангенс своего аргумента *x*, измеряемого в радианах.

Описание: double tanh (double x);

4.3.2. Библиотека макросов, описывающих характеристики целого типа системы программирования

Описание макросов содержится в заголовочном файле *limits.h*.

Основные макросы библиотеки:

CHAR_BIT — содержит информацию о количестве битов, формирующих данное типа *char*;

CHAR_MAX — содержит информацию о максимальном значении объекта типа *char*;

CHAR_MIN — содержит информацию о минимальном значении объекта типа *char*;

INT_MAX — содержит информацию о максимальном значении объекта типа *int*;

INT_MIN — содержит информацию о минимальном значении объекта типа *int*;

LONG_MAX — содержит информацию о максимальном значении объекта типа *long int*;

LONG_MIN — содержит информацию о минимальном значении объекта типа *long int*;

MB_LEN_MAX — обозначает максимальное число байт в многобайтовом символе;

SCHAR_MAX — содержит информацию о максимальном значении объекта типа *signed char*;

SCHAR_MIN — содержит информацию о минимальном значении объекта типа*signed char*;

CHRT_MAX — содержит информацию о максимальном значении объекта типа *short int*;

CHRT_MIN — содержит информацию о минимальном значении объекта типа *short int*;

UCHAR_MAX — содержит информацию о максимальном значении объекта типа *unsigned char*;

UINT_MAX — содержит информацию о максимальном значении объекта типа *unsigned int*;

ULONG_MAX — содержит информацию о максимальном значении объекта типа *unsigned long int*;

UCHRT_MAX — содержит информацию о максимальном значении объекта типа *unsigned short int*.

4.3.3. Библиотека макросов, описывающих свойства арифметики с плавающей точкой

Описание макросов содержится в заголовочном файле *float.h*.

Основные макросы библиотеки:

DBL_DIG — указывает такое количество десятичных цифр, что при выполнении преобразования с округлением числа типа *double* с этим количеством значащих цифр в число с плавающей точкой и обратно указанные десятичные цифры не претерпевают изменений;

DBL_EPSILON — обозначает разницу между 1.0 и наименьшим из значений, бóльших 1.0, которое представимо в виде числа типа *double*;

DBL_MANT_DIG — обозначает количество значащих цифр числа с плавающей точкой типа *double*, имеющей основание системы счисления *FLT_RADIX*;

DBL_MAX — обозначает максимальное представимое конечное число типа *double*;

DBL_MAX_10_EXP — максимальное целое число, которое, будучи степенью числа 10, дает число, не выходящее за диапазон представимых конечных чисел с плавающей точкой;

DBL_MAX_EXP — обозначает максимальное целое число, которое, будучи уменьшенным на единицу степенью числа *FLT_RADIX*, дает представимое конечное число с плавающей точкой;

DBL_MIN — обозначает минимальное нормализованное положительное число типа *double*;

DBL_MIN_10_EXP — минимальное отрицательное число, которое, будучи степенью числа 10, дает число, не выходящее за диапазон значений нормализованных чисел с плавающей точкой;

DBL_MIN_EXP — обозначает минимальное целое число, которое, будучи уменьшенным на единицу степенью числа *FLT_RADIX*, дает представимое конечное число с плавающей точкой;

FLT_DIG — указывает такое количество десятичных цифр, что при выполнении преобразования с округлением числа типа *float* с этим количеством значащих цифр в число с плавающей точкой и обратно указанные десятичные цифры не претерпевают изменений;

FLT_EPSILON — обозначает разницу между 1.0 и наименьшим из значений, бóльших 1.0, которое представимо в виде числа типа *float*;

FLT_MANT_DIG — обозначает количество значащих цифр числа с плавающей точкой типа *float*, имеющей основание системы счисления *FLT_RADIX*;

FLT_MAX — обозначает максимальное представимое конечное число типа *float*;

FLT_MAX_10_EXP — максимальное целое число, которое, будучи степенью числа 10, дает число, не выходящее за диапазон представимых конечных чисел с плавающей точкой;

FLT_MAX_EXP — обозначает максимальное целое число, которое, будучи уменьшенным на единицу степенью числа *FLT_RADIX*, дает представимое конечное число с плавающей точкой;

FLT_MIN — обозначает минимальное нормализованное положительное число типа *float*;

FLT_MIN_10_EXP — минимальное отрицательное число, которое, будучи степенью числа 10, дает число, не выходящее за диапазон значений нормализованных чисел с плавающей точкой;

FLT_MIN_EXP — обозначает минимальное целое число, которое, будучи уменьшенным на единицу степенью числа *FLT_RADIX*, дает представимое конечное число с плавающей точкой;

FLT_RADIX — обозначает основание степени (экспоненциальное представление числа);

FLT_ROUNDS— обозначает текущий режим выполнения округления;

LDBL_DIG — указывает такое количество десятичных цифр, что при выполнении преобразования с округлением числа типа *long double* с этим количеством значащих цифр в число с плавающей точкой и обратно указанные десятичные цифры не претерпевают изменений;

LDBL_EPSILON — обозначает разницу между 1.0 и наименьшим из значений, бóльших 1.0, которое представимо в виде числа типа *long double*;

LDBL_MANT_DIG — обозначает количество значащих цифр числа с плавающей точкой типа *long double*, имеющей основание системы счисления *FLT_RADIX*;

LDBL_MAX — обозначает максимальное представимое конечное число типа *long double*;

LDBL_MAX_10_EXP — максимальное целое число, которое, будучи степенью числа 10, дает число, не выходящее за диапазон представимых конечных чисел с плавающей точкой;

LDBL_MAX_EXP — обозначает максимальное целое число, которое, будучи уменьшенным на единицу степенью числа *FLT_RADIX*, дает представимое конечное число с плавающей точкой;

LDBL_MIN — обозначает минимальное нормализованное положительное число типа *long double*;

LDBL_MIN_10_EXP — минимальное отрицательное число, которое, будучи степенью числа 10, дает число, не выходящее за диапазон значений нормализованных чисел с плавающей точкой;

LDBL_MIN_EXP — обозначает минимальное целое число, которое, будучи уменьшенным на единицу степенью числа *FLT_RADIX*, дает представимое конечное число с плавающей точкой.

Рассмотрим пример на использование библиотеки математических функций.

Необходимо написать программу, которая считает десятичный логарифм числа 800,6872. Текст программы:

```
#include <math.h>
#include <stdio.h>
int main(void)
{
/* Инициализация переменных */
      double result;
      double x = 800.6872;
/* Вычисление результата */
      result = log10(x);
/* Вывод данных на экран */
      printf("Десятичный логарифм числа %lf равен %lf\n", x, result);
      return 0;
}
```

§ 4.4. Библиотеки преобразования символов и строк

Язык *C++* содержит несколько библиотек преобразования символов и строк. Рассмотрим их.

4.4.1. Библиотека функций работы с символами и строками, имена которых имеют форму *str** и *mem**

Описание функций содержится в заголовочном файле *string.h*. В данном файле содержатся объявления семейства библиотечных функций работы с символами и строками, имена которых имеют форму *str** и *mem**. Основные функции библиотеки:

memchr ( ) — осуществляет поиск символа *c* среди первых *n* символов строки *s*. Если символ *c* найден, то функция возвращает указатель на него, в противном случае — *NULL*.

Описание: void *memchr (const void *s, int c, size_t n);

memcmp ( ) — сравнивает *n* символов, на местоположение которых указывает *s*2, с символами, на местоположение которых указывает *s*1. Если символ найден, то функция возвращает указатель на него, в противном случае — *NULL*.

Описание: void *memcmp (const void *s1, const void *s2, size_t n);

memcpy ( ) — копирует *n* символов из области памяти, на которую указывает *s*2, в область хранения символов, на которую указывает *s*1. Если *s*1 и *s*2 перекрываются, то поведение функции не определено. Функция возвращает *s*1.

Описание: void *memcpy (void *s1, const void *s2, size_t n);

memmove ( ) — копирует *n* символов из области памяти, на которую указывает *s*2, в область памяти для хранения символов, на которую указывает *s*1. Копирование будет произведено корректно, даже если блоки перекрываются. Функция возвращает *s*1.

Описание: void *memmove (void *s1, const void *s2, size_t n);

memset ( ) — присваивает значение с первым *n* символам объекта, на который указывает *s*. Возвращаемым значением функции является *s*.

Описание: void *memset (void *s, int c, size_t n);

strcat ( ) — копирует строку символов, на которую указывает *s*2, в конец строки символов, на которую указывает *s*1. В процессе копирования символ '\0', замыкающий строку *s*1, стирается, а на его место помещается первый символ строки *s*2. Возвращаемое значение адресуется указателем *s*1.

Описание: char *strcat (char *s1, const char *s2);

strchr ( ) — выполняет поиск символа *c* в строке символов *s*. При обнаружении символа функция возвращает указатель на местоположение *c* в строке *s*.

Описание: char *strchr (const char *s, int c);

strcmp ( ) — выполняет беззнаковое сравнение строк *s*1 и *s*2, начиная с первого символа в каждой строке и продолжая до тех пор, пока не встретятся несовпадающие символы.

Описание: int strcmp (const char *s1, const char *s2);

strcoll ( ) — сравнивает строку символов, на которую указывает *s*2, со строкой символов, на которую указывает *s*1 в соответствии со списком, определяемым с помощью *setlocale*.

Описание: int strcoll (const char *s1, const char *s2);

strcpy ( ) — копирует строку символов, на которую указывает *s*2, в область памяти, где находится строка символов , адресуемая указателем *s*1. Возвращаемым значением является *s*1.

Описание: char *strcpy (char *s1, const char *s2);

strspn ( ) — выполняет поиск самой длинной строки символов в области памяти, адресуемой указателем *s*1; условием поиска является отсутствие в искомой строке символов, содержащихся в строке, на которую указывает *s*2.

Описание: char *strspn (char *s1, const char *s2);

strerror ( ) — возвращает адрес строки символов, содержащей сообщение, которое соответствует коду сообщения, используемому в качестве параметра. Функция пишет сообщение непосредственно в *stderr* вместе с текстом, определяемым пользователем.

Описание: char *strerror (int errnum);

strlen ( ) — возвращает число символов в строке, на которую указывает *s*.

Описание: size_t strlen (const char *s);

strncat ( ) — копирует не более чем *n* первых символов из строки, на которую указывает *s*2, в конец строки, на которую указывает *s*1. Возвращаемое значение адресуется указателем *s*1.

Описание: char *strncat (char *s1, const char *s2, size_t n);

strncmp ( ) — сравнивает не более чем *n* символов из строки, на которую указывает *s*2, со строкой, на которую указывает *s*1.

Описание: int strncmp (const char *s1, const char *s2, size_t n);

strncpy ( ) — копирует не более чем *n* символов из строки, на которую указывает *s*2, в строку, адресуемую указателем *s*1. Возвращаемое значение адресуется указателем *s*1.

Описание: char *strncpy (char *s1, const char *s2, size_t n);

strpbrk ( ) — выполняет поиск в строке символов, на которую указывает *s*1, первого из символов строки, на которую указывает *s*2. Возвращаемое значение — указатель на символ.

Описание: char *strpbrk (const char *s1, const char *s2);

strspn ( ) — ищет самую длинную строку символов, начиная с адреса *s*1, при условии, что она должна содержать только те символы, которые находятся в строке, адресуемой указателем *s*2. Возвращаемое значение — длина строки в *s*1.

Описание: size_t strspn (const char *s1, const char *s2);

strstr ( ) — выполняет поиск в строке, на которую указывает *s*1, подстроки, на которую указывает *s*2. Возвращаемое значение — местоположение подстроки, адресуемой *s*2.

Описание: char *strstr (const char *s1, const char *s2);

strtok ( ) — применяется для расчленения строки, адресуемой указателем *s*1, на цепочку, завершаемую символом '/ 0' лексем, используя символы-терминаторы лексем, задаваемые строкой, на которую указывает *s*2. Возвращаемое значение — указатель на найденную лексему, или *NUL*L, если лексема не обнаружена.

Описание: char *strtok (char *s1, const char *s2);

strxfrm ( ) — преобразует строку символов, на которую указывает *s*2, в другую строку, на которую указывает *s*1. В строке *s*1 может быть не более *n* преобразованных символов.

Описание: size_t strxfrm (char *s1, const char *s2, size_t n);

4.4.2. Библиотека функций преобразования и тестирования символов

Функции данной библиотеки описаны в заголовочном файле *ctype.h*. Файл *ctype.h* содержит описание следующих функций:

isalnum ( ) — проверяет, является ли аргумент с алфавитным или десятично-цифровым символом. Если результат проверки отрицательный, значение — ноль.

Описание: int isalnum (int c);

isalpha ( ) — проверяет, является ли аргумент *c* символом алфавита. Если результат проверки отрицательный, значение функции — ноль.

Описание: int isalpha (int c);

isascii ( ) — проверяет, принадлежит ли значение аргумента *c* диапазону допустимых значений кода *ASCII*. Если результат проверки отрицательный, значение функции — ноль.

Описание: int isascii (int c);

iscntrl ( ) — проверяет, является ли ее аргумент *c* элементом набора управляющих символов. Если результат проверки отрицательный, значение функции — ноль.

Описание: int iscntrl (int c);

isdigit ( ) — проверяет, является ли ее аргумент *c* одной из десятичных цифр 0–9. Если результат проверки отрицательный, значение функции — ноль.

Описание: int isdigit (int c);

isgraph ( ) — проверяет, является ли ее аргумент *c* каким-либо воспроизводимым символом, исключая пробел. Если результат проверки отрицательный, значение функции — ноль.

Описание: int isgraph (int c);

islower ( ) — проверяет, является ли ее аргумент *c* строчной буквой. Если результат проверки отрицательный, значение функции — ноль.

Описание: int islower (int c);

isprint ( ) — проверяет, является ли ее аргумент *c* каким-либо воспроизводимым символом, включая пробел. Если результат проверки отрицательный, значение функции — ноль.

Описание: int isprint (int c);

ispunct ( ) — проверяет, является ли ее аргумент *c* каким-либо воспроизводимым символом, кроме пробела, или символом, для которого применение функции *isalnum* дает истину. Если результат проверки отрицательный, значение функции — ноль.

Описание: int ispunct (int c);

isspace ( ) — проверяет, является ли ее аргумент *c* символом пробела. Если результат проверки отрицательный, значение функции — ноль.

Описание: int isspace (int c);

isupper ( ) — проверяет, является ли ее аргумент *c* прописной буквой. Если результат проверки отрицательный, значение функции — ноль.

Описание: int isupper (int c);

isxdigit ( ) — проверяет, является ли ее аргумент *c* шестнадцатеричной цифрой. Если результат проверки отрицательный, значение функции — ноль.

Описание: int isxdigit (int c);

tolower ( ) — возвращает строчный эквивалент своего аргумента *c* при условии, что аргумент — прописная буква. В противном случае аргумент возвращается, не претерпевая изменений.

Описание: int tolower (int c);

toupper ( ) — возвращает прописной эквивалент своего аргумента *c* при условии, что аргумент — строчная буква. В противном случае аргумент возвращается, не претерпевая изменений.

Описание: int toupper (int c);

Рассмотрим примеры на основные функции данных библиотек:

1. Необходимо написать программу, которая переписывает текст из одной строки в другую, а далее выводит вторую строку на экран. Текст программы:

```
#include<stdio.h>
#include <string.h>
int main(void)
{
    char string[10];
    char *str1 = "abcdefghi";
    stpcpy(string, str1);
    printf("%s\n", string);
    return 0;
}
```

2. Необходимо написать программу, которая выводит на экран строку символов, при этом размер строки изначально неизвестен. Текст программы:

```
#include<stdio.h>
#include <string.h>
int main(void)
{
    char *string = "Здравствуй, мир";
    printf("%d\n", strlen(string));
    return 0;
}
```

3. Необходимо написать программу, которая выводит на экран номер полученной ошибки. Текст программы:

```
#include<stdio.h>
#include <errno.h>
int main(void)
{
    char *buffer;
    buffer = strerror(errno);
    printf("Error: %s\n", buffer);
    return 0;
}
```

§ 4.5. Библиотека функций общего назначения

Данная библиотека содержит определения типов, макросов и различных функций общего назначения. Функции, макросы и типы описаны в заголовочном файле *stdlib.h*. Состав библиотеки:

abort ( ) — вызывает анормальное завершение работы программы. В зависимости от конкретной реализации (системы программирования) выполняются очистка буферов выходных потоков, закрытие открытых потоков, удаление временных файлов.

Описание: void abort (void);

abs ( ) — вычисляет абсолютное значение аргумента *x*.

Описание: int abs (int x);

atexit ( ) — позволяет зарегистрировать требуемую функцию таким образом, что она будет вызываться автоматически средой трансляции Си при нормальном завершении программы. Регистрируемая функция не должна иметь аргументов и не должна возвращать какое-либо значение. Одну и ту же функцию можно регистрировать более одного раза. При успешной регистрации функция возвращает ноль.

Описание: int atexit (void (*func)(void));

atof ( ) — преобразует ведущую часть строки символов, на которую указывает *nptr*, в значение *double*.

Описание: double atof (const char *nptr);

atoi ( ) — преобразует ведущую часть строки символов, на которую указывает *nptr*, в значение *int*.

Описание: int atoi (const char *nptr);

bsearch ( ) — выполняет поиск в массиве объектов *nmemb*, первый элемент которого адресуется указателем *base*. Осуществляется поиск объекта, эквивалентного объекту, на который указывает *key*. Если поиск не обнаружил соответствующего объекта, возвращаемое значение — *NULL*, иначе возвращается указатель на искомый объект.

Описание: void *bsearch (const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

где *size* — размер каждого объекта массива; *compar*( ) — функция, получающая два параметра, первый из которых указывает на объект *key*, второй — на элемент массива.

calloc ( ) — динамически выделяет область памяти для объектов *nmemb*, размер каждого из которых равен *size* (в байтах). Выдаваемое функцией значение — адрес первого байта выделения памяти.

Описание: void *calloc (size_t nmemb, size_t size);

div ( ) — вычисляет частное и остаток от деления *numer* на *denom*.

Описание: div_t div (int numer, int denom);

exit ( ) — вызывает нормальное завершение программы.

Описание: void exit (int status);

где параметр-статус принимает значения:

EXIT_SUCCESS — в случае удачного завершения;
EXIT_FAILURE — при возникновении ошибки.

free ( ) — освобождает область памяти, ранее выделенную функциями *calloc* ( ), *malloc* ( ), *realloc* (), на которую указывает *ptr*.

Описание: void free (void *ptr);

getenv ( ) — осуществляет поиск в списке переменных строки символов, которая совпадает со строкой, на которую указывает *name*. Функция возвращает указатель на строку символов.

Описание: char *getenv (const char *name);

labs ( ) — вычисляет абсолютное значение своего аргумента *x*, тип которого — *long*.

Описание: long labs (long x);

ldiv ( ) —вычисляет частное и остаток от деления *numer* на *denom* типа *long*.

Описание: ldiv_t ldiv (long numer, long denom);

malloc ( ) — динамически выделяет непрерывную область памяти размером *size* байт. Если память не выделена, то возвращаемое значение — *NULL*.

Описание: void *malloc (size_t size);

mblen ( ) — подсчитывает количество байт в многобайтовом символе, на который указывает *s*.

Описание: int mblen (const char *s, size_t n);

mbstowcs ( ) — преобразует последовательность многобайтовых символов в строку соответствующих символов расширенной формы.

Описание: int mbstowcs (wchar_t *pwcs, const char *s, size_t n);

mbtowc ( ) — подсчитывает количество байт в многобайтовом символе, на который указывает *s*. Далее она определяет код для значения данного типа *wchar_t*, которое соответствует указанному многобайтному символу.

Описание: int mbtowc (wchar_t *pwc, const char *s, size_t n);

qsort ( ) — выполняет сортировку массива объектов *nmemb*, на начальный объект которого указывает *base*. Параметр *size* задает размер каждого элемента в массиве.

Описание: void qsort (void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *));

rand ( ) — генерирует псевдослучайное целое число. Повторный вызов функции генерирует последовательность псевдослучайных целых чисел в диапазоне значений от 0 до RAND_MAX.

Описание: int rand (void);

realloc ( ) — функция меняет размер динамически выделяемой области памяти, адресуемой указателем *ptr*, на *size* (новый размер). Функция возвращает адрес новой области памяти.

Описание: void *realloc (void *ptr, size_t size);

srand ( ) — функция использует свой аргумент *seed* в качестве исходного данного, из которого возникает новая последовательность псевдослучайных чисел, возвращаемых при следующих друг за другом обращениях к *rand* ( ).

Описание: void srand (unsigned int seed);

strtod ( ) — преобразует ведущую часть строки, на которую указывает *nptr*, в данное типа *double*.

Описание: double strtod (const char *nptr, char **endptr);

strtol ( ) — преобразует ведущую часть строки, на которую указывает *nptr*, в данное типа *long*.

Описание: long strtol (const char *nptr, char **endptr, int base);

strtoul ( ) — преобразует ведущую часть строки, на которую указывает *nptr*, в данное типа *unsigned long*.

Описание: unsigned long strtoul (const char *nptr, char **endptr, int base);

system ( ) — передает строку символов, на которую указывает *string*, процессору командной строки среды, в которой функционирует программа.

Описание: int system (const char *string);

wcstombs ( ) — преобразует последовательность символов расширенной формы, адресуемую указателем *pwcs*, в последовательность многобайтовых символов, на которые указывает *s*.

Описание: int wcstombs (char *s, wchar_t wchar);

wctomb ( ) — определяет число байт, необходимых для представления многобайтового символа, соответствующего коду, значение которого — *wchar*. Данная функция также преобразует символ расширенной формы *wchar* в многобайтовый символ, который хранится по адресу *s*.

Описание: int wctomb (char *s, wchar_t wchar);

EXIT_SUCCESS — макрос, применяемый в качестве значения кода выхода из программы, символизирующего успех;

EXIT_FAILURE — макрос, применяемый в качестве значения кода выхода из программы, символизирующего неудачу;

MB_CUR_MAX — макрос, расширяемый до положительного целочисленного выражения, значение которого — максимальное число байт в многобайтовом символе;

RAND_MAX — макрос, являющийся целочисленным константным выражением, представляющим максимальное возможное значение, возвращаемое функцией *rand*();

div_t — тип, являющийся структурой, используемой как возвращаемое значение функции *div*;

ldiv_t — тип, являющийся структурой, используемой как возвращаемое значение функции *ldiv*( );

size_t — тип результата, порождаемого операцией вычисления размера объекта (*sizeof*);

wchar_t — тип целочисленных данных, диапазон значений которых обеспечивает представление отличающихся друг от друга кодов всех членов наибольшего расширенного набора символов.

Рассмотрим примеры на основные функции данной библиотеки:

1. Необходимо написать программу, которая выводит минимальный элемент введенного с клавиатуры массива целых чисел. Обязательное требование — динамическое выделение памяти под массив. Текст программы:

```
#include<stdio.h>
#include <conio.h>
#include <stdlib.h>
#define N 5
void main( )
{
    int *a;                                  //Указатель на массив
    int min;                                 //Количество ненулевых элементов
    int i;
                                             //Выделение памяти под элементы массива
    a=(int *)malloc(N *sizeof(int));
    printf ("\nПоиск минимального элемента массива.\n");
    printf("Введите в одной строке элементы массива,\n");
    printf("%i целых чисел, и нажмите <Enter>\n", N);
    printf("–>");
    for (i=0; i<5; i++)
        scanf("%d", &a[i]);
    min=0;                                   //Предположим, что первый элемент минимальный
                                             //Сравним оставшиеся элементы с минимальным
    for (i=1;i<N;i++)
        if(a[i]<a[min])
            min=i;
    printf("Минимальный элемент массива:");
    printf("a[%d]=%d ", min+1, a[min]);
    printf("\nДля завершения нажмите любую клавишу");
                                             //Освобождение выделенной памяти
    free(a);
    getch( );
}
```

2. Необходимо написать программу, которая выделяет память под массив символов «Привет» и выводит его на экран. Текст программы:

```
#include<stdio.h>
#include <string.h>
#include <stdlib.h>
int main(void)
{
    char *str;                                   //Указатель на массив символов
/* Выделение памяти под элементы массива*/
    if ((str = (char *) malloc(10)) == NULL)
    {
        printf("Не могу выделить память\n");
```

```
        exit(1);  /* Аварийный выход из программы */
    }
/* Запись "Привет" в массив символов */
    strcpy(str, "Привет");                          //Перезапись строки в массив
/* Вывод записи массива на дисплей */
    printf("Массив строк состоит из слова %s\n", str);
/* освобождение памяти */
    free(str);
    return 0;
}
```

3. Необходимо написать программу проверки умения складывать и вычитать числа в пределах 100. Программа должна вывести 10 примеров, причем в каждом примере уменьшаемое должно быть больше или равно вычитаемому, т. е. не допускается предлагать испытуемому примеры с отрицательным результатом. Оценка выставляется по следующему правилу: 10 правильных ответов —«5», 8–9 — «4», 7–6 —«3», менее 6 — «2». Текст программы:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
#define LEVEL 97+2                      //Действия над числами от 2 до 99
void main( )
{
    int numb1, numb2;                   //Слагаемые
    int op;                             //Действия над числами: 0 — сложение, 1 — вычитание
    char zop;                           //Знак между слагаемыми
    int res;                            //Результат, рассчитанный программой
    int otv;                            //Результат, введенный пользователем
    int kol=0;                          //Количество правильных ответов
    int buf;
    int i;
    time_t t; /*Текущее время для инициализации генератора случайных чисел*/
    printf("\***Проверка умения складывать и вычитать числа***\n");
    printf("После примера введите ответ и нажмите <Enter>\n");
                                        // Инициализация генератора случайных чисел
    srand((unsigned)time(&t));
                                        // Цикл по расчету примеров
    for(i=1; i<=10; i++)
    {
        numb1=rand ( ) %LEVEL;          //Число от 2 до 99
        numb2=rand ( ) %LEVEL;
        op=rand ( ) %2;                 //Выбор знака (+ или–)
        if(op==0)
        {
            res=numb1+numb2;
            zop='+';
        }
        else
        {
            zop='–';
                                        // Выбор наибольшего из двух чисел
            if (numb1<numb2)
            {
            buf=numb2;
            numb2=numb1;
            numb1=buf;
            }
            res=numb1 – numb2;
        }
        printf("%i%c%i=", numb1, zop, numb2);
        scanf("%i", &otv);
        if (otv==res)
```

```
            kol++;
        else printf ("Вы ошиблись! %i%c%i=%i\n",
            numb1, zop, numb2, res);
    }
    printf ("\nПравильных ответов: %i\n", kol);
    printf ("Ваша оценка:");                    // Выставление оценки
    switch (kol)
    {
        case 10: puts("5"); break;
        case 9: puts("4"); break;
        case 8: puts("4"); break;
        case 7: puts("3"); break;
        default: puts("2"); break;
    }
    printf ("\nДля завершения нажмите любую клавишу");
    getch( );
}
```

§ 4.6. Библиотеки системных функций

4.6.1. Библиотека функций, позволяющих программе выполнять прерывания *BIOS* и использовать операции *BIOS* в прикладной программе

Функции данной библиотеки описаны в заголовочном файле *bios.h.*

Основные функции библиотеки:

bioscom ( ) — выполняет ввод / вывод из последовательного порта.

Описание: int bioscom (int cmd, char abyte, int port);

где *cmd* — устанавливает параметры связи; *abyte* — комбинация битов, устанавливающая режим передачи данных;*port* — номер порта, при этом для *com*1 значение *port* равно 0 и т. д.

biosdisk ( ) — использует прерывание 0x13 для того, чтобы передать выполнение дисковых операций непосредственно *BIOS*.

Описание: int biosdisk (int cmd, int drive, int head, int track, int sector, void *buffer);

где *cmd* — вид выполняемой операции; *drive* — число, которое задает используемый дисковод; остальные параметры характеризуют диск, с которым производятся операции.

biosequip ( ) — проверяет оборудование, подключенное к системе.

Описание: int biosequip();

bioskey ( ) — посредством прерывания 0x16 выполняет различные операции с клавиатурой.

Описание: int bioskey (int cmd);

где *cmd* — определяет операцию.

biosprint ( ) — ввод / вывод на принтер при помощи *BIOS*.

Описание: int biosprint (int cmd, int abyte, int port);

где *cmd* — параметры работы принтера; *abyte* — символ для ввода / вывода;*port* — номер порта (для *lpt*1 равно 0).

biostime ( ) — считывание или установка таймера *BIOS*.

Описание: long biostime (int cmd, long newtime);

где *cmd* — устанавливает режим работы таймера (считывание — установка); *newtime* — новое значение таймера.

4.6.2. Библиотека функций, позволяющих программе выполнять прерывания *DOS* и использовать операции *DOS* в прикладной программе

Описание функций содержится в заголовочном файле *dos.h*.
Основные функции библиотеки:

getdisk ( ) — получает номер текущего устройства.

Описание: int getdisk (void);

mktemp ( ) — создает уникальное имя файла.

Описание: char *mktemp (char *template);

где *template* — уникальное имя файла.

mkdir ( ) — создает директорию.

Описание: int mkdir (const char *path);

где *path*— содержит маршрут до директории.

rmdir ( ) — удаляет директорию.

Описание: int rmdir (const char *path);

где *path* — содержит маршрут до директории.

setdisk ( ) — устанавливает спецификацию текущего дисковода.

Описание: int setdisk (int drive);

bdos ( ) — доступ к системным вызовам *DOS*.

Описание: int bdos (int dosfun, unsigned dosdx, unsigned dosal);

country ( ) — возвращает информацию, зависимую от конкретной страны.

Описание: struct country *country (int xcode, struct country *cp);

delay ( ) — задерживает выполнение программы на интервал в миллисекундах.

Описание: void delay (unsigned milliseconds);

disable ( ) — запрещает прерывания.

Описание: void disable (void);

enable ( ) — разрешает прерывания.

Описание: void enable (void);

getdate ( ) — получает системную дату.

Описание: void getdate (struct date *datep);

gettime ( ) — получает системное время.

Описание: void gettime (struct time *timep);

setdate ( ) — устанавливает системную дату.

Описание: void setdate (struct date *datep);

settime ( ) — устанавливает системное время.

Описание: void settime (struct time *timep);

geninterrupt ( ) — генерирует программное прерывание.

Описание: void geninterrupt (int intr_num);

hardresum ( ) — функция обработки ошибок электронного оборудования.

Описание: void hardresume (int rescode);

inport ( ) — считывает слово из порта.

Описание: int inport (int portid);

inportb ( ) — считывает байт из порта.

Описание: unsigned char inportb (int portid);

outport ( ) — вывод слова в порт.

Описание: void outport (int portid, int value);

outportb ( ) — вывод байта в порт.

Описание: void outport (int portid, unsigned char value);

4.6.3. Библиотека функций времени

Функции и макросы библиотеки описаны в заголовочном файле *time.h*.

Основные функции библиотеки:

asctime ( ) — преобразует время, содержащееся в виде структуры в **tblock*, в код *ASCII*. Функция возвращает указатель на символьную строку, содержащую дату и время.

Описание: char *asctime (const struct tm *tblock);

clock ( ) — определяет время процессора. Может быть использована для определения интервала времени между двумя событиями. Для определения времени в секундах значение, возвращаемое функцией, должно быть поделено на значение макроса *CLK_TCK* (новое название макроса *CLOCKS_PER_SEC*), задает число интервалов *clock_t* в секунду. Тип данных *clock_t* является арифметическим типом, который зависит от конкретной реализации системы программирования. Функция *clock* возвращает время процессора с момента начала выполнения программы.

Описание: clock_t clock (void);

ctime ( ) — преобразует дату и время в строку. Функция возвращает указатель на символьную строку, содержащую дату и время.

Описание: char *ctime (const time_t *time);

difftime ( ) — вычисляет разницу в секундах между *time*1 и *time*2.

Описание: double difftime (time_t time1, time_t time2);

gmtime ( ) — преобразует дату и время по стандарту Гринвича.

Описание: struct tm *gmtime (long *timer);

localtime ( ) — преобразует дату и время в структуру.

Описание: struct tm *localtime (const time_t *clock);

mktime ( ) — преобразует время из структуры, на которую указывает параметр *t*, в календарное время в формат, используемый функцией *time*.

Описание: time_t mktime (struct tm *t);

stime ( ) — устанавливает системные дату и время.

Описание: int stime (time_t *tp);

time ( ) — получает текущее время. Функция выдает текущее время (в секундах), прошедшее со времени, установленного в *BIOS* (например, 00:00:00 *GMT*, 1 января, 1970, или 00:00:00 *GMT*, 1 января, 2000), и сохраняет это значение по адресу, на который указывает *timer*.

Описание: time_t time (time_t *timer);

getdate ( ) — получает системную дату. Функция заполняет структуру *date* (с указателем *datep*) системной информацией о текущей дате.

Описание: void getdate (struct date *datep);

gettime ( ) — заполняет структуру *time*, на которую указывает параметр *timer*, текущим системным временем.

Описание: void gettime (struct time *timer);

setdate ( ) — устанавливает системную дату (месяц, день, год) в структуре *date*, на которую указывает *datep*.

Описание: void setdate (struct date *datep);

settime ( ) — устанавливает системное время в соответствии с содержимым структуры *time*, на которую указывает параметр *timer*.

Описание: void settime (struct time *timer);

4.6.4. Библиотека функций поддержки интернациональной среды

Функции и макросы данной библиотеки описаны в заголовочном файле *locale.h*.

В ее состав входят:

localeconv ( ) — вызывает инициализацию структуры типа *lconv* значениями, соответствующими текущей «культурной среде» (*locale*).

Описание: struct lconv *localeconv (void);

setlocale ( ) — дает программе возможность изменить *locale* полностью или только ее подкатегорию.

Описание: char *setlocale (int category, char *locale);

LC_ALL — макрос, используемый в качестве первого параметра функции *setlocale* ( ). Применение данного макроса влечет за собой назначение всех других *LC_** категорий для данной *locale*;

LC_COLLATE — макрос, применяемый для сравнения и объединения данных в функциях *strcoll* и *strxfrm*;

LC_CTYPE — макрос, используемый для выбора параметров в функциях, описанных в *ctype.h* и *stdlib.h*;

LC_MONETARY — макрос, применяемый для форматирования монетарных данных в функции *localeconv*;

LC_NUMERIC — макрос, применяемый для выбора символа десятичной точки в функциях ввода / вывода и для преобразования строк символов;

LC_TIME — макрос, применяемый для форматирования параметров функции *strtime* ( );

struct lconv — структура числового формата.

4.6.5. Библиотека макросов ошибок

Данная библиотека содержит объявление макросов для идентификации ошибок через их код. Описание средств библиотеки произведено в заголовочном файле *errno.h*. Основные макросы библиотеки:

errno — выражение, указывающее область памяти, через которую стандартная библиотека может пересылать значения ошибок. В процессе старта системы значение макроса *errno* обязательно должно быть принудительно обнулено.

EDOM — макрос, применяемый для указания ошибки выхода значения за пределы области определения.

ERANGE — макрос, используемый для указания диапазона значений ошибок.

4.6.6. Библиотека средств диагностики программ

Описание средств данной библиотеки произведено в заголовочном файле *assert.h*.

В состав библиотеки входит:

ASSERT — макрос, направляющий диагностическое сообщение в поток *stderr*, а далее инициирующий вызов *abort* при условии, что значением аргумента является «ложь».

4.6.7. Библиотека функций для сохранения и восстановления контекста программы

Библиотека содержит типы и функции, необходимые для сохранения и восстановления контекста программы, которые описаны в файле *setjmp.h*.

В состав библиотеки входят:

setjmp ( ) — записывает текущий контекст программы, определяемый в *jmp_buf* так, что программа может быть восстановлена в виде этого контекста посредством последующего обращения к функции *longjmp* ( ).

Описание: int setjmp (jmp_buf env);

longjmp ( ) — восстанавливает контекст программы.

Описание: void longjmp (jmp_buf env, int val);

jmp_buf — тип буфера для записи контекста.

§ 4.7. Библиотека графических функций

Данная библиотека содержит описание графических функций, которые описаны в заголовочном файле *graphic.h*. Основные функции библиотеки:

_graphfreemem ( ) — функция графической библиотеки для освобождения графической памяти, ранее выделенной функцией *_graphgetmem* ( ).

Описание: void far _graphfreemem (void far *ptr, unsigned size);

_graphgetmem ( ) — функция графической библиотеки для захвата графической памяти.

Описание: void far _graphgetmem (unsigned size);

arc ( ) — рисует текущим цветом дугу окружности с центром в точке с координатами (*x*, *y*) и радиусом *radius*. Дуга рисуется от угла *stangle* до угла *endangle*.

Описание: void far arc (int x, int y, int stangle, int endangle, int radius);

bar ( ) — рисует двухмерный заполненный цветом прямоугольник. Левый верхний и правый нижний углы прямоугольника заданы соответственно параметрами (*left*, *top*) и (*right*, *bottom*).

Описание: void far bar (int left, int top, int right, int bottom);

bar3d ( ) — рисует трехмерный, заполненный цветом прямоугольник. Глубина столбца в точках экрана задается параметром *depth*, вершина столбца задается параметром *topflag*.

Описание: void far bar3d (int left, int top, int right, int bottom, int depth, int topflag);

circle ( ) — рисует окружность текущим цветом. Центр окружности задается координатами *x* и *y*, радиус — параметром *radius*.

Описание: void far circle (int x, int y, int radius);

closegraph ( ) — прекращает работу графической системы.

Описание: void far closegraph (void);

detectgraph ( ) — определяет оптимальный графический драйвер и графический режим при проверке аппаратного обеспечения.

Описание: void far detectgraph (int far *graphdriver, int far *graphmode);

где **graphdriver* — определяет используемый графический драйвер; **graphmod*e — определяет исходный графический режим.

drawpoly ( ) — рисует текущими цветом и шириной линии контур многоугольника, состоящий из *numpoints* вершин. Координаты каждой вершины многоугольника задаются парами параметров *x* и *y*, являющихся элементами массива, на который указывает **polypoints*.

Описание: void far drawpoly (int numpoints, int far *polypoints);

ellipse ( ) — рисует эллиптическую дугу.

Описание: void far ellipse (int x, int y, int stangle, int endangle, int xradius, int yradius);

fillellipse ( ) — рисует и закрашивает эллипс.

Описание: void far fillellipse (int x, int y, int xradius, int yradius);

fillpoly ( ) — рисует и закрашивает многоугольник.

Описание: void far fillpoly (int numpoints, int far *polypoints);

floodfill ( ) — заполняет текущими цветом закраски и шаблоном ограниченную область, внутри которой находится точка с заданными координатами *x* и *y*. В параметре *border* указывается цвет границы ограниченной закрашиваемой области.

Описание: void far floodfill (int x, int y, int border);

getaspectratio ( ) — возвращает текущее отношение ширины изображения к его высоте.

Описание: void far getaspectratio(int far *xasp, int far *yasp);

getbkcolor ( ) — возвращает текущий цвет фона.

Описание: int far getbcolor (void);

getcolor ( ) — возвращает текущий цвет рисования.

Описание: int far getcolor (void);

getdrivername ( ) — возвращает указатель на строку, содержащую имя текущего графического драйвера.

Описание: char *far getdrivername(void);

getmaxx ( ) — возвращает максимальную координату *x* экрана.

Описание: int far getmaxx (void);

getmaxy ( ) — возвращает максимальную координату *y* экрана.

Описание: int far getmaxy (void);

getpixel ( ) — возвращает цвет заданного пикселя.

Описание: unsigned far getpixel (int x, int y);

getx ( ) — возвращает координату *x* текущей графической позиции.

Описание: int far getx (void);

gety ( ) — возвращает координату *y* текущей графической позиции.

Описание: int far gety (void);

grapherrormsg ( ) — возвращает указатель на строку сообщения об ошибке.

Описание: char *fargrapherrormsg(interrorcode);

graphresult ( ) — возвращает код ошибки последней выполненной графической операции (*grOk* — нет ошибок).

Описание: int far graphresult(void);

initgraph ( ) — инициализирует графическую систему путем загрузки графического драйвера с диска.

Описание: void far initgraph (int far *graphdriver, int far *graphmode, char far *pathtodriver);

где **graphdriver* — определяет используемый графический драйвер; **graphmode* — определяет исходный графический режим; **pathtodriver* — определяет маршрут поиска графического драйвера.

line ( ) — рисует линию между двумя указанными точками.

Описание: void far line (int x1, int y1, int x2, int y2);

linerel ( ) — рисует линию на заданное расстояние от текущей позиции.

Описание: void far linerel (int dx, int dy);

lineto ( ) — рисует линию от текущей позиции в точку с координатами (*x*, *y*).

Описание: void far lineto (int x, int y);

moverel ( ) — перемещает текущую позицию на заданное расстояние.

Описание: void far moverel (int dx, int dy);

moveto ( ) — перемещает текущую позицию в точку с координатами (*x*, *y*).

Описание: void far moveto (int x, int y);

outtext ( ) — отображает строку, на которую указывает **textstring* в окне экрана.

Описание: void far outtext (char far *textstring);

outtextxy ( ) — отображает строку, на которую указывает **textstring* в заданной позиции экрана.

Описание: void far outtextxy (int x, int y, char far *textstring);

pieslice ( ) — рисует и закрашивает сектор круга.

Описание: void far pieslice (int x, int y, int stangle, int endangle, int radius);

putpixel ( ) — выводит пиксель в заданную точку экрана.

Описание: void far putpixel (int x, int y, int color);

rectangle ( ) — рисует прямоугольник линией текущего вида, толщины и цвета.

Описание: void far rectangle (int left, int top, int right, int bottom);

sector ( ) — рисует и закрашивает эллиптические секторы.

Описание: void far sector (int x, int y, int stangle, int endangle, int xradius, int yradius);

setaspectrati ( ) — изменяет текущее отношение ширины изображения к его высоте.

Описание: void far setaspectratio(int xasp, int yasp);

setbkcolor ( ) — устанавливает текущий цвет фона.

Описание: void far setbcolor (int color);

где параметр *color*, обозначающий номер выбранного цвета, для *VGA* драйвера принимает значения, представленные в табл. 4.9.

Таблица 4.9.

Цвет	Значение	Имя
Черный	0	BLACK
Голубой	1	BLUE
Зеленый	2	GREEN
Синий	3	CYAN
Красный	4	RED
Фиолетовый	5	MAGENTA
Коричневый	6	BROWN
Светло-серый	7	LIGHTGRAY
Темно-серый	8	DARKGRAY
Светло-голубой	9	LIGHTBLUE
Светло-зеленый	10	LIGHTGREEN
Светло-синий	11	LIGHTCYAN
Светло-красный	12	LIGHTRED
Светло-фиолетовый	13	LIGHTMAGENTA
Желтый	14	YELLOW
Белый	15	WHITE

setcolor ( ) — устанавливает текущий цвет рисования.

Описание: void far setcolor (int color);

setfillstyle ( ) — устанавливает шаблон и цвет заполнения. Параметр *pattern*определяет шаблон заполнения (при сплошном заполнении — 1 либо *SOLID_FILL*), *color* — цвет заполнения.

Описание: void far setfillstyle(int pattern, int color);

setgraphbufsize ( ) — изменяет размер внутреннего графического буфера.

Описание: unsigned far setgraphbufsize (unsigned bufsize);

setgraphmode ( ) — переводит систему в графический режим, очищает экран.

Описание: void far setgraphmode (int mode);

setlinestyle ( ) — устанавливает толщину и тип линии.

Описание: void far setlinestyle (int linestyle, unsigned upattern, int thickness);

где *linestyle* — параметр, который определяет тип линии:
SOLID_LINE (0) — сплошная;
DOTTED_LINE (1) — пунктирная;
CENTER_LINE (2) — штрихпунктирная;
DASHED_LINE (3) — штриховая;
USERBIT_LINE (4) — заданная пользователем;
upattern — шаблон, используемый при выборе пользовательского типа линии (4);
thickness — определяет, будут ли последующие линии толстыми или тонкими:
NORM_WIDTH (1) — толщина в 1 пиксель;
THICK_WIDTH (3) — толщина в 3 пикселя.

settextstyle ( ) — устанавливает текущие характеристики текста для графического вывода.

Описание: void far settextstyle (int font, int direction, int charsize);

где *font* — параметр, определяющий тип шрифта:

DEFAULT_FONT (0) — матрица 8 × 8 бит;
TRIPLEX_FONT (1) — утроенный шрифт;
SMALL_FONT (2) — малый шрифт;
SANSSERIF_FONT (3) — шрифт *Sans Serif*;
GOTIC_FONT (4) — готический шрифт;

direction — параметр, определяющий расположение текста:

HORIZ_DIR (0) — горизонтальное;
VERT_DIR (1) — вертикальное;

charsize — параметр, определяющий размер шрифта (1 — 8 × 8; 2 — 16 × 16).

textheight ( ) — возвращает высоту строки в пикселях.

Описание: int far textheight (char far *textstring);

textwidht ( )— возвращает ширину строки в пикселях.

Описание: int far textwidht (char far *textstring);

Рассмотрим пример на использование основных функций графической библиотеки.

Необходимо написать программу, которая переводит систему в графический режим и рисует в центре экрана квадрат с размером сторон 100 пикселей. Текст программы:

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
    int gdriver = DETECT, gmode, errorcode;
    int left, top, right, bottom;
/* Инициализация графического режима */
    initgraph(&gdriver, &gmode, "");
/* Запись результата инициализации */
    errorcode = graphresult( );
    if (errorcode != grOk)                          // Возникновение ошибки
    {
        printf("Графическая ошибка: %s\n", grapherrormsg(errorcode));
        printf("Нажмите любую клавишу для выхода:");
        getch( );
        exit(1);                                    // Аварийный выход из программы
    }
/* Вычисление координат прямоугольника */
    left = getmaxx( ) / 2 – 50;
    top = getmaxy( ) / 2 – 50;
    right = getmaxx( ) / 2 + 50;
    bottom = getmaxy( ) / 2 + 50;
/* Рисование прямоугольника на экране */
    rectangle(left, top, right, bottom);
    getch( );
/* Очистка графического режима */
    closegraph( );
    return 0;
}
```

Глава 5
Программирование в среде *Borland C/C++* 3.1

§ 5.1. Элементы управления в *Borland C/C++* 3.1

Среда программирования *Borland C/C++ ver*. 3.1, разработанная фирмой *Borland International* в 1992 г., являлась одной из наиболее популярных инструментальных систем программирования языка Си в операционной системе *DOS*. Данная инструментальная система включает в себя языки программирования Си и C++ и позволяет создавать программные продукты под операционную систему *DOS* и *Windows ver*. 3.1. Эта среда до сих пор считается одной из лучших сред программирования на языках *C/C++* и используется в ограниченных по возможностям аппаратных системах. Поэтому изучение сред *C/C++* целесообразно начать именно с *Borland C/C++ ver*. 3.1.

При запуске программы раскрывается графическая оболочка, включающая в себя заголовочное меню, окно редактора текста программы и окна отладки. В заголовочном меню находятся следующие подзаголовки:

Ё — позволяет производить переход к другим программам, встроенным в *Borland C/C++ ver*. 3.1 (ассемблер, отладчик, профилировщик, компилятор ресурсов и т. д.).

File — включает команды управления файлами:
- *New* —создать новый файл в новом окне редактирования;
- *Open* — расположить и открыть файл;
- *Save* — сохранить файл в активном окне редактирования;
- *Save as* — сохранить файл в активном окне под новым названием;
- *Save all* — сохранить все изменяемые файлы;
- *Change dir* — изменить текущий рабочий каталог;
- *Print* — печатать содержание активного окна;
- *Dos shell* — временно выйти в *DOS*;
- *Quit* — выход из *Borland C++*.

Edit — включает команды редактирования операций и доступа к буферу обмена:
- *Undo* — отменить предыдущее действие редактора;
- *Redo* — восстановить ранее уничтоженное действие редактора;
- *Cut* — удалить выбранный текст и поместить его в буфер обмена;
- *Copy* — копировать выбранный текст и поместить его в буфер обмена;
- *Paste* — вставить выбранный текст из буфера обмена в позицию курсора;
- *Clear* — удалить выбранный текст;
- *Copy example* — копировать пример программы из справочного окна в буфер обмена;
- *Show clipboard* — открыть окно буфера обмена.

Search — включает команды обработки текста и поиска ошибок:
- *Find* — поиск текста;
- *Replace* — поиск текста и замена его новым текстом;
- *Search again* — повторение последнего поиска или замена операции;
- *Go to line number* — перемещение курсора в строку с определенным номером;
- *Previous error* — перемещение курсора в предыдущую позицию ошибки;
- *Next error*—переместить курсор в последующую позицию ошибки;
- *Locate function* —поиск декларированной функции при отладке.

Run — включает команды компилирования и пошагового выполнения программы:
- *Run* — запустить текущую программу;
- *Program reset* — перезапустить программу с начала;
- *Go to cursor* — выполнить программу до позиции курсора;
- *Trace intro* — остановиться в пределах текущей функции;

– *Step over* — остановиться до текущей функции;
– *Arguments* — показать командные строки, пропускаемые в программе.

Compile — включает программы компилирования и компоновки программы:
– *Compile*—компилирование файла в активном окне редактирования;
– *Make* —компилирование участков программы по мере необходимости;
– *Link* —компоновка без перетранслирования;
– *Build all*—перекомпилировать все файлы;
– *Information* —вывести информацию компилятора на дисплей;
– *Remove messages*—очистить все окна сообщений.

Debug — включает программы инспектирования, установки контрольных точек и окон просмотра:
– *Inspect* —открытие инспектирующего окна для исследования значений данных;
– *Evaluate / modify* —оценка значения переменной или выражения и отображение значений;
– *Call stack*—просмотр функции программы с контрольной точкой;
– *Watches* —добавление, удаление и редактирование окон просмотра:
  1) *Add watch*—вставить выражение для проверки в окно просмотра;
  2) *Delete watch*—удалить текущее выражение из окна просмотра;
  3) *Edit watch*—редактировать текущее выражение в окне просмотра;
  4) *Remove all watches*—удалить все выражения из окна просмотра;
– *Toggle breakpoint*— установка или очистка контрольной точки в позиции курсора;
– *Breakpoints* — установка контрольных точек; просмотр и редактирование контрольных точек.

Project — включает команды управления проектом:
– *Open project* — загрузить проектный файл и его окружение;
– *Close project* — закрыть текущий проект и загрузить проект, заданный по умолчанию;
– *Add item* — добавить элемент к проекту;
– *Delete item* — удалить элемент из проекта;
– *Local option* — показать опции элемента проекта;
– *Include files* — показать подключаемые файлы проекта.

Options — включает опции *IDE* (интегрированной среды разработки), компилятора, отладчика, установки свойств файлов программы:
– *Application* — свойства проекта;
– *Compiler* — установка значений для компилятора, генерации объектного кода, сообщений об ошибках и именах:
  1) *Code generation* — определение объектного кода компилятора (модели памяти и т. д.);
  2) *Advanced code generation* — установка опций генерации объектного кода;
  3) *Entry / Exit Code* — изменение кода, сгенерированного для функций;
  4) *C++options* — определение объектного кода компилятора *C++*;
  5) *Advanced C++options* — определение параметров настройки элементов генерации объектного кода *C++*;
  6) *Optimizations* — определение оптимизации компилятора;
  7) *Source* — определение опций исходного кода;
  8) *Messages* — определение обработки ошибок и предупреждений *IDE*:
     • *Display* — конфигурация ошибок и предупреждений;
     • *Portability* — выбор сообщений об ошибках для отображения;
     • *Ansi violations* — выбор сообщений нарушений *ANSI* для отображения;
     • *C++warnings* — выбор сообщений предупреждений *C++* для отображения;
     • *Frequent errors* — выбор часто встречаемых сообщений для отображения;
     • *Less frequent errors* — выбор менее общих сообщений для отображения;
  9) *Names* — замена заданного сегмента, группы и имени класса;
– *Transfer* — создание или замена программы перехода;
– *Make* — набор опций *make*;
– *Linker* — параметры настройки компоновщика и библиотек:
  1) *Settings* — опции компоновщика;

2) *Libraries* — опции подключаемых библиотек;

– *Librarian* — опции библиотек;
– *Debugger* — опции отладчика;
– *Directories* — пути для включения файлов, библиотек и двоичных выходных файлов;
– *Environment* — выбор среды, редактора, мыши и т. д.:
1) *Preferences* — определение параметров настройки среды;
2) *Editor* — определение параметров настройки редактора;
3) *Mouse* — определение параметров настройки мыши;
4) *Desktop* — определение параметров настройки рабочего стола;
5) *Startup* — определение параметров настройки запуска;
6) *Colors* — выбор цветовой палитры среды;
– *Save* — сохранение всех опций.

Window — команды открытия и упорядочения окон, списки окон:
– *Size / Move* — изменение размера или позиции активного окна;
– *Zoom* — увеличение или восстановление размера активного окна;
– *Cascade* — каскад окон на рабочем столе;
– *Tile* — окна на весь рабочий стол;
– *Next* — сделать следующее окно активным;
– *Close* — закрыть активное окно;
– *Close all* — закрыть все окна на рабочем столе, очистить списки хронологии;
– *Message* — открыть окно сообщений;
– *Output* — открыть окно для вывода;
– *Watch* — открыть окно для просмотра;
– *User screen* — включить полноэкранный режим;
– *Register* — открыть окно регистра;
– *Project* — открыть окно организатора проекта;
– *Project notes* — открыть окно организатора проекта для *notebook*;
– *List all* — показать список окон.

Help — доступ к интерактивной справке:
– *Contents* — показать контексты для интерактивной справки;
– *Index* — показать индексы для интерактивной справки;
– *Topic search* — показать справку для слова, выделенного курсором;
– *Previous topic* — восстановить изображение последней из рассматриваемых страниц интерактивной справки;
– *Help on help* — как использовать интерактивную справку;
– *Active file* — выбор справочного файла, который вы хотите использовать;
– *About* — показать версию и информацию об авторских правах.

§ 5.2. Создание и разработка проекта

Запуск *Borland C/C++* осуществляется запуском файла *bc.exe* в директории *BIN* среды разработки. При раскрытии оболочки программы в меню выбирается *Project–>Open*, далее открывается файл с названием проекта с расширением **.prj*, если проект уже был создан, либо создается новый проект. Среда обработки позволяет включать в проект любое количество файлов. Для этого в меню выбирается *Project–>Add item* и далее указывается имя файла с текстом программы. Для создания файла и его открытия для редактирования в меню открывается *File–>New*. Если необходимо открыть для редактирования уже существующий файл, то в меню выбирается *File–>Open*. Файлы, включенные в проект, можно открывать в окне редактора двойным нажатием левой клавиши мыши по названию файла в окне проекта.

После создания проекта необходимо установить опции проекта в соответствии с необходимыми потребностями разрабатываемой программы. Для этого нужно войти в меню в*Options* и произвести проверку опций проекта, установленных по умолчанию.

Пример настройки основных опций проекта:

• **Applications** — Dos Standard.

• **Compiler:**

– Code generation:
  1) Model — *тип модели памяти*, *обычно*Large*или*Huge;
  2) Assume SS Equals DS — Default for memory model;
– Advanced Code Generation:
  1) Floating Point — Emulation;
– Entry/Exit Code Generation:
  1) Prolog/Epilog Code Generation — Dos standard;
  2) Calling Convention — C;
– Optimization — Fastest Code;
– Source options:
  1) Keywords — *рекомендуется* Borland *C++*.

• **Make:**

– Break Make On — Errors;
– After Compiling — Run linker;
– Generate Import Library — Use DLL file exports.

• **Linker:**

– Settings:
  1) Map File — off;
  2) Output — Standard DOS EXE;
– Libraries:
  1) Libraries — Graphics library.

• **Directories:**

– Include Directories — *Диск*:*Директорий среды разработки*\INCLUDE;
– Library Directories — *Диск*:*Директорий среды разработки*\LIB.

После установки опций необходимо произвести их сохранение в *Options–>Save*.

В инструментальной системе *Borland C*/*C++* 3.1 широко используется система подсказок *Help*. Причем помимо непосредственного обращения в меню *Help* достаточно перевести курсор на функцию, тип данных или идентификатор, по которым необходимо получить дополнительную информацию, и нажать правую кнопку мыши. В результате высветится информация по интересующему вас вопросу, включая пример, который можно скопировать и перенести в текст программы.

После набора программы осуществляется ее компиляция. Управление компиляцией осуществляется в меню *Compile*. Если проект включает в себя большое количество файлов, рекомендуется выполнять операцию *Build all*. Если необходимо произвести трансляцию активированного файла проекта либо если проект состоит из одного файла, то выполнять операцию *Make*. При положительной трансляции проекта осуществляется запуск программы в меню *Run–>Run*.

Для ускорения процесса набора программы в *Borland C*/*C++* 3.1 необходимо широко применять функции редактирования, которые позволяют выделить участок текста программы, вырезать его, копировать в другое место программы, как в обычном текстовом редакторе. Данные функции расположены в меню *Edit*. Также имеется возможность автоматического поиска слова или участка программы, автозамена его на другой участок в единичном месте или по всему тексту программы. Данные функции располагаются вменю *Search*. С помощью меню *Window* производится доступ к текстам программы в файлах в окне редактора, а также доступ к служебным окнам (окно проекта, окно просмотра, окно сообщения и т. д.) в окне отладки. Осуществляется также просмотр всех открытых окон проекта и управление ими (открытие, закрытие, расположение и т. д.). Более подробно вопросы отладки программы и устранения ошибок рассмотрены далее.

§ 5.3. Отладка программ в *BorlandC/C++* 3.1

Для рассмотрения вопроса отладки программ в среде *Borland* C / *C++* 3.1 создадим приложение и заведомо внесем в него несколько ошибок, которые невозможно обнаружить на этапе компиляции. Далее с помощью средств отладчика *Turbo Debugger* выявим заложенные ошибки.

Создадим программу, в которой рассчитывается среднее арифметическое введенного с клавиатуры массива, состоящего из 5 элементов. Для этого создадим проект под названием *Bed*, в него включим файл *Arifm.c*, в который введем следующий код:

```
#include <stdio.h>
#include <conio.h>

#define SIZE 5                                  //Размер массива
void main( )
{
    int a[SIZE];                                //Массив
    int sum;                                    //Сумма всех элементов
    int Average;                                //Среднее арифметическое элементов
    int i;
    clrscr( );
    printf ("\nРасчет среднего арифметического массива.\n");
    printf("Введите в одной строке элементы массива,\n");
    printf("%i целых чисел, и нажмите<Enter>\n", SIZE);
    printf("–>");
    for (i=0; i<SIZE; i++)
        scanf ("%d", &a [i]);
                                                //Сумма элементов массива
    for (i=1; i<SIZE; i++)
        sum=sum + a [i];
    Average = sum / SIZE;
    printf("Среднее арифметическое массива =");
    printf("%d", Average);
    printf("\nДля завершения нажмите любую клавишу");
    getch( );
}
```

Откомпилируйте данное приложение и запустите его. В качестве элементов массива введите числа 10, 11, 12, 13, 14. В результате будет получен ответ «Среднее арифметическое массива = 298». Возможно и другое неопределенное значение, которое не соответствует действительности. Перейдем к отладке программы. Запустим программу в режиме пошагового выполнения, чтобы проверить, какое значение принимают переменные на каждом шаге выполнения программы. Однако выполнять весь код программы в режиме пошагового просмотра неэффективно, желательно начать ближе к месту, в котором вычисляется среднее арифметическое. Для того чтобы начать отладку программы с произвольной строки, необходимо установить ***точку прерывания***. Во время выполнения программа остановится на ней, а далее появится возможность просмотреть код программы и перейти на пошаговый просмотр. Для установки точки прерывания следует установить указатель на необходимой строке программы и далее выбрать в меню *Debug –> Toggle breakpoint*. В результате выбранная строка программы будет выделена красным цветом. Для снятия точки прерывания необходимо повторить эти действия. Далее введем в окно для просмотра имена переменных, значения которых необходимо контролировать. Для этого выберите *Debug –> Watches –> Add watch…* и введите необходимые имена переменных.

Установим в нашем приложении точку прерывания на цикл, в котором происходит суммирование элементов массива для вычисления среднего арифметического. Для инспектирования введем переменные *sum*, *Average*, *a* [*i*]. Запустим приложение в режиме выполнения до точки прерывания. Для этого необходимо выбрать меню *Run –> Run*. В программе есть возможность запуска программы до произвольного положения курсора *Run –> Go to cursor*. Данные на входе введите аналогично предыдущему примеру: 10, 11, 12, 13, 14. В результате программа выполнилась до точки прерыва-

ния. В окне инспектирования появились первые значения переменных: *sum* = 1440, *Average* = 1824, *a* [*i*] = 0. Возможны и другие значения. Сразу видна первая ошибка. Поскольку, вычисляя сумму элементов массива, мы складываем значения всех его элементов в переменной *sum*, то ее необходимо первоначально инициализировать нулем. Добавим этот код в наше приложение:

```
#include<stdio.h>
#include<conio.h>

#define SIZE 5                                  //Размер массива
void main( )
{
    int a[SIZE];                                //Массив
    int sum;                                    //Сумма всех элементов
    int Average;                                //Среднее арифметическое элементов
    int i;
    clrscr( );
    printf ("\nРасчет среднего арифметического массива.\n");
    printf("Введите в одной строке элементы массива,\n");
    printf("%i целых чисел, и нажмите<Enter>\n", SIZE);
    printf("–>");
    for (i=0; i<SIZE; i++)
        scanf("%d", &a[i]);
    sum=0;                                      //Инициализация нулем
                                                //Сумма элементов массива
    for (i=1; i<SIZE; i++)
        sum=sum + a[i];
    Average = sum / SIZE;
    printf("Среднее арифметическое массива = ");
    printf("%d", Average);
    printf("\nДля завершения нажмите любую клавишу");
    getch( );
}
```

Вновь запустим приложение до точки прерывания. Переменной *sum*присвоено значение 0. Будем выполнять программу в режиме пошагового просмотра. Для этого необходимо выбирать *Run –> Step Over* или *F8* для выполнения каждого шага. Обнаружена следующая ошибка: в расчетах не учитывается значение *a* [0] = 10. Это происходит потому, что в цикле по вычислению среднего арифметического инициализация переменной *i* происходит не от 0, а от 1. Исправим эту ошибку:

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5                                  //Размер массива
void main( )
{
    int a [SIZE];                               // Массив
    int sum;                                    //Сумма всех элементов
    int Average;                                //Среднее арифметическое элементов
    int i;
    clrscr( );
    printf ("\nРасчет среднего арифметического массива.\n");
    printf("Введите в одной строке элементы массива,\n");
    printf("%i целых чисел, и нажмите<Enter>\n", SIZE);
    printf("–>");
    for (i=0; i<SIZE; i++)
        scanf("%d", &a[i]);
    sum=0;                                      //Инициализация нулем
                                                //Сумма элементов массива
//  for (i = 1; i<SIZE; i++)                    // Ошибочное значение i
    for (i = 0; i < SIZE; i++)
        sum = sum + a [i];
    Average = sum / SIZE;
```

```
    printf("Среднее арифметическое массива =");
    printf("%d", Average);
    printf("\nДля завершения нажмите любую клавишу");
    getch( );
}
```

Запустим приложение вновь. Изменим данные на входе. Введем числа, следующие не по порядку, например 0, 12, 5, 7, 9. В результате получим, что сумма всех элементов равна 33, а среднее арифметическое — 6. Ошибка в данном случае произошла потому, что значение среднего арифметического округляется до целого значения. Для исправления присвоим переменной *Average* тип *float*, а не *int* (при этом не забудьте изменить параметр формата функции *printf*( )с "% *d*" на "% *f*" при выводе значения переменной на экран). Кроме того, при вычислении среднего арифметического переназначим переменной *SIZE* тип *float*, в противном случае значение на выходе все равно будет округлено до целого значения:

```
#include <stdio.h>
#include <conio.h>
#define SIZE 5                                  //Размер массива
void main( )
{
    int a[SIZE];                                //Массив
    int sum;                                    //Сумма всех элементов
    float Average;                              //Среднее арифметическое элементов
    int i;
    clrscr( );
    printf ("\nРассчет среднего арифметического массива.\n");
    printf("Введите в одной строке элементы массива,\n");
    printf("%i целых чисел, и нажмите<Enter>\n", SIZE);
    printf("–>");
    for (i=0; i<SIZE; i++)
        scanf("%d", &a[i]);
    sum=0;                                      //Инициализация нулем
                                                //Сумма элементов массива
    for (i = 0; i<SIZE; i++)
        sum=sum + a[i];
    Average = sum / (float)SIZE;
    printf ("Среднее арифметическое массива = ");
    printf ("%g", Average);
    printf ("\nДля завершения нажмите любую клавишу");
    getch( );
}
```

Запустим приложение со значениями переменных на входе, аналогичными предыдущему примеру: 0, 12, 5, 7, 9. В результате среднее арифметическое равно 6,6. Теперь все ошибки устранены, приложение работает без ошибок.

Глава 6
Программирование в среде *Visual C++*

§ 6.1. Основы программирования в среде *Windows*

Уровень сложности современного программного обеспечения настолько высок, что разработка приложений в среде *Windows* с использованием одного из современных языков программирования значительно затрудняется. Программист должен затратить массу времени на решение стандартных задач по созданию многооконного интерфейса. Реализация технологии связывания и встраивания объектов — *OLE* (*Object Linking and Embedding*) — потребует от программиста еще более сложной работы. Чтобы облегчить работу программиста, практически все современные компиляторы с языка *C++* содержат специальные библиотеки классов. Такие библиотеки включают в себя практически весь программный интерфейс *Windows* и позволяют пользоваться при программировании средствами более высокого уровня, чем обычные вызовы функций. За счет этого значительно упрощается разработка приложений, имеющих сложный интерфейс пользователя, облегчается поддержка технологии *OLE* и взаимодействие с базами данных.

Современные интегрированные средства разработки приложений *Windows* позволяют автоматизировать процесс создания приложения. Для этого используются генераторы приложений. Программист отвечает на вопросы генератора приложений и определяет свойства приложения: поддерживает ли оно многооконный режим, технологию *OLE*, трехмерные органы управления, справочную систему. Генератор приложений создаст приложение, отвечающее требованиям, и предоставит исходные тексты. Пользуясь им как шаблоном, программист сможет быстрее разрабатывать свои приложения.

Подобные средства автоматизированного создания приложений включены в компилятор*Microsoft Visual C++*, который объединяет в себе две законченные системы разработки *Windows*-приложений.

Первая из них — программирование на основе *MFC AppWizard* (*Microsoft Foundation Class Library Application Wizard*). Заполнив несколько диалоговых панелей, можно указать характеристики приложения и получить его тексты, снабженные обширными комментариями. *MFC AppWizard* позволяет создавать однооконные (однодокументные) и многооконные (многодокументные) приложения, а также приложения, не имеющие главного окна, — вместо него используется диалоговая панель. Можно также включить поддержку технологии *OLE*, баз данных, справочной системы. Конечно, *MFC AppWizard* не всесилен. Прикладную часть приложения программисту придется разрабатывать самостоятельно. Исходный текст приложения, созданный *MFC AppWizard*, станет только основой, к которой нужно подключить остальное. Но работающий шаблон приложения — это уже половина всей работы. Исходные тексты приложений, автоматически полученных от *MFC AppWizard*, могут составлять сотни строк текста. Набор его вручную был бы очень утомителен. Следует отметить, что *MFC AppWizard* создает тексты приложений только с использованием библиотеки классов *MFC* (*Microsoft Foundation Class Library*). Поэтому только изучив язык *C++* и библиотеку *MFC*, можно пользоваться средствами автоматизированной разработки и создавать свои приложения в кратчайшие сроки.

Вторая система основывается на библиотеке шаблонов *ActiveX* (*ActiveX Template Library — ATL*). *ATL* представляет собой средство построения элементов управления *ActiveX*. Следует отметить, что писать элементы управления *ActiveX* можно как на *MFC*, так и на *ATL*, но *ATL*-элементы намного меньше по объему кода и быстрее загружаются по Интернету.

Перед изучением технологии использования шаблонов для создания *Windows*-приложений необходимо рассмотреть принципы работы операционной системы *Windows* и ее взаимодействия с прикладными программами.

Благодаря ***интерфейсу вызова функций в Windows*** доступ к системным ресурсам осуществляется через целый ряд системных функций. Совокупность таких функций называется приклад-

ным программным интерфейсом, или *API* (*Application Programming Interface*). Для взаимодействия с *Windows*приложение запрашивает функции *API*, с помощью которых реализуются все необходимые системные действия, такие как выделение памяти, вывод на экран, создание окон и т. п.

Поскольку *API* состоит из большого числа функций, может сложиться впечатление, что при компиляции каждой программы, написанной для *Windows*, к ней подключается код довольно значительного объема. В действительности это не так. Функции *API* содержатся в ***библиотеках динамической загрузки*** (*Dynamic Link Libraries — DLL*), которые загружаются в память только в тот момент, когда к ним происходит обращение, т. е. при выполнении программы. Динамическая загрузка обеспечивает ряд существенных преимуществ.

Во-первых, поскольку практически все программы используют *API*-функции, то благодаря *DLL*-библиотекам существенно экономится дисковое пространство, которое в противном случае занималось бы большим количеством повторяющегося кода, содержащегося в каждом из исполняемых файлов.

Во-вторых, изменения и улучшения в *Windows*-приложениях сводятся к обновлению только содержимого *DLL*-библиотек. Уже существующие тексты программ не требуют перекомпиляции.

В настоящее время наибольшее распространение получила версия *API*, которая имеет название *Win32*. Данная версия *API* пришла на смену версии *Win16*, используемой в *Windows* 3.1. Фактически 32-разрядная *Win32*, используемая в операционных системах линейки 9x, является надмножеством для *Win16* (т. е. фактически включает в себя этот интерфейс), так как большинство функций имеет то же название и применяется аналогичным образом. Однако будучи в принципе похожими, оба интерфейса все же отличаются друг от друга. *Win32* поддерживает 32-разрядную линейную адресацию, тогда как *Win1*6 работает только с 16-разрядной сегментированной моделью памяти. Это привело к тому, что некоторые функции были модифицированы таким образом, чтобы принимать 32-разрядные аргументы и возвращать 32-разрядные значения. Часть из них пришлось изменить с учетом 32-разрядной архитектуры. Была реализована поддержка потоковой многозадачности, новых элементов интерфейса и прочих нововведений *Windows 9x*. Версия *API Win32* операционной системы линейки *NT* уже полностью 32-разрядная и существенно отличается от *Win32 Windows 9x*.

Так как *Win32* поддерживает полностью 32-разрядную адресацию, то логично, что целые типы данных (*integers*) также объявлены 32-разрядными. Это означает, что переменные типа *int* и *unsigned* будут иметь длину 32 бита, а не 16, как в *Windows* 3.1 или *DOS*. Если же необходимо использовать переменную или константу длиной 16 бит, они должны быть объявлены как *short* (далее будет показано, что для этих типов определены независимые *typedef*-имена). Следовательно, при переносе программного кода из 16-разрядной среды необходимо убедиться в правильности использования целочисленных элементов, которые автоматически будут расширены до 32 бит, что может привести к появлению побочных эффектов.

Другим следствием 32-разрядной адресации является то, что указатели больше не нужно объявлять как *near* и *far*. Любой указатель может получить доступ к любому участку памяти. В современных ОС *Windows* константы *near* и *far* объявлены (с помощью директивы # *define*) пустыми.

Одним из подмножеств *API* является *GDI* (*Graphics Device Interface* — интерфейс графического устройства)—та часть *Windows*, которая обеспечивает поддержку аппаратно-независимой графики. Благодаря функциям *GDI Windows*-приложение может выполняться на различных ПЭВМ.

Еще одной особенностью *Windows 9x* является многозадачность, причем поддерживаются два типа многозадачности: основанная на процессах и основанная на потоках. Рассмотрим их подробнее.

Процесс—это программа, которая выполняется. При многозадачности такого типа две или более программы могут выполняться параллельно. Конечно, они по очереди используют ресурсы центрального процессора и в техническом плане выполняются неодновременно, но благодаря высокой скорости работы компьютера это практически незаметно.

Поток—это отдельная часть исполняемого кода. Название произошло от понятия «направление протекания процесса». В многозадачности данного типа отдельные потоки внутри одного процесса также могут выполняться одновременно. Все процессы имеют по крайней мере один поток, но в самой *Windows* их может быть несколько.

Взаимодействие программ и *Windows.* Во многих операционных системах взаимодействие между системой и программой инициализирует программа. Например, в *DOS* программа запрашивает разрешение на ввод и вывод данных. Говоря другими словами, не *Windows*-программы сами вызывают операционную систему. Обратного процесса не происходит. В *Windows* все совершенно наоборот: именно система вызывает программу. Это осуществляется следующим образом: программа ожидает получение сообщения от *Windows*. Когда это происходит, то выполняется некоторое действие. После его завершения программа ожидает следующего сообщения.

Windows может посылать программе сообщения множества различных типов. Например, каждый раз при щелчке мышью в окне активной программы посылается соответствующее сообщение. Другой тип сообщений посылается, когда необходимо обновить содержимое активного окна. Сообщения посылаются также при нажатии клавиши, если программа ожидает ввода с клавиатуры. Необходимо запомнить одно: по отношению к программе сообщения появляются случайным образом. Вот почему *Windows*-программы похожи на программы обработки прерываний: невозможно предсказать, какое сообщение появится в следующий момент.

Поскольку архитектура *Windows*-программ основана на принципе сообщений, все эти программы содержат некоторые общие компоненты. Обычно их приходится в явном виде включать в исходный код. Но, к счастью, при использовании библиотеки *MFC* это происходит автоматически; нет необходимости тратить время и усилия на их написание. Тем не менее для того, чтобы до конца разобраться, как работает *Windows*-программа, написанная с использованием *MFC*, и почему она работает именно так, необходимо в общих чертах понять назначение этих компонентов.

Функция *WinMain* ( ). Все *Windows*-программы начинают выполнение с вызова функции *WinMain* ( ). При традиционном методе программирования это нужно делать явно. С использованием библиотеки *MFC* такая необходимость отпадает, но функция все-таки существует.

Функция окна. Все *Windows*-программы должны содержать специальную функцию, которая не используется в программе, но вызывается самой операционной системой. Эту функцию обычно называют ***функцией окна,*** или ***процедурой окна***. Она вызывается *Windows*, когда системе необходимо передать сообщение в программу. Именно через нее осуществляется взаимодействие между программой и системой. Функция окна передает сообщение в своих аргументах. Согласно терминологии *Windows* функции, вызываемые системой, называются ***функциями обратного вызова***. Таким образом, функция окна является функцией обратного вызова. Помимо принятия сообщения от *Windows* функция окна должна вызывать выполнение действия, указанного в сообщении. Конечно, программа не обязана отвечать на все сообщения, посылаемые *Windows*. Поскольку их может быть сотни, то большинство сообщений обычно обрабатывается самой системой, а программе достаточно поручить *Windows* выполнить действия, предусмотренные по умолчанию.

Следует отметить, что при использовании библиотеки *MFC* функция окна создается автоматически. В этом заключается одно из преимуществ библиотеки. Но в любом случае если сообщение получено, то программа должна выполнить некоторое действие. Хотя она может вызывать для этого одну или несколько *API*-функций, само действие было инициировано *Windows*. Поэтому именно способ взаимодействия с операционной системой через сообщения диктует общий принцип построения всех программ для *Windows*, написанных как с использованием *MFC*, так и без нее.

Цикл сообщений. Как уже сообщалось, *Windows* взаимодействует с программой, посылая ей сообщения. Все приложения *Windows* должны организовать так называемый цикл сообщений (обычно внутри функции *WinMain* ( )). В этом цикле каждое необработанное сообщение должно быть извлечено из очереди сообщений данного приложения и передано назад в *Windows*, которая затем вызывает функцию окна программы с данным сообщением в качестве аргумента. В традиционных *Windows*-программах необходимо самостоятельно создавать и активизировать такой цикл. При использовании *MFC* это также выполняется автоматически. Однако важно помнить, что цикл сообщений все же существует. Он является неотъемлемой частью любого приложения *Windows*.

Класс окна. Как будет показано далее, каждое окно в *Windows*-приложении характеризуется определенными атрибутами, называемыми классом окна. (Здесь понятие «класс» не идентично используемому в *C++*. Оно скорее означает стиль или тип.) В традиционной программе класс окна должен быть определен и зарегистрирован прежде, чем будет создано окно. При регистрации необходимо сообщить *Windows*, какой вид должно иметь окно и какую функцию оно выполняет. В то же

время регистрация класса окна еще не означает создания самого окна. Для этого требуется выполнить дополнительные действия. При использовании библиотеки *MFC* создавать собственный класс окна нет необходимости. Вместо этого можно работать с одним из заранее определенных классов, описанных в библиотеке.

Структура *Windows*-программ отличается от структуры программ других типов. Это вызвано двумя обстоятельствами: во-первых, способом взаимодействия между программой и *Windows*, описанным выше; во-вторых, правилами, которым следует подчиняться для создания стандартного интерфейса *Windows*-приложения, т. е. чтобы сделать программу «похожей» на *Windows*-приложение.

Главная задача *Windows*, поставленная перед ее разработчиками — дать человеку, который хотя бы немного знаком с системой, возможность сесть за компьютер и запустить любое приложение без предварительной подготовки. Для этого *Windows*предоставляет дружественный интерфейс пользователя, который необходимо поддерживать всем программистам, создающим программное обеспечение в данной операционной системе.

Типы данных в Windows. В *Windows*-программах вообще (и в использующих библиотеку *MFC* в частности) не слишком широко применяются стандартные типы данных из *C* или *C++*, такие как *int* или *char**. Вместо них используются типы данных, определенные в различных библиотечных (*header*) файлах. Наиболее часто используются типы *HANDLE*, *HWND*, *BYTE*, *WORD*, *DWORD*, *UNIT*, *LONG*, *BOOL*, *LPSTR* и *LPCSTR*.

Тип *HANDLE* обозначает 32-разрядное целое, используемое в качестве дескриптора. Есть несколько похожих типов данных, но все они имеют ту же длину, что и *HANDLE*, и начинаются с литеры *H*. ***Дескриптор*** — это просто число, определяющее некоторый ресурс:

– тип *HWND* обозначает 32-разрядное целое — дескриптор окна;
– тип *BYTE* обозначает 8-разрядное беззнаковое символьное значение;
– тип *WORD* обозначает 16-разрядное беззнаковое короткое целое;
– тип *DWORD* обозначает беззнаковое длинное целое;
– тип *UNIT* обозначает беззнаковое 32-разрядное целое;
– тип *LONG* эквивалентен типу *long*;
– тип *BOOL* обозначает целое и используется, когда значение может быть либо истинным, либо ложным;
– тип *LPST*R определяет указатель на строку;
– тип *LPCSTR* — константный (*const*) указатель на строку.

§ 6.2. Обзор среды *Microsoft Visual C++*

6.2.1. Структура *Microsoft Visual C++*

Студия разработчика фирмы *Microsoft* (*Microsoft Developer Studio*) — это интегрированная среда для разработки, позволяющая функционировать различным средам разработки, одна из которых *Visual C++*.

В студии разработчика можно строить обычные программы на *C* и *C++*, создавать статические и динамические библиотеки, но основным режимом работы является создание *Windows*-приложений с помощью инструмента *MFC AppWizard* (мастер-приложений) и библиотеки базовых классов *MFC*. Такие приложения называются *MFC*-приложениями. Главная особенность этих *Windows*-приложений состоит в том, что они работают как совокупность взаимодействующих объектов, классы которых определены библиотекой *MFC*.

Как ранее указывалось, главная часть библиотеки *MFC* состоит из классов, используемых для построения компонентов приложения. С каждым *MFC*-приложением связывается определяющий его на верхнем уровне объект *theApp*, принадлежащий классу, производному от *CWinApp*. Как правило, структура приложения определяется архитектурой *Document-View* (документ-вид). Это означает, что приложение состоит из одного или нескольких документов-объектов, классы которых

являются производными от класса *CDocument* (класс «документ»). С каждым из документов связаны один или несколько обликов — объектов классов, производных от *CView* (класс «вид») и определяющих облик документа.

Класс *CFrameWnd* («окна-рамки») и производные от него определяют окна-рамки на дисплее. Элементы управления, создаваемые при проектировании интерфейса пользователя, принадлежат семейству классов элементов управления. Появляющиеся в процессе работы приложения диалоговые окна — это объекты классов, производных от *CDialog*.

Классы *CView*, *CFrameWnd*, *CDialog* и все классы элементов управления наследуют свойства и поведение своего базового класса *CWnd* («окно»), определяющего по существу *Windows*-окно. Этот класс, в свою очередь, является наследником базового класса *CObject* («объект»).

Одна из трудностей в понимании принципов устройства *MFC*-приложения заключается в том, что объекты, из которых оно строится, наследуют свойства и поведение всех своих предков, поэтому необходимо знать базовые классы. У всех *Windows*-приложений есть фиксированная структура, определяемая функцией *WinMain*. Структура приложения, построенного из объектов классов библиотеки *MFC*, является еще более определенной. Приложение состоит из объекта *theApp*, функции *WinMain* и некоторого количества других объектов. Сердцевина приложения — объект *theApp* — отвечает за создание всех остальных объектов и обработку очереди сообщений. Объект *theApp* является глобальным и создается еще до начала работы функции *WinMain*. Работа функции *WinMain* заключается в последовательном вызове двух методов объекта *theApp*: *InitInstance* и *Run*. В терминах сообщений можно сказать, что *WinMain* посылает объекту *theApp* сообщение *InitInstance*, которое приводит в действие метод *InitInstance*.

Получив сообщение *InitInstance*, *theApp* создает внутренние объекты приложения. Процесс создания выглядит как последовательное порождение одних объектов другими. Набор объектов, порождаемых в начале этой цепочки, определен структурой *MFC* практически однозначно — это главная рамка, шаблон, документ, облик. Их роли в работе приложения будут рассмотрены далее.

Следующее сообщение, получаемое *theApp* — *Run* — приводит в действие метод *Run*. Оно как бы говорит объекту: «Начинай работу, начинай процесс обработки сообщений из внешнего мира». Объект *theApp* циклически выбирает сообщения из очереди и инициирует обработку сообщений объектами приложения.

Некоторые объекты имеют графический образ на экране, с которым может взаимодействовать пользователь. Эти интерфейсные объекты обычно связаны с *Windows*-окном. Среди них особенно важны главная рамка и облик. Именно им объект прежде всего распределяет сообщения из очереди через механизм *Windows*-окон и функцию *Dispatch*.

Когда пользователь выбирает команду меню окна главной рамки, то возникают ***командные сообщения***. Они отправляются сначала объектом *theApp* объекту «главная рамка», а затем обходят по специальному маршруту целый ряд объектов, среди которых первыми являются «документ» и «облик», информируя их о пришедшей от пользователя команде.

При работе приложения возникают и обычные вызовы одними объектами методов других объектов. В объектно-ориентированной терминологии такие вызовы могут называться ***сообщениями***. В *Visual C++* некоторым методам приписан именно этот статус (например, методу *OnDraw*).

Важное значение имеют также объекты «***документ***», «***облик***» и «***главная рамка***». Здесь отметим только, что «документ» содержит данные приложения, «облик» организует представление этих данных на экране, а окно «главной рамки» — это окно, внутри которого размещены все остальные окна приложения.

Наследование — одна из фундаментальных идей объектно-ориентированного программирования. Именно этот механизм наследования позволяет программисту дополнять и переопределять поведение базового класса, не вторгаясь в библиотеку *MFC*, которая остается неизменной. Все изменения делаются в собственном производном классе. Именно в этом и заключается работа программиста.

Объекты, из которых состоит приложение, являются объектами классов, производных от классов библиотеки *MFC*. Разработка приложения состоит в том, что программист берет из библиотеки *MFC* классы *CWinApp*, *CFrameWnd*, *CDocument*, *CView* и т. д. и строит производные классы. При-

ложение создается как совокупность объектов этих производных классов. Каждый объект несет в себе как наследуемые черты, определяемые базовыми классами, так и новые черты, добавленные программистом. Наследуемые черты определяют общую схему поведения, свойственную таким приложениям. Новые же черты позволяют реализовать специфические особенности поведения приложения, необходимые для решения стоящей перед ним задачи.

При определении производного класса программист может:

– переопределить некоторые методы базового класса, причем те методы, которые не были переопределены, будут наследоваться в том виде, в каком они существуют в базовом классе;

– добавить новые методы;

– добавить новые переменные.

Приложение, построенное на основе библиотеки *MFC*, — «айсберг», бóльшая часть которого невидима, но является основой всего приложения. Часть приложения, лежащая в библиотеке *MFC* — *framework*, — называется ***остовом приложения***. Рассмотрим работу приложения как процесс взаимодействия между остовом и частью приложения, разработанной программистом. Совершенно естественно, что в методах, определенных программистом, могут встречаться вызовы методов базового класса, что вполне можно рассматривать как вызов функции из библиотеки. Важнее, однако, что и метод производного класса, определенный программистом, может быть вызван из метода родительского класса. Другими словами, остов и производный класс в этом смысле равноправны — их методы могут вызывать друг друга. Такое равноправие достигается благодаря виртуальным методам и полиморфизму, имеющимся в арсенале объектно-ориентированного программирования.

Если метод базового класса объявлен виртуальным и разработчик переопределил его в производном классе, это значит, что при вызове данного метода в некоторой полиморфной функции базового класса в момент исполнения будет вызван метод производного класса и, следовательно, остов вызывает метод, определенный программистом. Точнее, обращение к этому методу должно производиться через ссылку на производный объект либо через объект, являющийся формальным параметром и получающий при вызове в качестве своего значения объект производного класса. Когда вызывается виртуальный метод (назовем его, например, *Method* 1), переопределенный разработчиком, то согласно терминологии *Visual C++* остов посылает сообщение *Method* 1 объекту производного класса, а метод *Method* 1 этого объекта обрабатывает полученное сообщение. Если сообщение *Method* 1 послано объекту производного класса, а обработчик этого сообщения не задан программистом, объект наследует метод *Method* 1 ближайшего родительского класса, в котором определен этот метод. Если же обработчик такого сообщения создан программистом, то он автоматически отменяет действия, предусмотренные родительским классом в отсутствие этого обработчика.

Рассмотрим, как создается приложение с помощью *Visual C++*. Сначала разберем одно важное понятие — проект. До сих пор приложение рассматривалось только как совокупность объектов базовых и производных классов. Но для обеспечения работы приложения требуется наряду с описанием классов и нечто бóльшее —необходимо описание ресурсов, связанных с приложением, нужна справочная система и т. п. Термин «проект» как раз и используется тогда, когда имеется в виду такой более общий взгляд на приложение.

В среде *Visual C++* можно строить различные типы проектов. Такие проекты после их создания можно компилировать и запускать на исполнение. Компания *Microsoft* разработала специальный инструментарий, облегчающий и ускоряющий создание проектов в среде *Visual C++*. Например, мастер *MFC AppWizard*(*exe*) позволяет создать проект *Windows*-приложения, которое имеет однодокументный, многодокументный или диалоговый интерфейс и использует библиотеку *MFC*.

Создаваемый остов приложения составлен так, что в дальнейшей работе с проектом можно использовать другое инструментальное средство — *ClassWizard* (мастер классов), предназначенное для создания остовов новых производных классов. Еще одно основное назначение *ClassWizard* состоит в том, что он создает остовы для переопределяемых методов. Он позволяет показать все сообщения, приходящие классу, и создать остов обработчика любого из этих сообщений. Это только две основные функции *ClassWizard*. Он не всесилен, но его возможности довольно велики.

6.2.2. Инструменты *Visual C++*

В состав компилятора *Microsoft Developer Studio* встроены средства, позволяющие программисту облегчить разработку приложений. В первую очередь к ним относятся *MFC AppWisard*, *ClassWizard* и редактор ресурсов.

Благодаря *MFC AppWizard* среда разработчика позволяет быстро создавать шаблоны новых приложений. При этом программисту не приходится писать ни одной строчки кода. Достаточно ответить на ряд вопросов, касающихся того, какое приложение требуется создать, и исходные тексты шаблона приложения вместе с файлами ресурсов готовы. Эти тексты можно оттранслировать и получить готовый загрузочный модуль приложения.

Для создания ресурсов приложения предназначен редактор ресурсов. Он позволяет быстро создавать новые меню, диалоговые панели, добавлять кнопки к панели управления *toolbar*, полосы прокрутки и т. д.

Средство *ClassWizard* позволяет подключить к созданным и отредактированным ресурсам управляющий ими код. Бóльшую часть работы по описанию и определению функций, обрабатывающих сообщения от меню, органов управления диалоговых панелей и т. д., также берет на себя средство *ClassWizard*.

Среда разработчика *Visual C++* предлагает выбор из множества различных типов проектов приложений. Такие проекты после их создания можно компилировать и запускать на исполнение. *Microsoft* разработала специальный инструментарий, облегчающий и ускоряющий создание проектов в среде *Visual C++*. Рассмотрим основные типы проектов, которые можно создавать при помощи различных средств (мастеров проектов) *Microsoft Visual C++*:

- **MFC AppWizard (exe)** —при помощи мастера приложений можно создать проект *Windows*-приложения, которое имеет однодокументный, многодокументный или диалоговый интерфейс. Однодокументное приложение может предоставлять пользователю в любой момент времени работать только с одним файлом. Многодокументное приложение, напротив, может одновременно представлять несколько документов, каждый в собственном окне. Пользовательский интерфейс диалогового приложения представляет собой единственное диалоговое окно;
- **MFC AppWizard (dll)** —мастер приложений позволяет создать структуру *DLL*, основанную на *MFC*. С его помощью можно определить характеристики будущей *DLL*;
- **AppWizard ATL COM** —средство позволяет создать элемент управления *ActiveX* или сервер автоматизации, используя новую библиотеку шаблонов *ActiveX*. Опции этого мастера дают возможность выбрать активный сервер (*DLL*) или исполняемый внешний сервер (*exe*-файл);
- **Custom AppWizard** — с помощью этого средства можно создать пользовательские мастера *AppWizard*. Пользовательский мастер может базироваться на стандартных мастерах для приложений *MFC* или *DLL*, а также на существующих проектах или содержать только определяемые разработчиком шаги;
- **MFC ActiveX Control Wizard** — мастер элементов управления реализует процесс создания проекта, содержащего один или несколько элементов управления *ActiveX*, основанных на элементах управления *MFC*;
- **Win32 Application** — мастер позволяет создать проект обычного *Window*-приложения. Проект создается незаполненным, файлы с исходным кодом в него следует добавлять вручную;
- **Win32 Console Application** — мастер создания проекта консольного приложения. Консольное приложение — это программа, которая выполняется из командной строки окна *DOS* или *Windows* и не имеет графического интерфейса (окон). Проект консольного приложения создается пустым, предполагая добавление файлов исходного текста в него вручную;
- **Win32 Dynamic-Link Library** — создание пустого проекта динамически подключаемой библиотеки. Установки компилятора и компоновщика будут настроены на создание *DLL*. Исходные файлы следует добавлять вручную.
- **Win32 Static Library** — средство создает пустой проект, предназначенный для генерации статической (объектной) библиотеки. Файлы с исходным кодом в него следует добавлять вручную.

Рассмотрим преимущества использования мастеров в процессе создания приложений. Прежде всего нужно отметить, что создание проекта — это не только творчество, но и большой объем технической работы, требующей внимания и аккуратности. Например, все *Windows*-приложения имеют достаточно общую структуру, следовательно, можно построить некоторые шаблонные заготовки, подходящие для того или иного типа проектов. Построению таких заготовок способствует то, что приложения, создаваемые на основе *MFC*, строятся из элементов фиксированных классов. Логическим развитием этой идеи было введение специальных классов и специальной архитектуры построения приложения, которая подходила бы широкому классу приложений. О такой архитектуре уже упоминалось, когда речь шла о библиотеке *MFC* — это архитектура *Document-View* (документ-вид). Она является основной, но не единственной при построении проектов в среде *Visual C++*.

Суть данной архитектуры состоит в том, что работу многих приложений можно рассматривать как обработку документов. При этом можно отделить сам документ, отвечающий за представление и хранение данных, от образа этого документа, видимого на экране и допускающего взаимодействие с пользователем, который просматривает и (или) редактирует документ. В соответствии с этой архитектурой библиотека *MFC* содержит два семейства классов, производных от базовых классов *CDocument* и *CView*.

В результате появилась двухэтапная технология создания проектов. Вначале создается некая заготовка проекта с общими свойствами, подходящими для многих проектов этого типа. На втором этапе производится уже настройка, учитывающая специфику задачи. Для каждого этапа фирма *Microsoft* разработала свое инструментальное средство.

Начальная заготовка — ***остов приложения*** — создается в диалоге с пользователем инструментальным средством *AppWizard*. В процессе диалога пользователь определяет тип и характеристики проекта, который он хочет построить. Определив, какие классы из *MFC* необходимы для этого проекта, *AppWizard* строит остовы всех нужных производных классов. Построенный *AppWizard* остов приложения содержит все необходимые файлы для создания стартового приложения, которое является законченным приложением и обладает разумными функциональными свойствами, общими для целого класса приложений. Естественно, никаких специфических для данного приложения свойств остов не содержит. Они появятся на следующем этапе, когда программист начнет работать с остовом, создавая из заготовки свое собственное приложение. Тем не менее стартовое приложение можно транслировать и запускать на исполнение.

Термин ***остов*** (приложения, класса, функции) применяется для заготовок, создаваемых инструментальными средствами *AppWizard*. Создаваемый остов приложения составлен так, что в дальнейшей работе с проектом можно использовать другое инструментальное средство — *ClassWizard* (мастер классов). Средство *ClassWizard* предоставляет широкий спектр услуг. Оно позволяет добавлять к существующему проекту следующие новые элементы:

• **Создание нового класса.** При помощи *ClassWizard* можно добавить новый класс, созданный на основе базовых классов. В качестве базового класса можно использовать классы, наследованные от классов *CCmdTarget* или *CRecordset*. Для наследования классов от других базовых классов нельзя использовать средства *ClassWizard*. Такие классы надо создавать вручную, непосредственно в текстовом редакторе. Объекты, порожденные от класса *CCmdTarget*, могут обрабатывать сообщения *Windows* и команды, поступающие от меню, кнопок, акселераторов. Класс *CCmdTarget* и другие наследованные от него классы имеют таблицу сообщений (*MessageMap*) — набор макрокоманд, позволяющий сопоставить сообщения *Windows*и команды метода класса. Полученная заготовка класса полностью работоспособна. Ее можно дополнить по своему усмотрению новыми методами и данными. Эту работу можно выполнить вручную, но гораздо лучше и проще воспользоваться услугами *ClassWizard*. За счет использования *ClassWizard* процедура создания собственного класса значительно ускоряется и уменьшается вероятность совершить ошибку во время объявления методов.

• **Включение в класс новых методов.** Очень удобно использовать *ClassWizard* для включения в состав класса новых методов. Можно добавлять к классу методы, служащие для обработки сообщений *Windows* и команд от объектов, а также методы, переопределяющие виртуальные методы базовых классов. *ClassWizard* позволяет не только добавить в класс новые методы, но и удалить их. *ClassWizard* самостоятельно удалит объявление метода из класса.

• **Включение в класс новых элементов данных.** *ClassWizard* позволяет включать в класс не только новые методы, но и элементы данных, связанные с полями диалоговых панелей, форм просмотра и форм для просмотра записей баз данных и полей наборов записей. *ClassWizard* использует специальные процедуры, чтобы привязать созданные им элементы данных класса к полям диалоговых панелей. Эти процедуры носят названия «обмен данными диалоговой панели» и «проверка данных диалоговой панели» (*Dialog Data Exchange and Dialog Data Validation — DDX/DDV*). Чтобы привязать поля из наборов записей к переменным, используется процедура обмена данными с полями записей (*Record Field Exchange — RFX*). Процедуры *DDX/DDV* и *RFX* значительно упрощают программисту работу с диалоговыми панелями. Они позволяют связать поля диалоговых панелей и переменные. Когда пользователь редактирует поля диалоговых панелей, процедуры *DDV* проверяют введенные значения и блокируют ввод запрещенных значений. Затем процедуры *DDX* автоматически копируют содержимое полей диалоговых панелей в привязанные к ним элементы данных класса. И наоборот, когда приложение изменяет элементы данных класса, привязанные к полям диалоговой панели, процедуры *DDX* могут сразу отобразить новые значения полей на экране компьютера.

6.2.3. Соглашения об именах, используемых в Visual *C++*

Библиотека *MFC* содержит большое количество классов, структур, констант и т. д. Для того чтобы текст *MFC*-приложений был более легким для понимания, принято применять ряд соглашений для используемых имен и комментариев. Названия всех классов и шаблонов классов библиотеки *MFC* начинаются с заглавной буквы *C*. При наследовании классов от классов *MFC* можно давать им любые имена. Рекомендуется начинать их названия с заглавной буквы *C*. Это сделает исходный текст приложения более ясным для понимания.

Для имен переменных *Microsoft* предлагает более сложную систему, предусматривающую обозначение именуемых типов данных. Для этого используется небольшой префикс из строчных букв, а собственно имя начинается с заглавной буквы, например: *lUnit* — переменная типа *long*. Типы префиксов представлены в табл. 6.1.

Таблица 6.1

Префикс	Тип данных
1	*2*
a	Массив
b	Булевский (байт)
by	Беззнаковый тип (байт)
c	Символ (байт)
cb	Счетчик байтов
cr	Цвет
cx, cy	Короткий тип (*short*)
dw	Длинное беззнаковое целое (*DWORD*)
f	16-битный флаг (битовая карта)
fn	Функция
h	Дескриптор (*handle*)
l	Длинное целое (*long*)
I	Данные типа *int*
m_	Переменная класса
lp	Длинный указатель (*long pointer*)
n	Целое (16 бит)
np	Ближний указатель
p	Указатель (*pointer*)

Окончание табл. 6.1

1	*2*
pt	Точка (два 32-битных целых)
w	Целое без знака (*WORD*, 16 бит)
s	Строка
sz	Указатель на строку, заканчивающуюся 0 (*string>zero*)
tm	Текстовая метрика
x, y	Короткий тип (координата *x* или *y*)
lpsz	Длинный указатель на *sz* (*long pointer string zero*)
rgb	Длинное целое, содержащее цветовую комбинацию *RGB*

Библиотека *MFC* включает в себя, помимо классов, набор служебных функций. Названия этих функций начинаются с символов *Afx*, например *AfxGetApp*. Символы *AFX* являются сокращением от словосочетания *Application Framework X*, означающих основу приложения, его внутреннее устройство. Символы *AFX* встречаются не только в названии функций *MFC*. Многие константы, макрокоманды и другие символы начинаются с этих символов. В общем случае *AFX* является признаком, по которому можно определить принадлежность того или иного объекта (функция, переменная, ключевое слово или символ) к библиотеке *MFC*.

Когда приложение разрабатывается средствами *MFC AppWizard* и *ClassWizard*, они размещают в исходном тексте приложения комментарии следующего вида:

```
//{{AFX_ …
//}}AFX_
```

Такие комментарии образуют блок кода программы, который управляется только средствами *MFC AppWizard* и *ClassWizard*. Пользователь не должен вручную вносить изменения в этом блоке. Для этого необходимо употреблять средства *ClassWizard*. В табл. 6.2 представлено краткое описание некоторых блоков //{{AFX_.

Таблица 6.2

Блок	Описание
//{{AFX_DATA //}}AFX_DATA	Включает объявление элементов данных класса. Используется в описании классов диалоговых панелей
//{{AFX_DATA_INIT //}}AFX_DATA_INIT	Включает инициализацию элементов данных класса. Используется в файле реализации классов диалоговых панелей
//{{AFX_DATA_MAP //}}AFX_DATA_MAP	Включает макрокоманды *DDX*, предназначенные для связывания элементов данных класса и органов управления диалоговых панелей. Используется в файле реализации классов диалоговых панелей
//{{AFX_MSG //}}AFX_MSG	Включает описание методов, которые предназначены для обработки сообщений. Этот блок используется при описании класса
//{{AFX_MSG_MAP //}}AFX_MSG_MAP	Включает макрокоманды таблицы сообщений класса. Используются совместно с *AFX_MSG*
//{{AFX_VIRTUAL //}}AFX_VIRTUAL	Включает описание переопределенных виртуальных методов класса. Блок *AFX_VIRTUAL* используется при описании класса

MFC AppWizard и *ClassWizard* помогают разрабатывать приложения. Они создают все классы и методы, необходимые для его работы. Программисту остается дописать к ним свой код. В тех местах, где можно вставить свой код, *MFC AppWizard* и *ClassWizard* помещают комментарии:

```
//TODO:
```

§ 6.3. Разработка проекта в *Microsoft Visual C++*

Профессиональный разработчик программного обеспечения вынужден бóльшую часть жизни проводить за компьютером, бесконечно повторяя цепочку одних и тех же действий, которые составляют суть процесса отладки проекта. Студия разработчика *Visual C++* предоставляет для этой цели целый набор достаточно удобных встроенных инструментов (*Tools*). Возможности студии, может быть, не столь эффектны, как в системе визуального дизайна *Borland C++ Builder*, но они позволяют разрабатывать проекты различных типов, организовывать зависимые (*dependent*) проекты, объединять проекты разных типов. При необходимости можно создавать и отлаживать как структурно-, так и объектно-ориентированные программы внутри интегрированной среды разработки *Integrated Development Environment* (*IDE*), используя следующие инструменты: редакторы текста и ресурсов, окна просмотра и встроенный отладчик. Также возможно изменять настройки компилятора и компоновщика.

Создаваемые файлы должны быть включены в *Visual C++ Project*, который, в свою очередь, необходимо поместить в рабочее пространство (*Workspace*). Так называется хранилище проектов или область памяти (папка на диске), где расположены проекты и их конфигурации. *Workspace*может содержать несколько разных проектов (*multiple projects*), в том числе проекты различных типов. При запуске оболочки *Visual C++* раскрывается окно редактора текста, а слева от него — окно, которое носит название *Workspace*. Вкладки или страницы этого окна представляют собой инструменты для просмотра (*Viewers*) файлов проекта:

- **Class View** — демонстрирует классы, их данные и методы;
- **File View** — показывает все файлы, включенные в проект;
- **Resource View** — позволяет быстро попасть в нужный редактор ресурсов.

При создании нового проекта (по команде меню *File–>New–>Projects*) по умолчанию создается один проект, но в двух конфигурациях:

- **Debug** — версия проекта, в которой выключается оптимизация кода и включается отладочная информация;
- **Release** — версия, в которой делается все наоборот (выключается отладочная информация и включается оптимизация кода).

Сначала можно увидеть только одну из папок (*Debug* или *Release*) в зависимости от выбора активной конфигурации в меню *Build–>Set Active Configuration*. Обычно при создании нового проекта (*Project*) он автоматически помещается во вновь созданное рабочее пространство (*Workspace*) в конфигурации *Debug*. Нужно отметить, что демонстрация файлов проекта в отдельной папке проекта не означает, что файлы физически расположены в одной папке. Оболочка может использовать ссылки на файлы из других директорий, но графически отображать файлы так, как будто они находятся в папке проекта. В процессе работы с проектом можно безболезненно удалять папки *Debug* или *Release*. Они автоматически восстановятся при последующей компиляции исходных файлов.

Когда создано рабочее пространство, в него можно добавлять:

- новые проекты;
- новые конфигурации (*configurations*);
- взаимозависимости (*interdependencies*) между проектами;
- подчиненные проекты (*sub projects*).

Как ранее было указано, *Visual C++* предлагает выбор из множества различных типов проектов приложений. Их все можно увидеть, выбрав пункт меню *File–>New–>Projects*. Инструмент студии разработчика *MFC App Wizard*, о котором уже упоминалось ранее, автоматизирует процесс создания начальной заготовки приложения и по выбранному типу проекта создает стартовый остов приложения, состоящий из одного-двух десятков файлов. Есть файлы, которые существуют во всех типах проектов. Рассмотрим их. Предположим, что выбран какой-то тип проекта (из возможных альтернатив, предложенных в диалоге *File–>New–>Projects*) и в качестве имени проекта (*ProjectName*) задали имя *First*. *AppWizard* создаст каталог *First* и поместит туда 4 файла:

First.dsw — это ***developer studio workspace***-файл, используемый средой *IDE*. Он помогает объединить все проекты в одном рабочем пространстве.

First.opt — этот файл содержит ***workspace options,*** используемые *IDE*. Здесь сохраняются все настройки рабочего пространства, выбранные в течение работы с проектом.

First.dsp — файл типа ***developer studio project.*** Здесь хранится информация, описывающая конкретный проект. Сколько проектов в ***workspace*** — столько и *dsp*-файлов.

First.ncb — файл типа ***no compile browser,*** в котором располагается информация, генерируемая синтаксическим анализатором и используемая такими инструментами, как *ClassView*, *WizardBar*, *Component Gallery*.

Помимо этих файлов (созданных изначально) в течение работы с проектом в директории *First* или в ее поддиректориях будут появляться другие файлы, которые представлены в виде списка (данный список можно использовать в качестве справочника):

StdAfx.h, **StdAfx.cpp** — используются для построения файла скомпилированных заголовков *First.pch* (***precompiled header***) и файла скомпилированных типов *StdAfx.obj*, которые значительно ускоряют повторные компиляции всех файлов проекта после внесенных изменений (если они незначительны).

First.h — главный файл заголовков (***header file***), который содержит определения глобальных переменных, классов и типов, директивы # *include* для подключения других файлов заголовков. В нем объявляется класс приложения *CFirstApp* (производный от класса *CWinApp*) и переопределяется метод *InitInstance*.

First.cpp — это главный файл исходного текста программы. Он создает один объект класса *CFirstApp* и дает тело метода *InitInstance*.

FirstDlg.h, **FirstDlg.cpp** — существуют, если приложение создано на основе диалоге (одна из возможностей, предоставляемая *AppWizard*).

FirstDoc.h, **FirstDoc.cpp** — содержат описание и остовы методов класса, производимого от *CDocument*. Модель программирования ***Документ-Вид*** и, в частности, класс *CDocument* поддерживают стандартные операции с абстрактным документом, такие как его создание, запись содержимого в файл и чтение из файла (***serialization***).

FirstView.h, **FirstView.cpp** — содержат описание и остовы методов класса, производимого от класса *CView*, который инкапсулирует данные и методы для воспроизведения содержимого документа. Один документ может иметь несколько обликов (видов). Класс *CFirstView* может происходить от классов *CEditView*, *CFormView*, *CRecordView*, *CDaoRecordView*, *CTreeView*, *CListView*, *CRichEditView*, *CScrollView*, которые помогают разработчику создавать различные облики документа.

MainFrm.h, **MainFrm.cpp** — содержат описание и остовы методов класса *CMainFrame*, производимого от класса *CFrameWnd* (для *SDI*-приложений) или *CMDIFrameWnd* (для *MDI*-приложений). Класс управляет поведением главного окна приложения, в частности таких его элементов, как *ToolBar* и *StatusBar*.

Resource.h, **First.rc** — файлы ресурсов, которые содержат определения меню, значков, картинок, клавиш быстрого доступа (***hot-keys***), диалогов, строк подсказок, высвечиваемых в строке состояния, и т. д. В них есть директивы для включения файлов *afxres.h* и *afxres.rc*, которые содержат определения стандартных ресурсов и ссылку на файлы *res\First.ico*и*res\toolbar.bmp*с *bitmap*-изображениями значков приложения.

First.rc2 — файл, который содержит общие ресурсы, используемые в нескольких проектах. Помещая в данный файл ресурсы, многократно используемые различными проектами, необходимо подключить *First.rc*2 директивой # *include* (в файле *First.rs*).

First.clw — используется одним из *Tools* (***ClassWizard***) для хранения информации о классах в вашем проекте.

First.odl — содержит исходный код языка описания объектов (***object description language***), необходимый для библиотеки элементов управления.

Readme.txt — содержит описание файлов проекта.

В проектах ранних версий студии, предназначенных для работы в режиме *Win16*, можно обнаружить файлы с расширением *def* — файлы описания *exe*-модуля (сегменты *DATA* и *CODE*, размер областей *heap* и *stack* и т. д.).

Программа в *Visual C++ AppWizard* состоит из четырех основных частей:

- объект приложения;
- объект главного окна;
- объект документа;
- объект вида.

• ***Объект приложения*** включает в себя файлы с расширением **.cpp* (содержит исходный текст приложения) и **.h* (содержит определения констант, переменных и методов классов).

• ***Объект главного окна*** — сама программа, меню, заголовок окна, панель инструментов. Рабочая зона программы — клиентская область окна.

• ***Объект вида*** осуществляет работу с клиентской областью — местом, где в соответствующем формате отображаются данные программы (например, текст, если приложение — текстовый редактор). В целом следует отметить, что объект вида — окно, которое накладывается поверх клиентской области.

• ***Объект документа*** — место, в котором хранятся данные программы.

§ 6.4. Отладка программ в *Visual C++*

Для того чтобы изучить процесс отладки, начнем с создания программы, заведомо содержащей несколько ошибок (она будет называться *bed*), а затем посмотрим, как с помощью отладчика найти и исправить эти ошибки. В частности, научимся устанавливать в программе ***точки прерывания*** и выполнять ее до заданной точки. Кроме того, программу можно выполнять последовательно, строка за строкой (метод пошагового выполнения). Этот режим позволяет следить за тем, как изменяются значения различных переменных, а также иногда он помогает понять, в чем заключается проблема: если переменная получает неожиданное значение, то скорее всего в программе допущена ошибка. После того как ошибки будут обнаружены, их можно исправить и заново отладить программу.

Создадим программу *bed*, определяющую среднее арифметическое первых пяти положительных целых чисел (1–5) и отладим ее. При помощи *AppWizard*(*exe*) создадим*SDI*-программу (однодокументный интерфейс). Для хранения пяти целых чисел воспользуемся массивом *data*. Начнем с включения этого массива в заголовочный файл документа:

```
                                        // Operations
public:
    int data[5];
. . .
```

Исходные числа будут заноситься в массив в конструкторе документа:

```
CBedDoc::CBedDoc( )
{
                                        // TODO: add one-time construction code here
    data[0]=1;
    data[1]=2;
    data[2]=3;
    data[3]=4;
    data[4]=5;
}
```

С помощью редактора меню добавьте новое меню *Calculate* (расположите его между *File* и *Edit*) с единственной командой — *CalculateAverage*. С помощью *ClassWizard* создайте в классе вида обработчик для этой команды и добавьте в него следующий код:

```
void CBedView::OnCalculateCalculateaverage ( )
{
    CBedDoc* pDoc = GetDocument( );
    ASSERT_VALID(pDoc);
                                // Переменные для вычисления суммы и среднего арифметического
    float Sum;
    float Average;
                                // Цикл для вычисления суммы
    for (int i=1; i<5; i++)
    {
        Sum+=pDoc–>data[i];
    }
                                // Вычисление среднего арифметического
    Average=Sum/(float)5.0;
                                // Вывод данных на экран
    OutString.Format("Среднее арифметическое пяти чисел равно:%.3f", Average);
    Invalidate( );
}
```

Объявим OutString в заголовочном файле вида:

```
. . .
protected:
    CString OutString;
. . .
```

Добавим код в метод OnDraw для отображения данных:

```
void CBedView::OnDraw(CDC* pDC)
{
    CBedDoc* pDoc = GetDocument( );
    ASSERT_VALID(pDoc);
      // TODO: add draw code for native data here
    pDC–>TextOut(0, 0, OutString);
}
```

Запустите ее и выберите команду *CalculateAverage*. Вы обнаружите, что программа выдает данные, не соответствующие действительности.

На следующем этапе перейдем к отладке программы. Желательно попасть в программу во время ее работы, чтобы можно было начать пошаговое выполнение, однако начинать с самого начала программы не хочется, поскольку в этом случае нам придется проходить через стартовый код *Visual C++*, который нет необходимости рассматривать. Отладка должна начинаться поближе к тому месту, где, как можно предположить, возникает проблема — к фрагменту, отвечающему за вычисление среднего арифметического. Чтобы начать отладку с произвольной строки, следует установить ***точку прерывания***. Во время выполнения программа остановится на ней, и в результате появляется возможность просмотреть свой код и при необходимости перейти в пошаговый режим. Поместим точку прерывания в начале фрагмента для вычисления среднего арифметического, конкретнее — в начале метода *OnCalculateCalculateaverage* ( ), в первой строке цикла *for*. Точку прерывания можно установить перед запуском программы или во время ее остановки на другой точке. Следует отметить, что компилятор не позволяет устанавливать точки прерывания на строках программы, содержащих объявления переменных. Для того чтобы разместить точку прерывания, установите на необходимой строке текстовый курсор и нажмите на панели инструментов клавишу *F9* или кнопку с изображением поднятой руки. В результате будет установлена точка прерывания, которая обозначается маленьким значком в виде стоп-сигнала на левом поле. Нажимая клавишу *F10*, можно перемещаться по коду программы. Однократное нажатие этой клавиши переводит к следующей строке программы. Многократно нажимая *F10*, можно перемещаться в программе и дальше. Одновременно можно просмотреть значения различных переменных.

Запустите программу *bed* в отладчике — для этого достаточно выполнить команду *Build–>StartDebug–>Go*. На экране появляется окно приложения и появляется возможность выполнить команду *Calculate–>CalculateAverage*. В результате код метода *OnCalculateCalculateAverage*( ) выполняется до строки, на которой установлена точка прерывания. Далее программа останавливается и отображает в окне *Visual C++* код метода. Следует обратить внимание на то, что в *Visual C++* меню *Build* заменилось на *Debug*. Среди различных команд этого меню особый интерес представляют команды *StepInto* (*F11*), *StepOver* (*F10*), *StepOut* (*Shift + F11*) и *RunToCursor* (*Ctrl + F10*). Они соответствуют различным способам выполнения программы, предусмотренным в режиме отладки *Visual C++*.

Во время отладки программ на *Visual C++* часто встречаются строки с вызовами различных методов (например, *PerformWork*(*data*)). Если продолжить пошаговое выполнение программы с такой строки, то осуществится переход к коду вызываемого метода (например, *PerformWork*( ), который может быть достаточно длинным). Это означает, что для возврата к отлаживаемому фрагменту необходимо пройти через весь код метода. Если необходимо пропустить данный код, воспользуйтесь клавишей *F*10. Для прохода через код вызываемого метода пользуйтесь клавишей *F11*. Если вы окажетесь внутри вызванного метода или другого блока, который не хотите отлаживать, то можете выйти за его пределы, нажав клавиши *Shift + F11*. Существует и другая возможность — установить курсор в некоторой точке программы за текущей выполняемой строкой и нажать клавиши *Ctrl + F10*, что приведет к выполнению кода до строки с курсором. В нашем примере для пошагового выполнения программы будет использоваться клавиша *F10*. Нажмите ее один раз, чтобы перейти к следующей строке программы. Текущей становится следующая строка программы. Именно в ней происходит суммирование чисел для получения накапливаемой суммы в переменной *Sum*. Чтобы выполнить текущую строку, обновить значение суммы и перейти к следующей строке, еще раз нажмите клавишу *F10*. К переменной *Sum* прибавлено первое целое число, значение которой можно проверить и убедиться, что все идет нормально.

Чтобы узнать значение переменной *Sum* (в которой должно храниться первое целое число, равное 1), задержите над ней указатель мыши. Рядом с именем переменной на экране появляется подсказка со значением, отличающимся от необходимого числа. Похоже, проблема найдена. Помимо экранной подсказки переменная *Sum* со своим значением отображается в окне *Auto*, расположенном в левом нижнем углу. В этом окне приведены значения последних переменных, с которыми работал *Visual C++*. Среди них есть и наша переменная *Sum*. Кроме того, вы можете щелкнуть на корешке *Locals* и перейти к одноименному окну, содержащему значения всех переменных (включая *Sum*), определенных в текущем методе или фрагменте кода. Если же вы хотите отслеживать значение конкретной переменной во время всего выполнения программы, то введите ее имя в окне *Watch*, расположенном в правом нижнем углу окна *Visual C++*. Кроме того, нужную переменную можно просто перетащить мышью в окно *Watch*.

Просмотр программы показывает, что не обнулено начальное значение суммы. Сделайте это. Прежде всего завершите сеанс отладки командой *Debug–>StopDebugging*, затем отредактируйте метод и присвойте переменной *Sum* начальное значение 0:

```
void CBedView::OnCalculateCalculateaverage ( )
{
    // TODO: Add your command handler code here
    CBedDoc* pDoc = GetDocument( );
    ASSERT_VALID (pDoc);
    float Sum;                                  // Набрано ошибочно
    float Sum=0;
    . . .
}
```

Ошибка с инициализацией исправлена. Снова запустите программу. Теперь программа сообщает, что среднее арифметическое первых пяти целых чисел равно 2,800. Конечно, это гораздо ближе к ожидаемому значению 3,000, но все-таки не оно. Пора возвращаться к отладчику. Снова начните отладку командой *Build–>StartDebug–>Go* и выполните программу до точки прерывания. Войдите в цикл *for* клавишей *F10*. Задержите указатель мыши над переменной *Sum*, чтобы убедиться, что ее значение действительно равно 0.

С переменной *Sum* все нормально; давайте проверим значение, которое к ней добавляется, — *pDoc –> data* [*i*]. Невозможно узнать значение выражения *pDoc –> data* [*i*], задерживая над ним указатель мыши (непонятно, что представляет интерес — только *pDoc* или все выражение?), но оно присутствует в окне *Auto* в левом нижнем углу. Видно, что к сумме прибавляется число 2 (а не 1, как ожидалось). Просмотр кода показывает, что начальное значение переменной цикла равно 1, а не 0, как положено. Чтобы узнать значение того или иного выражения во время выполнения программы, можно выполнить команду *Debug–>QuickWatch* и ввести выражение в диалоговом окне *QuickWatch*. *Visual C++* вычислит его значение. Остановите отладку и исправьте ошибку:

```
void CBedView::OnCalculateCalculateaverage ( )
{
    // TODO: Add your command handler code here
    . . .
    //for (int i=1; i<5; i++)                          // Набрано ошибочно
    for (int i=0; i<5; i++)
    {
        Sum+=pDoc–>data[i];
    }
    . . .
}
```

Запустите программу. Получен желаемый результат — среднее арифметическое чисел 1–5 равно 3. Встроенные отладочные средства *Visual C++* помогли ликвидировать ошибки. Чтобы удалить из программы точку прерывания, установите курсор на строке с ней и нажмите клавишу *F9*. Закончив отладку, следует удалить отладочную информацию из исполняемого файла. Выполните команду *Build–>SetActiveConfiguration* и выберите из раскрывающегося списка строку *bed — Win32Release* (по умолчанию, в программы на *Visual C++* включается большой объем информации, используемой отладчиком).

Список литературы

1. *Карпов Б., Баранова Т.* С++:спец. справ. — СПб.: Питер, 2001. — 479 с.
2. *Керниган Б., Ритчи Д.* Язык программирования Си. — М.: Финансы и статистика, 1992. — 272 с.
3. *Климова Л. М.* С++. Практическое программирование. — М.: Кудиц-образ, 2001. — 587 с.
4. *Кнут Д.* Искусство программирования для ЭВМ. Ч.1. — М.: Мир, 1976. — 726 с.
5. *Марапулец Ю. В.* Основы программирования на языке Си. — Петропавловск-Камчатский: КамчатГТУ, 2002. — 100 с.
6. *Марапулец Ю. В.* Основы программирования на языке С++. — Петропавловск-Камчатский: КамчатГТУ, 2003. — 157 с.
7. *Намиот Д. Е.* Основные особенности языка программирования С++. — М.:Память, 1991. — 96 с.
8. *Павловская Т. А.* С/С++. Программирование на языке высокого уровня. — СПб.:Питер, 2001. — 460 с.
9. *Подбельский В. В.* Язык Си++. — М.:Финансы и статистика, 1996. — 559 с.
10. *Романов Е. Л.* Практикум по программированию на С++. — СПб.:БХВ-Петербург, 2004. — 427 с.
11. *Страуструп Б.* Язык программирования Си++. — М.:Радио и связь, 1991. — 352 с.

ПРИЛОЖЕНИЕ

ОПТИМИЗАЦИЯ ПРОГРАММ НА ЯЗЫКЕ *C++*

Уменьшение времени доступа в память

1. Групповое присваивание:

использование a=b=1;.
вместо a=1; b=l;

2. Использование оператора присваивания в условии:

```
if ((c=getchar( ))!=EOF)
{   }
```

вместо

```
c=getchar( );
if (c!=EOF)
{...}
```

3. Использование условного оператора:

```
a=(x0)?x:x+1;
```

вместо

```
if (x0)
    a=x;
else a=x+1;
```

4. Использование операции сдвига при умножении на степень 2:

```
i=(j<<3)+j; // j*8+j
```

вместо

```
i=j*9;
```

5. Использование операций ++, −−, +=, −=:

```
mas [i+2] ++;
```

вместо

```
mas [i+2] =mas [i+2] +1;
```

Преобразования типов

Следует отметить, что наиболее быстро выполняются операции, не связанные с преобразованием типа. Например, аргументы вещественного типа в выражениях при вычислении преобразуются в тип *double*. Поэтому если они уже описаны как *double*, то программа будет работать быстрее. Также необходимо обратить внимание на операции присваивания для типов разной длины (*long* и *int*, *char* и *int*). Реализация таких операций требует в общем случае включения специальных команд расширения (например, знакового расширения).

Условия в операторе *if*

По стандарту языка *C++* условия в условном операторе вычисляются последовательно. При этом если условие есть, например, некоторая конъюнкция элементарных посылок, то при ложности одной из них все остальные уже не вычисляются. Это можно использовать, располагая условия в порядке, обеспечивающем минимальное число вычислений. Аналогичные рассуждения применимы и для дизъюнкции элементарных посылок.

Функции и передача параметров

Остановимся на следующих моментах:

– временной выигрыш можно получить, реализуя функции как текстовые макросы;

– функции могут быть реализованы как *inline* (открытая подстановка);

– для сложных агрегатов данных передача указателей осуществляется быстрее, чем передача самих значений, поскольку в последнем случае выполняется полное копирование аргументов.

Использование указателей вместо индексации

При работе с массивами использование указателей может обеспечить выигрыш во времени по сравнению с индексацией. Например, вместо

```
mas [0] =mas [i] +mas [j];
```

можно использовать

```
*mas=*(mas+i) + *(mas+j);
```

Оглавление

Учебное издание

Марапулец Юрий Валентинович

ЯЗЫК C++. ОСНОВЫ ПРОГРАММИРОВАНИЯ

Издание второе, исправленное и дополненное

Авторская редакция

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Камчатский государственный университет имени Витуса Беринга»
683032, Петропавловск-Камчатский, ул. Пограничная, 4
Тел. 8(415-2) 42-68-42, www.kamgu.ru

Подписано в печать 23.04.2019. Формат 60 × 84 / 8
Бумага офсетная. Печать цифровая
Гарнитура «TimesNewRoman». Усл. печ. л. 18,36. Уч.-изд. л. 9,52
Тираж 500 экз.

Отпечатано в КамГУ им. Витуса Беринга