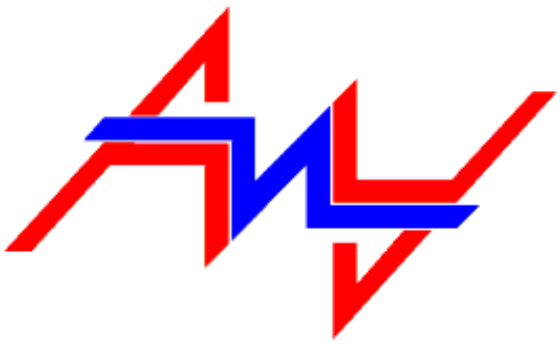
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Государственное образовательное учреждение высшего профессионального образования

Казанский национальный исследовательский технический университет им. А.Н.Туполева - КАИ



А.Г. Аузяк, Ю.А. Богомолов, А.И. Маликов, Б.А. Старостин

Программирование и основы алгоритмизации

Учебное пособие

Казань 2013

УДК 62.52/07

Программирование и основы алгоритмизации: Для инженерных специальностей технических университетов и вузов. /А.Г. Аузяк, Ю.А. Богомолов, А.И. Маликов, Б.А. Старостин. Казань: Изд-во Казанского национального исследовательского технического ун-та - КАИ, 2013, 153 с.

Пособие соответствует требованиям государственного образовательного стандарта высшего профессионального образования по направлению подготовки бакалавров 220400.62 -"Управление в технических системах".

Учебное пособие предназначено для студентов, изучающих дисциплину "Программирование и основы алгоритмизации". В пособии рассматривается классификация вычислительных алгоритмов, приводятся примеры составления различных алгоритмов для прикладных задач и изложены основы программирования на языке С++.

Предисловие

Цель данного учебного пособия – помочь студенту в изучении основ алгоритмизации и элементов программирования на языке C++.

В первом разделе пособия наряду с изложением общих понятий приводится принятая классификация вычислительных алгоритмов. Рассматриваются линейные, разветвляющиеся и циклические алгоритмы, а также приводятся примеры составления алгоритмов для различных прикладных задач.

Второй раздел пособия связан с изучением основ программирования на языке C++. Методика изложения учебного материала пособия, в основном, связана с разбором примеров, а не голой формулировкой правил. Примеры, приведенные в учебном пособии, в их большей части являются законченными реальными программами, а не отдельными фрагментами. Примеры напечатаны в виде, пригодном для ввода в машину.

При работе над учебным пособием использовался компилятор, входящий в состав интегрированной среды разработки Borland C++ 3.1, который подходит как нельзя лучше для цели обучения основам программирования на C++. К тому же выбор транслятора абсолютно не принципиален. Следует лишь иметь в виду, что выполнение примеров в других инструментальных средах в ряде случаев может привести к иным результатам.

Раздел I. Понятие об алгоритмах

1.1. Определение алгоритма

Слово «Алгоритм» происходит от algorithmi – латинского написания имени аль-Хорезми, под которым в средневековой Европе знали величайшего математика из Хорезма (город в современном Узбекистане) Мухаммеда бен Мусу, жившего в 783-850 гг. В своей книге «Об индийском счете» он сформулировал правила записи натуральных чисел с помощью арабских цифр и правила действий над ними столбиком. В дальнейшем алгоритмом стали называть точное предписание, определяющее последовательность действий, обеспечивающую получение требуемого результата из исходных данных. Алгоритм может быть предназначен для выполнения его человеком или автоматическим устройством. Создание алгоритма, пусть даже самого простого, - процесс творческий. Он доступен исключительно живым существам, а долгое время считалось, что только человеку. Другое дело – реализация уже имеющегося алгоритма. Ее можно поручить субъекту или объекту, который не обязан вникать в существо дела, а возможно, и не способен его понять. Такой субъект или объект принято называть формальным исполнителем.

Примером формального исполнителя может служить стиральная машина-автомат, которая неукоснительно исполняет предписанные ей действия, даже если вы забыли положить в нее порошок. Человек тоже может выступать в роли формального исполнителя, но в первую очередь формальными исполнителями являются различные автоматические устройства, и компьютер в том числе.

Каждый алгоритм создается в расчете на вполне конкретного исполнителя. Те действия, которые может совершать исполнитель, называются его допустимыми действиями. Совокупность допустимых действий образует систему команд исполнителя. Алгоритм должен содержать только те действия, которые допустимы для данного исполнителя.

1.2. Свойства алгоритмов

Данное выше определение алгоритма нельзя считать строгим - не вполне ясно, что такое «точное предписание» или «последовательность действий, обеспечивающая получение требуемого результата». Поэтому обычно формулируют несколько общих свойств алгоритмов, позволяющих отличать алгоритмы от других инструкций.

Такими свойствами являются:

- **Дискретность** (прерывность, раздельность) - алгоритм должен представлять процесс решения задачи как последовательное выполнение простых (или ранее определенных) шагов. Каждое действие, предусмотренное алгоритмом, исполняется только после того, как закончилось исполнение предыдущего.
- **Определенность** - каждое правило алгоритма должно быть четким, однозначным и не оставлять места для произвола. Благодаря этому свойству выполнение алгоритма носит механический характер и не требует никаких дополнительных указаний или сведений о решаемой задаче.
- **Результативность** (конечность) - алгоритм должен приводить к решению задачи за конечное число шагов.
- **Массовость** - алгоритм решения задачи разрабатывается в общем виде, то есть, он должен быть применим для некоторого класса задач, различающихся только исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется областью применимости алгоритма.

Правила выполнения арифметических операций или геометрических построений представляют собой алгоритмы. При этом остается без ответа вопрос, чем же отличается понятие алгоритма от таких понятий, как «метод», «способ», «правило». Можно даже встретить утверждение, что слова «алгоритм», «способ»,

«правило» выражают одно и то же (т.е. являются синонимами), хотя такое утверждение, очевидно, противоречит “свойствам алгоритма”.

Само выражение «свойства алгоритма» не совсем корректно. Свойствами обладают объективно существующие реальности. Можно говорить, например, о свойствах какого-либо вещества. Алгоритм – искусственная конструкция, которую мы сооружаем для достижения определенных целей. Чтобы алгоритм выполнил свое предназначение, его необходимо строить по определенным правилам. Поэтому нужно говорить все же не о свойствах алгоритма, а о правилах построения алгоритма, или о требованиях, предъявляемых к алгоритму.

Первое правило – при построении алгоритма, прежде всего, необходимо задать множество объектов, с которыми будет работать алгоритм. Формализованное (закодированное) представление этих объектов носит название данных. Алгоритм приступает к работе с некоторым набором данных, которые называются входными, и в результате своей работы выдает данные, которые называются выходными. Таким образом, алгоритм преобразует входные данные в выходные.

Это правило позволяет сразу отделить алгоритмы от “методов” и “способов”. Пока мы не имеем формализованных входных данных, мы не можем построить алгоритм.

Второе правило – для работы алгоритма требуется память. В памяти размещаются входные данные, с которыми алгоритм начинает работать, промежуточные данные и выходные данные, которые являются результатом работы алгоритма. Память является дискретной, т.е. состоящей из отдельных ячеек. Поименованная ячейка памяти носит название переменной. В теории алгоритмов размеры памяти не ограничиваются, т. е. считается, что мы можем предоставить алгоритму любой необходимый для работы объем памяти.

В школьной «теории алгоритмов» эти два правила не рассматриваются. В то же время практическая работа с алгоритмами (программирование) начинается именно с реализации этих правил. В языках программирования распределение памяти осуществляется декларативными операторами (операторами описания пе-

ременных). При запуске программы транслятор языка анализирует все идентификаторы в тексте программы и отводит память под соответствующие переменные.

Третье правило – дискретность. Алгоритм строится из отдельных шагов (действий, операций, команд). Множество шагов, из которых составлен алгоритм, конечно.

Четвертое правило – детерминированность. После каждого шага необходимо указывать, какой шаг выполняется следующим, либо давать команду остановки.

Пятое правило – сходимость (результативность). Алгоритм должен завершать работу после конечного числа шагов. При этом необходимо указать, что считать результатом работы алгоритма.

Итак, алгоритм – неопределяемое понятие теории алгоритмов. Алгоритм каждому определенному набору входных данных ставит в соответствие некоторый набор выходных данных, т.е. вычисляет (реализует) функцию. При рассмотрении конкретных вопросов в теории алгоритмов всегда имеется в виду какая-то конкретная модель алгоритма.

1.3. Виды алгоритмов и их реализация

Алгоритм применительно к вычислительной машине – точное предписание, т.е. набор операций и правил их чередования, при помощи которого, начиная с некоторых исходных данных, можно решить любую задачу фиксированного типа.

Виды алгоритмов как логико-математических средств отражают указанные компоненты человеческой деятельности и тенденции, а сами алгоритмы в зависимости от цели, начальных условий задачи, путей ее решения, определения действий исполнителя подразделяются следующим образом:

Механические алгоритмы, или иначе *детерминированные, жесткие* (например, алгоритм работы машины, двигателя и т.п.);

Гибкие алгоритмы, например *стохастические (вероятностные)* и *эвристические*.

Механический алгоритм задает определенные действия, обозначая их в *единственной и достоверной последовательности*, обеспечивая тем самым однозначный требуемый или искомый результат, если выполняются те условия процесса, задачи, для которых разработан алгоритм.

Вероятностный (стохастический) алгоритм дает программу решения задачи *несколькими путями или способами*, приводящими к *вероятному* достижению результата.

Эвристический алгоритм (от греческого слова “эврика”) – это такой алгоритм, в котором достижение конечного результата программы действий однозначно не предопределено, так как не обозначена вся последовательность действий, не выявлены все действия исполнителя. К эвристическим алгоритмам относят, например, инструкции и предписания. В этих алгоритмах используются универсальные логические процедуры и способы принятия решений, основанные на аналогиях, ассоциациях и прошлом опыте решения сходных задач.

Линейный алгоритм – набор команд (указаний), выполняемых последовательно во времени друг за другом.

Разветвляющийся алгоритм – алгоритм, содержащий хотя бы одно условие, в результате проверки которого ЭВМ обеспечивает переход на один из двух возможных шагов.

Циклический алгоритм – алгоритм, предусматривающий многократное повторение одного и того же действия (одних и тех же операций) над новыми исходными данными. К циклическим алгоритмам сводится большинство методов вычислений, перебора вариантов.

Цикл программы – последовательность команд (серия, тело цикла), которая может выполняться многократно (для новых исходных данных) до удовлетворения некоторого условия.

Вспомогательный (подчиненный) алгоритм (процедура) – алгоритм, ранее разработанный и целиком используемый при алгоритмизации конкретной задачи. В некоторых случаях при наличии одинаковых последовательностей указаний

(команд) для различных данных с целью сокращения записи также выделяют вспомогательный алгоритм.

1.4. Методы изображение алгоритмов

На практике наиболее распространены следующие формы представления алгоритмов:

- словесная (записи на естественном языке);
- графическая (изображения из графических символов);
- псевдокоды (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.);
- программная (тексты на языках программирования).

Словесное описание алгоритма

Данный способ получил значительно меньшее распространение из-за его многословности и отсутствия наглядности.

Рассмотрим пример на алгоритме нахождение максимального из двух значений:

Определим форматы переменных X, Y, M, где X и Y – значения для сравнения, M – переменная для хранения максимального значения;

получим два значения чисел X и Y для сравнения;

сравним X и Y.

если X меньше Y, значит большее число Y.

Поместим в переменную M значение Y.

Если X не меньше (больше) Y, значит большее число X.

Поместим в переменную M значение X.

Словесный способ не имеет широкого распространения по следующим причинам:

- такие описания строго не формализуемы;
- страдают многословностью записей;
- допускают неоднозначность толкования отдельных предписаний.

Блок-схема алгоритма

А этот способ оказался очень удобным средством изображения алгоритмов и получил широкое распространение в научной и учебной литературе.

Структурная (блок-, граф-) схема алгоритма – графическое изображение алгоритма в виде схемы связанных между собой с помощью стрелок (линий перехода) блоков – графических символов, каждый из которых соответствует одному шагу алгоритма. Внутри блока дается описание соответствующего действия.

Графическое изображение алгоритма широко используется перед программированием задачи вследствие его наглядности, т.к. зрительное восприятие обычно облегчает процесс написания программы, ее корректировки при возможных ошибках, осмысливание процесса обработки информации.

Можно встретить даже такое утверждение: «Внешне алгоритм представляет собой схему – набор прямоугольников и других символов, внутри которых записывается, что вычисляется, что вводится в машину и что выдается на печать и другие средства отображения информации». Здесь форма представления алгоритма смешивается с самим алгоритмом.

Принцип программирования «сверху вниз» требует, чтобы блок-схема поэтапно конкретизировалась и каждый блок «расписывался» до элементарных операций. Но такой подход можно осуществить при решении несложных задач. При решении сколько-нибудь серьезной задачи блок-схема «расползется» до такой степени, что ее невозможно будет охватить одним взглядом.

Блок-схемы алгоритмов удобно использовать для объяснения работы уже готового алгоритма, при этом в качестве блоков берутся действительно блоки ал-

горитма, работа которых не требует пояснений. Блок-схема алгоритма должна служить для упрощения изображения алгоритма, а не для усложнения.

Вспомним основные условные обозначения, используемые при графической записи алгоритма (рис 1.1)

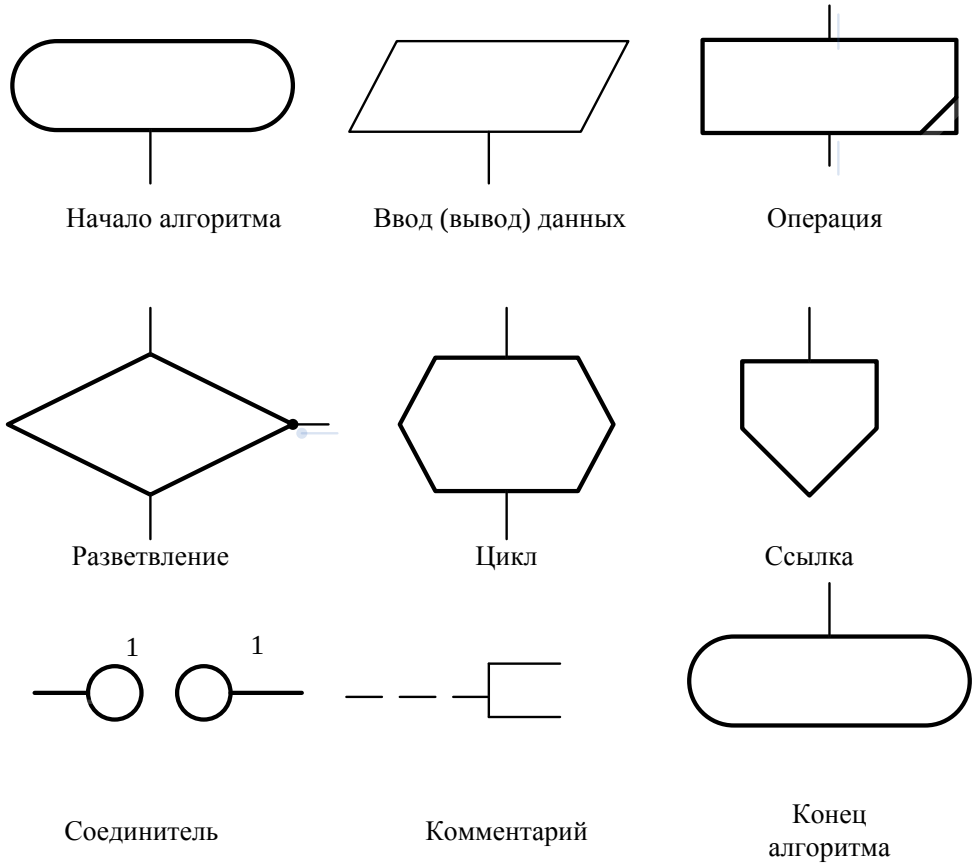


Рис. 1.1

Псевдокод

Псевдокод представляет собой систему обозначений и правил, предназначенную для единообразной записи алгоритмов. Он занимает промежуточное место между естественным и формальным языками.

С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

В псевдокоде не приняты строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя. Однако в псевдокоде обычно имеются некоторые конструкции, присущие формальным языкам, что облегчает переход от записи на псевдокоде к записи алгоритма на формальном языке. В частности, в псевдокоде, так же, как и в формальных языках, есть служебные слова, смысл которых определен раз и навсегда. Они выделяются в печатном тексте жирным шрифтом, а в рукописном тексте подчеркиваются. Единого или формального определения псевдокода не существует, поэтому возможны различные псевдокоды, отличающиеся набором служебных слов и основных (базовых) конструкций.

Программное представление алгоритма

При записи алгоритма в словесной форме, в виде блок-схемы или на псевдокоде допускается определенный произвол при изображении команд. Вместе с тем такая запись точна настолько, что позволяет человеку понять суть дела и исполнить алгоритм.

Однако на практике в качестве исполнителей алгоритмов используются специальные автоматы — компьютеры. Поэтому алгоритм, предназначенный для исполнения на компьютере, должен быть записан на «понятном» ему языке. И здесь на первый план выдвигается необходимость точной записи команд, не оставляющей места для произвольного толкования их исполнителем.

Следовательно, язык для записи алгоритмов должен быть формализован. Такой язык принято называть языком программирования, а запись алгоритма на этом языке — программой для компьютера.

1.5. Порядок разработки иерархической схемы реализации алгоритмов

К основным методам структурного программирования относится, прежде всего, отказ от бессистемного употребления оператора непосредственного перехода и преимущественное использование других структурированных операторов, методы нисходящего проектирования разработки программы, идеи пошаговой детализации и некоторые другие соглашения, касающиеся дисциплины программирования.

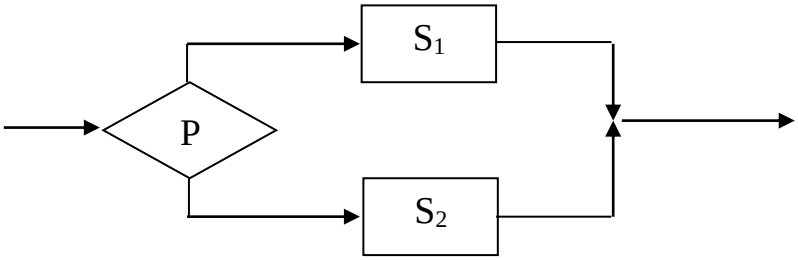
Всякая программа, в соответствии со структурным подходом к программированию, может быть построена только с использованием трех основных типов блоков.

1. Функциональный блок, который на блок-схеме изображается в виде прямоугольников с одним входом и одним выходом:

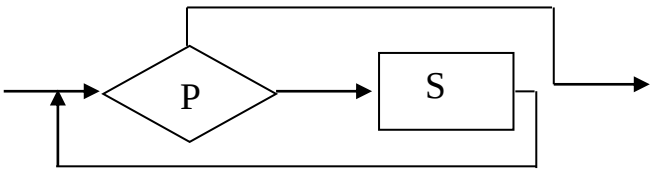
Функциональному блоку в языках программирования соответствуют операторы ввода и вывода или любой оператор присваивания.

В виде функционального блока может быть изображена любая последовательность операторов, выполняющихся один за другим, имеющая один вход и один выход.

2. Условная конструкция. Этот блок включает проверку некоторого логического условия (P), в зависимости от которого выполняется либо один (S1), либо другой (S2) операторы:



3. Блок обобщенного цикла. Этот блок обеспечивает многократное повторение выполнения оператора S пока выполнено логическое условие P:



При конструировании программы с использованием рассмотренных типов блоков эти блоки образуют линейную цепочку так, что выход одного блока подсоединяется ко входу следующего. Таким образом, программа имеет линейную структуру, причем порядок следования блоков соответствует порядку, в котором они выполняются.

Такая структура значительно облегчает чтение и понимание программы, а также упрощает доказательство ее правильности. Так как линейная цепочка блоков может быть сведена к одному блоку, то любая программа может, в конечном итоге, рассматриваться как единый функциональный блок с один входом и одним выходом.

При проектировании и написании программы нужно выполнить обратное преобразование, то есть этот блок разбить на последовательность подблоков, затем каждый подблок разбить на последовательность более мелких блоков до тех пор, пока не будут получены «атомарные» блоки, рассмотренных выше типов. Такой метод конструирования программы принято называть нисходящим («сверху вниз»).

При нисходящем методе конструирования алгоритма и программы первоначально рассматривается вся задача в целом. На каждом последующем этапе задача разбивается на более мелкие подзадачи, каждая подзадача, в конечном итоге на еще более мелкие подзадачи и так до тех пор, пока не будут получены такие подзадачи, которые легко кодируются на выбранном языке программирования. При этом на каждом шаге уточняются все новые и новые детали («пошаговая детализация»).

В процессе нисходящего проектирования сохраняется строгая дисциплина программирования, то есть разбиение на подзадачи осуществляется путем применения только рассмотренных типов конструкций (функциональный блок, условная конструкция, обобщенный цикл), поэтому, в конечном итоге, получается хорошо структурированная программа.

2. Классификация алгоритмов

По типу используемого вычислительного процесса различают линейные (прямые), разветвляющиеся и циклические алгоритмы.

Линейные алгоритмы описывают линейный вычислительный процесс, этапы которого выполняются однократно и последовательно один за другим. Он включает последовательное выполнение следующих этапов:

- ввод исходных данных в память ЭВМ;
- вычисление искомых величин по формулам;
- вывод результатов из памяти ЭВМ на информационный носитель.

Пример 1. Составить алгоритм вычисления площади круга по формуле $S = \pi R^2$. Решение показано на рис. 2.1.

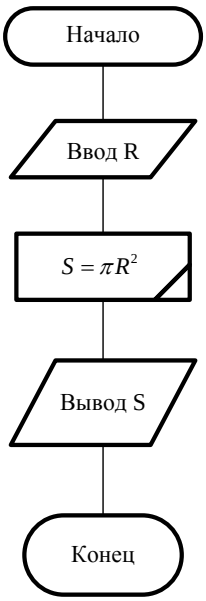


Рис. 2.1

Разветвляющийся алгоритм описывает вычислительный процесс, реализация которого происходит по одному из нескольких заранее предусмотренных направлений. Направления, по которым может следовать вычислительный процесс, называются ветвями. Выбор конкретной ветви вычисления зависит от результатов проверки выполнения некоторого логического условия. Результатами проверки являются: "истина" (да), если условие выполняется, и "ложь" (нет), при невыполнении условия.

Пример 2. Составить алгоритм решения для функции F(x) = 1 при x > 0 и F(x) = 0 при x < 0. Блок - схема разветвляющегося алгоритма показана на рис. 2.2.

Циклический алгоритм описывает вычислительный процесс, этапы которого повторяются многократно. Различают простые циклы, не содержащие внутри себя других циклов, и сложные (вложенные), содержащие несколько циклов. В зависимости от ограничения числа повторений выделяют циклы с известным числом повторений и циклы, число повторений которых заранее неизвестно.

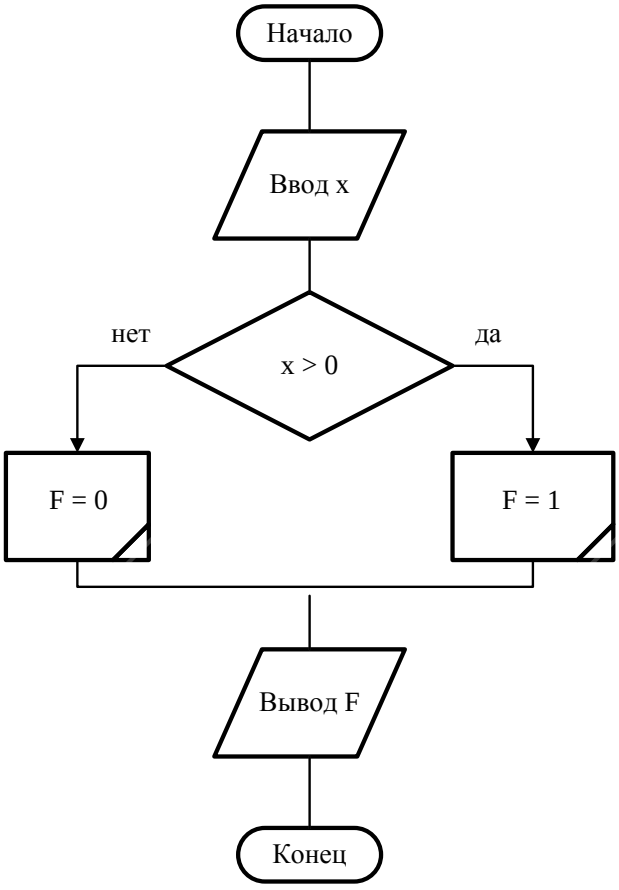


Рис. 2.2

2.1. Циклы с известным числом повторений

При организации этих циклов присутствуют стандартные элементы, сопровождающие любой цикл:

- подготовка первого выполнения цикла (присвоение счетчику цикла начального значения);
- тела цикла, которое образуют блоки , выполняемые многократно;
- изменение значения счетчика циклов и сравнение его с конечным значением.

Блок-схемы циклических алгоритмов существенно отличаются структурами повторения "повторять ДО "(повторять до выполнения условия окончания цикла) или "повторять ПОКА " (повторять пока выполняются условия продолжения циклического процесса). В первом варианте проверка условий окончания циклических вычислений осуществляется в конце цикла (рис. 2.3, а), а во втором - в начале цикла (рис. 2.3, б). Как видно из рисунка, цикл "повторять ДО "выполняется, по крайней мере, один раз, а цикл "повторять ПОКА"может сразу привести к выходу из цикла.

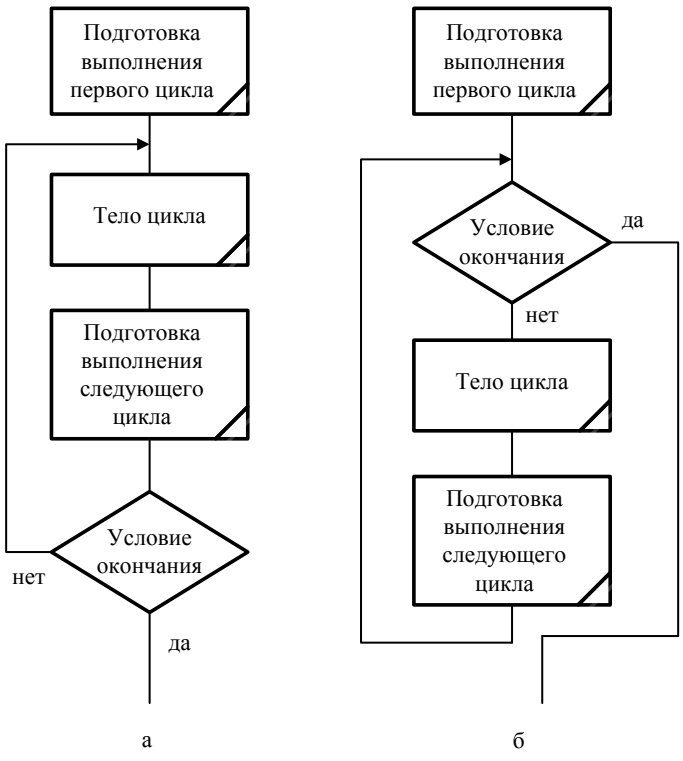
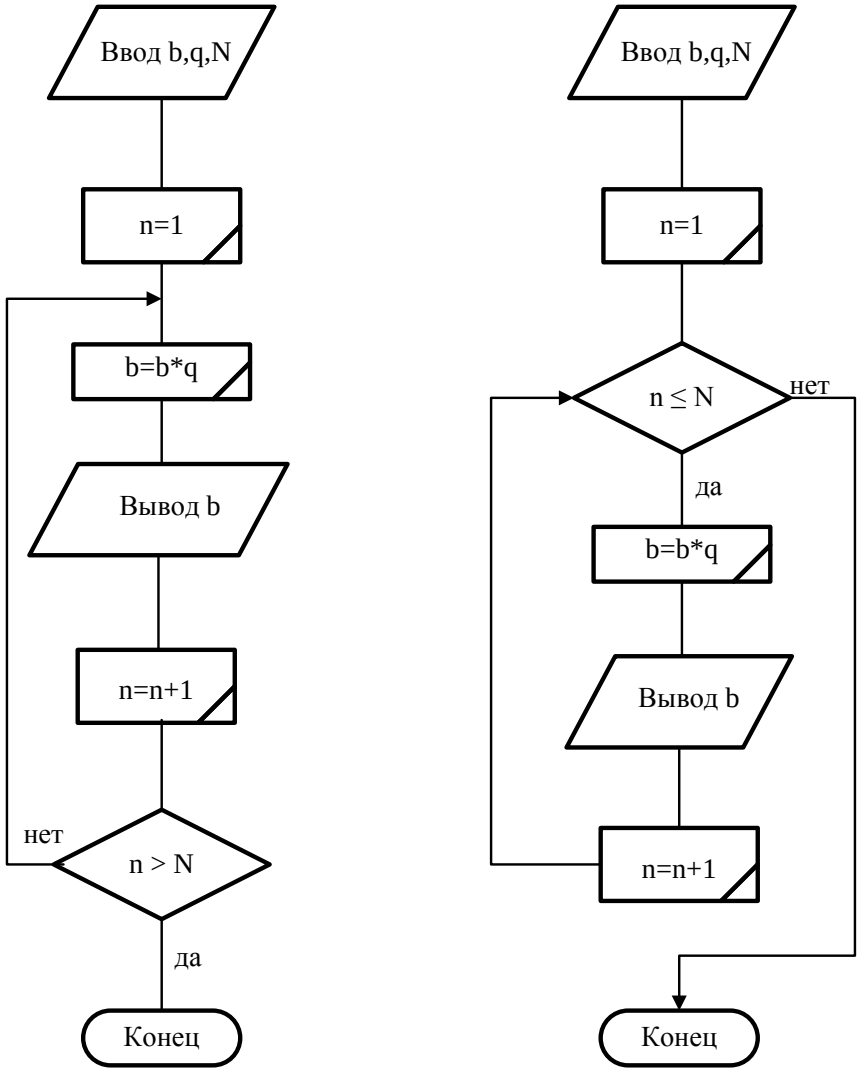


Рис. 2.3

Пример 3. Составить алгоритм решения задачи вычисления N первых членов геометрической прогрессии, используя формулу $b_{n+1}=b_n*q$ для любых b и q, где n - текущий член геометрической прогрессии.

Блок-схема алгоритма решения данного примера показана в двух вариантах: с использованием цикла "ДО" (рис. 2.4, а) и цикла "ПОКА"(рис. 2.4, б).



а б

Рис. 2.4

2.2. Циклы с неизвестным числом повторений

Примером циклов, число повторений которых не задано, являются итерационные вычислительные процессы. В них решение задачи реализуется путем последовательного приближения к искомому результату. Процесс является циклическим, поскольку заключается в многократных вычислениях. Начальное приближение Y0 выбирается заранее или задается по определенным правилам. Заканчивается итерационное вычисление при выполнении условия $|Y_i - Y_{i-1}| < d$,

где d - допустимая ошибка вычисления.

Типовая структура алгоритма итерационных вычислений имеет вид, показанный на рис. 2.5.

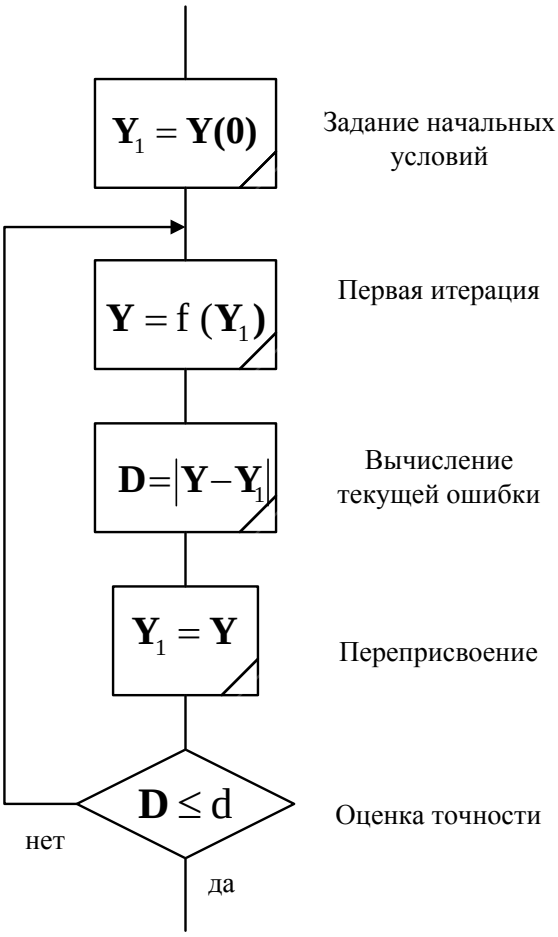


Рис. 2.5

Пример 4. Составить алгоритм вычисления функции $y=\sqrt{x}$ с точностью d, используя рекуррентную формулу $y_{i+1}=0.5*\left(x/y_i+y_i\right)$.

Если начальное приближение $y_1 = x$, тогда на первом цикле вычисления будем иметь $y=0.5*\left(x/y_1+y_1\right)$ Блок - схема алгоритма решения примера 4 приведена на рис. 2.6.

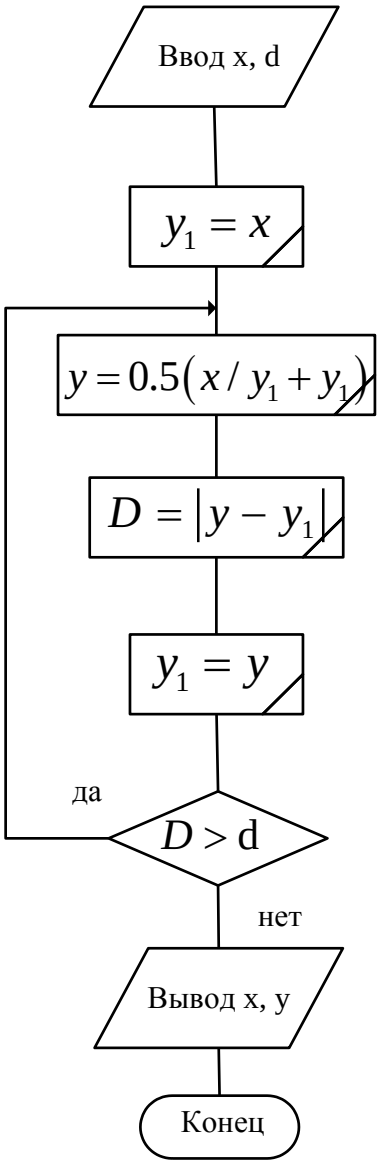


Рис. 2.6

2.3. Сложные циклы

Вычислительные процессы, содержащие два или более включенных друг в друга циклов, называют сложными циклическими процессами (алгоритмами). Цикл, который содержит внутри себя другой цикл, называют внешним в противоположность внутреннему (вложенному). Необходимо учитывать, что за одно исполнение внешнего цикла внутренний цикл повторяется многократно.

Пример 5. Составить алгоритм вычисления и вывод на печать функции y = x*z / (A + B) при изменении аргументов 1< x < 10 с шагом Δx = 2 и 1< z <4 с шагом Δz = 0.5.

Решение данного примера показано на рис. 2.7.

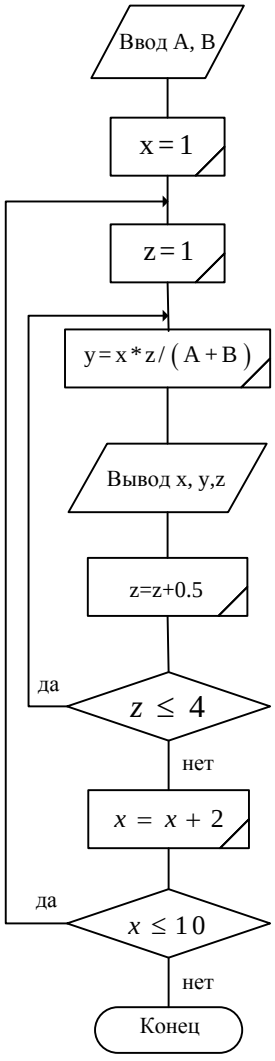


Рис. 2.7

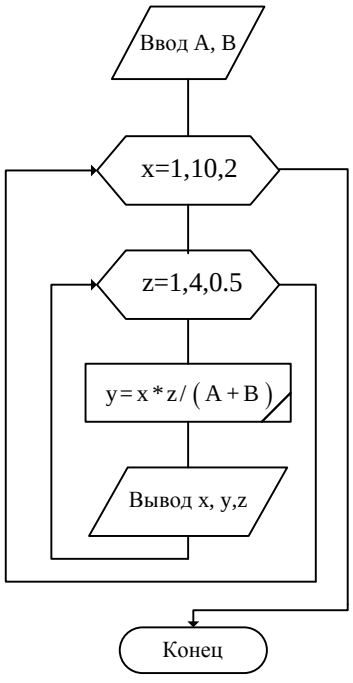


Рис. 2.8

Внутренний цикл организован по переменной z , а внешний - по переменной x . На каждом шаге изменения переменной x (переменной внешнего цикла) переменная z (переменная внутреннего цикла) проходит весь заданный диапазон изменения от 1 до 4 с шагом 0.5.

Блок вывода на печать помещен во внутреннем цикле, что позволяет регистрировать переменные во всем диапазоне их изменения. На рис. 2.8 эта же задача решена с помощью модифицированной блок-схемы алгоритма. В ней циклы представлены с помощью более компактных условных обозначений, принципы организации которых становятся ясными из рис. 2.9. Первая цифра внутри фигуры (рис. 2.9) определяет начальное значение переменной, вторая ее конечное значение, а третья - шаг изменения переменной. По умолчанию (при отсутствии последней цифры) шаг изменения переменной принимается равным 1.

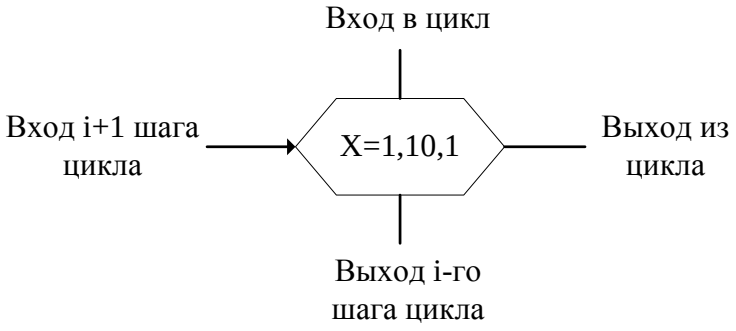


Рис. 2.9

2.4. Алгоритмы с массивами

Наряду с одиночными переменными в задачах часто используются организованные переменные - массивы. Под массивом понимают совокупность данных, имеющих общее имя. По способу организации чаще всего различают одномерные и двумерные массивы. Одномерный массив - вектор, элементы которого снабжаются одним индексом, определяющим их порядковый номер в массиве. Двумерный массив - это матрица. Элементы одного массива снабжаются двумя индексами, первый из которых определяет номер строки, а второй - номер столбца. На

пересечении номера строки и номера столбца расположен искомый элемент. Например, в массиве

$$A(2,2)=\begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix}$$

элемент a_{11} расположен на пересечении первой строки и первого столбца, элемент a_{12} - на пересечении первой строки и второго столбца, элемент a_{21} - на пересечении второй строки и первого столбца и т. д. Элементы массива называются переменными с индексами или индексными переменными.

Пример 6. Составить алгоритм вычисления элементов массива Y(10) по элементам массива X(10), если $y_i = \dfrac{x_i + a}{\sqrt{x_i^2 + 1}}$.

Алгоритм решения данной задачи приведен на рис. 2.10. В нем показано, что после блоков ввода переменной a и массива X(10) организуется цикл по индексной переменной i для вычисления элементов массива y; по элементам массива Xi.

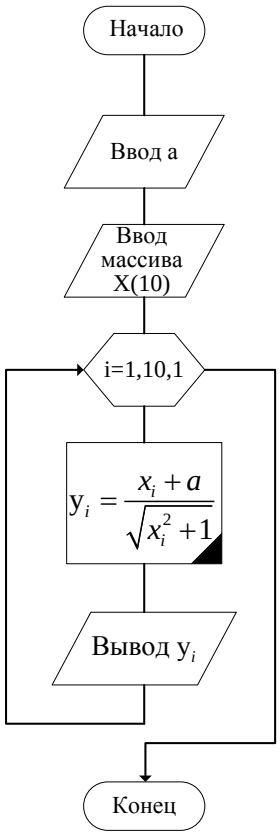


Рис. 2.10

Пример 7. Составить алгоритм получения элементов массива Y(5,4) по элементам массива X(5,4), если

$$y_{i,j} = \frac{x_{i,j} + a}{\sqrt{x_{i,j}^2 + 1}}$$

Алгоритм решения задачи из примера 7 приведен на рис. 2.11. Он отличается от предыдущего алгоритма наличием второго цикла по переменной j, так как адрес каждого элемента двумерного массива определяется двумя индексами.

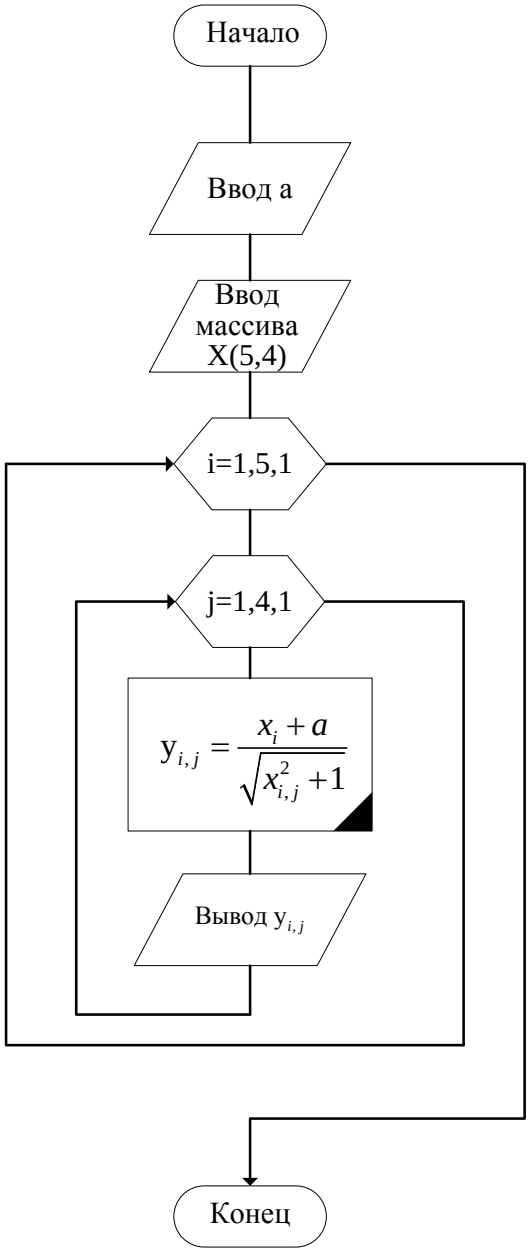


Рис. 2.11

Пример 8. Дан массив ненулевых элементов A(10). Необходимо определить количество положительных и отрицательных элементов.

Пусть P - количество положительных элементов, а Q - количество отрицательных элементов, тогда алгоритм решения задачи будет иметь вид, показанный на рис. 2.12.

Принцип работы данного алгоритма ясен из блок-схемы, однако надо иметь в виду, что перед входом в цикл необходимо выполнить обнуление переменных P и Q. В этих переменных формируются суммы по количеству положительных и отрицательных элементов, которые должны быть равны нулю перед входом в цикл.

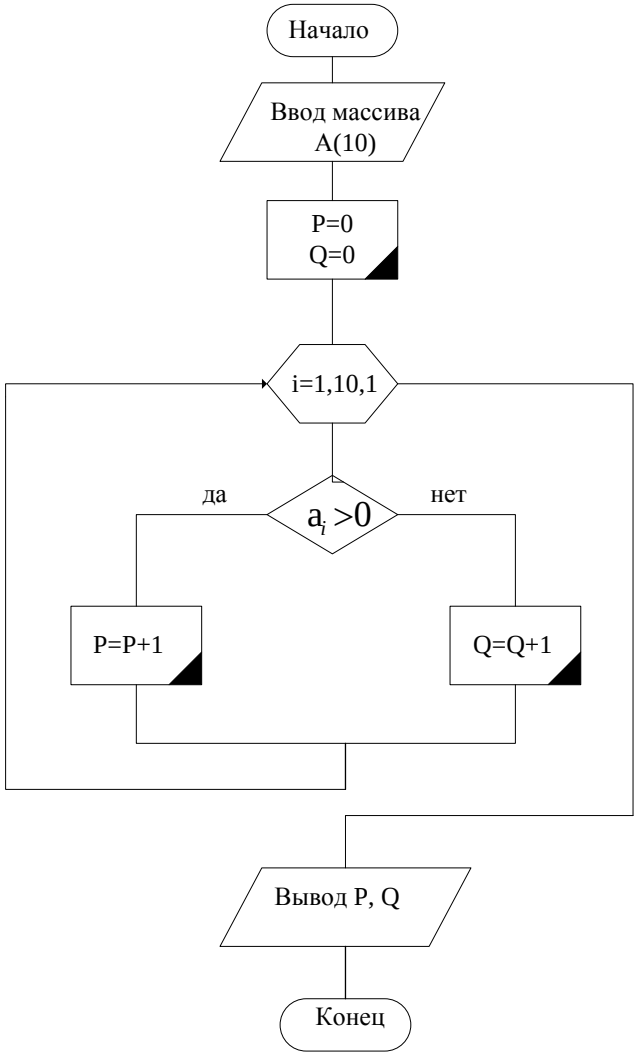


Рис. 2.12

Пример 9. Задан массив ненулевых элементов B(2,3). Определить количество положительных и отрицательных элементов.

В примере 9 рассматривается двумерный массив, элементы которого являются переменными с двумя индексами. Поэтому для перебора всех элементов массива в блок-схеме алгоритма на рис. 2.13 необходимо изменять два параметра: i - номер строки; j - номер столбца.

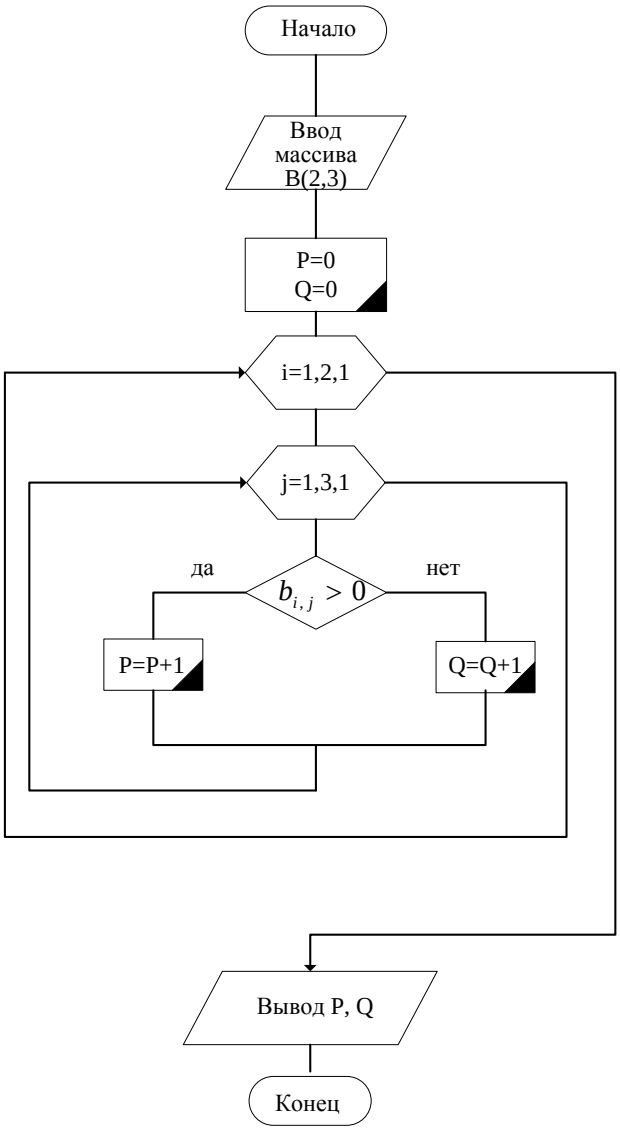


Рис. 2.13

2.5. Алгоритмы вычисления степенных полиномов

Вычисление степенных полиномов вида $Y=a_1x^n+a_2x^{n-1}+\cdots+a_nx+a_{n+1}$ наиболее целесообразно проводить по схеме Горнера

$$Y=\left(.\left(\left(a_1x+a_2\right)x+a_3\right)x+\cdots a_n\right)x+a_{n+1}.$$

Такая запись полинома существенно сокращает время вычислений, так как наиболее трудоемкая операция, связанная с возведением в степень переменной x в данном случае заменяется расчетом по рекуррентной формуле $Y=a_kx+a_{k+1}$, где k = 1,2,...,n. Коэффициенты полинома сводятся в массив, включающий (n+1) элемент. Начальное значение переменной Y, задаваемое перед циклом, должно быть равно коэффициенту a_1 при x в старшей степени, а параметр цикла должен изменяться от 2 до n+1.

Пример 10. Составить алгоритм вычисления полинома степени n при заданных значениях массива коэффициентов полинома A(n+1).Блок-схема алгоритма приведена на рис. 2.14.

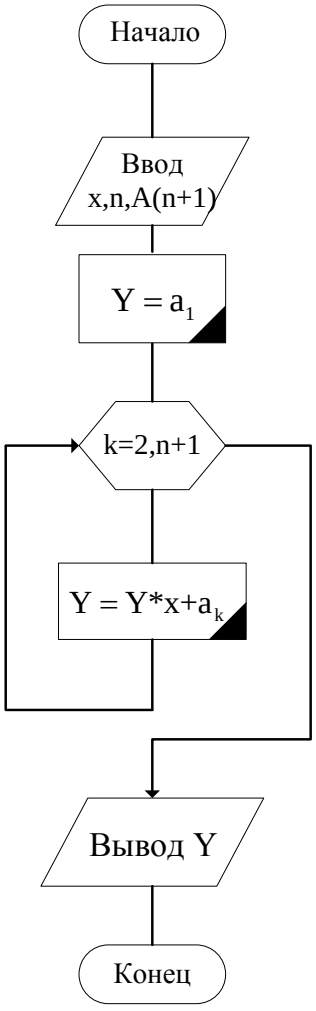


Рис. 2.14

2.6. Алгоритмы нахождения наибольшего (наименьшего) из множества значений

Реализация этих алгоритмов осуществляется в цикле путём сравнения некоторого текущего значения с наибольшим из всех предыдущих. При этом если текущее значение больше наибольшего из всех предыдущих, то наибольшему присваивается значение текущего. В противном случае наибольшее сохраняет прежнее значение.

При первом выполнении цикла необходимо в качестве начального значения Y_{max} взять заведомо небольшое число, например (-10^{30}). Тогда после первого выполнения Y_{max} примет значение Y_1, так как Y_1 наверняка больше (-10^{30}). При втором выполнении цикла Y_{max} сравнивается с Y_2 и находится большее из Y_1 и Y_2 и так далее.

Аналогичным образом находится наименьшее среди набора элементов. Начальное значение переменной в этом случае можно принимать $Y_{min} = 10^{30}$.

Пример 11. Найти наибольшее значение функции $Y=abs(a)e^{a*x-x/2}$ при изменении аргумента x от 0 до a с шагом h. Блок-схема алгоритма решения данного примера приведена на рис. 2.15. Значения коэффициента a и шага h вводятся в режиме диалога. После присвоения переменной Y_{max} минимального значения организуется цикл по переменной x, для каждого значения которой вычисляется функция Y. На каждом шаге вычисления осуществляется сравнение значений функций Y и Y_{max}.

Если $Y > Y_{max}$, то Y_{max} присваивается значение Y, в противном случае значение Y_{max} остается неизменным. По завершении цикла на печать выводится максимальное значение функции Y_{max}.

Пример12. Найти наименьший элемент массива $(x_1,x_2,\cdots,x_{10})$ и его порядковый номер. Особенностью решения является то, что необходимо найти не только минимальный по значению элемент, но и его порядковый номер. Для этого следует всякий раз, когда в цикле выполняется условие $x_i<x_{min}$, присваивать не

только $x_{min} = x_i$, но и $i_{min} = i$. В качестве начального значения x_{min} целесообразно задать значения x_1, а цикл выполнить при изменении i от 2 до 10.

Блок-схема алгоритма решения данного примера приведена на рис. 2.16. По окончании цикла значения x_{min} и i_{min} выводятся на печать.

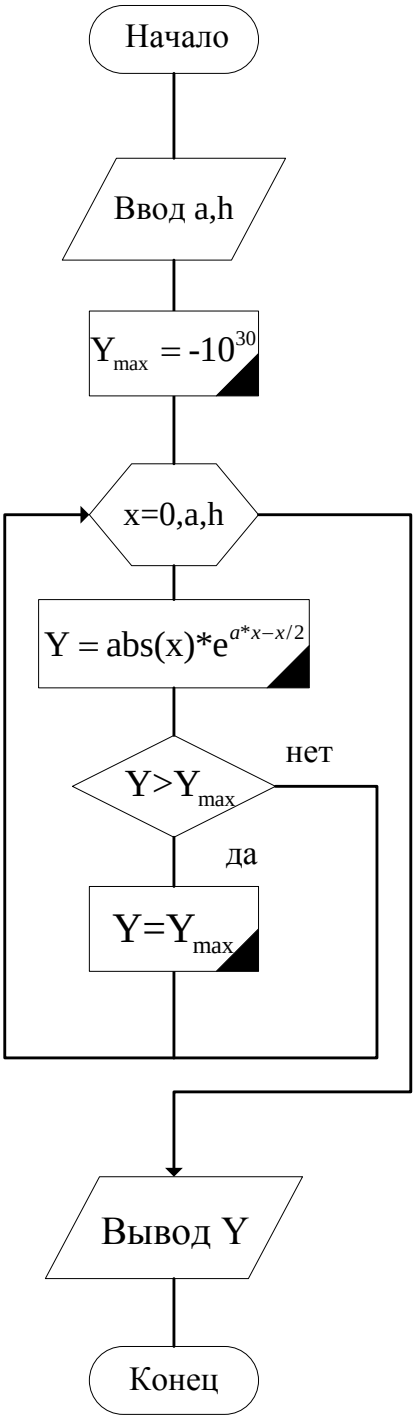


Рис. 2.15

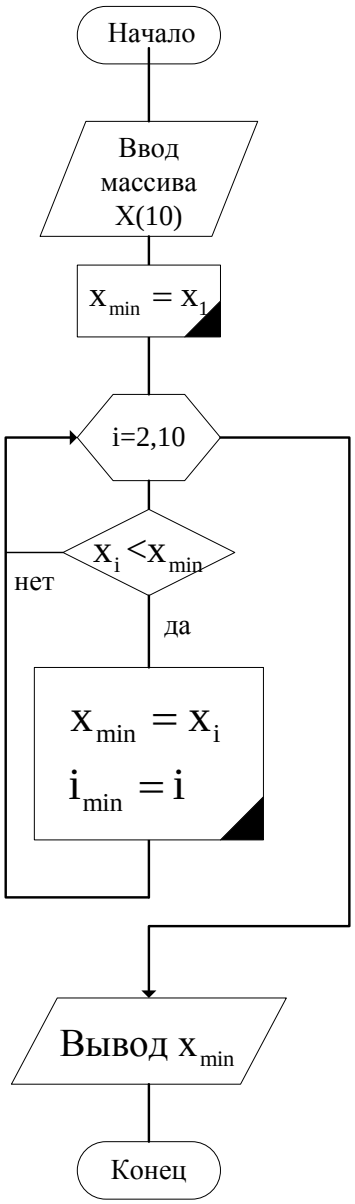


Рис. 2.16

Пример 13. Упорядочить элементы массива (a_1 , a_2 , .. .,a_{10}), расположив их по убыванию в том же массиве.

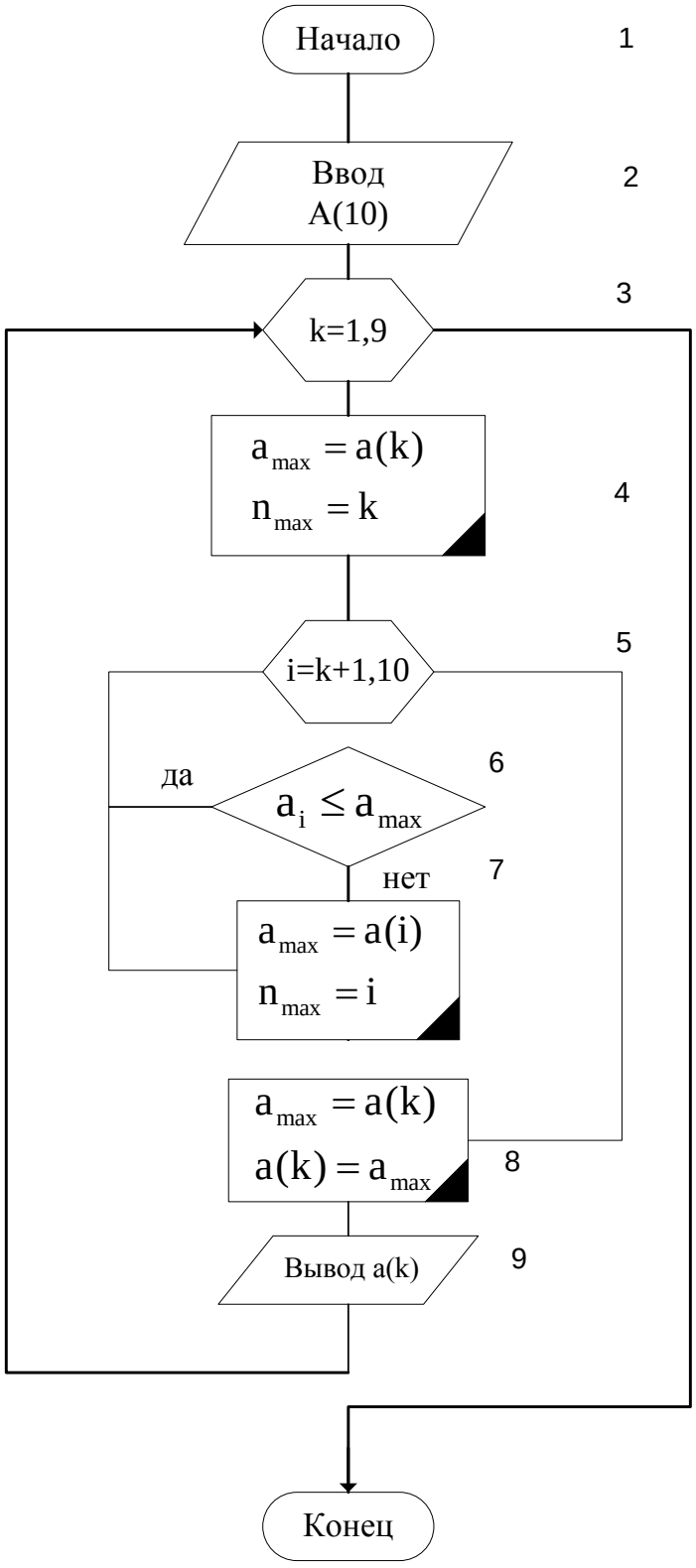


Рис. 2.17

Для решения этой задачи необходимо организовать цикл, в котором осуществить поиск наибольшего элемента массива и его порядкового номера. После окончания этого цикла следует поменять местами наибольший элемент с первым элементом. Если этот процесс повторить во внешнем цикле 9 раз, начиная с первого элемента массива, затем со второго и т. д., то все элементы массива будут упорядочены по убыванию.

Блок-схема алгоритма решения данного примера приведена на рис. 2.17, в котором основные элементы схемы обозначены цифрами. В блоке 2 осуществляется ввод массива A(10). В блоке 3 организуется внешний цикл по параметру k. Блок 4 задает начальное значение наименьшего элемента массива и его номер. Блок 5 организует внутренний цикл, в котором находится наибольший элемент массива и его порядковый номер (блоки 6 и 7). Блок 8 осуществляет перестановку наибольшего и k элемента массива.

Раздел II. Элементы программирования на языке C++

3. Из истории развития языка C++

В начале 1970-х годов Деннис Ритчи (Dennis Ritchie) вместе с программистом Кеннетом Томпсоном (Kenneth Thompson) разработали язык программирования Си. Представленный в 1973 году, Си широко применяется до сих пор и оказал влияние на многие более современные языки. Ритчи как создатель Си также внес большой вклад в создание операционной системы UNIX.

Язык C++ был разработан Бьерном Страуструпом в начале 80-х годов, как дальнейшее развитие языка Си. Язык C++ обладает рядом свойств, которые делают его более совершенным языком по сравнению с Си, однако наиболее важным является то, что он обеспечивает возможность объектно-ориентированного программирования.

Объектно-ориентированное программирование существенно повышает производительность разработчиков по сравнению с традиционными методами. Объектно-ориентированные программы легче понимать, корректировать и модифицировать. C++ - это гибридный язык, он предоставляет возможность программировать и в стиле Си, и в объектно-ориентированном стиле, и в обоих стилях сразу [1].

4. Структура программы на языке C++

Структуру программы на языке C++ рассмотрим на примере простой программы, печатающей строку текста. Листинг, характеризующий структуру построения простейшей программы на языке C++, приведен на рис. 4.1.

```
// Моя первая программа
# include < iostream .h >
main ( )
{
cout << " Это моя первая программа на языке C++";
```

```
return 0;
}
```

Рис. 4.1

Первая строка данной программы начинается с символа //, показывающего, что следующий за ним текст является комментарием, который компилятор игнорирует. Комментарии вставляются для документирования программы и облегчения ее чтения. Они помогают другим людям читать и понимать вашу программу. Комментарий, начинающийся с символа //, называется однострочным, потому что он должен заканчиваться в конце текущей строки. При использовании многострочных комментариев целесообразно применять символы /* и */. Все, что помещено между ними компилятор игнорирует. Строка # include <iostream.h> является директивой препроцессора.

Препроцессор - это специальная программа, которая обрабатывает строки программы, начинающиеся со знака #. Данная строка дает указание препроцессору перед компиляцией программы включить в нее информацию, содержащуюся в файле iostream.h. Следом идет обязательная функция main (), а круглые скобки прямо указывают на то, что main - имя функции. Открывающая фигурная скобка отмечает начало последовательности операторов, образующих тело функции.

Строка cout << " "; - оператор вывода, с помощью которого выводится на экран дисплея фраза, заключенная в кавычки. Функция может возвращать значение в программу с помощью оператора возврата (return). Этот оператор также означает выход из функции. Если же указанный оператор отсутствует, то функция автоматически возвращает значение типа void (пустой). Закрывающая фигурная скобка отмечает конец последовательности операторов, образующих тело функции. На этой скобке выполнение функции и программы завершается.

Программа на C++ состоит из одной или более функций, причем только одна из них обязательно должна называться main().

Функция - это блок программы, который выполняет одно или несколько действий. Описание функции состоит из заголовка и тела (см. рис.4.2). Круглые

скобки являются частью имени функции, и ставить их надо обязательно, так как именно они указывают компилятору, что имеется в виду функция, а не просто английское слово main. Фактически каждая функция включает в свое имя круглые скобки, но в большинстве случаев в них содержится некая информация, передаваемая функции. Если же информация не передается, то в фигурных скобках можно указать ключевое слово void (пустой). Перед именем функции указывается ключевое слово, соответствующее типу возвращаемого функцией значения. Если значение не возвращается, то также можно указать ключевое слово void.

```
void main ()                                  Заголовок функции
{
int num;
num=1;                                        Тело функции
cout <<"хорошее число="<< num
}
```

Рис. 4.2

Заголовок функции состоит из имени функции, а тело функции заключено в фигурные скобки и представляет собой набор операторов, каждый из которых оканчивается символом «**;**». Оператор описания int num определяет num как переменную целого типа (integer). Любая переменная в языке C++ должна быть описана раньше, чем она будет использована.

В C++ используются правила, регулирующие употребление прописных и строчных букв [2]. Команды и стандартные имена функций (т.е. имена функций языка C++) всегда пишутся строчными буквами. Заглавные буквы в языке C++ обычно используются для задания имен констант. В именах своих функций и переменных Вы можете использовать как заглавные, так и строчные буквы. Однако следует помнить, что язык C++ различает использование прописных и строчных букв. Например, если Вы определите в своей программе переменные name, Name, NAME, то для компилятора это три различные переменные. В работе [3] даются следующие рекомендации относительно использования прописных и строчных

букв в идентификаторах. Так, в именах переменных целесообразно использовать строчные буквы (нижний регистр), а прописные буквы (верхний регистр) использовать для обозначения констант, макросов и т.д.

После того как компьютер заканчивает выполнение инструкций, заданных в вашей программе, программа завершается, и компьютер возвращается в исходное состояние (в то состояние, которое было перед запуском программы). Возврат в исходную среду в случаях, когда функция не возвращает значения, как правило, осуществляется автоматически. Исключение составляют отдельные компиляторы языка C++, которые требуют, чтобы Вы явно указали возврат. Для таких компиляторов вводится инструкция return 0;, которую помещают непосредственно перед фигурной скобкой, завершающей тело функции main(). Если функция возвращает значение, то тело функции должно содержать как минимум один оператор return следующего формата:

return выражение ; ,

где выражение определяет значение, возвращаемое данной функцией.

5. Ввод и вывод в C++

Ввод-вывод в языке C++ осуществляется потоками байтов [4]. Поток - это просто последовательность байтов. В операциях ввода байты пересылаются от устройства ввода (например, клавиатуры, дисковода или соединений сети) в оперативную память. При выводе байты пересылаются из оперативной памяти на устройства (например, экран дисплея, принтер или дисковод). Язык C++ предоставляет возможности для ввода-вывода как на низком, так и на высоком уровнях. Ввод-вывод на низком уровне обычно сводится к тому, что некоторое число байтов данных следует переслать от устройства в память или из памяти в устройство. При такой пересылке каждый байт является самостоятельным элементом данных. Передача на низком уровне позволяет осуществлять пересылку больших по объему потоков ввода-вывода с высокой скоростью, но такая передача обычно оказы-

вается неудобной для программиста и пользователя. Операции ввода-вывода на высоком уровне осуществляются путем преобразования байтов в такие значащие элементы данных, как целые числа, числа с плавающей запятой, символы, строки и т. д. Стандартные библиотеки C++ имеют расширенный набор средств ввода-вывода, при этом большая часть программ включает заголовочный файл <iostream.h>, который содержит основные сведения, необходимые для всех операций с потоками ввода-вывода. Так, например, он включает объекты cin, cout, cerr, clog, которые соответствуют стандартным потокам ввода-вывода и стандартным потокам вывода сообщений об ошибках. Объект стандартного потока ввода cin связан со стандартным устройством ввода, обычно с клавиатурой. Операция взять из потока (cin - the standard input stream - стандартный поток ввода), показанная в приведенном ниже операторе, означает, что величина переменной x должна быть введена из объекта cin в память cin >> x ;. Объект стандартного потока вывода cout связан со стандартным устройством вывода, обычно с экраном дисплея. Операция поместить в поток (cout - standard output stream - стандартный поток вывода), показанная в приведенном ниже операторе, означает, что величина переменной x должна быть выведена из памяти на стандартное устройство вывода cout << x;.

Объекты cerr и clog связаны со стандартным устройством вывода сообщений об ошибках. Их различие состоит в том, что при использовании cerr сообщение об ошибках выводится мгновенно, тогда как в случае применения объекта clog сообщения об ошибках помещаются в буфер, где они хранятся до тех пор, пока буфер полностью не заполнится или пока содержимое буфера не будет выведено принудительно.

Рассмотрим примеры практической реализации операции ввода-вывода. В программе на рис. 5.1 показан вывод строки, использующий одну операцию поместить в поток. Пример многократного использования операции «поместить в поток» приведен на рис. 5.2. Выполнение этой программы дает те же результаты , что и в примере на рис. 5.1.

```
// Вывод строки
# include <iostream.h>
main( )
{
 cout << "Добро пожаловать в мир C++ ! \n";
 return 0 ;
}
```

Результаты выполнения программы:

Добро пожаловать в мир C++ !

// Вывод строки с помощью двух операций поместить в поток

```
# include <iostream.h>
main( )
 {
 cout << "Добро пожаловать в" ;
 cout << " мир C++ ! \n";
 return 0 ;
 }
```

Результаты выполнения программы:

Добро пожаловать в мир C++ !

Рис. 5.2

Переход на новую строку в этих программах осуществляется с помощью управляющей последовательности \n. Эту же операцию можно осуществить и с помощью манипулятора потока endl (end line - конец строки), как показано на рис. 5.3.

```
// Вывод строки с использованием манипулятора потока endl
# include <iostream.h>
main( )
{
 cout << "Добро пожаловать в мир C++ ! "<<endl;
 return 0 ;
}
```

Результаты выполнения программы:

Добро пожаловать в мир C++ !

Рис. 5.3

Пример вывода значений выражений показан в программе на рис. 5.4.

```
// Вывод выражений
# include <iostream.h>
main( )
{
 cout << "47 плюс 53 равняется" ;
 cout << ( 47 + 53 ) ;        //выражение
 cout << endl ;
 return 0 ;
}
```

Результаты выполнения программы:

47 плюс 53 равняется 100

Рис. 5.4

На рис. 5.5 приведена программа, в которой последняя задача решается с помощью одного выражения, использующего способ сцепления операций.

```
// Вывод выражений путем сцепления операций
# include <iostream.h>
main( )
{
 cout << "47 плюс 53 равняется"<< ( 47 + 53 ) << endl ;
 return 0 ;
}
```

Результаты выполнения программы:

47 плюс 53 равняется 100

Рис. 5.5

На рис.5.6 приведена программа, в которой осуществляется вычисление суммы двух целых чисел, вводимых с клавиатуры при помощи объекта cin и операции взятия из потока >>.

```
// Вычисление суммы двух чисел в режиме диалога
# include <iostream.h>
main( )
 {
```

```
int x, y ;
cout << "Введите два целых числа :" ;
cin >> x >> y ;
cout << "Сумма чисел " << x ,<< "" <<y <<"равна"
      << (x + y) << endl ;
return 0 ;
}
```

Результаты выполнения программы:

Введите два целых числа: 30 92
Сумма чисел 30 и 92 равна 122

Рис. 5.6

На рис. 5.7 приведен пример программы вывода данных различного типа [2].

```
// Вывод данных различного типа
#include <iostream.h>
main()
 {
 int i, j;                  // объявление переменных
 float f;
 i = 10;                    // присваивание переменным
 j = 20;                    // значений
 f = 99.101
 cout << "Вот несколько чисел: ";   // операторы вывода
 cout << i << ' ' << j << ' ' << f;
 return 0;
 }                                   // конец функции
```

Результат работы программы:

Вот несколько чисел: 10 20 99.101

Рис. 5.7

Здесь в строке cout << i << ' ' << j << ' ' << f; выводится несколько элементов, данных в одном выражении. В общем случае можно использовать единственную инструкцию для вывода любого требуемого количества элементов данных. Обратите внимание, что по мере необходимости следует включить в программу пробе-

лы между элементами данных. При их отсутствии выводимые на экран данные будет неудобно читать.

На рис. 5.8 приведен пример ввода значения целого числа пользователем.

```
// Ввод данных в режиме диалога
#include <iostream.h>
main()
{                              // начало функции
 int i;                        // объявление целой переменной
 cout << "Введите целое число: ";
 cin >> i;
 cout << "Вот Ваше число: " << i << "\n";
 return 0;
}                              // конец функции
```

Результат работы программы:

Введите число: 100

Вот Ваше число: 100

Рис. 5.8

6. Основные элементы языка C++

Под элементами языка понимают его базовые конструкции, используемые при написании программ [2]. К ним относятся: алфавит, правила записи констант и идентификаторов, основные типы данных и действия над ними.

6.1. Алфавит

Алфавитом называют совокупность символов, используемых в языке. В C++ они образуют буквы, цифры и специальные символы. В качестве букв используются прописные буквы латинского алфавита от A до Z и строчные от a до z, а также знак подчеркивания {_}. В качестве десятичных цифр используются арабские цифры от 0 до 9. Специальные символы в языке C++ применяются для

различных целей: от организации текста программы до определения указаний компилятору языка C++. Специальные символы перечислены в табл. 6.1.

Таблица 6.1

Символ	Наименование	Символ	Наименование	Символ	Наименование
,	Запятая	<	Меньше	%	Процент
.	Точка	>	Больше	&	Амперсант
;	Точка с запятой	[	Левая квадратная скобка	^	Крышка
:	Двоеточие	]	Правая квадратная скобка	-	Минус
?	Знак вопроса	!	Восклицательный знак	=	Знак равенства
‘	Одиночная кавычка (апостроф)	|	Вертикальная черта	+	Плюс
(	Левая круглая скобка	/	Наклонная черта вправо (прямой слеш)	*	Звездочка
)	Правая круглая скобка	\	Наклонная черта влево (обратный слеш)	“	Двойная кавычка
{	Левая фигурная скобка	~	Тильда		
}	Правая фигурная скобка	#	Знак номера		

Из символов алфавита формируются лексемы языка [1]:

- идентификаторы;
- знаки операций;
- константы;
- разделители.

6.2. Идентификаторы

Идентификаторы используются как имена переменных, функций и типов данных. Они записываются по следующим правилам.

1. Идентификаторы начинаются с буквы (знак подчеркивания также является буквой).
2. Идентификатор может состоять из латинских букв и цифр (пробелы, точки и другие специальные символы при написании идентификаторов недопустимы).

3. Между двумя идентификаторами должен быть, по крайней мере, хотя бы один пробел.
4. Идентификатор может быть произвольной длины, но значимыми являются только первые 32 символа в среде WINDOWS и 8 символов в среде DOS; остальные символы игнорируются.

Правильные идентификаторы	Неправильные идентификаторы
wiggly	**$A^****
cat	**don't**
HOT_key	**HOT-key**
_grab1	**1grab**

При написании идентификаторов можно использовать как прописные, так и строчные буквы. В отличие от других языков программирования, компилятор языка C++ различает их в записи идентификатора. Для более простого чтения и понимания идентификаторов рекомендуется использовать имена идентификаторов, состоящие из строчных и прописных букв. Каждый идентификатор имеет тип, который устанавливается при его объявлении. Компилятор языка C++ не допускает использование идентификаторов, совпадающих по написанию с ключевыми словами. Например, идентификатор auto недопустим (однако допустим идентификатор AUTO).

6.3. Переменные и константы

Данные, обрабатываемые компилятором, - это переменные и константы.

Переменные - это данные, которые могут изменять свои значения в процессе выполнения программы. Все переменные в языке C++ должны быть описаны явно. Это означает, что, во-первых, в начале каждой программы или функции Вы должны привести список имен (идентификаторов) всех используемых переменных, а во-вторых, указать тип каждой из них. Оператор описания состоит из спецификации типа и списка имен переменных, разделенных запятой. В конце обязательно должна стоять точка с запятой. При описании возможно задание начально-

го значения переменной. Имя переменной - любая последовательность прописных и строчных букв английского алфавита, цифр и символа подчеркивания ‘_’. Имя должно начинаться с буквы или символа подчеркивания. Имя может быть произвольной длины, но значимыми являются только первые 32 символа; остальные символы имени игнорируются.

Спецификация типа формируется из ключевых слов, указывающих на различные типы данных. Основные типы в C++ подразделяются на две группы: целочисленные типы и типы с плавающей точкой. К целочисленным типам относятся типы, представленные следующими именами основных типов: char, short, int, long. Имена целочисленных типов могут использоваться в сочетании с парой модификаторов типа signed и unsigned. Эти модификаторы изменяют формат представления данных, но не влияют на размеры выделяемых областей памяти. Модификатор типа signed указывает, что переменная может принимать как положительные, так и отрицательные значения. Модификатор типа unsigned указывает, что переменная принимает только положительные значения. Основные характеристики целочисленных типов выглядят следующим образом (табл. 6.2):

Таблица 6.2

Тип данных	Байты	Биты	Min	Max
signed char	1	8	- 128	127
unsigned char	1	8	0	255
signed short	2	16	-32768	32767
unsigned short	2	16	0	65535
signed int	2	16	-32768	32767
unsigned int	2	16	0	65535
signed long	4	32	-2147483648	2147483647
unsigned long	4	32	0	4294967295

По умолчанию считается, что данные типов char, int, short int, long используются со знаком. Поэтому ключевое слово signed можно не указывать. Данные типа char используются для хранения символов. Под символом подразумевается одиночная буква, цифра или знак, занимающий только один байт памяти. Переменные типа char могут использоваться как данные со знаком (signed char) и как данные без знака (unsigned char). Если тип char рассматривается как signed, то

старший байт его кода определяет знак. В этом случае диапазон значений типа char - от -128 до + 127. В случае unsigned char все восемь бит рассматриваются как код, а диапазон возможных значений - от 0 до 255.

К плавающим типам относятся три типа, представленные следующими именами типов, модификаторов и их сочетаний: float, double, long double. Они используются для работы с вещественными числами, которые представляются в форме записи с десятичной точкой и в "научной нотации". Разница между нотациями становится очевидной из простого примера, который демонстрирует запись одного и того же вещественного числа в различных нотациях.

297.7
2.977E2
и ещё один пример...
0.002355
2.355E-3

В научной нотации слева от символа E записывается мантисса, справа - значение экспоненты, которая всегда равняется показателю степени 10. Ниже представлены основные характеристики типов данных с плавающей точкой (табл. 6.3):

Таблица 6.3

Тип данных	Байты	Биты	Min	Max
float	4	32	3.4E-38	3.4E+38
double	8	64	1.7E-308	1.7E+308
long double	10	80	3.4E-4932	3.4E+4932

Константы - это данные, которые устанавливаются равными определенным значениям еще до выполнения программы и сохраняют их на протяжении выполнения всей программы. Имеется 4 типа констант: целые, с плавающей запятой, символьные и перечисляемые. Например, 25 и -5 - целые константы, 4.8 , 5E15, -5.1E8 - константы с плавающей запятой, ‘A’, ‘\0’, ‘\n’, ‘007’ - символьные константы. Набор символов внутри двойных кавычек: "Это строка "- есть строковая константа. Целые константы могут быть десятичные, восьмеричные и шестнадцатеричные.

Десятичные константы могут принимать значения от 0 до 4 294 967 295. Константы, выходящие за указанные пределы, вызывают ошибку. Отрицательные десятичные константы - это просто константы без знака, к которым применена унарная операция минус.

Восьмеричные константы начинаются с символа нуля, после которого следуют восьмеричные цифры (от 0 до 7), например 032. Если восьмеричная константа содержит недопустимые цифры 8 или 9, выдается сообщение об ошибке. Ошибка будет также выдаваться при превышении восьмеричной константой значения **037777777777.**

Шестнадцатеричные константы начинаются с **0x** (или **0X).** Значения шестнадцатеричных констант, превышающие **0xFFFFFFFF,** приводят к ошибке.

Символьная константа - это один или более символов, заключенных в одинарные кавычки, например **'F',** '=', '**\n'.** В **C++** константа из одного символа имеет тип **char.**

Для введения управляющих последовательностей, позволяющих получить визуальное представление некоторых, не имеющих графического аналога символов, используется группа символьных констант, которые начинаются со специального символа обратной косой черты ('\').

В таблице 6.4 показаны допустимые управляющие последовательности.

Строковые константы образуют специальную категорию констант, используемых для работы с фиксированными последовательностями символов. Строковая константа имеет тип данных array of char и записывается как последовательность произвольного количества символов, заключенных в двойные кавычки: "Это строковая константа!". Нулевая (пустая) строка записывается как "".

Перечисляемые константы представляют собой идентификаторы, определенные в объявлениях типа enum. Эти идентификаторы обычно выбираются как мнемонические обозначения для удобства обращения с данными. Перечисляемые константы имеют целочисленный тип данных. Они могут быть использованы в любых выражениях, в которых допустим целочисленный тип данных. Используе-

мые идентификаторы должны быть уникальными в пределах контекста объявления enum. Значения, принимаемые этими константами, зависят от формата объявления перечислимого типа и присутствия опциональных инициализаторов. Например, в операторе enum color { red, yellow, green }; объявляется переменная с именем color, которая может принимать константные значения red, yellow или green.

Таблица 6.4

Последовательность символов	**Обозначение в ASCII-таблице**	**Выполняемое действие**
\a	BEL	При выводе на экран вызывает звуковой сигнал
\b	BS	При выводе на принтер и на экран вызывает сдвиг текущей позиции влево
\f	FF	Перевод формы. При выводе на принтер вызывает прогон бумаги до начала следующей страницы
\n	LF	При выводе на принтер и на экран переводит строку
\r	CR	При выводе на принтер и на экран перемещает текущую позицию на начало строки, не переводит строку
\t	HT	Горизонтальная табуляция
\v	VT	Вертикальная табуляция
\\	\	Символ обратной косой черты
\'	‘	Символ одиночной кавычки
\"	“	Символ двойной кавычки
\?	?	Знак вопроса
\0oo		Последовательность цифр, начинающаяся с 0, трактуется как код внутреннего представления, заданный в восьмеричной системе счисления числом oo
\xhh		Последовательность цифр, начинающаяся с x , трактуется как код внутреннего представления, заданный в шестнадцатеричной системе счисления числом hh

6.4. Определение констант с помощью директивы препроцессора #define

Константы в языке C++ можно задавать либо в явном виде (т.е. указывать непосредственно значение константы), либо использовать идентификатор, которому присваивается значение константы. Определение константы с помощью идентификатора осуществляется в заголовке программы по следующей форме [2,4]:

#define имя строка,

где имя - идентификатор; строка - любая последовательность символов, отделяемая от имени хотя бы одним пробелом и заканчиваемая в текущей строке.

Директива #define выполняет простую текстовую подстановку, т.е. когда препроцессор встречает имя, он заменяет его на строку.

Примеры:

```
#define   I   5 // ставит в соответствие имени I число 5
#define   J   4
#define   PI 3.1415
```

Необходимо обратить внимание на то, что при использовании директивы define тип константы не имеет значения (константы I, J, PI не имеют никакого конкретного типа). Определение констант с помощью директивы define наиболее предпочтительно, так как в случае изменения их значений в программе понадобится внести изменения только в одном месте.

7. Операции и выражения

7.1. Выражение и его интерпретация

Выражение в языке C++ (C) - это последовательность операндов, операций и символов-разделителей [2]. Операнды - это переменные, константы либо другие выражения. Разделителями в C++ являются символы [ ] ( ) { } , ; : . * = #, каждый из которых выполняет свою функцию. Выражение может состоять из одной или более операций и определять выполнение целого ряда элементарных шагов по преобразованию информации. Компилятор соблюдает строгий порядок интерпретации выражений, называемый правилами предшествования. Этот порядок может быть изменен, если отдельные части выражения заключить в круглые скобки. Элементарная операция по преобразованию информации задается знаком операции.

По числу операндов, участвующих в операции, различают следующие типы:

- унарные (имеющие один операнд);

- бинарные (имеющие два операнда);
- тернарные (имеющие три операнда).

По типу выполняемой операции различают:

- арифметические операции;
- логические операции и операции отношения;
- операцию условия;
- операцию присваивания;
- операцию sizeof;
- операцию преобразования типов.

-

7.2. Арифметические операции

Язык C++ (C) включает [3]:

- арифметические операции сложения (задается знаком +), вычитания и унарный минус (-), умножения (*), деления (/), операцию определения остатка (%);
- операции инкремента (++) и декремента (--).

Операции сложения, вычитания, умножения и деления являются стандартными и выполняются так же, как и в большинстве других алгоритмических языков.

Операция определения остатка (%) (или деления по модулю) служит для определения остатка от деления числа целого типа, стоящего слева от знака, на целое число, расположенное справа от знака. Ее поясняет следующий пример:

```
int x =7, y = 2, z ;
z = x % y;      // z=1 - остаток от целого деления 7/3
z = x/y;    // z =2 - результат деления целых чисел
```

Операции инкремента и декремента соответственно увеличивают и уменьшают операнд на 1. Операции ++ и -- могут применяться только к переменным.

Используются две формы их записи: префиксная (знак операции располагается слева от операнда) и постфиксная (знак операции справа от операнда). В префиксной форме сначала выполняется увеличение операнда на 1, и увеличенное значение используется в выражении. В постфиксной форме сначала берется значение операнда, и только после этого его значение увеличивается на 1. Например:

```
int x=0, y = 1, z=0;
z = x++;     // в результате z = 0, x = 1
z = ++x;     // z=2, x =2
z = ++y;     // z=2, y= 2
```

Ниже приведен пример программы, иллюстрирующей работу префиксной и постфиксной форм инкремента.

```
# include < iostream .h>
main{}
{
int x;
x=5;
cout << x <<"\n" ;
cout << x++ <<"\n" ; // Постфиксная форма инкремента cout <<
x <<"\n" ;

x=5;
cout << x <<"\n" ;
cout << ++ x <<"\n" ; // Префиксная форма инкремента
cout << x <<"\n" ;
return 0;
} ----------------------------------------------------------------
Результаты работы программы:
55
65
66
```

Форма записи операций ++ и -- сказывается в составных выражениях. Например, три оператора присваивания вида

```
number = number + 1;
sum = sum + 1;
x = x + 1;
```

могут быть записаны более кратко

```
number += 1;
sum + = 1;
x += 1; ,
```

а с использованием операций инкремента в префиксной форме:

```
++number;
++sum ;
++x;
```

или в постфиксной форме:

```
number++;
sum++;
x ++;
```

7.3. Логические операции и операции отношения

Логические операции и операции отношения используются при формировании логических выражений, имеющих только два значения: 1, если логическое выражение ИСТИННО; 0, если логическое выражение ЛОЖНО. Логические выражения наиболее часто используют вместе с операторами управления потоком вычислений - операторами циклов и ветвлений [2]. С++ поддерживает следующие логические операции:

&& - логическое И; дает результат 1 (ИСТИНА), если все операнды имеют значение 1 (ИСТИНА); в противном случае дает значение 0 (ЛОЖЬ);

|| - логическое ИЛИ; дает результат 1 (ИСТИНА), если хотя бы один из операндов имеет значение 1 (ИСТИНА); в противном случае дает значение 0 (ЛОЖЬ);

! - логическое НЕ; дает результат 1 (ИСТИНА), если операнд справа от знака имеет значение 0 (ЛОЖЬ); в противном случае дает значение 1 (ИСТИНА).

Возможные результаты выполнения логических операций представлены в табл.7.1.

Таблица 7.1

a	b	a&&b	a b	!a
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Операнды логических выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, то второй операнд не вычисляется. Логические операции не вызывают стандартных арифметических преобразований. Они оценивают каждый операнд с точки зрения его эквивалентности нулю. Результатом логической операции является 0 или 1, тип результата int.

C++ поддерживает следующие операции отношения:

- > - больше; дает результат 1, если операнд слева от знака больше операнда справа; в противном случае дает 0;
- < - меньше; дает результат 1, если операнд слева меньше операнда справа; в противном случае дает 0;
- = = - равно; дает результат 1, если операнд слева от знака равен операнду справа; в противном случае дает 0;
- >= - больше или равно; дает результат 1, если операнд слева от знака больше или равен операнду справа; в противном случае дает 0;
- <= - меньше или равно; дает результат 1, если операнд слева от знака меньше или равен операнду справа; в противном случае дает 0;
- != - не равно; дает результат 1, если операнд слева от знака не равен операнду справа; в противном случае дает значение 0.

Приведем пример программы, получающей со стандартного ввода два значения и выводящей на стандартный вывод результат выполнения логических операций и операций отношения:

```
#include <iostream.h>
#include <conio.h>
```

```
void main( )
{
  clrscr(); float p1,p2;
  cout << "Введите первое значение p1 = ";
  cin >> p1;
  cout << "Введите второе значение p2 = ";
  cin >> p2;
  cout << " p1 > p2 дает " << (p1 > p2) << "\n";
  cout << " p1 < p2 дает " << (p1 <p2) << "\n";
  cout << " p1 == p2 дает " << (p1==p2) << "\n";
  cout << " p1 >= p2 дает " << (p1>=p2) << "\n";
  cout << " p1 <= p2 дает " << (p1<=p2) << "\n";
  cout << " p1 != p2 дает " << (p1!=p2) << "\n";
  cout << "     !p1 дает " << !p1 << "\n";
  cout << "     !p2 дает " << !p2 << "\n";
  cout << " p1 || p2 дает " << (p1||p2) << "\n";
  cout << " p1 && p2 дает " << (p1&&p2) << "\n";
  cout<<"\n---------------------------- \n";
  cout«ᴹ\nНажмите любую клавишу
  getch();
}_____________________________________________
```

Результаты работы программы:

Введите первое значение **p1 = 2**
Введите второе значение **p2 = 3**

p1 > p2 дает **0**
p1 < p2 дает **1**
p1 == p2 дает **0**
p1 >= p2 дает **0**
p1 <= p2 дает **1**
p1 != p2 дает **1**
!p1 дает **0**
!p2 дает **0**
p1 || p2 дает **1**
p1 && p2 дает **1**

7.4. Операция условия

В языке C++ (C) имеется одна тернарная операция - условная операция, которая имеет следующий формат:

операнд-1 ? операнд-2 : операнд-3.

Операнд-1 должен быть целого или плавающего типа или быть указателем. Он оценивается с точки зрения его эквивалентности 0. Если операнд-1 не равен 0, то вычисляется операнд-2, и его значение является результатом операции. Если операнд-1 равен 0, то вычисляется операнд-3, и его значение является результатом операции. Следует отметить, что вычисляется либо операнд-2, либо операнд-3, но не оба.

Пример: max = (d<=b) ? b : d; Переменной max присваивается максимальное значение из переменных d и b.

7.5. Операция присваивания

Язык C++ имеет несколько особенностей выполнения операции присваивания. Помимо простого присваивания посредством операции "=" , C++ поддерживает составные операции присваивания, которые перед присваиванием выполняют дополнительные операции над своими операндами [1].

Формат операции простого присваивания (=): операнд 1 = операнд2; .

Сначала вычисляется выражение, стоящее в правой части операции, а потом его результат записывается в область памяти, указанную в левой части.

Например:

```
int a=3,b=5,c=7; // Результаты работы фрагмента программы:
a=b;b=a;c=c+1; // a=5;b=5;c=8.
```

Операция составного присваивания состоит из простой операции присваивания, скомбинированной с другой бинарной операцией. В составном присваивании вначале выполняется операция, специфицированная аддитивным оператором, а затем результат присваивается левому операнду. Выражение составного присваивания, например, имеет вид

операнд 1> += < операнд 2>

и может быть понято как

операнд 1> = < операнд 1> + < операнд 2>

Однако выражение составного присваивания неэквивалентно расширенной версии, поскольку в выражении составного присваивания < операнд 1> вычисляется только один раз, в то время как в расширенной версии оно вычисляется дважды: в операции сложения и в операции присваивания.

В общем случае формат операции составного присваивания можно представить так:

< операнд1> (бинарная операция) = < операнд2>,

где (бинарная операция) - одна из операций, задаваемых знаками *, /, +, -,%.

Результатом операции составного присваивания являются значение и тип левого операнда. Например:

```
a += b;      // a= a + b;
a -= b;      // a= a - b;
a *= b;      // a= a * b;
a /= b;      // a= a / b;
a %= b;      // a= a % b;
```

7.6. Операция sizeof

С помощью операции sizeof можно определить размер памяти, которая соответствует идентификатору или типу. Она имеет следующий формат:

sizeof (выражение) .

В качестве выражения может быть использован любой идентификатор (кроме имени функции), либо имя типа, заключенное в скобки. Если в качестве выражения указано имя массива, то результатом является размер всего массива (т. е. произведение числа элементов на длину типа). Результатом операции sizeof является размер в байтах типа или объявленной переменной, а применительно к массивам операция возвращает число байтов, необходимое для размещения всех элементов массива. Если определяется размер переменной, то имя переменной можно также указывать через пробел после sizeof без круглых скобок.

Пример:

```
#include <iostream.h>
void main()
{
 int i; char c;
 long double ff;
 cout << " Данные типа char занимают " << sizeof(char) << "байт \n";
 cout << " Данные типа int занимают " << sizeof(int) << "байт \n";
 cout << " Данные типа long занимают " << sizeof(long) << "байт \n";
 cout << " Переменная i занимает " << sizeof i << "байт \n";
 cout << " Переменная c занимает " << sizeof c << "байт \n";
 cout << " Переменная ff занимает " << sizeof ff << "байт \n";
}
```

В результате выполнения программы на экран будет выведено:

Данные типа char занимают 1 байт
Данные типа int занимают 2 байт
Данные типа long занимают 4 байт
Переменная i занимает 2 байт
Переменная c занимает 1 байт
Переменная ff занимает 10 байт

7.7. Преобразование типов

Операнды бинарной операции могут быть разного типа. В этом случае перед ее выполнением компилятор предварительно приводит операнды к одному типу. Преобразование типов выполняется по следующим правилам [2]:

1. операнды разных типов приводятся к "старшему", т.е. более длинному типу. Ниже перечислены типы в порядке убывания старшинства: самый старший - long double, double, float, unsigned long, long int, unsigned int, char - самый младший;
2. при выполнении операции присваивания результат приводится к типу переменной слева от знака операции.

Явное приведение типа операндов выполняется с помощью операции type(expression), где type - любой допустимый тип языка C++. Результатом опера-

ции type (expression) является значение выражения expression, преобразованное к типу type. Пример:

```
---
int x; float y=1.6;
…
x = int (5.2) + int (y);
---
```

В результате выполнения данного фрагмента программы получим x=6. Преобразование младшего (меньшего по размеру типа) к старшему происходит корректно, а старшего к младшему может вызвать затруднения (все число может не поместиться в выделенной под него области памяти, и тогда результат операции будет не верный).

7.8. Порядок выполнения операций

Все операции, распознаваемые компилятором, упорядочены по приоритету. Выражения с более приоритетными операциями вычисляются первыми. Порядок вычислений может изменяться круглыми скобками. Если внутри скобок или при отсутствии скобок вообще в выражении есть несколько операций одного приоритета, компилятор учитывает дополнительно порядок выполнения операций, например, слева направо или справа налево. Заключенное в скобки выражение интерпретируется компилятором в первую очередь. В сложном выражении первым выполняется выражение в самых «глубоких» скобках. В табл.7.2 приведен перечень всех операций языка, упорядоченных в порядке убывания приоритета (тонкая черта отделяет операции с одинаковым приоритетом).

Таблица 7.2

Название	Символ операции	Порядок выполнения
Обращение к функции	( )	Слева направо
Выделение элемента массива	[]	То же
Выделение поля структурной переменной	.	-“-
Выделение поля структурной переменной по указателю на ее начало	->	-“-
Логическое отрицание	!	Справа налево
Поразрядное логическое НЕ	~	То же
Изменение знака (унарный минус)	-	-“-
Инкремент	++	-“-

Декремент Определение адреса переменной Обращение к памяти по значению указателя Преобразование к типу Определение размера в байтах	-- & * (type) sizeof	-“- -“- -“- -“- -“- -“-
Умножение Деление Определение остатка целого деления	* / %	Слева направо То же -“-
Сложение Вычитание	+	Слева направо То же
Сдвиг влево Сдвиг вправо	<< >>	Слева направо То же
Меньше Меньше или равно Больше Больше или равно	 < > >=	Слева направо То же -“- -“-
Равно Не равно	!=	Слева направо То же
Поразрядное логическое И	&	Слева направо
Поразрядное исключающее ИЛИ (XOR)	^	Слева направо
Поразрядное логическое ИЛИ	|	Слева направо
Логическое И	**&&**	Слева направо
Логическое ИЛИ	||	Слева направо
Операция условия	**?:**	Справа налево
Присваивание	= += -= *= /= %= <<= >>= != ^= &=	Слева направо То же -“- -“-
Операция «запятая»	,	Слева направо

8. Операторы управления

8.1. Общие сведения

Операторы управления вычислительным процессом позволяют выполнять ветвление, циклическое повторение одного или нескольких операторов, передачу управления в нужное место кода программы. Под вычислительным процессом понимают процесс выполнения операторов программы. Операторы программы могут быть простыми или составными. Простой оператор - это оператор, не содержащий другие операторы. Разделителем простых операторов служит точка с запятой. Специальным случаем простого оператора является пустой оператор, состоящий из единственного символа ';'. Составной оператор, или блок, - это любая совокупность простых операторов, заключенная в фигурные скобки {}. Составной

оператор идентичен простому оператору и может находиться в любом месте программы, где синтаксис языка допускает наличие оператора. Все операторы языка могут быть условно разделены на следующие категории:

- условные операторы, к которым относятся оператор условия if и оператор выбора switch;
- операторы цикла (for, while, do while);
- операторы перехода (break, continue, return, goto);
- другие операторы (оператор "выражение", пустой оператор).

8.2. Оператор if

Оператор условия if выбирает в программе из группы альтернатив возможное продолжение вычислительного процесса. Выбор выполняется, исходя из значения заданного логического выражения.

Оператор if имеет следующую общую форму записи:

if (логическое выражение) оператор A; [else оператор B;]

При выполнении оператора if сначала вычисляется логическое выражение. При результате ИСТИНА (любое отличное от нуля значение) выполняется оператор A, в противном случае (результат логического выражения ЛОЖЬ (равен 0)) выполняется оператор B. Если ключевое слово else отсутствует, а результат логического выражения ЛОЖЬ, то в этом случае оператор A пропускается, а управление передается на следующий после if оператор. Если в какой-либо ветви требуется выполнить несколько операторов, их необходимо заключать в блок. Блок может содержать любые операторы, в том числе и другие условные операторы, образуя так называемые вложенные if [1].

Примеры:

```
if (a<0) b=1;                                                          //1
if (a<0 &&(a > d || a = = 0)) b++; else {b*=a;a=0;}                    //2
if (a<b) { if (a<c) m=a; else m=c;} else { if (b<c) m=b; else m=c;}     //3
if (a++) b++;                                                          //4
 if (b>a) max = b; else max = a;                                       //5
```

В примере 1 отсутствует ветвь else. В примере 2 показано, что при необходимости проверки нескольких условий их объединяют знаками логических операций. Оператор в примере 3 вычисляет наименьшее значение из трех переменных. В примере 4 показано, что в логическом выражении могут использоваться не только операции отношения, а в примере 5 показано, как с помощью оператора if можно найти максимальное из двух заданных чисел. При использовании вложенных циклов может возникнуть неоднозначность в понимании того, к какой из вложенных конструкций if относится элемент else. Например, в конструкции

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

элемент else будет отнесен компилятором ко второй конструкции if , т.е. оператор 2 будет выполняться в случае, если первое условие истинно, а второе ложно. Иначе говоря, конструкция будет прочитана как

```
if (условие 1 )
 {
   if(условие 2) оператор 1;
   else оператор 2;
 }
```

Если же необходимо отнести else к первому оператору if, то с помощью соответствующих установок фигурных скобок фрагмент программы запишется в виде

```
if (условие 1 ) {
    if(условие 2) оператор 1;
  }
else оператор 2;
```

Рассмотрим еще один пример программы:

```
#include <iostream.h>
void main()
{
  int a=1, b=10;
```

```
 if (a ==1)
    if (b == 1)
      cout << " \n a=1, b = 1\n";
 else
    cout << "\n a не равно 1\n";
}
```

В результате выполнения этой программы на экран будет выводиться строка "a не равно 1", хотя на самом деле a=1. Ошибка происходит потому, что компилятор сопоставляет if с ближайшим else (т.е. со 2-м if).

Ошибку можно исправить, поставив фигурные скобки, ограничивающие блок. В результате данный пример будет иметь вид

```
#include <iostream.h>
void main( )
{
  int a=1, b=10;
  if (a ==1)
     {
       if (b == 1)
          cout << " \n a=1, b = 1\n";
      }
   else
      cout << "\n a не равно 1 \n"; cout
   << "Конец";
}
```

Рассмотрим еще один пример:

```
int main ( )
 {
  int t=2, b=7, r=3;
    if (t>b)
    {
       if (b < r) r=b;
    }
   else r=t;
   return (0);
 }
```

В результате выполнения этой программы r станет равным 2. Если же в программе опустить фигурные скобки, стоящие после оператора if, то программа будет иметь следующий вид:

```
int main ( )
 {
  int t=2, b=7, r=3;
  if ( a>b )
    if ( b < c ) t=b;
    else     r=t;
  return (0);
 }
```

В этом случае r получит значение, равное 3, так как ключевое слово else относится ко второму оператору if, который не выполняется, поскольку не выполняется условие, проверяемое в первом операторе if. При выборе результата из нескольких возможных вариантов широко используется следующая конструкция оператора if, часто определяемая как else - if :

```
if (логическое выражение 1)
   оператор 1;
else if (логическое выражение 2)
        оператор 2;
     else if (логическое выражение 3)
             оператор 3;
……..
                  else if (логическое выражение N)
                          оператор N;
                       else оператор;
```

В этой конструкции все логические выражения просматриваются последовательно. Если какое-то выражение оказывается истинным, то выполняется относящийся к нему оператор, и этим вся цепочка заканчивается. Каждый оператор может быть либо отдельным оператором, либо группой операторов в фигурных скобках. Последняя часть с else реализуется, когда ни одно из проверяемых условий не выполняется. Иногда по условиям задачи этот оператор может быть опущен.

Для иллюстрации выбора из четырех возможных вариантов приведем программу, определяющую номер квадранта в декартовой системе координат для точки, значения координат которой заранее не известны, а задаются в режиме диалога.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include<iostream.h>
main()
{
float x=0, y=0;
int   n;                              // номер квадранта
clrscr();                             // очистка экрана
cout<<"\n Найти квадрант для точки с заранее неизвестными координатами\n";
cout<<"\nВведите координату X=";
cin>> x;
cout<<"\nВведите координату Y=";
cin>> y;
if( x > 0 && y>0 )
   n = 1;
else if (x < 0 && y > 0)
   n = 2;
else if (x < 0 && y < 0)
   n = 3;
else
   n = 4;
cout<<"\n\nТочка находится в "<<n<<" квадранте";
cout<<"\n\nНажмите любую клавишу…";
getch();
return 0;
}
```

8.3. Операторы switch

Иногда алгоритм задачи содержит ряд альтернативных решений, причем некоторую переменную надо проверять отдельно для каждого постоянного целого значения, которое она может принимать. В зависимости от результатов этой проверки должны выполняться различные действия. Для принятия подобных решений в C++ имеется структура множественного выбора - оператор switch. Оператор switch производит сопоставление значения с множеством констант. Структура

switch, состоящая из ряда меток case и необязательной метки default (умолчание), имеет следующий вид:

```
switch (выражение выбора)
 {
 case значение 1:
        оператор 1;
        break;
.....
 case значение N:
        оператор N;
        break;
 default: оператор;
 }
```

Пример программы на использование оператора switch [2] показан на рис. 8.1:

```
// Пример программы на использование оператора switch
#include <iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    char x;
    cout << "Введите первую букву имени функции\n";
    cout << "S- Sin\nC - Cos\nA - Atan\n";
    cin >> x;
    switch (x)
    {
     case 's':
     case 'S':
        cout << "Вычисление синуса аргумента в радианах\n";
        break;
     case 'c':
     case 'C':
        cout << "Вычисление косинуса аргумента в радианах\n";
        break;
     case 'a':
     case 'A':
        cout << " Вычисление арктангенса аргумента ";
        cout << " в радианах\n";
        break;
```

```
      default:
         cout << "Ошибка\n";
   }
     cout<<"\n ------------------------------ \n";
  cout«"\nНажмите любую клавишу…"
getch();
}
```

Результаты работы программы:

Введите первую букву имени функции
S- Sin
C - Cos
A - Atan
C

Вычисление косинуса аргумента в радианах

Рис. 8.1

Программа на рис. 8.2 использует оператор switch для подсчета числа различных оценок, полученных студентами на экзаменах.

```
//Использование оператора switch для подсчета числа оценок
# include <iostream .h>
#include<conio.h>
main()
{
     int i = 1 , grade;                                    // оценка
     int otl = 0, xor = 0, udov = 0, neudov=0;
     while (i<=8){
     cout << "Введите значение оценки =";
     cin>>grade;
     switch (grade) {
      case 5:
         ++ otl ;
         break;
      case 4:
         ++xor;
         break;
      case 3:
         ++ udov;
         break;
      case 2:
         ++ neudov;
         break;
      default:
```

```
        cout << "\nВведена неправильная оценка\n" ;
        break;
     }
  ++i ;
}
Cout<<кол.отл.оценок=<<otl<<\n"<<кол.хор.оценок="<<\n"<<
      кол.удов.оценок= "<< udov<<"\n" <<"
      кол.неудов.оценок="<<neudo<<"\n";
cout<<"\n .....\n";
cout<<Нажмите любую клавишу";
getch();
return(0);
}
```

Рис. 8.2

Результаты работы программы:

```
Введите значение оценки = 5
Введите значение оценки = 5
Введите значение оценки = 4
Введите значение оценки = 3
Введите значение оценки = 3
Введите значение оценки = 4
Введите значение оценки = 4
Введите значение оценки = 2
кол.отл.оценок=2
кол.хор.оценок=3 кол.удов.оценок=2
кол.неудов.оценок=1
```

Рис. 8.2

Оператор break применяется для выхода из оператора switch и вызывает передачу управления на первый оператор после структуры switch. Константы в вариантах case должны быть различными, и если проверяемое значение не совпадает ни с одной из констант, выбирается вариант default. После каждой метки case может быть предусмотрено одно или несколько действий, при этом в последнем случае операторы не нужно заключать в скобки.

8.4. Оператор while

Структура оператора:

```
while (логическое условие)
{
```

```
    операторы
}
```

Оператор while позволяет осуществлять циклическое выполнение рабочих операторов, пока логическое условие остается истинным. Рабочие операторы, записанные в структуре оператора while, составляют его тело, которое может быть отдельным или составным оператором.

Как пример использования while с единственным рабочим оператором рассмотрим фрагмент программы, определяющей первое значение степени 2, превышающее 1000. Предположим, что имеется целая переменная number, с начальным значением 2. Когда приведенный ниже фрагмент программы закончит свою работу, number будет содержать искомую величину.

Пример программы показан на рис. 8.3.

```
//Пример использования оператора while
#include<iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
main()
{
   clrscr();
   int number=2; while(number<=1000)
   number=2*number ;
   cout<<\nчисло="<<number;
   cout<<"\n ------------------------------\n";
   cout<<\nНажмите любую клавишу…
   getch();
   return 0;
   }
```

Результаты работы программы:

число= 1024

Рис. 8.3

На рис. 8.4 приведен пример использования оператора while для расчета суммы четных чисел.

```
//Пример на использование оператора цикла while
//Расчет суммы четных чисел от 1 до 10
```

```
#include <iostream.h>
#include <conio.h>
void main( )
{
  clrscr( );
  int n=2, sum=0, i=1;
  while (n <= 10)
  {
    sum += n;                    // sum = sum + n;
    cout << "В"<<" "<<i<<" "<< "цикле n = " << n << "\t
     sum = " << sum << "\n";
     n += 2;
    i++ ;
  }
  cout << " \n Окончательный результат: \n";
  cout << " n = " << n << "\t sum = " << sum << "\n";
  cout<<"\nНажмите любую клавишу…
  getch( );
} ____________________________________________________
```

Результаты работы программы:

В 1 цикле **n** = 2 **sum** = 2
В 2 цикле **n** = 4 **sum** = 6
В 3 цикле **n** = 6 **sum** = 12
В 4 цикле **n** = 8 **sum** = 20
В 5 цикле **n** = 10 **sum** = 30

Окончательный результат:
n=12 **sum**=30

Рис. 8.4

Другой пример на рис. 8.5 иллюстрирует работу структуры while при использовании составного оператора в теле цикла. В приведенной программе внутри тела цикла оператора while вычисляются и выводятся на дисплей значения функции y=sin(x) с шагом 0,5 до значения x <= X_{max}, а при выходе из цикла функция y вычисляется по формуле y = cos(x).

```
#include<iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
main()
{
```

```
clrscr();
int i;
float x,y,Xmax;
cout<<"Введите максимальное значение переменной Xmax =";
cin >> Xmax;
x=0;
while(x<=Xmax)
  {
   y=sin(x);
   x+=0.5;
   cout<<"\ny="<<y;
  }
cout<<"\nМомент переключения";
y=cos(x);
cout<<"\ny="<<y;
cout<<"\n---------------------------- \n";
cout<<"\nНажмите любую клавишу ... ";
getch();
return 0;
}
```

Результаты работы программы:

Введите максимальное значение переменной Xmax = 1
y=0
y=0.479426
y=0.841471
Момент переключения **x**=1.5
y=0.070737

Рис. 8.5

Следующее после while условие должно быть заключено в круглые скобки. Условие вычисляется, и, если его значение не ноль, выполняется непосредственно следующий за ним оператор. Это повторяется до тех пор, пока вычисление условия не даст ноль. Разновидностью оператора while является структура do/while, в которой проверка условия продолжения циклов производится после того, как тело цикла выполнено, следовательно, тело цикла будет выполнено, по крайней мере, один раз. Структура оператора имеет вид

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

Пример использования структуры do/while показан на рис. 8.6.

```
//Пример на использование оператора цикла do ... while
//Расчет суммы нечетных чисел от 0 до 10.
#include <iostream.h>
#include <conio.h>
void main()
{
 clrscr();
 int n=1,sum=0;
 do
  {
   sum += n;   // sum = sum +n;
   cout << "Сейчас n = " << n << "\t sum = " << sum << "\n";
   n += 2;
  }
 while (n <= 10);
 cout << "\nОкончательный результат: \n";
 cout << " n = " << n << "\t sum = " << sum << "\n";
 cout<<"\n\n";
 cout«"\nНажмите любую клавишу…";
  getch();
}
```

Результаты работы программы:

Сейчас **n** = 1 **sum** = 1
Сейчас **n** = 3 **sum** = 4
Сейчас **n** = 5 **sum** = 9
Сейчас **n** = 7 **sum** = 16
Сейчас **n** = 9 **sum** = 25
Окончательный результат:
Сейчас **n** = 11 **sum** = 25

Рис. 8.6

8.5. Оператор for

Оператор for повторяет блок рабочих операторов указанное число раз и имеет следующую структуру:

for *(имя переменной = начальное значение; конечное значение; приращение)* **оператор,** где

- имя переменной - имя управляющей переменной, используемой как счетчик цикла;

- начальное значение - начальное значение управляющей переменной;
- конечное значение - конечное значение управляющей переменной;
- приращение - шаг изменения значения управляющей переменной;
- оператор - один или несколько рабочих операторов, образующих тело цикла.

Если тело цикла содержит более одного оператора, последние должны заключаться в фигурные скобки. Способы изменения управляющей переменной в структуре оператора for покажем на следующих примерах:

1. Изменение управляющей переменной от 1 до 100 с шагом 1
   for (int i = 1; i<=100);
2. Изменение управляющей переменной от 100 до 1 с шагом -1
   for (int i = 100; i >=1; i--);
3. Изменение управляющей переменной от 7 до 77 с шагом 7
   for (int i = 7; i <= 77; i += 7);
4. Изменение управляющей переменной от 20 до 2 с шагом -2
   for (int i = 20; i >= 2; i -= 2);
5. Изменение управляющей переменной от 7 до 77 с шагом 7
   for (int i = 7; i <= 77; i += 7);

Следующие два примера содержат простые приложения структуры **for.** Первая программа использует структуру **for** для суммирования всех четных чисел от 2 до 100.

```
//Суммирование с помощью for
# include <iostream.h>
#include <conio.h>
main ( )
{
 int sum = 0;
  for (int i = 2; i <= 100; i +=2)
       sum += i;
 cout << "Сумма равна" << sum<<"\n";
 return 0;
 }
```

Здесь первое выражение в структуре **for** вводит целую переменную **i,** являющуюся счетчиком циклов, и инициализирует ее значение 2. Второе выражение

проверяет условие завершения цикла. В данном случае цикл завершается, когда переменная **i** примет значение >100. Третье выражение структуры **for** увеличивает значение **i** на 2 после каждого выполнения цикла.

Следующий пример вычисляет с помощью структуры **for** значения функции **y = x** при изменении **x** от -3 до 3 с шагом 0,5.

```
#include<iostream.h> #in-
clude <conio.h>
#include <stdlib.h>
#include <math.h>
main()
{
clrscr();
float x,y;
for ( x = -3 ; 3; x += 0.5)
{
   y = x*x;
   cout<<"\ny="<<y;
}
cout<<"\n---------------------------------- \n";
cout<<"\nНажмите любую клавишу…";
getch();
return 0;
}
```

Цикл, организованный в теле другого цикла, называется вложенным. В этом случае внутренний цикл полностью выполняется на каждой итерации внешнего цикла. Пример программы, иллюстрирующей работу вложенного цикла, приведен на рис. 8.7.

```
// Пример использования вложенных циклов for
#include <iostream.h>
#include <conio.h>
void main( )
{
  clrscr();
  int y=0,n,m;
```

```
for (n = 1; n <= 2; n++)
   for ( m = 1; m <= 2; m++)
    {
      y = y + (n*n + m*m);
      cout << " n = " << n << "\t";
      cout << "m = " << m << "\ty = " << y << "\n";
    }
cout << "\nОтвет: y = " << y;
cout<<"\n\n";
cout << "\nНажмите любую клавишу…";
getch();
} _____
```

Результаты работы программы:

```
n=1 m=1 y=2 n=1
m=2 y=7 n=2 m=1
y=12 n=2 m=2 y=20
Ответ: y =20
```

Рис. 8.7

В этой программе внутренний цикл **for** содержит несколько операторов, объединенных в один блок с помощью фигурных скобок.

8.6. Операторы break и continue

Эти операторы изменяют поток управления. Когда оператор **break** выполняется в структурах **while, for, do/while** или **switch,** происходит немедленный выход из структуры. Программа продолжает выполнение с первого оператора после структуры. Обычное назначение оператора **break -** досрочно прерывать цикл или пропустить оставшуюся часть структуры **switch,** например:

```
// Применение оператора break в структуре for
#include <iostream .h>
main ( )
{
        for (int x=1; x<=10; x++)
           {
              if (x == 5)
                 break;
              cout << "x=" << x;
```

```
        }
    cout << "\n Цикл прерван при x== " <<x<<"\n";
    return 0 ;
}
```

Данный пример демонстрирует прерывание в структуре повторения **for.** Когда структура **if** определяет, что **x** стал равным 5, выполняется оператор **break.** Это вызывает окончание работы оператора **for,** и программа заканчивает свою работу на четвертом цикле.

Оператор **continue** в структурах **while, for** или **do/while** вызывает пропуск оставшейся части тела структуры и начинается выполнение следующей итерации цикла, например:

```
// Применение оператора continue в структуре for
# include <iostream.h>
main()
{
    for ( int x = 1; x <= 10; x++) {
        if (x == 5)
            continue;     // пропуск оставшейся части цикла
        // только при x == 5
        cout <<" x " <<x ;
    }
    cout << "\nИспользован continue для пропуска печати
                    при x == 5"<<"\n";
    return 0;
}
```

В этой программе оператор **continue** используется в структуре **for,** чтобы пропустить оператор вывода при x == 5 и начать следующую итерацию цикла.

9. Функции

9.1.Описание функции

Самый распространенный способ задания в **C++** каких-то действий - это вызов функции, которая выполняет такие действия. Функция - это именованная часть программы (блок кода, не входящий в основную программу), к которой

можно обращаться из других частей программы столько раз, сколько потребуется. Основная форма описания функции имеет вид [4]

```
Тип < Имя функции> (Список параметров)
                    {
                      Операторы тела функции
                    }
```

Тип определяет тип значения, которое возвращает функция с помощью оператора **return.** Если тип не указан, то по умолчанию предполагается, что функция возвращает целое значение (типа **int).**

Список параметров состоит из перечня типов и имен параметров, разделенных запятыми. Функция может не иметь параметров, но круглые скобки необходимы в любом случае.

Первая строка описания функции, содержащая тип возвращаемого значения, имя функции и список параметров, называется заголовком функции. Параметры, перечисленные в заголовке описания функции, называются формальными, а записанные в операторе вызова функции - фактическими. Тип возвращаемого значения может быть любым, кроме массива и функции. В приведенном ниже фрагменте программы функция перемножает два числа и возвращает результат в основную программу с помощью оператора **return** через переменную **z.**

```
int multiply(int x, int y) // заголовок функции
{
int z = (x * y ); // тело функции
return z ;
}
```

До использования функции ее необходимо объявить. Объявление функции осуществляется с помощью прототипа, который сообщает компилятору, сколько аргументов принимает функция, тип каждого аргумента и тип возвращаемого значения. Пример использования функции для вычисления произведения двух чисел показан на рис. 9.1.

```
// Вычисление произведения двух чисел
#include<iostream.h>
#include <conio.h>
#include <stdlib.h>
```

```
int multiply(int x, int y); // объявление прототипа функции
main( )
{
clrscr();
int x, y, result;
cout <<"\nВведите первое число:";
cin >> x;
cout <<"\nВведите второе число:";
cin>>y;
result = multiply(x, y); // вызов функции
cout <<"\nРезультат =" <<
result;
cout <<"\nНажмите любую клавишу
getch();
return 0 ;
}
int multiply(int x, int y) //заголовок функции
{
return x * y ; // тело функции
}
```

Результаты работы программы:

```
Введите первое число: 3
Введите второе число: 2
Результат = 6
```

Рис. 9.1. Окончание

Эта программа запрашивает у пользователя два числа, а затем вызывает функцию **multiply()** для вычисления произведения. Прототип функции объявляется в голове программы и представляет собой тот же заголовок функции, но с точкой запятой “ **;** ” в конце. При желании имена переменных в прототипах можно не указывать. Пример подобной программы показан на рис. 9.2.

```
// Нахождение максимального числа из трех заданных чисел
#include<iostream.h>
#include <conio.h>
int maxs(int, int, int); // прототип функции
main()
{
  clrscr(); //очистка экрана int x,
  y, z, s;
```

```
  cout <<"\nВведите три числа:\n";
  cin>>x>>y>>z;
  s=maxs(x, y, z) ;
  cout<<"\nРезультат ="<< s;
  cout<<"\nНажмите любую клавишу…";
  getch();
  return 0;
}

int maxs(int x, int y, int z)
{
  int max=x;
  if(y>max)
  max=y;
  if(z>max)
  max=z;
  return max;
}
```

Результаты работы программы:

```
Введи-
 те три числа:
 5
 15
  1
Результат =15
```

Рис. 9.2

Прототип функции **maxs** имеет вид **int maxs(int, int, int);.** Этот прототип указывает, что **maxs** имеет три аргумента типа **int** и возвращает результат типа i**nt.** Отметим, что прототип функции имеет такой же вид, что и заголовок описания функции **maxs,** за исключением того, что в него не включены имена параметров **(x,y,z)**. Значение возвращается из функции с помощью оператора **return.** Тип возвращаемого значения может быть любым, кроме массива и функции. Могут быть также функции, не возвращающие никакого значения. В заголовке таких функций тип возвращаемого значения объявляется **void.** Если тип возвращаемого значения не указан, он по умолчанию считается равным **int.**

Пример функции, не возвращающей значения, приведен в программе на рис. 9.3.

```
//Возведение произвольного числа m в степень i
#include <iostream.h>
#include <conio.h>
void step(float, int); // прототип функции
main( )
{
  int i;
  float m, s;
  clrscr();    // очистка экрана
  cout<<"\nВведите m = ";
  cin>> m;
  cout<<"\nВведите i = ";
  cin>> i; step(m, i);
  cout<<"\nНажмите любую клавишу…";
  getch();
}
void step(float base, int k) // заголовок функции
{
 int i; float p = 1;
 for(i=1;i<=k;++i)
 p*=base;
 cout<<"\n"<< base << " в степени "<< k << "
      равно "<<p<<"\n";
}
```

Результаты работы программы:

Введите **m** = 2
Введите **i** = 3
2 в степени 3 равно 8

Рис. 9.3

В этой программе реализован алгоритм возведения в целую степень любого числа. Функция **step** принимает по значению два числа типов **float** и **integer,** а результат ее вычисления не возвращается в основную программу. При вызове тип каждого параметра функции сопоставляется с ожидаемым типом точно так же, как если бы инициализировалась переменная описанного типа. Это гарантирует надлежащую проверку и преобразование типов. Например, обращение к функции **step(float x, int n)** с помощью оператора **step(12.3,"abcd")** вызовет недовольство компилятора, поскольку **"abcd"** является строкой, а не **int.** При вызове **step(2, i)**

компилятор преобразует 2 к типу **float,** как того требует функция. Функция **step** может быть определена, например, так:

```
float step(float x, int n)
{
 if (n < 0) error(" извините, отрицательный показатель для step( )");
   switch (n)
   {
   case 0: return 1;
   case 1: return x;
   default:
     return x*step(x, n -1);
   }
}
```

Первая часть определения функции задает имя функции**(step),** тип возвращаемого ею значения **(float)** и типы и имена ее параметров **(float x, int n).** Значение возвращается из функции с помощью трех операторов **return.**

9.2. Правила работы с функциями

- Функция может принимать любое количество аргументов или не иметь их вообще.
- Функция может возвращать значение, но это не является обязательным.
- Если для возвращаемого значения указан тип **void**, функция не возвращает никакого значения. При этом функция не должна содержать оператора **return,** однако при желании его можно оставить.
- Если в объявлении функции указано, что она возвращает значение, в теле функции должен содержаться оператор **return,** возвращающий это значение. В противном случае компилятор выдаст предупреждение.
- Функции могут иметь любое количество аргументов, но возвращаемое значение всегда одно.

Аргументы могут передаваться функции по значению, через указатели или по ссылке.

Основные преимущества построения программ на основе функций сводятся к следующему:

- Программирование с использованием функций делает разработку программ более управляемой.
- Повторное использование программных кодов, т.е. использование существующих функций как стандартных блоков для создания новых программ.
- Возможность избежать в программе повторения каких-либо фрагментов.

9.3. Передача параметров

При вызове функции выделяется память для ее формальных параметров, и каждый формальный параметр инициализируется значением соответствующего фактического параметра. Имеются два способа обращения к функциям -вызов по значению и вызов по ссылке. Когда аргумент передается вызовом по значению, создается копия аргумента, и она передается вызываемой функции. Основным недостатком вызова по значению является то, что при передаче большого количества данных создание копии может привести к большим потерям времени выполнения. В случае вызова по ссылке оператор вызова дает вызываемой функции возможность прямого доступа к передаваемым данным, а также возможность изменения этих данных. Ссылочный параметр - это псевдоним соответствующего аргумента. Чтобы показать, что параметр функции передан по ссылке, после типа параметра в прототипе и заголовке функции ставится символ амперсанта. Рассмотрим функцию

```
int multiply (int x, int &y) //заголовок функции
{
 return x * y ;
}
```

В функцию **multiply** параметр **x** передается по значению, а параметр **y** по ссылке. Вызов по ссылке хорош в смысле производительности, так как исключает процедуру копирования данных, однако при этом появляется вероятность того, что вызываемая функция может испортить передаваемые в нее данные.

Покажем это на примере программы, приведенной на рис. 9.4.

```
#include<iostream.h>
#include <conio.h>
int multiply(int, int &);
main( )
{
clrscr();   //очистка экрана
int x1,y1,z;
    cout<<"\nВведите первый сомножитель=;
    cin>>x1;
    cout<<"\nВведите второй сомножитель=";
    cin>>y1;
    cout<<"\nРезультат ="<<myltiply(x1, y1) ;
    cout<<"\nx1="<<x1<<"\ny1="<<y1;
    cout<<"\n ---------------------------- \n";
    cout<<"\nНажмите любую клавишу…";
getch();
}
int multiply(int x, int &y)
{
return ++x * ++y;
}
```

Результаты работы программы при введенных значениях x1=1 и y1=2: Результат= 6 x1=1, y1=3

Рис. 9.4

Приведенные данные показывают, что при передаче параметров по ссылке их исходные значения портятся. Так, исходное значение переменной **y1** было равно 2, а после вызова функции оно стало равным 3, тогда как значение переменной **x1**, передаваемой по значению, осталось неизменным. Это объясняется тем, что при вызове функции multiply в выражении **++x** увеличивается локальная копия

первого фактического параметра, тогда как в **++y** фактический параметр увеличивается сам. Поэтому функции, которые изменяют свой передаваемый по ссылке параметр, трудно интерпретируются, и их по возможности лучше избегать. Однако при наличии громоздких аргументов типа больших массивов целесообразно использовать передачу параметров по ссылке, запрещая функции изменять значения аргументов. Это может быть осуществлено передачей в функцию аргументов, как констант, что достигается установкой ключевого слова **const** перед соответствующими переменными в списке аргументов.

Для этого заголовок функции **myltiply( )** должен иметь следующий вид:

int multiply(int x, const int &y).

Указание ключевого слова **const** в прототипе функции не обязательно. При таком описании функции аргумент **y** не будет копироваться при вызове функции, но внутри функции изменить значение **y** будет невозможно. На любую попытку изменить аргумент **y** компилятор выдаст сообщение об ошибке.

Альтернативной формой передачи параметра по ссылке является использование указателей. При этом адрес переменной передается в функцию не операцией адресации (**&**), а операцией косвенной адресации (*****). В списке параметров подобной функции перед именем переменной указывается символ **(*)**, свидетельствуя о том, что передается не сама переменная, а указатель на нее. В теле функции тоже перед именем параметра ставится символ разыменовывания (*), чтобы получить доступ через указатель к самой переменной. А при вызове функции в нее в качестве аргумента должна передаваться не сама переменная, а ее адрес, полученный с помощью операции адресации **&.**

Приведем пример рассмотренной ранее функции **myltiply(**) на рис. 9.4, но с передачей параметра по ссылке с помощью указателя:

```
#include<iostream.h>
#include <conio.h>
int multiply(int ,int *); // прототип функции
main()
```

```
{
clrscr(); //очистка экрана
int x1,y1,z;
     cout<<"\n Введите первый сомножитель=";
     cin>>x1;
     cout<<"\nВведите второй сомножитель=";
     cin>>y1;
     z=myltip(x1, &y1); // вызов функции
     cout<<"\nРезультат ="<< z;
     cout<<"\n ------------------------------ \n";
     cout«"\nНажмите любую клавишу…";
     getch();
}
int multip(int x, int *y) // заголовок функции
{
return *y*x; // изменение значения параметра
}
```

Рис. 9.5

10. Указатели

10.1. Назначение указателей

Указатели позволяют применять язык **C++** в самом широком диапазоне задач - от драйверов устройств на уровне аппаратного обеспечения и управляющих систем реального времени до операционных систем и компиляторов, анимации и мультимедийных приложений. **C++** - это идеальный инструмент для решения задач в поразительно широком многообразии прикладных областей.

Указатель - это особый тип переменной, содержащей в памяти адрес того элемента, на который он указывает. При этом имя элемента отсылает к его значению прямо, а указатель косвенно. Поэтому ссылка на значение посредством указателей называется косвенной адресацией. Смысл использования указателей состоит в том, что, отводя место только для запоминания адреса в памяти, Вы получаете идентификатор (переменную типа указатель), который может ссылаться на любой элемент в памяти указанного типа, причем в разный момент указатель мо-

жет указывать на разные элементы, т.е. не надо перемещать в памяти с места на место кучи байтов, достаточно просто изменить указатель.

Указатель может указывать на любые объекты: переменные, массивы, классы, структуры, и даже на функции. При объявлении указателя необходимо указать тип элемента, на который он будет указывать. Сам указатель занимает ровно столько места, сколько требуется для хранения адреса элемента. Признаком указателя является символ (*), который означает, что следующая за ним переменная является указателем. Рассмотрим примеры объявления указателей для различных типов переменных.

int *r; - *указатель на целое число,*
float*f; - *указатель на действительное число***,**
char*ch; - *указатель на символьную переменную***,**
long *g[10]; - *массив из 10указателей на длинное целое***,**
long (*t)[10]; - *указатель на массив из десяти длинных целых***,**
int(*fun)(double, int); *указатель на функцию с именем* **fun.**

Для инициализации указателей используется операция присваивания "=". Присваивание начального значения означает, что при определении указателя ему следует присвоить либо адрес объекта, либо адрес конкретного места в памяти, либо число 0. Примеры каждого типа инициализации выглядят так:

1. Присваивание указателю адреса существующего объекта:
   - С помощью операции получения адреса:

int a=5; // *целая переменная*
int*p = &a; // *указателю присваивается адрес переменной a*
int*p (&a); // *то же самое другим способом*

   - С помощью значения другого инициализированного указателя:

int*r = p;

   - С помощью имени массива или функции, которые трактуются как адрес:

int b[10]; // *массив*
int *t = b; // *присвоение адреса начала массива*

2. Присваивание указателю адреса области памяти в явном виде:
   char*z = (char*)0x00000417;
   Здесь **0x00000417** - шестнадцатеричная константа **,**
   (char*) - операция приведения константы к типу указатель на **char.**

3. Присвоение пустого значения:
   int*p = 0;

Поскольку гарантируется, что объектов с нулевым адресом нет, пустой указатель можно использовать для проверки, с помощью которой можно оценить: ссылается указатель на конкретный объект или нет.

4. Выделение участка динамической памяти и присвоение ее адреса указателю с помощью операции **new:**

- **int*n = new int;** // 1
- **int*m = new int(10);** // 2
- **int*q = new int[10];** // 3

В операторе 1 операция **new** выполняет выделение достаточного для размещения величины типа **int** участка динамической памяти и записывает адрес начала этого участка в переменную **n.** В операторе 2, кроме того, производится инициализация выделенной памяти значением 10. В операторе 3 операция **new** выделяет память под массив из 10 элементов и записывает адрес начала этого участка в переменную **q,** которая может трактоваться как имя массива.

10.2. Операции над указателями

С указателями связаны два специальных оператора: **&** и*. Обе эти операции унарные, т.е. имеют один операнд, перед которым они ставятся. Операция **&** соответствует действию "взять адрес". Операция * соответствует словам "значение, расположенное по указанному адресу". Например:

```
int y = 5;
int *py;
py = &y; .
```

Здесь оператор **py = &y;** присваивает адрес переменной **y** указателю **py.** Говорят, что переменная **py** указывает на **y.** Оператор * обычно называют оператором косвенной адресации, или операцией разыменования, возвращающей значение объекта (элемента), на который указывает ее операнд (т.е. указатель). Например, оператор **cout << *py <<"\n";** печатает значение переменной **y,** а именно 5.

Использование * указанным способом позволяет обеспечить доступ к величине, адрес которой хранится в указателе.

На рис. 10.1 приведен листинг программы, в которой рассматриваются примеры использования операций **&** и *****.

```
#include<iostream.h>
#include <conio.h>
main()
{
   int a;            // a - целое число
   int *pa ;         // pa - указывает на адрес целого числа
   clrscr();
   a = 7;
   pa = &a;          // pa устанавливаем равным адресу переменной a

   cout<<"\nАдрес a:"<<&a<<"\n"<<"Значение pa:"<<pa<<"\n"; cout<<"\nЗначение
   a :"<<a<<"\n"<<" Значение * pa :"<<*pa<<"\n\n";
   cout<<"\nДоказывают, что & и * дополняют друг друга"
       <<"\n"<<"&*pa="<<&*pa<<"\n"<<"*&ap="<<*&pa<<"\n"; cout<<"\n\n";
   cout<<"\nНажмите любую клавишу
  getch();
  return 0;
}
```

Значение **pa:0xfff4**
Значение **a: 7**
Значение ***pa: 7**
Доказательство того, что **&** и * дополняют друг друга
&*pa: 0xfff4
***&pa: 0xfff4**

Рис. 10.1

Программа на рис. 10.1 демонстрирует операцию с указателями. Адреса ячеек памяти выводятся как шестнадцатеричные целые с помощью оператора вывода **cout.** В программе показано, что адрес переменной **a** и значение указателя **pa** идентичны, а операция разыменовывают указателя ***pa** выводит на печать значение переменной **a.** Операции **&** и ***** взаимно дополняют друг друга. При их пооче-

редном применении к **pa** в любой последовательности будет напечатан один и тот же результат.

10.3. Выражения и арифметические действия с указателями

Указатели могут использоваться как операнды в арифметических выражениях, выражениях присваивания и выражениях сравнения [3]. Число арифметических операций с указателями ограничено. Указатели можно увеличивать (++), уменьшать (--), складывать с указателем целые числа (+ или +=), вычитать из него целые числа (- или -=) или вычитать один указатель из другого. Арифметические действия над указателями имеют свои особенности. Рассмотрим это на простейшем примере (рис. 10.2).

```
#include<iostream.h>
#include <conio.h>
main()
{
  int   x; // x - целое число
  int *p ,*p1;   // указывают на целые числа
  clrscr();   // очистка экрана
  p = &x; // указателю присваивается адрес целого числа x p1=p+3;
    cout<<"\nНачальное значение p:"<<p<<"\n";
    cout«"Конечное значение ++p:"<<++p<<"\n";
    cout<<"Конечное значение --p:"<<--p<<"\n";
    cout<<"Конечное значение   p1:"<<p1<<"\n";
    cout<<"\n\n";
    cout<<"\nНажмите любую клавишу…";
    getch();
    return 0;
}
```

Результаты работы программы:

Начальное значение **p: 0xfff4**
Конечное значение ++**p: 0xfff6**
Конечное значение --**p: 0xfff4**
Конечное значение   **p1: 0xfffa**

Рис. 10.2

По результатам выполнения этой программы мы видим, что при операции ++**p** значение указателя **p** увеличивается не на 1, а на 2. И это правильно, так как

новое значение указателя должно указывать не на следующий адрес, а на адрес следующего целого. А целое, как известно, занимает два байта памяти. Если бы базовый тип указателя был не **int,** а **double,** то были бы напечатаны адреса, отличающиеся на 8, именно столько байт памяти занимает переменная типа **double,** т.е. при каждой операции **++p** значение указателя будет увеличиваться на количество байт, занимаемых переменной базового типа указателя, а при операции **-- p** соответственно уменьшаться. К указателям можно прибавлять или вычитать некоторое целое. В данной программе указатель **p1** представляет собой сумму значений указателя **p** и целого числа 3. Результат равен **0xfffa,** т.е. увеличился на 6 по сравнению с исходным значением указателя **p.** Общая формула для вычисления значения указателя после выполнения операции **p = p + n;** будет иметь вид < **p** > = < **p** > + **n***< количество байт памяти базового типа указателя>. Можно так же вычитать один указатель из другого. Так, если **p** и **p1** - указатели на элементы одного и того же массива, то операция **p - p1** дает такой же результат, как и вычитание соответствующих индексов массива. Указатели можно сравнивать, при этом применимы все 6 операций:

<, >, <=, >=, =, ==, !=.

Сравнение **p<g** означает, что адрес, находящийся в **p,** меньше адреса, находящегося в **g.** Указателю можно присваивать значение другого указателя, а можно явно задать присваивание указателю адреса переменной того типа, на который он указывает. Для этого используют операцию взятия адреса **&:**

int r=10; int *t=&r; .

Здесь мы явно инициализировали указатель так, чтобы он указывал на переменную. Для того чтобы можно было использовать то, на что ссылается указатель через его имя, используют оператор разадресации *, ***т.е. при объявлении переменной символ * служит признаком указателя, а при реализации * служит знаком использования данных по адресу, если стоит перед указателем.*** Пример использования операции разадресации показан на рис. 10.3.

```
#include <iostream.h>
void main()
{
int a=1, b=2;
int *c=&b, *d=&a;
cout<< a <<' '<< b<<' '<<*c«' '<< d<<' '<< c<<' '<< *d<<' '<< "\n";
}
```

Результаты работы программы:

a=1, b=2, *c=2, d=0xfff2, c=0xfff4, *d=1.

Рис. 10.3

11. Массивы

Массивы - это группа элементов одинакового типа **(double, float, int** и т.п.), хранящихся под одним именем. Массив характеризуется своим именем, типом хранимых элементов, размером (количеством элементов), нумерацией элементов и размерностью. Различают одномерные и многомерные массивы. Основная форма объявления массива размерности N имеет следующий формат:

тип < имя массива> [размер 1][размер2]... [размер N];

тип - базовый тип элементов массива,

[размер1][размер2]... [размер N] - количество элементов одномерных массивов, входящих в многомерный массив.

11.1. Одномерные массивы

Чаще всего используются одномерные массивы, форма объявления которых будет иметь вид **Тип <имя массива> [размер].** Например, оператор **int A[10];** объявляет массив с именем **A,** содержащий **10** целых чисел. Доступ к элементам массива осуществляется выражением **A[i]** , где **i** - индекс элементов массива, который начинается с нуля и в данном примере заканчивается цифрой 9. Поэтому элемент **A[0]** характеризует значение первого элемента массива, **A[1]** - второго, **A[9]** - последнего. Объявление массива можно совмещать с заданием элементам массива начальных значений. Эти значения перечисляются в списке инициализации после знака равенства, разделяются запятыми и заключаются в фигурные скобки, например: **int A[10] = {1,2,3,4,5,6,7,8,9,10};.** Элементы массива могут иметь любой

тип. Так, например, оператор **char S[10];** объявляет массив из символов. Массив символов - это фактически строка, и число символов, помещаемых в строку, должно быть на единицу меньше объявленного размера массива. Это обусловлено тем, что строка кончается нулевым символом и будет, к примеру, иметь вид **char S[10] = {"abcdefghi\0"};.** Нулевой символ в конце можно не указывать, поэтому нормально будет воспринято такое объявление: **char S[10] = {"abcdefghi"};.**

11.2. Многомерные массивы

Многомерным называется массив, элементами которого являются одномерные массивы. Например, двумерный массив может быть объявлен таким образом: **int A2[10][3];.** Этот оператор описывает двумерный массив, который можно представить себе как таблицу, состоящую из 10 строк и 3 столбцов. Доступ к значениям элементов многомерного массива обеспечивается через индексы, каждый из которых заключается в квадратные скобки. Например, A2[3][2] - значение элемента, лежащего на пересечении четвёртой строки и третьего столбца (напоминаем, что индексы начинаются с 0). Если многомерный массив инициализируется при его объявлении, список значений по каждой размерности заключается в фигурные скобки. Приведённый ниже оператор объявляет и инициализирует двумерный массив **A2** размерностью 3 на 5:

int A2[3][5] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};,

однако допустим и упрощенный способ инициализации:

int A2[3][5] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};.

11.3. Примеры использования массивов

Так как массив является структурным типом, то основные операции с ним необходимо проводить с помощью оператора цикла. Покажем это на ряде примеров. В примере на рис. 11.1 с помощью оператора цикла осуществляется присвоение начальных нулевых значений элементам массива **n[5],** содержащего пять целых чисел. Печать массива осуществляется в табулированном формате. Первый оператор вывода печатает на экране заголовки столбцов, а второй выводит эле-

менты массива и их значения. Функция **setw()** указывает ширину поля, в котором будет выведено следующее значение.

```
#include<iostream.h>
#include <conio.h>
#include <stdlib.h>
#include<iomanip.h>
main( )
{
  clrscr();                          // очистка экрана
  int n[5];
  for(int i=0;i<5;i++)n[i]=0;        // обнуление элементов массива
  cout<<"элемент"«setw(13)<<"значение\n"; // вывод заголовков
  for(i=0;i<5;i++)
     cout<<setw(7)<<i<<setw(13)<<n[i]<<"\n"; // вывод элементов
  cout<<"\nНажмите любую клавишу…";
  getch();
  return 0;
}
```

Результаты работы программы:

элемент	значение
0	0
1	0
2	0
3	0
4	0

Рис. 11.1

В примере на рис. 11.2 реализуются операция копирования массивов **(mes, mcopy)** и вывод их элементов на экран дисплея. При копировании элементам массива **mcopy** последовательно (в цикле) присваиваются значения элементов массива **mes.** При этом для доступа к элементам массива используются индексы. Вывод элементов массива также осуществляется в цикле.

```
#include<iostream.h>
#include<iomanip.h>
#include <conio.h>
#include <stdlib.h>
main()
{
clrscr(); int i;
```

```
int mes[12]={31,28,31,30,31,30,31,31,30,31,30,31}; // инициализация массива
int mcopy[12];
for(i=0;i<12;i++) mcopy[i]=mes[i];                    // копирование массива
cout<<"Исходный массив "<<setw(29)<<" Скопированный   массив"; //вывод
                                                              //заголовков
for(i=0;i<12;i++) cout<<"\nmes="<<mes[i]<<"\t\t\tmcopy="<<mcopy[i]; //вывод
                                                              //массивов
Cout<<"\nНажмите любую клавишу…";
getch();
return 0;
}
```

Результаты работы программы:

Исходный массив	Скопированный массив
mes= 31	mcopy =31
mes =28	mcopy =28
…	…
mes = 31	mcopy =31

Рис. 11.2

На рис. 11.3 приведен пример вычисления одномерного массива y[10] по заданному массиву x[10], элементы, которых связаны следующим соотношением:

$$y[i] = (x[i]+a)/sqrt((x[i]^2+1));.$$

```
#include<iostream.h>
#include <conio.h>
#include <math.h>
main()
{
float   a=0;
float y[10], x[10];
int i=0;
clrscr();                                          // очистка экрана
cout<<"\nВведите a : ";
cin>>a;                                            //ввод переменной a
for( i=0; i<10;i++)
      {
      cout<<"\nВведите x["<<i<<"] : ";
      cin>>x[i];                                   //ввод массива x[i]
      y[i]= (x[i] + a) / sqrt((x[i]*x[i])+1);
      cout<<"\t\t\t y["<<i<<"] ="<< y[i] ;        //вывод  массива  y[i]
      }
```

```
cout<<"Нажмите любую клавишу…";
getch();
}
```

Рис. 11.3

В данной программе ввод переменной **a** и элементов массива **x[i]** организован в режиме диалога.

В примере на рис. 11.4 аналогичная задача решается для двумерных массивов. В этой программе ввод и вывод массивов осуществляется с помощью вложенных циклов.

```
#include<iostream.h>
#include <conio.h>
#include <math.h>
main()
{
  float a=0;
  float y[3][2], x[3][2]; int i=0, j=0;
  clrscr();
  cout<<"\nВведите a : ";
  cin>> a;                                          // ввод переменной a
  for( i=0; i<3; i++)
   {
     for( j=0; j<2; j++)
     {
     cout<<"\nВведите x["<<i<<","<<j<<"]:" ;
     cin>> x[i][j];                                 // ввод элементов массива x[i][j]
     y[i][j]=(x[i][j] + a)/sqrt((x[i][j] * x[i][j])+1);
     cout<<"\t\t\t y["<<i<<","<<j<<"]="<<y[i][j]; // вывод элементов массива
     }
     cout<<"\n";                                    //вывод пустой строки
   }
  cout<<"\n\n";
  cout<<"\nНажмите любую клавишу ...";
  getch();
  return 0;
}
```

Рис. 11.4.

На рис. 11.5 приведен пример программы, в которой вычисляется количество положительных, отрицательных и нулевых элементов одномерного массива.

Количество положительных элементов суммируется в переменной **p**, отрицательных - в переменной **n**, а нулевых - в переменной **zero**.

```
#include <conio.h>
#include <math.h>
#include<iostream.h>
main()
{
float a[10];
int   i=0, n=0, p=0, zero=0;                    // обнуление переменных
clrscr();
cout<<"\nОпределить количество положительных и отрицательных элементов
            массива a[10]\n";
 for( i=0; i<10;i++)
{
cout«"\nВведите a["<<i+1<<"] : ";
cin>>a[i];
}
for( i=0; i<10; i++)
{
   if( a[i] > 0) p += 1; // определение количества положительных .элементов
   if( a[i] < 0) n += 1; // определение количества отрицательных элементов
   if( a[i] == 0) zero += 1; // определение количества нулевых элементов
}
cout<<"\n\n";
cout<<"\nЧисло положительных элементов ="<<p;
cout<<"\nЧисло отрицательных элементов ="<<n;
cout<<"\nЧисло нулевых элементов ="<<zero;
cout<<"\n\n";
cout<<"\nНажмите любую клавишу…";
getch();
}
```

Рис. 11.5.

Для задания размера массива часто удобно использовать константы. В примере на рис. 11.6 число строк двумерного массива задается константой **row,** а число столбцов - константой **col.** Использование констант для задания размера массивов делает программу более наглядной и масштабируемой, так как при любом изменении размеров массива в программе достаточно будет изменить только значения констант. Этот прием наиболее эффективен в больших программах. В данной

программе решается задача вычисления количества положительных, отрицательных и нулевых элементов двумерного массива размерностью 2 на 3.

```
#include<iostream.h>
#include <conio.h>
#include <math.h>
 #define row 2 // строки
#define col 3   // столбцы
main()
{
   float b[row][col];                              // объявление массива
   int i=0, j=0,n=0, p=0, zero=0;
   clrscr();
   cout<<"\nОпределить количество положительных и отрицательных элементов";
   cout<<"\n массива b["<<row<<","<<col<<"]\n";
   for( i=0; i<row; i++){
       for( j=0; j<col; j++){
          cout<<"\n Введите b["«i+1«","«j+1<<"]=";
          cin>> b[i][j];                           // ввод элементов двумерного массива
       }
       cout<<"\n\n";
   }
   for(i=0; i<row; i++) {
       for(j=0; j<col; j++){
            if(b[i][j] > 0) p += 1;
            if(b[i][j] < 0) n += 1;
            if(b[i][j] == 0) zero += 1;
       }
   }
  cout<<"\nЧисло положительных элементов = "<<p;
  cout<<"\nЧисло отрицательных элементов = "<<n;
  cout<<"\nЧисло нулевых элементов = "<<zero;
  cout<<"\nНажмите любую клавишу…";
  getch();
  return 0;
}
```

Рис. 11.6.

11.4. Массивы и функции

При передаче массива в функцию в качестве параметра в заголовке функции необходимо указывать тип и имя массива с последующими пустыми квадратными скобками, а также тип и имя переменной, определяющей размерность массива. В

прототипе функции имена массива и переменной могут быть опущены. На рис. 11.7 приведена программа вычисления минимальной компоненты вектора с использованием функции vec( ), в которой заголовок функции имеет вид **vec(int x[ ], int k),** а аргументы **int x[ ]** и **int k** соответствуют имени массива и его размерности. В прототипе функции **vec(int, int)** эти имена опущены.

```
#include<iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#define n 5
vec(int [ ], int); // прототип функции вычисления min компоненты вектора
main()
{
  int y[n], i, min;
  clrscr();
  for(i=0;i<n;i++){
      cout<<"\nВведите y["<<i<<"]=";
      cin>>y[i];                          // ввод элементов вектора
   }
  min=vec(y, n);                          // вызывающая функция
  cout<<"\nМинимальная компонента = "<<min;
  cout<<"\nНажмите любую клавишу…";
  getch();
  return 0;
}
vec(int x[ ], int k)                      // заголовок функции
{
      int min1=x[0];
      int i;
      for(i=1; i < k;i++)
         if(x[i]<min1) min1=x[i];
      return min1;
}
```

Рис. 11.7.

Для передачи многомерных массивов в функцию необходимо учитывать ряд особенностей, которые связаны с тем, что при описании функции необходимо указывать размерность второго индекса и всех последующих индексов массива. Размерность первого индекса многомерного массива также не указывается, но

значения размерностей всех последующих индексов необходимы. Это обусловлено тем, что в памяти компьютера элементы массивов хранятся последовательно, независимо от количества индексов (размерности массива), а знание величин индексов дает возможность компилятору сообщить функции о том, как расположены элементы в массиве. Пример программы, в которой функция **mas** принимает двумерный массив как параметр, приведен на рис. 11.8.

```
//Вычисление минимальной компоненты двумерного массива
#include<iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#define n 2                          // число строк
#define m 3                          // число столбцов
mas(int [ ][m], int, int);           // прототип функции
main( )
{
  int y[n][m], i, j, min ;
  clrscr();                          // очистка экрана
  for(i=0;i<n;i++)
     for(j=0;j<m;j++)
    {
      cout<<"\nВведите y["<<i<<"]["<<j<<"]=";
      cin>>y[i][j];                  // ввод массива
    }
  min=mas(y, n, m);                  // обращение к функции и получение результата
  cout<<"\n\nМинимальная компонента = "<<min;
  cout<<"\nНажмите любую клавишу…";
  getch( );
  return 0;
}
mas(int x[ ][m], int k, int d)       //заголовок функции
 {
      int min1=x[0][0];
      int i, j;
      for( i=0; i < k; i++)
        for(j=0;j<d;j++){
         if( x[i][j]<min1) min1=x[i][j];
        }
      return min1;
  }
```

Рис. 11.8.

В этой программе значения индексов массива задаются с помощью констант **n** и **m,** а в заголовке функция описывается следующим образом:

mas(int x[][m], int k, int d),

где **m** - значение второго индекса массива, а **k** и **d** - фиктивные параметры, численные значения которым присваиваются после вызова функции с помощью оператора вида **mas(y, n, m);.**

В прототипе функции также необходимо указывать размерность второго индекса, в соответствии с выражением **mas(int [][m], int, int).**

11.5. Массивы и указатели

Массивы и указатели в языке **C++** тесно связаны и могут использоваться почти эквивалентно. Так, имя массива является константным указателем на первый элемент массива, а указатели можно использовать для выполнения любой операции, включая индексирование массива. Пусть сделано следующее объявление: **int b[5] = {1,2,3,4,5}, *p;,** которое означает, что объявлены массив целых чисел **b[5]** и указатель на целое **p.** Поскольку имя массива является указателем на первый элемент массива, можно задать указателю **p** адрес первого элемента массива с помощью оператора **p = b;** . Это эквивалентно присвоению адреса первого элемента массива другим способом: **p = &b[0];.** Теперь можно сослаться на элемент массива **b[3]** с помощью выражения ***(p+3).** В этой записи скобки необходимы, потому что приоритет ***** выше, чем +. На рис. 11.9 показан пример программы, которая с помощью указателей заносит данные в одномерный массив **a[5].** Определяет сумму и количество положительных элементов. Выводит на экран полученный массив и адреса его элементов, а также результаты расчетов.

```
#include <iostream.h>
#include <conio.h>
void main()
{ clrscr();
  int a[5], sum = 0, *p;
  int kol = 0, i;
  p = &a[0];                    // инициализация указателя адресом первого элемента
  cout << " Ввод данных в массив a[ ]\n";
  for ( i = 0; i <5; i++){
```

```
      cout << " a [ " << i << " ] = ";
      cin >> *(p+i);                    // разыменовывание смещенного указателя
    }
   // расчет суммы и количества положительных элементов
   for ( i = 0; i < 5; i ++)
     if ( *(p+ i) > 0 ) {
        sum += *( p+i );
        kol ++;
     }
    // вывод исходных данных и результатов
    cout << "\n\n\n Элемент массива Адрес элемента массива \n";
    for ( i = 0; i < 5; i++ ){
       cout << *( p+ i) << "\t\t " << (p+i) << "\n";          // вывод результатов
    }
    cout << "\nсумма = " << sum << "^количество = " << kol; cout<<"\n\n";
    cout<<"\nНажмите любую клавишу…";
    getch();
}
```

Результаты работы программы:

Ввод данных в массив **a[ ] :**

a[0]=1
a[1]=2
a[2]=5
a[3]=5
a[4]=4

Элемент массива	Адрес элемента массива
1	Oxffec
2	Oxffee
3	Oxfff0
4	Oxfff2
5	Oxfff4

сумма = 17
количество=5

Рис. 11.9.

На рис.11.10 приведен пример программы, в которой с помощью указателей формируется двумерный массив **a[2][2],** а из минимальных элементов его столбцов формируется массив **b[2].** Значения полученных массивов выводятся на дисплей.

```
#include <iostream.h>
#define I 2
```

```
#define J 2
#include <conio.h>
void main()
{
   clrscr();
   int a[I][J], b[J], min, *p ;
   int i,j;
   p = &a[0][0];                    // инициализация указателя адресом первой ячейки
   cout << "Введите данные в массив a["<< I <<"]["<<J<<"]:\n";
   for ( i = 0; i < I; i++ )
    for ( j = 0; j < J; j++ ){
      cout << "a[" << i << "][" << j << "]=";
      cin >> *(p + i*I + j);          // ввод массива
    }
   // расчет массива b[2]
   for ( j = 0; j < J; j++ ) {      // цикл по столбцам
       min = *(p + j); // присваивание   min значения первого элемента столбца
       for (i = 1; i < I; i+)       // цикл по строкам, начиная со второго элемента
          if ( ( *(p + i*I + j)) < min) min = *(p + i*I + j);
          *(b + j) = min;
   }
   cout << "nВывод исходного массива a[,]:";
   for (i = 0; i < I;i++){
     cout << "\n";
     for ( j = 0; j < J;j++){
       cout <<"\t"<< *(p + i*I + j);
     }
   }
   cout << "\n\nВывод полученного массива b[]:\n";
   for ( j = 0; j < J; j++ ){
    cout <<"\t"<< *( b + j);
   }
   cout<<"\n\n";
   cout<<"\nНажмите любую клавишу…";
   getch();
}
```

Результаты работы программы:

Введите данные в массив **a[2][2]:**

a[0][0]=1
a[0][1]=4
a[1][0]=5
a[1][1]=3

Вывод исходного массива **a[2][2]:**

1 4

5 3

Вывод полученного массива **b[]:**

1 3

Рис. 11.10.

12.Форматирование ввода-вывода

12.1. Форматированный ввод-вывод

В языке C++ задача форматирования решается с помощью манипуляторов потока.

Манипуляторами называются функции, которые можно включать в цепочку операций ввода и вывода для форматирования данных.

Они позволяют выполнять следующие операции: задание ширины полей, задание точности, установку и сброс флагов формата и т. д. Манипуляторы потоков **dec, oct и hex** задают основания чисел. Чтобы установить шестнадцатеричный формат представления элементов данных (с основанием 16), необходимо в потоке использовать манипулятор **hex.** Манипулятор **oct** используется для установки восьмеричного формата представления данных, а манипулятор **dec** осуществляет возврат представления данных к десятичному формату. Установленный формат потока сохраняется до тех пор, пока он не будет изменен явным образом. В программе на рис. 12.1 показано использование манипуляторов потока **dec , oct и hex.**

```
// Пример программы на использование манипуляторов потока
#include <iostream.h>
#include <conio.h>
main()
{
  int n;
  clrscr();
  cout<<" Введите десятичное число"<<" ";
  cin>>n;
  cout <<n<<" "<<"В шестнадцатеричном формате равно <<hex«n«endl
     <<" В восьмеричном формате =" <<oct<<n<<endl
     <<" В десятичном формате= "<<dec<<n<<endl;
  cout<<"\n Нажмите любую клавишу…";
```

```
  getch();
  return 0;
}
```

Результаты выполнения программы:

Введите десятичное число 20
20 В шестнадцатеричном формате равно 14
В восьмеричном формате = 24
В десятичном формате = 20

Рис. 12.1

Точностью чисел с плавающей запятой, выводимых на печать, т.е. числом разрядов справа от десятичной точки, можно управлять с помощью манипулятора потока **setprecision** или функции-элемента **precision.** Вызов любой из этих установок точности действует для всех последующих операций вывода до тех пор, пока не будет произведена следующая установка точности. Функция-элемент **precision** не имеет никаких аргументов и возвращает текущее значение точности. Программа на рис. 12.2 является примером использования как функции-элемента **precision,** так и манипулятора **setprecision** для печати таблицы корня квадратного из числа 2 с точностью, варьируемой от 0 до 9. Величина 0 (по умолчанию) соответствует значению точности, равной 6.

```
// Управление точностью печати чисел с плавающей запятой
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
#include <math.h>
main()
{
  int d;
  double root2 = sqrt(2.0);
  clrscr();
  cout<<" корень квадратный из 2 с точностью d от 0 до 9"<<endl
     <<"с помощью функции-элемента precision"<<endl;
  for (d=0;d<=9;d++){
      cout.precision(d); cout<<root2<<endl
      ;
  }
  cout<<" с помощью манипулятора setprecision"<<endl;
  for (d=0;d<=9;d++){
      cout<<setprecision(d)<<root2<<endl ;
```

```
  }
 cout<<"\nнажмите любую клавишу…";
 getch();
}
```

Результаты выполнения программы:

корень квадратный из 2 с точностью **d** от 0 до 9

с помощью функции-элемента **precision**

1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136 1.41421356
1.414213352

с помощью манипулятора **setprecision**

1.414214
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213352

Манипулятор потока **setw** и метод **width** устанавливают ширину поля (т.е. число символов позиций, в которые значение будет выведено, или число символов, которые будут введены). Если обрабатываемые значения имеют меньше символов, чем заданная ширина поля, то для заполнения лишних позиций используются заполняющие символы. По умолчанию заполняющими символами являются пробелы, и они вставляются перед значащими символами, т. е. происходит выравнивание вправо. Если число символов в обрабатываемом значении больше, чем заданная ширина поля, то лишние символы не отсекаются и число печатается полностью. Установка ширины поля влияет только на следующую операцию поместить в поток; затем ширина поля устанавливается неявным образом на 0, т.е.

поля для представления выходных значений будут просто такой ширины, которая необходима. Функция **width** без аргументов возвращает текущую установку ширины поля. Заполняющие символы могут устанавливаться манипулятором **setfill.** Программа на рис. 12.3 демонстрирует влияние ширины поля на результат вывода числа 25:

```
// Пример программы на использование манипулятора setw
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
main()
{
  int n=25;
  clrscr();
  for(int i=0; i< 5; i++)
    cout<<setw(i)<<n<<endl;
   cout<<"\nНажмите любую клавишу…";
  getch( );
  return 0;
}
```

Результаты выполнения программы:

```
25
25
25
  25
    25
```

Рис. 12.3

Из этого результата видно, что пока ширина поля меньше числа символов в выводимом числе, она ни на что не влияет, а при большей ширине поля происходит выравнивание числа вправо. Такой же результат дает и программа на рис. 12.4, в которой используется метод **width.**

```
// Пример программы на использование манипулятора width
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
main()
{
  int n=25;
  clrscr();
  for(int i=0; i< 5; i++){
     cout.width(i);
```

```
    cout <<n<<endl;
  }
  cout<<"\nНажмите любую клавишу…";
  getch();
  return 0;
}
```

Рис.12.4

Если в программе на рис. 12.3 оператор **cout<<setw(i)<<n<<endl;** заменить на **cout<<setfill('*')<<setw(i)<<n<<endl;** , т.е. ввести модификатор **setfill,** то заполняющие символы изменятся, и результаты выполнения программы будут иметь вид:

```
25
25
25
*25
**25
```

12.2. Неформатированный ввод - вывод

Неформатированный ввод-вывод выполняется с помощью функций-элементов **read и write.** Функция **read** вводит в память, а функция **write** выводит из памяти некоторое число байтов символьного массива. Эти байты не подвергаются какому-либо форматированию. Они просто вводятся или выводятся в качестве сырых байтов данных. Например, вызов

```
char buffer[ ] = "Поздравляем с днем рождения ";
cout.write (buffer,12);
```

выводит первые 12 байтов символьного массива **buffer[ ].** Поскольку символьная строка указывает на адрес своего первого символа, то вызов

```
cout.write ("ABCDEFGYIJKLMNOPRST",10);
```

отобразит на экране первые 10 символов алфавита. Функция-элемент **read** вводит в память указанное число символов из массива. Функция-элемент **gcount** сообщает о количестве символов, прочитанных последней операцией ввода. Программа на рис. 12.5 показывает работу функций-элементов **read, write** и **gcount.** Она вводит 20 символов (из более длинной входной последовательности) в буфер памяти с помощью функции-элемента **read,** определяет число введенных символов с помо-

щью **gcount** и выводит символьный массив из буфера памяти с помощью **write.** Пример программы на неформатированный ввод- вывод показан на рис. 12.5.

```
// Пример программы на неформатированный ввод-вывод
#include <iostream.h>
#include <conio.h>
const int size = 80;
main()
{
   clrscr();
   char buffer[size] ;
   cout << "Введите предложение :"<<endl;
   cin.read(buffer, 20);
   cout << "Введенное предложение:"<<endl;
   cout.write (buffer ,cin.gcount());
   cout<<"\nНажмите любую клавишу…";
   getch();
   return 0;
}
```

Результаты выполнения программы:

```
Введите предложение:
Использование функций-элементов read, write, gcount
Введенное предложение:
Использование функций
```

Рис. 12.5

13. Область видимости переменных

Областью видимости (областью действия) переменной называется область программы, в которой на данную переменную можно сослаться. На некоторые переменные можно сослаться в любом месте программы, тогда как на другие - только в определенных ее частях [1]. Класс памяти определяется, в частности, местом объявления переменной. Локальные переменные объявляются внутри некоторого блока или функции. Эти переменные видны только в пределах того блока, в котором они объявлены. *Блоком называется фрагмент кода,* ограниченный фигурными скобками "{ }". Глобальные переменные объявляются вне какого-либо блока или функции. Спецификации класса памяти могут быть разбиты на два класса: автоматический класс памяти с локальным временем жизни и статический

класс памяти с глобальным временем жизни. Ключевые слова **auto** и **register** используются для объявления переменных с локальным временем жизни. Эти спецификации применимы только к локальным переменным. Локальные переменные создаются при входе в блок, в котором они объявлены, существуют лишь во время активности блока и исчезают при выходе из блока. Спецификация **auto,** как и другие спецификации, может указываться перед типом в объявлении переменных. Например: **auto float x, y;.** Локальные переменные являются переменными с локальным временем жизни по умолчанию, так что ключевое слово **auto** используется редко. Далее будем ссылаться на переменные автоматического класса памяти просто как на автоматические переменные.

Пусть, например, имеется следующий фрагмент кода:

```
{
int i = 1;
…
i++;
…
},
```

к которому в ходе работы программы происходит неоднократное обращение. При каждом таком обращении переменная **i** будет создаваться заново (под нее будет выделяться память) и будет инициализироваться единицей. Затем в ходе работы программы ее значение будет увеличиваться на 1 операцией инкремента. В конце выполнения этого блока переменная исчезнет и выделенная под нее память освободится. Следовательно, в такой локальной переменной невозможно хранить какую-то информацию между двумя обращениями к блоку. Спецификация класса памяти **register** может быть помещена перед объявлением автоматической переменной, чтобы компилятор сохранял переменную не в памяти, а в одном из высокоскоростных аппаратных регистров компьютера. Например: **register int i = 1;.**

Если интенсивно используемые переменные, такие как счетчики или суммы могут сохраняться в аппаратных регистрах, накладные расходы на повторную загрузку переменных из памяти в регистр и обратную загрузку результата в память

могут быть исключены. Это сокращает время вычислений. Компилятор может проигнорировать объявления **register.** Например, может оказаться недостаточным количество регистров, доступных компилятору для использования. К тому же оптимизирующий компилятор способен распознавать часто используемые переменные и решать, помещать их в регистры или нет. Так что явное объявление спецификации **register** применяется редко.

Ключевые слова **extern** и **static** используются, чтобы объявить идентификаторы переменных как идентификаторы статического класса памяти с глобальным временем жизни. Такие переменные существуют с момента начала выполнения программы. Для таких переменных память выделяется и инициализируется сразу после начала выполнения программы. Существует два типа переменных статического класса памяти: глобальные переменные и локальные переменные, объявленные спецификацией класса памяти **static.** Глобальные переменные по умолчанию относятся к классу памяти **extern.** Глобальные переменные создаются путем размещения их объявлений вне описания какой-либо функции и сохраняют свои значения в течение всего времени выполнения программы. На глобальные переменные может ссылаться любая функция, которая расположена после их объявления или описания в файле. Локальные переменные, объявленные с ключевым словом **static,** известны только в том блоке, в котором они определены. Но, в отличие от автоматических переменных, локальные переменные **static** сохраняют свои значения в течение всего времени выполнения программы. При каждом следующем обращении к этому блоку локальные переменные содержат те значения, которые они имели при предыдущем обращении.

Вернемся к уже рассмотренному выше примеру, но укажем для переменной **i** статический класс:

```
{
static int i = 1;
…
i++;
…
}
```

Инициализация переменной **i** произойдет только один раз за время выполнения программы. При первом обращении к этому блоку значение переменной **i** будет равно 1. К концу выполнения блока ее значение станет равно 2. При следующем обращении к блоку это значение сохранится и при окончании повторного выполнения блока **i** будет равно 3.

Таким образом, статическая переменная способна хранить информацию между обращениями к блоку и, следовательно, может использоваться, например, как счетчик числа обращений. Все числовые переменные статического класса памяти принимают нулевые начальные значения, если программист явно не указал другие начальные значения. Статические переменные - указатели тоже имеют нулевые начальные значения.

Спецификации класса памяти **extern** используются в программах с несколькими файлами. Пусть, например, в модуле **Unit1** или в файле **Unit1.cpp** объявлена глобальная переменная **int a = 5; .**

Тогда, если в другом модуле **Unit2** или в файле **Unit2.cpp** объявлена глобальная переменная **extern int a;,** то компилятор понимает, что речь идет об одной и той же переменной. И оба модуля могут с ней работать. Для этого даже нет необходимости связывать эти модули директивой **#include,** включающей в модуль **Unit1** заголовочный файл второго модуля.

Таким образом, ***область действия идентификатора*** - это часть программы, в которой на идентификатор можно ссылаться. Например, на локальную переменную, объявленную в блоке, можно ссылаться только в этом блоке или в блоке, вложенном в этот блок. Существует четыре области действия идентификатора - ***функция, файл, блок и прототип функции.***

Идентификатор, объявленный вне любой функции (на внешнем уровне), имеет область действия файл. Такой идентификатор известен всем функциям.

14. Работа с файлами

Файлы предназначены для постоянного хранения больших объемов данных, так как хранение данных в переменных и массивах является временным. Существует множество способов организации записей в файле, но наиболее распространенными являются [4]:

1. Файлы последовательного доступа.
2. Файлы произвольного доступа.

В файлах последовательного доступа записи хранятся в последовательности, соответствующей ключевому полю. Например, в платежной ведомости записи располагаются в последовательности, соответствующей идентификационному номеру служащего. Первая запись содержит наименьший идентификационный номер, а последующие записи располагаются в порядке возрастания идентификационных номеров (ключей). Ключ показывает, что запись относится к конкретному объекту, т. е. является уникальной среди других записей в файле. Структура файла последовательного доступа задается программистом в соответствии с требованием прикладных программ, так как язык **C++** не предписывает никаких требований к структуре файла. При этом могут быть использованы следующие классы файлового ввода-вывода:

ifstream - файлы для чтения ;
ofstream - файлы для записи ;
fstream - файлы для чтения и записи.

Чтобы приступить к записи в файл, нужно сначала создать объект **ofstream,** а затем связать его с определенным файлом на диске. В заголовке программы необходимо включить файл заголовка **fstream.h,** вместо файла **iofstream.h,** поскольку этот файл уже содержится в файле **fstream.h.** Для открытия файла в режиме диалога с помощью объекта **ofstream** нужно объявить экземпляр этого объекта, передав ему в качестве параметра переменную **fname,** определяющую имя файла **ofstream fout (fname);.** Открытие файла для чтения выполняется аналогичным образом, за исключением того, что для этого используется объект **ifstream fin**

(fname);. В этих выражениях задаются имена объектов **fout** и **fin,** которые можно использовать так же, как объекты **cout** и **cin** соответственно. По завершении работы с файлом (записи или чтения информации) его нужно закрыть с помощью функции **close().**

Пример программы создания простейшего файла последовательного доступа для чтения и записи показан на рис. 14.1.

```
1. #include <conio.h>
2. #include <fstream.h>
3. void main()
4. {
5. clrscr();
6. cout << "Введите имя файла для ввода: ";
   7. char fname[80];    //создание массива для записи имени файла
   8. cin >> fname;
   9. ofstream fout(fname);              // открыть файл для записи
10. if (!fout)                          // проверить открытие файла
11. {
12. cout << "Невозможно открыть файл";
13. return;
14. }
15. char c;
16. cout << "Вводите информацию :\n";
17. cout << "Конец ввода - символ *\n";
18. while ( fout )
19. {
20. fout.put(c);    // запись информации в файл
21. cin.get(c);
22. if ( c == '*')
23. break;
24. }
25. fout.close();
26. cout << "Введите имя файла для считывания : ";
27. cin >> fname;
28. ifstream fin (fname);
29. if ( !fin)
30. {
31. cout << "Невозможно открыть файл";
32. return;
33.   }
34. while ( fin )
35. {
36. fin.get(c);    // считывание информации с файла
```

```
37. cout << c;
38. }
39. fin.close();
40. cout<<"\nНажмите любую клавишу... ";
41. getch();
42. }
```

Рис. 14.1.

В строке 7 создается массив для записи имени файла, а в строке 8 предлагается ввести имя файла, которое записывается в массив **fname.** В строке 9 создается объект **ofstream** с именем **fout,** который связывается с введенным ранее именем файла. В строках 10-12 осуществляется проверка открытия файла. При неудачной попытке выводится сообщение: "Невозможно открыть файл". В строках 14-23 осуществляется запись вводимой информации в файл и контроль конца записи. Запись информации осуществляется с помощью функции **put(c),** которая посимвольно вводит данные в файл. Контроль конца записи реализуется с помощью функции **get(c),** которая считывает вводимые в файл данные и заносит их в символьную переменную **c.** Ввод информации в файл продолжается до тех пор, пока в переменной **c** не появится ключевой символ **'*'**, определяющий конец записи. В строке 25 файл закрывается. В строке 28, после введения имени в строке 26, файл открывается заново, но в этот раз для чтения, и его содержимое посимвольно выводится в программу в строках 34-37. В строке 39 файл закрывается окончательно.

Другой пример создания файла последовательного доступа показан на рис. 14.2. В этом примере для каждого клиента программа получает номер счета, имя клиента и сумму денег, которую клиент должен компании за товары и услуги, полученные в прошлом. Данные, полученные от каждого клиента, образуют записи, в которых номер счета используется в качестве ключа записи. Таким образом, файл будет создаваться, и обрабатываться, в соответствии с номером счета. В программе файл должен быть открыт для записи, так что создается объект класса **ofstream.** Конструктору объекта передаются два аргумента - имя файла и режим открытия файла. Для объекта **ofstream** режим открытия файла может быть или

ios::out - для записи данных в файл, или **ios::app** - для добавления данных в конец файла.

```
#include <iomanip.h>
#include <conio.h>
#include <fstream.h>
void outputLine(int,char*,float) ;
void main()
{
clrscr();
cout << "Введите имя файла для ввода: ";
char fname[80];       // создание массива для записи имени файла
cin >> fname;
ofstream fout(fname);   //открыть файл для записи
if (!fout) {     // проверить открытие файла
    cerr << "Невозможно открыть файл"<<endl ;
    exit(1);
}
cout << "Введите счет имя сумму"<<endl
<<"Введите EOF для окончания ввода "<<endl<<"?";
int accout;
char name[10];
float sum;
while(cin>>accout >>name>>sum) {
     fout<< accout<<" " <<name<<" "<<sum<<endl ;
     cout<<"?" ;
}
fout.close();
ifstream fin ("n1");
if ( ! fin) {
      cout << "Невозможно открыть файл";
      return;
}
cout<<setiosflags(ios::left)<<setw(10)<<" счет"<<setw(13)<<" Имя"
       <<" сумма "<<endl;
while ( fin>> accout>>name>>sum) {
   outputLine(accout, name, sum);     // запись информации в файл
}
fin.close();
cout<<"\nНажмите любую клавишу...";
getch();
}
void outputLine(int acct, char*name, float bal){
     cout<<setiosflags(ios::left)<<setw(10)<<acct<<setw(13)<<name<<setw(7)
     <<setprecision(2)<<setiosflags(ios::showpoint|ios::right)<<bal<<endl;
```

}

Рис. 14.2

Объявление **ofstream fout("clients.dat", ios::out);** создаёт объект **fout** класса **ofstream,** связанный с файлом **clients.dat,** который открывается для записи. По умолчанию объекты класса **ofstream** открыты для ввода, поэтому для открытия файла **clients.dat** для записи может быть использован оператор **ofstream fout ("clients.dat");**

Список режимов открытия файлов приведён в табл.14.1.

Таблица 14.1

Режим	Описание
ios::app	Записать все данные в конец файла
ios::ate	Переместиться в конец исходного открытого файла. Данные могут быть записаны в любом месте файла
ios::in	Открыть файл для чтения
ios::out	Открыть файл для записи

Объект класса **ofstream** может быть создан без открытия, какого-либо файла. В этом случае файл может быть связан с объектом позднее. Например, объявление **ofstream fout;** создает объект **fout** класса **ofstream.** Функция-элемент **open** класса **ofstream** открывает файл и связывает его с существующим объектом этого класса **fout.open ("clients.dat"); .**

После создания объекта класса **ofstream** и попытки открыть его программа проверяет, была ли операция открытия файла успешной. Фрагмент программы имеет вид

```
if (!fout) {
  cerr << " Невозможно открыть файл"<< endl ;
  exit(1);
}
```

Если файл не открывается, то об этом выводится сообщение и вызывается функция **exit()** для завершения программы. Аргумент 0 функции **exit( )** означает, что программа завершена нормально, а любое другое значение указывает среде окружения, что программа завершена по ошибке. Если файл открылся успешно, то программа начинает обрабатывать данные. Следующий оператор запрашивает

пользователя о вводе различных полей каждой записи или информацию о конце файла, если ввод данных завершен:

cout << "Введите счет имя сумму**"«endl**
<<"Введите **EOF** для окончания ввода **"<<endl<<"?";.**

Строка **while (cin>>accout >>name>>sum)** вводит каждый набор данных и определяет, не введен ли признак конца файла. Когда достигнут признак конца файла или вводятся неверные данные, условное выражение возвращает 0 и оператор **while** завершает свою работу. После этого файл закрывается явным образом с помощью функции-элемента **clouse: fout.close();.**

Для чтения файла последовательного доступа его открывают путем создания объекта класса **ifstream.** Объекту передаются два аргумента - имя файла и режим открытия файла. Объявление

ifstream fin ("clients.dat", ios::in);

создаёт объект **fin** класса **ifstream,** связанный с файлом **clients.dat,** который открывается для чтения. По умолчанию объекты класса **ifstream** открыты для чтения, поэтому для открытия файла **clients.dat** для чтения может быть использован оператор **ifstream fin ("clients.dat"); .**

Строка **while (fin>>accout >>name>>sum)** читает из файла набор данных, т.е. записи. Всякий раз, когда выполняется оператор **while,** считывается следующая запись из файла. Записи выводятся на экран с помощью функции **output(),** которая использует параметризованные манипуляторы потока для форматирования данных, изображаемых на экране. Когда достигается конец файла, входная последовательность в операторе **while** возвращает 0, и программа выходит из цикла. Файл закрывается оператором **close(),** и программа завершает свою работу.

При поиске данных в последовательном файле программа начинает чтение данных с начала файла и читает все данные последовательно до тех пор, пока не будут найдены требуемые данные. Это приводит к необходимости обрабатывать файл последовательно несколько раз (каждый раз с начала), если искомые данные расположены в разных местах файла. Классы **istream** и **ostream** содержат функ-

ции-элементы для позиционирования, которые определяют порядковый номер следующего байта в файле для считывания или записи. Этими функциями-элементами являются **seekg** (позиционировать при извлечении из потока) для класса **istream** и **seekp** (позиционировать при помещении в поток) для класса **ostream.** Кроме того, любой объект класса **istream** имеет указатель **get,** показывающий номер очередного вводимого в файл байта, а любой объект класса **ostream** имеет указатель **set,** показывающий номер очередного выводимого из файла байта. Оператор **fin.seekg(0);** позиционирует указатель на начало файла. Аргумент функции **seekg( )** обычно является целым типа **long,** а второй аргумент, который может быть задан, показывает направление позиционирования. Оно может быть **ios::beg** при позиционировании относительно начала потока (используется по умолчанию), **ios::cur** - для позиционирования относительно текущей позиции и **ios::end** - для позиционирования относительно конца потока.

Примеры:

```
// Позиционировать fileObject на n-й байт(полагаем ios::beg)
                fileObject . seekg(n );
// Позиционировать fileObject на n байтов вперед
                fileObject . seekg(n , ios::cur );
// Позиционировать fileObject на i-й байт от конца файла
                fileObject . seekg(n , ios::end);
// Позиционировать fileObject на конец файла
                fileObject.seekg(0 , ios::end);
```

Те же самые операции могут быть выполнены с помощью функции-элемента **seekp** класса **ostream.** Функции-элементы **tellg** и **tellp** возвращают текущие позиции указателей соответственно **get** и **set.** Следующий оператор присваивает переменной **location = fileObject.tellg();** значение указателя **get.**

Программа на рис. 14.3 позволяет менеджеру по кредитам отображать на экране информацию о клиентах с нулевой задолженностью, информацию о клиентах-задолжниках и клиентах, которым должна компания. Программа отображает меню и позволяет по каждой из трех опций получить соответствующую информацию по кредитам.

```
// Программа запроса кредитной информации
#include <iomanip.h>
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
void output( int , char* ,float ) ;
int main()
{
   clrscr();
   ifstream fin("Clients");     //открытие файла для записи
   if(!fin){
     cerr<<" файл не открыть";
     exit(1);
   }
   cout<<"Запрос на ввод \n"
    <<"1-список счетов с нулевым балансом\n"
    <<"2- список счетов с кредитным балансом\n"
    <<"3- список счетов с дебитовым сальдо\n"
    <<"4 -конец счета \n";
    int zapros;
    cin >>zapros;
    while(zapros !=4){
      int account;
      char name[10];
      float sum;
      fin>>account>>name>>sum;
      switch( zapros){
        case 1:
          cout<<endl<<" счета с нулевым балансом :"<<endl;
          while(!fin.eof()){
             if (suma==0)
                output(account, name, sum);
             fin>>account>>name>>sum;
          }
          break;
        case 2:
          cout<<endl<<" счета с кредитным балансом :"<<endl;
          while(!fin.eof()){
             if (suma<0)
                output(account, name, sum);
             fin>>account>>name>>sum;
          }
          break;
        case 3:
```

```
        cout<<endl<<" счета с дебетовым балансом: "<<endl;
        while(!fin.eof()){
           if (sum > 0)
              output(account, name, sum );
           fin>>account>>name>>sum;
        }
        break;
      }
      fin.clear();
      fin.seekg(0);
      cout<<endl<<"?";
      cin>> zapros ;
    }

    cout<<"\nНажмите любую клавишу…";
    getch();
    return 0;
}
void output(int a, char* n, float s)
{
     cout<<setiosflags(ios::left) <<setw(10) <<a<<setw(13) <<n<<setw(7)
     <<setprecision(2)<<setiosflags( ios::showpoint|ios::right) << s<<endl;
}
```

Рис. 14.3.

Опция 1 выводит список счетов с нулевым балансом. Опция 2 выводит список счетов с кредитным балансом. Опция 3 выводит список счетов с дебетовым сальдо. Опция 4 завершает выполнение программы. Пример вывода приведен на рис. 14.4.

```
Запрос на ввод
        1- список счетов с нулевым балансом
        2- список счетов с кредитным балансом
        3  список счетов с дебетовым сальдо
        4 -конец счета
?       1
        Счета с нулевым балансом :
        20      Петров      0.00
        25      Беляев      0.00

?       2
        Счета с кредитным балансом
        41      Иванов     -45.20
        58      Сидоров    -85.30
```

? **3**

Счета с дебетовым сальдо

5 **Шичко** **24.99**

87 **Райков** **345.83**

? **4**

Конец счета.

Рис. 14.4.

15. Структуры

Структура - это набор взаимосвязанных данных, возможно и разных типов, объединенных в единое целое. Возможностью объединять данные разных типов, структура отличается от массива, а массив, как известно, объединяет данные одного типа. Структуру целесообразно использовать в тех случаях, когда необходимо иметь одну переменную, содержащую набор взаимосвязанных данных. Например, для хранения списка сотрудников организации, удобно иметь одну переменную(**Kadry**), содержащую такие данные как фамилию сотрудника, его должность и отдел. При использовании структуры ее вначале необходимо объявить, а затем создать экземпляр этой структуры. Структура описывается с помощью ключевого слова **struct** [4]:

```
struct Kadry {      // описание структуры
   int nzap;        // номер записи
   char fam[20];    // фамилия
   char dol[10];    // должность
   int otdel;       // отдел
};
```

Идентификатор данной структуры **Kadry** определяет ее тег (имя). Элементы структуры называются членами-данными или полями. Каждый член структуры объявляется так же как обычная переменная. В приведенном примере структура содержит два массива типа **char** и два члена типа **int.** В описании структуры после закрывающейся фигурной скобки ставится точка с запятой. Когда структура опи-

сана, ее можно использовать, предварительно создав экземпляр структуры, которая выглядит следующим образом:

Kadry sotr ;

В результате для структуры выделяется необходимая память, и эта область памяти связывается с переменной, имеющей имя sotr. После чего можно присваивать значения членам-данным:

```
sotr. nzap = 1;
strcpy(sotr.fam, "Сидоров");
strcpy(sotr.dol, "лаборант");
sotr.otdel = 12;
```

Для доступа к членам-данным используется оператор доступа к членам структуры, который представляет собой точку между именем переменной и именем члена структуры. Оператор доступа позволяет обращаться к конкретному члену структуры, как для чтения, так и для изменения его значения. При желании можно сразу инициализировать все члены вновь созданного экземпляра структуры:

```
Kadry engineer = {
        2,
        "Петров", "ин-
        женер" , 15
}
```

Этот способ короче предыдущего, но он не всегда применим в реальных ситуациях. Обычно структура заполняется в результате ввода данных пользователем или чтения их из файла. В этих случаях присвоение значений подобным образом невозможно.

На рис. 15.1 показан пример ввода структуры **Vkladchik** пользователем. В этом примере форматированный вывод элементов структуры на дисплей осуществляется с помощью функции, прототип которой имеет вид **void output (int, char*, float);**.

//Пример ввода - вывода простейшей структуры на дисплей

```
#include <iomanip.h>
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
void output(int, char*, float); //прототип функции ввода
struct Vkladchik {  // определение структуры "Вкладчик"
     int account;    // номер счета
     char name[10]; // имя
     float suma;       // сумма вклада
};
int main()
{
  clrscr();
  Vkladchik k ; // создание экземпляра объекта
  cout<<"Введите счет, имя , сумму \n";
  cin>>k.account>>k.name>>k.suma;
  cout<<"Счет"«setw(9 )<<"Имя" <<setw(16)<<"Сумма"<<endl;
  output(k.account,k.name,k.suma);
  cout<<"\n\n";
  cout<<"\nНажмите любую клавишу…";
  getch();
  return 0;
}
void output(int a, char* n, float s)
{
   cout<<setiosflags(ios::left)<<setw(10)<< a<<setw(13)<<n
   <<setw(7)<<setprecision(2)<<setiosflags(ios::showpoint|ios::right)
   << s<<endl;
}
```

Результаты работы программы:

Введите счет, имя, сумму

1
Victor
125.45

Счет	Имя	Сумма
1	**Victor**	**125.45**

Рис. 15.1.

Структуры, как и элементы других типов, могут объединяться в массивы структур. Чтобы объявить массив структур, надо сначала создать экземпляр структуры (например, **struct Vkladchik),** а затем объявить массив структур:

Vkladchik k[10]; .

Этот оператор создает в памяти 10 переменных типа структуры с шаблоном **Vkladchik** и именами **k[0], k[1]** и т.д.

На рис. 15.2 приведен пример программы, позволяющей вводить и выводить массив структур на дисплей.

```
//Пример ввода - вывода массива структур на дисплей
#include <iomanip.h>
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
void output(int , char* ,float ) ;
struct Vkladchik {          // определение структуры "Вкладчик"
            int account;    // номер счета
            char name[10]; // имя
            float suma;     // сумма вклада
};
int main()
{   int i, n;
    clrscr();
    cout<<"Введите число записей n= ";
    cin>>n;
    Vkladchik k[10]; // создание массива экземпляров объектов
    cout<<"Введите счет, имя, сумму \n";
    for(i=0;i<n;i++){
        cin>>k[i].account>>k[i].name>>k[i].suma;
        cout<<"?"<<endl;
    }
    cout<<"Счет"<<setw(9)<<"Имя" <<setw(16) <<"Сумма"<<endl;
    for(i=0;i<n;i++)
        output(k[i].account, k[i].name, k[i].suma);
    cout<<"\n\n";
    cout<<"\nНажмите любую клавишу…";
    getch();
    return 0;
}
```

```
void output(int a, char* n, float s)
{
  cout<<setiosflags( ios::left)<<setw(10)<< a<<setw(13)<<n
  <<setw(7)<<setprecision(2)<<setiosflags( ios::showpoint|ios::right) <<
  s<<endl;
}
```

Результаты работы программы:

Введите число записей **n=2**
Введите счет, имя, сумму
1
Викторов
125.45 ? 2
Петров
458.63

Счет	Имя	Сумма
1	**Викторов**	**125.45**
2	**Петров**	**458.63**

Рис. 15.2.

Примеры ввода структуры в файл и вывода ее из файла показаны на рис. 15.3 и 15.4.

```
//Пример ввода - вывода структуры в файл
#include <iomanip.h>
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
void output(int, char*, char*, int ); // прототип функции вывода
struct book{                    // определение структуры
        int nzap;          // номер записи
        char fam[20];   // фамилия
        char dol[10];   // должность
        int otdel;         // отдел
};
int main(){
clrscr();
book b;                              // создание экземпляра объекта
ofstream outfile("kniga");        //открытие файла для записи
if(!outfile){
   cerr<<"файл не открыть";
   exit(1);
}
```

```
cout<<"Введите номер записи, фамилию, должность, номер отдела"
      <<" и символ eof по окончании ввода\n";
while(cin>>b.nzap>>b.fam>>b.dol>>b.otdel){
   outfile«b.nzap<<" "<<b.fam<<" "<<b.dol<<" "<<b.otdel<<endl;
   cout<<" ? ";
}
outfile.close();
ifstream infile("kniga"); // открывает файл для чтения
if(!infile){
   cerr<<"файл не открыть";
   exit(1);
}
cout<<"Это содержание файла:\n";
cout<<setw(10)<<" номер записи"
     <<setw(10)<<" фамилия "<<setw(15)<<" должность "
     <<setw(l3)<<"отдел"<<endl;
while(infile>>b.nzap>>b.fam>>b.dol>>b.otdel)
   output(b.nzap, b.fam, b.dol, b.otdel);       // вызов функции вывода
infile.close();
cout<<"\n\n";
cout<<"Нажмите любую клавишу…";
getch();
return 0;
}
void output(int z, char* a, char* n, int s) { // описание функции вывода
  cout<<setiosflags( ios::left)<<setw(15)<< z<<setw(13)<< a<<setw(13)
  <<n<<setw(8)<<setiosflags( ios::right) << s<<endl;
}
```

Результаты работы программы:

```
Введите номер записи, фамилию, должность, номер отдела
        и символ eof по окончании ввода
1
Иванов лаборант
12
? 2
Петров инженер
15
? eof
```

Номер записи	Фамилия	Должность	Отдел
1	Иванов	лаборант	12
2	Петров	инженер	15

Рис. 15.3.

```
//Ввод и вывод структуры в файл и на дисплей в режиме диалога
#include <fstream.h>
#include <string.h>
#include <conio.h>
#include <iomanip.h>
#define FAM 25
#define DOL 15
struct SOTR{              // объявление структуры -."Сотрудники "
       char fam [FAM];    // фамилия
       char dol[DOL];     // должность
       int otdel;         // отдел
};
void sozdanie();           //прототип функции :" Создание "
void prosmotr();           // прототип функции :" Просмотр "
  // операция-функция ввода в структуру с клавиатуры
istream &operator >> (istream &in, SOTR &x) {
     cout<<"\nФамилия:";
     in.seekg(0,ios::end);
     in.get(x.fam,FAM-1,'\n');
     cout<<"\nДолжность:";
     in.seekg(0,ios::end);
     in.get(x.dol,DOL-1,'\n');
     cout<<"\nОтдел:";
     in.seekg(0,ios::end);
     in >> x.otdel;
     return in;
}
// операция-функция вывода структуры на дисплей
ostream &operator << (ostream &out, SOTR x){   // печать объекта
   out << "\n|" << x.fam << "|" << x.dol << "|" << x.otdel <<"|";
   return out;
}
// операция-функция ввода структуры с МД
ifstream &operator >> (ifstream &in, SOTR &x){
   in.setf(ios::left); in.width(FAM);
   in.get(x.fam,FAM,'\n'); in.width(DOL);
   in.get(x.dol,DOL,'\n'); in >> x.otdel;
   return in;
}
// операция-функция вывода структуры на МД
ofstream &operator << (ofstream &out, SOTR &x) {
   out.width(FAM-1); out.setf(ios::left); out << x.fam;
   out.width(DOL-1);
   out.setf(ios::left); out << x.dol; out << x.otdel;
   return out;
```

```
}
void main(void) {
   clrscr();
   char c;
   while (1) {
     cout << endl << "1. Создание файла";
     cout << endl << "2. Просмотр содержимого";
     cout << endl << "3. Выход";
     cout << endl << "Ваш выбор -> ";
     cin.seekg(0,ios::end);
     c = cin.get();

     switch(c){
         case '1':
           sozdanie();
           break;
         case '2':
           prosmotr();
           break;
         case '3':
           return;
         default:
           cout << "Вводите только цифры от 1 до 3" << endl;
       }
   }
}

void sozdanie(){
   char c;
   // поток ff для вывода файла kniga.txt
   ofstream ff;
   // создание экземпляра объекта
   SOTR s;
   ff.open("kadry.txt", ios::binary );
   // цикл записи элементов в файл
   do{
      cin >> s;       // ввод с клавиатуры
      ff << s;        // вывод в файл
      cout<<"\nПродолжить ввод?(Y/N\ или Д/Н)";
   }
   while ((c = getch())=='y'||c=='Y'||c=='д'||c=='Д');
   ff.close();     // закрытие файла
}

void prosmotr(){
   ifstream finp;
```

```
    SOTR s;
    // поток finp для ввода из файла kniga.txt
    finp.open("kadry.txt", ios::binary);
    finp.seekg(0,ios::beg);
    cout<<"\nСписок элементов из файла\n";
    while ( finp ) {    // пока не конец файла
       finp >> s ;    // вывод из файла
       cout << s;     // вывод на дисплей
    }
    finp.close();    // закрытие файла
  }
```

Результаты работы программы:

1. Создание файла
2. Просмотр содержимого
3. Выход

Ваш выбор -> 1

Фамилия: ***Иванов***
Должность: ***лаборант***
Отдел: ***12***
Продолжить ввод**?(Y/N или Д/Н)**
Y

Фамилия: ***Петров***
Должность: ***инженер***
Отдел: ***15***
продолжить ввод**?(Y/N или Д/Н)**
N

1. Создание файла
2. Просмотр содержимого
3. Выход

Ваш выбор -> 2

Фамилия	Должность	Отдел
Иванов	лаборант	12
Петров	инженер	15

Рис. 15.4

16. Классы

16.1. Определение класса

Программисты на С основное внимание уделяют написанию функций, тогда как при программировании на **C++** большее внимание уделяется созданию классов. Классы - это типы данных, определяемые программистом. Каждый класс содержит данные и набор функций, манипулирующих с этими данными. Компоненты - данные класса называются данными-элементами. Компоненты - функции класса называются функциями-элементами. Класс, определяемый пользователем, называется объектом. Классы в **C++** являются естественным продолжением структуры (**struct**) в **C.** Прежде чем рассматривать специфику разработки классов, проанализируем недостатки структуры от которых удается легко избавиться в классах.

Программа на рис. 16.1 создает определенный пользователем тип структуры **Time** с тремя целыми элементами: **hour, minute, second.**

```
//Создание структуры, задание и печать ее элементов
#include <iostream.h>
#include <conio.h>
struct Time {              // определение структуры
     int hour;             // 0-23
     int minute;           // 0-59
     int second;           // 0-59
};
void printMilitary (int, int, int);                    // прототип
void printStandard (int, int, int);                    // прототип
main (){
   Time dinnerTime;       // переменная нового типа Time
   dinnerTime.hour = 18; // задание элементам правильных значений
   dinnerTime.minute = 30;
   dinnerTime.second = 0;
   cout << "Обед состоится в ";
printMilitary(dinnerTime.hour, dinnerTime.minute, dinnerTime.second );
cout << "по военному времени," << endl << "что соответствует";
cout << "по стандартному времени," << endl;
```

```
printStandard(dinnerTime.hour, dinnerTime.minute, dinnerTime.second);
dinnerTime.hour = 29;          // задание элементам неправильных значений
dinnerTime.minute = 73;
dinnerTime.second = 103;
cout << endl << "Время с неправильными значениями:";
printMilitary(dinnerTime.hour, dinnerTime.minute, dinnerTime.second);
cout << endl;
getch();
return 0;
}
 // Печать времени в военном формате
 void printMilitary (int h, int m, int s){
   cout << ( h < 10 ? "0" : "") << h
        << ":" << (m < 10 ? "0" : "") << m
        << ":" << (m < 10 ? "0" : "") << s ;
}
 // Печать времени в стандартном формате
 void printStandard(int h, int m, int s){
   cout << ( h == 0 || h == 12 ? 12 : h %12 )
        << ":" << (m < 10 ? "0" : "") << m
        << ":" << (m < 10 ? "0" : "") << s ;
}
```

Результаты работы программы:

Обед состоится в **18: 30: 00** по военному времени, что соответствует **6: 30: 00 PM** по стандартному времени.
Время с неправильными значениями: **29: 73: 103**

Рис. 16.1.

Программа определяет единственную структуру типа **Time,** названную **dinnerTime,** и использует операцию точка для присвоения элементам структуры начальных значений **18** для **hour, 30** для **minute** и **0** для **second.** Затем программа печатает время в военном (24-часовом) и стандартном (12-часовом) форматах. Основной недостаток структур связан с тем, что в них не существует никакого интерфейса по оценке правильности использования типов данных и оценки противоречивости их начальных значений. Существуют и другие проблемы, связанные со структурами в стиле **C.** В **C** структуры не могут быть напечатаны как еди-

ное целое, а их печать возможна только по одному элементу с соответствующим форматированием каждого.

Классы предоставляют программисту возможность моделировать объекты, которые имеют атрибуты, представленные как данные - элементы и варианты поведения или операции, представленные как функции - элементы. Типы, содержащие данные - элементы и функции - элементы, определяются в **C++** с помощью ключевого слова **classs.** Когда класс определен, имя класса может быть использовано для объявления объекта этого класса.

На рис. 16.2 дано простое определение класса **Time.** Определение класса начинается с ключевого слова **classs.** В фигурных скобках записывается тело класса, а его определение заканчивается точкой с запятой. Определение класса **Time,** как и в структуре, содержит три целых элемента: **hour, minute и second.**

```
//Простое определение класса Time
class Time {
public:
    Time();
    void setTime (int, int, int);
    void printMilitary();
    void printStandart();
private:
     int hour;     // 0-23
     int minute;   // 0-59
     int second;   // 0-59
};
```

Рис. 16.2

Остальные части определения класса - новые. Метки **public:** (открытая) и **private:** (закрытая) называются *спецификаторами доступа к элементам.* Любые данные и функции-элементы, объявленные после спецификатора доступа **public:**, доступны при любом обращении программы к объекту класса **Time**, а эти же элементы, объявленные после спецификатора доступа **private:**, доступны только функциям-элементам этого класса. Спецификаторы доступа к элементам всегда заканчиваются двоеточием и могут появляться в определении класса много раз и

в любом порядке. Определение класса (рис.16.2) содержит после спецификатора доступа **public:** прототипы следующих четырёх функций элементов: **Time, setTime, printMilitary, printStandart.** Это - открытые функции-элементы или открытый интерфейс услуг класса. Эти функции будут использоваться для манипуляций с данными класса. Функция-элемент с тем же именем, что и класс, называется конструктором этого класса. Конструктор – это специальная функция-элемент, которая присваивает начальные значения данным-элементам этого класса. После спецификатора **private:** следуют три целых элемента. Они являются доступными только функциям-элементам класса, т. е. функциям, прототипы которых включены в определение этого класса. Когда класс определен, его можно использовать в качестве типа в объявлениях, например, следующим образом:

```
Time sunset.                  // объект типа Time
ArrayOfTimes[5] .             // массив объектов типа Time
*pointerToTime.               // указатель на объект типа Time
&dinnerTime = sunset;         // ссылка на объект типа Time
```

Программа на рис.16.3 использует класс **Time** и создает единственный объект класса **Time,** названный **t.** После создания объекта, автоматически вызывается конструктор **Time,** который явно присваивает нулевые начальные значения всем данным-элементам закрытой части **private.** Затем печатается время в военном и стандартном форматах, с тем, чтобы подтвердить, что элементы получили правильные начальные значения. После этого с помощью функции-элемента **setTime** устанавливается время, и оно снова печатается в обоих форматах. Затем функция-элемент **setTime** пытается дать данным-элементам неправильные значения, и время снова печатается в обоих форматах.

```
// Определение класса Time
class Time {
public :
     Time ( );                      // конструктор
     void setTime(int, int, int );  // установка часов, минут и секунд
      void printMilitary( );        // печать времени в военном и
      void printStandard( );        // стандартном форматах
```

```
private:
        int hour ;              // 0 - 23
        int minute;             // 0 - 59
        int second;             // 0 - 59
};
```

*/*Конструктор **Time ( ) ;** присваивает нулевые начальные значения каждому элементу данных и обеспечивает согласованное начальное состояние всех объектов **Time ( ) ;** */*

```
Time :: Time( ) { hour = minute = second = 0; }
```

/ Задание нового значения **Time** в военном формате.*
Проверка правильности значений данных.
*Обнуление неверных значений. */*

```
void Time : : setTime( int h, int m, int s ){
   hour = ( h>= 0 && h < 24 ) ? h : 0; minute =
   (m>=0 && m<60) ? m : 0; second = (s >= 0
   && s < 60) ? s : 0;
   }
```

// Печать времени в военном формате

```
void Time : : printMilitary( ){
   cout << ( hour < 10 ? "0" : "") << hour
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second ;
}
```

// Печать времени в стандартном формате

```
void Time : : printStandard( ){
     cout << ( (hour == 0 || hour == 12 ) ? 12 : hour %12 )
          << ":" << (minute < 10 ? "0" : "") <<minute
          << ":" << (second < 10 ? "0" : "") << second ;
          << (hour < 12 ? "AM " : "PM ") ;
}
main ( ){
         Time t ;          // определение экземпляра объекта t класса Time
       cout <<"Начальное значение военного времени равно ";
                                t. printMilitary( );
       cout << endl
            << " Начальное значение стандартного времени равно "; t.
                                printStandard( );
       t.setTime (13, 27, 6);
       cout << endl << endl << "Военное время после setTime равно";
                                t. printMilitary( );
        cout << endl << "Стандартное время после setTime равно";
                                t. printStandard( );
```

```
    t.setTime (99, 99, 99); //попытка установить неправильные значения
    cout << endl << endl
         << "После попытки неправильной установки:"
         << endl << "Военное время:";
                    t. printMilitary( ); cout
     << endl << "Стандартное время :";
                    t. printStandard( );
     cout << endl ; re-
     turn 0;
}_____________________________________________________________
Начальное значение военного времени равно 00 : 00 : 00
Начальное значение стандартного времени равно 12 : 00 : 00 AM

Военное время после setTime равно 13 : 27 : 06
Стандартное время после setTime равно 1 : 27 : 06 PM

После попытки неправильной установки :
Военное время : 00 : 00 : 00
Стандартное время : 12 : 00 : 00 AM
```

Рис. 16.3.

В этой программе конструктор **Time** просто присваивает начальные значения, равные 0, данным-элементам. Неправильные значения не могут храниться в данных-элементах объекта типа **Time,** поскольку они не могут получать начальные значения в теле цикла, где они объявляются, а получают значения через конструктор, который автоматически вызывается при создании объекта **Time.** Все последующие попытки изменить данные-элементы тщательно анализируются функцией **setTime.**

Функция с тем же именем, что и класс, снабженная символом тильда (**~**), называется деструктором этого класса. Деструктор удаляет объекты класса, которые завершили свою работу, тем самым, освобождая память для повторного использования системой. Деструкторы редко используются с простыми классами. Их применение имеет смысл в классах с динамическим распределением памяти под объекты (например, для массивов и строк).

В рассматриваемой программе функции-элементы **printMilitary** и **printStandard** не получают никаких аргументов, потому что они печатают данные-

элементы определенного объекта типа **Time.** Это уменьшает вероятность передачи неверных аргументов.

Обычно классы не создаются на пустом месте. Часто они являются производными от других классов или включают объекты других классов как элементы.

Создание новых классов на основе уже существующих ***называется наследованием.***

Включение классов как элементов других классов ***называется композицией.***

16.2. Доступ к элементам класса и их область действия

Данные-элементы класса и функции-элементы имеют областью действия класс. Функции, не являющиеся элементом класса, имеют областью действия файл. При области действия класс элементы класса непосредственно доступны всем функциям-элементам этого класса, и на них можно ссылаться просто по имени. Вне класса к его элементам можно обращаться либо через имя объекта, либо ссылкой на объект, либо с помощью указателя на объект. Переменные, определенные в функции- элементе, известны только этой функции. Если функция-элемент определяет переменную с тем же именем, что и переменная в области действия класс, то последняя становится невидимой в области действия функция. Операции, используемые для доступа к элементам класса, аналогичны операциям, используемым для доступа к элементам структуры. Программа, приведенная на рис. 16.4, иллюстрирует доступ к открытым элементам класса с помощью операций выбора элемента. Она использует простой класс, названный **count,** с открытым элементом данных **x** типа **int** и открытой функцией-элементом **print.** Программа создает три экземпляра переменных типа **Count -counter, counterRef** (ссылка на объект типа **Count)** и **counterPtr** (указатель на объект типа **Count).** Переменная **counterRef** объявлена, чтобы ссылаться на **counter,** а переменная **counterPtr** объявлена, чтобы указывать на **counter.**

```
// Демонстрация операций доступа к элементам класса . и ->
#include <iostream.h>
```

```
class Count { // простой класс Count
public:
   int x;
   void print () { cout << x << endl; }
};

main ( ){
    Count counter,              // создает объект counter
    * counterPtr = &counter ,   // указатель на counter
    & counterRef = counter;     // ссылка на counter
    cout << " Присвоение x значения 7 и печать по имени объекта :";
    counter.x = 7;              // присвоение 7 элементу данных x
    counter.print ( ) ;         // вызов функции-элемента для печати
    cout << " Присвоение x значения 8 и печать по ссылке :";
    counterRef.x = 8;           // присвоение 8 элементу данных x
    counter.print ( ) ;         // вызов функции-элемента для печати
    cout << " Присвоение x значения 10 и печать по указателю :";
    counterPtr ->x = 10;        // присвоение 10 элементу данных x
    counterPtr->print ( ) ;     // вызов функции-элемента для печати
}
```

Результаты работы программы:

```
Присвоение x значения 7 и печать по имени объекта : 7
Присвоение x значения 8 и печать по ссылке : 8
Присвоение x значения 10 и печать по указателю : 10
```

Рис . 16.4

На рис 16.5 показан усложненный вариант этой программы на примере класса **Clients** (Клиенты). В этом классе используются открытые элементы данных **account** (счет) типа **int** и **sum**(сумма) типа **float,** а также открытая функция-элемент **print( ).** Программа создает три экземпляра переменных типа **Clients -vcladchik** (вкладчик), **vcladchikRef** (ссылка на объект типа **Clients)** и **vcladchik Ptr** (указатель на объект типа **Clients).** Переменная **vcladchik Ref** объявлена, чтобы ссылаться на **vcladchik, а п**еременная **vcladchik Ptr** объявлена, чтобы указывать на **vcladchik.**

```
// Демонстрация операций доступа к элементам класса . и ->
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
class Clients{                  // простой класс Clients
    public:
```

```
        int account;
        char name[20];
        float sum;
        void print () { cout<<endl<<"счет="<<account<<";\n"<<"имя=" << name
                        <<";\n"<<"сумма="<<sum«";\n"; }
};
main ( ){
   clrscr();
   Clients vcladchik,                    // создает объект vcladchik
   * vcladchikPtr = & vcladchik ,        // указатель на vcladchik
   & vcladchikRef = vcladchik;           // ссылка на vcladchik
   cout << " Присвоение счет =1, имя=Bobby, сумма=125.45 и печать по имени
         объекта :";
   vcladchik.account = 1;   // присвоение значения элементу данных account
   vcladchik.account = 1;   // присвоение 1 элементу данных account
   strcpy( vcladchik.name,"Bobby"); // запись имени в массив строковой
                                    // переменной name
   vcladchik.sum = 125.45;          // присвоение значения элементу данных sum
   vcladchik.print ( ) ;            // вызов функции - элемента для печати
   cout <<"Присвоение счет =2, имя=Kat, сумма=458.95 и печать по ссылке: ";
   vcladchikRef.account = 2;    // присвоение 2 элементу данных account
   strcpy(vcladchikRef.name,"Kat");
   vcladchik.sum = 458.95;   // присвоение значения элементу данных sum
   vcladchik.print ( ) ;     // вызов функции - элемента для печати
   cout <<"Присвоение счет =3, имя=Mary, сумма=858.35
          и печать по указателю :";
   vcladchikPtr ->account = 3;    // присвоение 3 элементу данных account
   strcpy(vcladchikPtr->name,"Mary");
   vcladchikPtr ->sum = 858.35;   // присвоение значения элементу данных sum
   vcladchikPtr ->print ( ) ; // вызов функции - элемента для печати
   getch();
   return 0;
}
```

Результаты работы программы:

```
Присвоение счет =1,имя= Bobby, сумма=125.45 и печать по имени объекта:
счет= 1
имя = Bobby
сумма = 125.45
Присвоение счет =2,имя=Kat, сумма=458.95 и печать по ссылке: счет= 2
имя = Bobby
сумма = 125.45
Присвоение счет =3,имя=Mary, сумма=858.35 и печать по указателю: счет= 1
имя = Boby
сумма = 125.45
```

Рис. 16.5.

На рис. 16.6 показан вариант программы, в которой с помощью функции-элемента **print()** осуществляется форматированный вывод данных на печать. Как и ранее, функция **print()** не получает никаких аргументов, потому что она печатает данные-элементы определенного объекта типа **Clients**. Это уменьшает вероятность появления ошибки при передаче аргументов. В этом классе используются открытые элементы данных **account** (счет) типа **int** и **sum** (сумма) типа **float**, а также открытая функция - элемент **print( )**. Программа создает три экземпляра переменных типа **Client**s - **vcladchik**(вкладчик), **vcladchikRef** (ссылка на объект типа **Clients**) и **vcladchikPtr** (указатель на объект типа **Clients**). Переменная **vcladchikRef** объявлена, чтобы ссылаться на **vcladchik**, а переменная **vcladchikPtr** объявлена, чтобы указывать на **vcladchik**.

```
// Демонстрация операций доступа к элементам класса . и ->
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
class Clients { // простой класс Clients
public:
    int account;
    char name[20];
    float sum;
    void print ();
};
//описание функции-элемента print(), принадлежащей классу Clients
void Clients::print () {
  cout<<"\nсчет "<<setw(9)<<" имя "<<setw(16)<<" сумма "<<endl;
  cout<<setiosflags( ios::left)<<setw(10)<< account<<setw(13)<<name
  <<setw(7)<<setprecision(2)<<setiosflags( ios::showpoint|ios::right)
  << sum<<endl<<endl;
}
main ( ){
   clrscr();
   Clients vcladchik,                  // создает объект vcladchik
   * vcladchikPtr = & vcladchik,       // указатель на vcladchik
   &vcladchikRef = vcladchik;          // ссылка на vcladchik
   cout << " Присвоение счет =1, имя=Bobby, сумма=125.45 и печать по имени
```

```
   объекта :";
   vcladchik.account = 1;   //присвоение 1 элементу данных account
   strcpy( vcladchik.name,"Bobby");       // запись имени в массив строковой
                                          //переменной name
   vcladchik.sum = 125.45;   //присвоение значения элементу данных sum
   vcladchik.print ( ) ;        // вызов функции - элемента для печати
   cout << "Присвоение счет =2, имя=Kate, сумма=458.95 и печать по ссылке :";
   vcladchikRef.account = 2;    // присвоение 2 элементу данных account
   strcpy(vcladchikRef.name,"Kate");
   vcladchik.sum = 458.95;   //присвоение значения элементу данных sum
   vcladchik.print ( ) ;      // вызов функции - элемента для печати
   cout <<"Присвоение счет =3, имя=Mary, сумма=858.35 и печать по указателю
   :";
   vcladchikPtr ->account = 3;    // присвоение 3 элементу данных account
   strcpy(vcladchikPtr->name,"Mary");
   vcladchikPtr ->sum = 858.35;   //присвоение значения элементу данных sum
   vcladchikPtr ->print ( ) ; // вызов функции - элемента для печати
   getch();
   return 0;
}

void print(int a, char* n, float s)
{
  cout<<" счет "<<setw(9)<<" имя "<<setw(16)<<" сумма "<<endl;
  cout<<endl<<setiosflags( ios::left)<<setw(10)<< a<<setw(13)<<n
  <<setw(7)<<setprecision(2)<<setiosflags( ios::showpoint|ios::right)
  << s<<endl;
    }
```

Результаты работы программы:

```
Присвоение счет =1, имя= Bobby, сумма=125.45 и печать по имени объекта :
      счет                имя                сумма
       1                  Bobby               125.45
Присвоение счет =2,имя=Ka1e, сумма=458.95 и печать по ссылке :
счет                      имя                сумма
       2                  Kate               458.95
Присвоение счет =3, имя=Mary, сумма=858.35 и печать по указателю :
      счет                имя                сумма
       3                  Mary               858.35
```

Рис. 16.6.

На рис. 16.7 приведена программа, которая иллюстрирует один из возможных способов организации доступа к открытому массиву элементов класса **Clients**, когда значения данных элементов вводятся пользователем в режиме диалога.

```
// Доступ к массиву элементов класса в режиме диалога #in-
clude <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
class Clients                          // простой класс Clients
{
    public:
      int account;
      char name[20]; float
      sum;
      void print ();
};
// описание функции-элемента print ();принадлежащей классу Clients
void Clients::print (){
  cout<<setiosflags( ios::left)<<setw(10)<< account<<setw(13)<<name
  <<setw(7)<<setprecision(2)<<setiosflags( ios::showpoint|ios::right)
  << sum<<endl<<endl;
}

main ( ){
   clrscr();
   Clients vcladchik[10];   // создание массива объектов
   int n, i ;
   cout<<"Введите число записей n = ";
   cin>>n;
   cout<<"Введите счет, имя, сумму :\n";
   // Ввод массива данных - элементов
   for(i=0;i<n;i++){
      cin>>vcladchik[i].account>>vcladchik[i].name>>vcladchik[i].sum;
      cout<<"?"<<endl;
   }
   //Вывод заголовка на дисплей
   cout<<"\nсчет "<<setw(9)<< " имя "<<setw(16)<<" сумма"<<endl;
   // Вывод массива данных - элементов
   for(i=0;i<n; i++)
      vcladchik[i].print();
```

```
    cout<<"\n\n";
    getch();
    return 0;
}
```

Результаты работы программы:

```
Введите число записей n=2
Введите счет, имя, сумму:
1
Bobby
125.45
? 2
Kate
458.95
        счет            имя             сумма
        1               Bobby            125.45
        2               Kate             458.95
```

Рис. 16.7.

16.3. Отделение интерфейса от реализации

Один из наиболее фундаментальных принципов разработки хорошего программного обеспечения состоит в отделении интерфейса от реализации. Это облегчает модификацию программ. Объявление класса помещается в заголовочный файл и становится доступным любому клиенту, который захочет использовать класс. На рис. 16.8 показано, как программу, приведенную на рис .16.3, можно разбить на ряд файлов. При построении программы на **C++** каждое определение класса обычно помещается в заголовочный файл, а определения функций - элементов этого класса помещаются в файлы исходных кодов с теми же базовыми именами. Заголовочные файлы включаются (посредством **#include)** в каждый файл, в котором используется класс. Файлы с исходными кодами компилируются с файлом, содержащим главную программу.

Программа на рис. 16.8 состоит из заголовочного файла **TIME1.H,** в котором объявляется класс **Time,** файла **TIME1.CPP,** где описываются функции-элементы

класса **Time,** и файла с функцией **main.** Выходные данные этой программы идентичны выходным данным программы на рис.16.3

```
//T1ME1.H
/*Определение класса Time. Функции-элементы определяются в фай-
 ле TIME1.CPP*/
#ifndef TIME1_H //предотвращение многократного включения заголовочного
#define T I M E 1 H // файла
class Time {
   public :
      Time ( );                          // конструктор
      void setTime(int, int, int );      // установка часов, минут и секунд
      void printMilitary( );             // печать времени в военном и
      void printStandard( );       // стандартном форматах
    private:
      int hour ;              // 0 - 23
      int minute;             // 0 - 59
      int second;             // 0 - 59
};
#endif

// T1ME1.CPP
// Определение функций-элементов для класса Time
/*Конструктор Time ( )  присваивает нулевые начальные значения каж-
дому элементу данных и обеспечивает согласованное начальное состоя-
ние всех объектов Time */
Time : : Time( ) { hour = minute = second = 0; }

/* Задание нового значения Time в военном формате.
   Проверка правильности значений данных.
   Обнуление неверных значений. */
void Time : : setTime( int h, int m, int s )
{
   hour = ( h>= 0 && h < 24 ) ? h : 0;
   minute = (m>=0 && m<60) ? m : 0;
   second = (s >= 0 && s < 60) ? s : 0;
   }

void Time : : printMilitary( ) // печать времени в военном формате
{
cout << ( hour < 10 ? "0" : "") << hour
        << ":" << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second ;
}
```

```
void Time : : printStandard( ) // печать времени в стандартном формате
{
cout << ( (hour == 0 || hour == 12 ) ? 12 : hour %12 )
        << ":" << (minute < 10 ? "0" : "") <<minute
        << ":" << (second < 10 ? "0" : "") << second ;
        << (hour < 12 ? "AM " : "PM ") ;
}

// Драйвер класса Time1
#include"time1.cpp"
#include"time1.h"
// Драйвер для проверки простого класса
main ( )
{
   Time t ;        // определение экземпляра объекта t класса Time
   cout <<"Начальное значение военного времени равно ";
   t.printMilitary( );
   cout << endl
        << " Начальное значение стандартного времени равно ";
   t.printStandard( );
   t.setTime (13, 27, 6);
   cout << endl << endl << "Военное время после setTime равно";
   t.printMilitary( );
   cout << endl << "Стандартное время после setTime равно";
   t.printStandard( );
   t.setTime (99, 99, 99); //попытка установить неправильные значения
   cout << endl << endl
           << "После попытки неправильной установки :" << endl
           << "Военное время :";
   t.printMilitary( );
   cout << endl << "Стандартное время :";
   t.printStandard( );
   cout << endl ;
   return 0;
}
```

Результаты работы программы:

Начальное значение военного времени равно 00 : 00: 00
Начальное значение стандартного времени равно 12: 00: 00 AM

Военное время после setTime равно 13 : 27 : 06
Стандартное время после setTime равно 1 : 27 : 06 PM

После попытки неправильной установки :
Военное время : 00 : 00 : 00 Стандартное
время : 12 : 00 : 00 AM

Рис. 16.8.

В этой программе объявление класса заключено в директивы препроцессора:

```
#ifndef TIME1H
#define TIME1H
.....
#endif .
```

При построении больших программ в заголовочные файлы будут помещаться также и другие определения и объявления. Приведенные директивы предотвращают включение кода между **#ifndef** и **#endif,** если определено имя **TIME1_H.** Если заголовок еще не включался в файл, то имя **TIME1_H** определяется директивой **#define,** и операторы заголовочного файла включаются в результирующий файл. Если же заголовок был включен ранее, **TIME1_H** уже определен, то операторы заголовочного файла повторно не включаются. В приведенных директивах использованы символические константы, определяемые именем заголовочного файла с символом подчеркивания вместо точки.

16.4. Управление доступом к элементам

Спецификаторы доступа к элементу **public, private** и **protected** используются для управления доступом к данным - элементам класса и функциям -элементам. По умолчанию режим доступа для классов - закрытый **(private),** так что все элементы после заголовка класса и до первого спецификатора доступа являются закрытыми. После каждого спецификатора режим доступа, определенный им, действует до следующего спецификатора или до завершающей правой скобки (**}**) определения класса. Спецификаторы **public, private** и **protected** могут быть повторены, но этот прием используется редко. Закрытые элементы класса могут быть доступны только для функций-элементов (и дружественных функций) этого класса. Открытые элементы класса доступны для любых функций в программе. Ос-

новная задача открытых элементов состоит в том, чтобы дать клиентам класса представление о возможностях (услугах), которые обеспечивает класс. Этот набор услуг составляет открытый интерфейс класса. Клиенты класса не должны знать, каким образом класс решает их задачи. Закрытые элементы класса и описания открытых функций-элементов напрямую недоступны для клиентов класса. Эти компоненты составляют реализацию класса. У клиентов класса имеется возможность изменять значения закрытых данных с помощью функций-элементов. Доступ к закрытым данным класса должен тщательно контролироваться с помощью функций-элементов, называемых функциями доступа. Например, чтобы разрешить клиентам прочитать закрытое значение данных, класс может иметь функцию **get** (получить), а возможность изменения закрытых данных можно реализовать через функцию **set** (установить). Функции-элементы **set и get** обеспечивают проверку правильности данных и обеспечивают уверенность в том, что данные установлены верно. Причем, функция **get** управляет форматированием и отображением данных, а функция **set** должна тщательно анализировать любую попытку изменить значение закрытого элемента данных. Например, должны быть отвергнуты попытки установить день месяца, равный 37, отрицательный вес человека, численную величину символьного значения и т.д. Клиенты класса должны быть уведомлены о попытке присвоить данным неверные значения.

Программа на рис. 16.9 расширяет класс **Time** за счет включения функций чтения и записи закрытых данных-элементов **hour, minute** и **second.** Функция записи жестко управляет установкой данных-элементов. Попытка задать данным-элементам неправильные значения вызывает присваивание этим данным-элементам нулевых значений. Программа сначала использует функции записи, чтобы задать правильные значения закрытым данным-элементам объекта **t** класса **Time,** затем использует функцию чтения для вывода значений на экран. Далее функции записи пытаются задать элементам **hour** и **second** неправильные значения, а элементу **minute** - правильное. После этого функции чтения направляют эти значения на экран. Результат подтверждает, что неправильные значения вызыва-

ют установку данных-элементов в нулевое состояние. В итоге программа устанавливает время **11:58:00** и прибавляет 3 минуты при вызове функции **incrementMinutes.** Эта функция не является элементом класса, поэтому она использует функции-элементы записи и чтения для соответствующего увеличения элемента **minute.** Из-за многократных вызовов функций этот способ снижает производительность. Одним из способов устранения этого недостатка является запись дружественных функций.

```
//TIME3.H
// Объявление класса Time
// Функции - элементы определены в TIME3.CPP
//Предотвращение многократного включения заголовочного файла
#ifndef TIME3H
#define TIME3H
// Определение класса Time
class Time {
public :
      Time (int = 0, int = 0, int = 0 );      // конструктор

      // Функции записи
      void setTime(int, int, int ); // установка часов, минут и секунд
      void setHour(int);                  // установка часа
      void setMinute(int);                // установка минут
      void setSecond(int);                // установка секунд

      // Функции чтения
      int getHour();                      // возвращает час
      int getMinute ();                   // возвращает минуты
      int getSecond ();                   // возвращает секунды

      void printMilitary( ); void         // печать времени в военном и
      printStandard( );                   // стандартном форматах
private:
       int hour ;                 // 0 - 23
       int minute;                // 0 - 59
       int second;                // 0 - 59
};
#endif

// TIME3.CPP
// определение функций-элементов для класса Time
```

```
// #include"time3.h"
#include <iostream .h>
/* Функция-конструктор для задания начальных значений
закрытых данных, вызывает функцию-элемент setTime, чтобы установить
значения переменных, которые по умолчанию равны нулю.*/
Time :: Time( int hr, int min, int sec) { setTime( hr, min, sec) ; }

// Установка значений часов, минут, секунд
void Time::setTime(int h, int m, int s)
{
      hour = (h >= && h<24) ? h : 0 ;
      minute = (m>= 0 && m< 60) ? m : 0;
      second = (s >=0 && s < 60) ? s : 0;
};

// Установка значения часа
      void Time::setHour(int h) { hour = (h>= 0 && h < 24) ? h : 0;}
// Установка значения минут
      void Time:: setMinute(int m) { minute = (m>=0 && m<60) ? m : 0;}
// Установка значения секунд
      void Time:: setSecond(int s)   { second = (s >= 0 && s < 60) ? s : 0;}

//Получение значения часа
int Time :: getHour() {return hour;}
int Time :: getMinute () {return minute;}
int Time :: getSecond () {return second;}

// Печать времени в военном формате
void Time :: printMilitary( )
{
cout << ( hour < 10 ? "0" : "") << hour
          << ":" << (minute < 10 ? "0" : "") << minute
          << ":" << (second < 10 ? "0" : "") << second ;
}
// Печать времени в стандартном формате
void Time :: printStandard( )
{
cout << ( (hour == 0 || hour == 12 ) ? 12 : hour %12 )
          << ":" << (minute < 10 ? "0" : "") <<minute
          << ":" << (second < 10 ? "0" : "") << second
          << (hour < 12 ? "AM " : "PM ") ;
}
```

```
// Демонстрация функций записи и чтения класса Time
#include <iostream.h>
// #include "time3. h"
void incrementMinutes( Time &, int);
main ()
{
   Time t; t.setHour(17);
   t.setMinute(34);
   t.setSecond(25);
   cout << "Результат установки всех правильных значений:"<< endl
        <<" Час: " << t.getHour()
        <<" Минуты: " << t.getMinute ()
        <<" Секунды: " << t.getSecond () << endl <<endl;
   t.setHour(234);
   t.setMinute(43);
   t.setSecond(6373);
      cout << "Результаты попытки установить неправильные часы, минуты и " <<"
      секунды: " <<endl << "Час: "<< t.getHour() <<" Минуты: " << t.getMinute () <<"
      Секунды: " << t.getSecond () << endl <<endl;
   t.setTime(11,58,0);
   incrementMinutes(t, 3);
   return 0;
}

void incrementMinutes(Time &tt, int count)
{
  cout << "Увеличение минут на "<< count<<endl <<"Начальное время:";
  tt.printStandard();
  for ( int i=1; I <= count ;i++){
     tt.setMinute((tt.getMinute() +1) % 60);
     if (tt.getMinute() == 0)
       tt.setHour((tt.getHour () +1) % 24);
     cout << endl << "минуты +1 :";
     tt.printStandard();
  }
  cout << endl;
}_____
Результат установки всех правильных значений:
  Час: 17 Минуты: 34 Секунды: 25
Результаты попытки установить неправильные часы и секунды:
Час: 0   Минуты: 43 Секунды: 0

Увеличение минут на 3
```

Начальное время: 11: 58: 00AM
минуты +1: 11: 59: 00AM
минуты +1: 12: 00: 00PM
минуты +1: 12: 01: 00PM

Рис. 16.9.

16.5. Дружественные функции и дружественные классы

Дружественные функции класса определяются вне области действия этого класса, но имеют право доступа к закрытым элементам **(private** и **protected)** данного класса. Функция или класс в целом могут быть объявлены другом **(friend)** другого класса. Чтобы объявить функцию как друга класса, перед ее прототипом в описании класса ставится ключевое слово **friend,** а объявление записывается в форме **friend ClassTwo** в определении класса **ClassOne.** Программа на рис. 16.10 демонстрирует объявление и использование дружественной функции **setX** для установки закрытого элемента данных **x** класса **count.** Объявление **friend** появляется первым в объявлении класса, даже раньше объявления закрытых функций-элементов.

```
//Пример использования дружественных функций
#include <iostream .h>
class Count {
      friend void setX(Count &, int);          //объявление друга
public:
     Count() { x = 0; }                        // конструктор
     void print() const {cout << x << endl; }  // вывод
private :
     int x;                                    // элемент данных
};

void setX(count &c, int val)
{
     c.x = val;
}

main ()
{
   count object;
   cout << "object.x после своего создания :";
   object.print();
```

```
   cout << "object.x после вызова дружественной функции setX:";
   setX(object, 8);
   object.print();
   return 0;
}
```
object.x после своего создания: **0**
object.x после вызова дружественной функции **setX: 8**

Рис. 16.10

Другой пример использования дружественной функции для доступа к закрытому элементу приведен на рис.16.11. В программе используется дружественная функция **SUM()** для установки закрытого элемента данных sum (сумма вклада) класса **Clients**.

Конструктор задает закрытому элементу данных sum начальное нулевое значение. Передача параметров в дружественную функцию осуществляется по ссылке.

```
// Пример использования дружественной функции
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
// Простой класс Clients
class Clients {
    friend void SUM (Clients& , float ) ; //объявление дружественной функции
public:
    Clients() { sum=0; }                  // конструктор
   // Открытые данные - элементы
    int account;                          // номер счета
    char name[20];                        // массив символьных переменных
    void print ();                        // функция-элемент
private:
   float sum;                             // закрытые данные-элементы
};
void SUM(Clients& c, float val)           // описание дружественной функции
{
    c.sum = val;
};

void Clients::print ()                    // описание функции-элемента
```

```
{
  cout<<endl<< " счет "<<setw(13)<<" имя "<<setw(15)<<" сумма "<<endl;
  cout<<setiosflags( ios::left)<<setw(14)<< account«setw(13)«name
  <<setprecision(2)<<setiosflags( ios::showpoint|ios::right)
  << sum<<endl<<endl;
}
main ( )
{
   clrscr();
   Clients vcladchik;                  // объявление объекта vcladchik
   cout << "Вывод объекта vcladchik после его создания :";
   vcladchik.account = 1;              //присвоение значения элементу данных account
   strcpy( vcladchik.name,"Bobby"); // запись имени в массив строковой
                                       //переменной name
   vcladchik.print ( ) ;               // вызов функции-элемента для печати
   sum(vcladchik , 458.45);            //вызов дружественной функции
   cout << "Вывод объекта vcladchik после вызова дружественной функции :";
   vcladchik.account = 1;
   strcpy(vcladchik.name,"Bobby");
   vcladchik.print ( ) ;
   getch();
   return 0;
}
```

Результаты работы программы:

Вывод объекта vcladchik после его создания:

счет	имя	сумма
1	Bobby	0

Вывод объекта vcladchik после вызова дружественной функции:

счет	имя	сумма
1	Bobby	458.45

Рис. 16.11.

Библиографический список

1. Павловская Т.А. С/С++: Программирование на языке высокого уровня. Учебник. - СПб.: Питер ,2001. - 460 с.
2. Шумова Е.О. Программирование на языке высокого уровня: Учеб. пособие. Ч.1. - СПб.: СЗТУ,2001. - 82 с.
3. Березин Б.И., Березин С.Б. Начальный курс С и С++. - М.: Диалог - Мифи,1997. - 288 с.

4. Либерти Джесс. Освой самостоятельно C++ за 21 день.- М.: Издательский Дом "Вильямс", 2001. - 814 с.
5. Строганов Р.П., Давыдов В.Г., Васильев В.П. Алгоритмическое обеспечение задач управления в АСУ ТП: Учеб. пособие. - Л.: ЛПИ , 1989.

ОГЛАВЛЕНИЕ