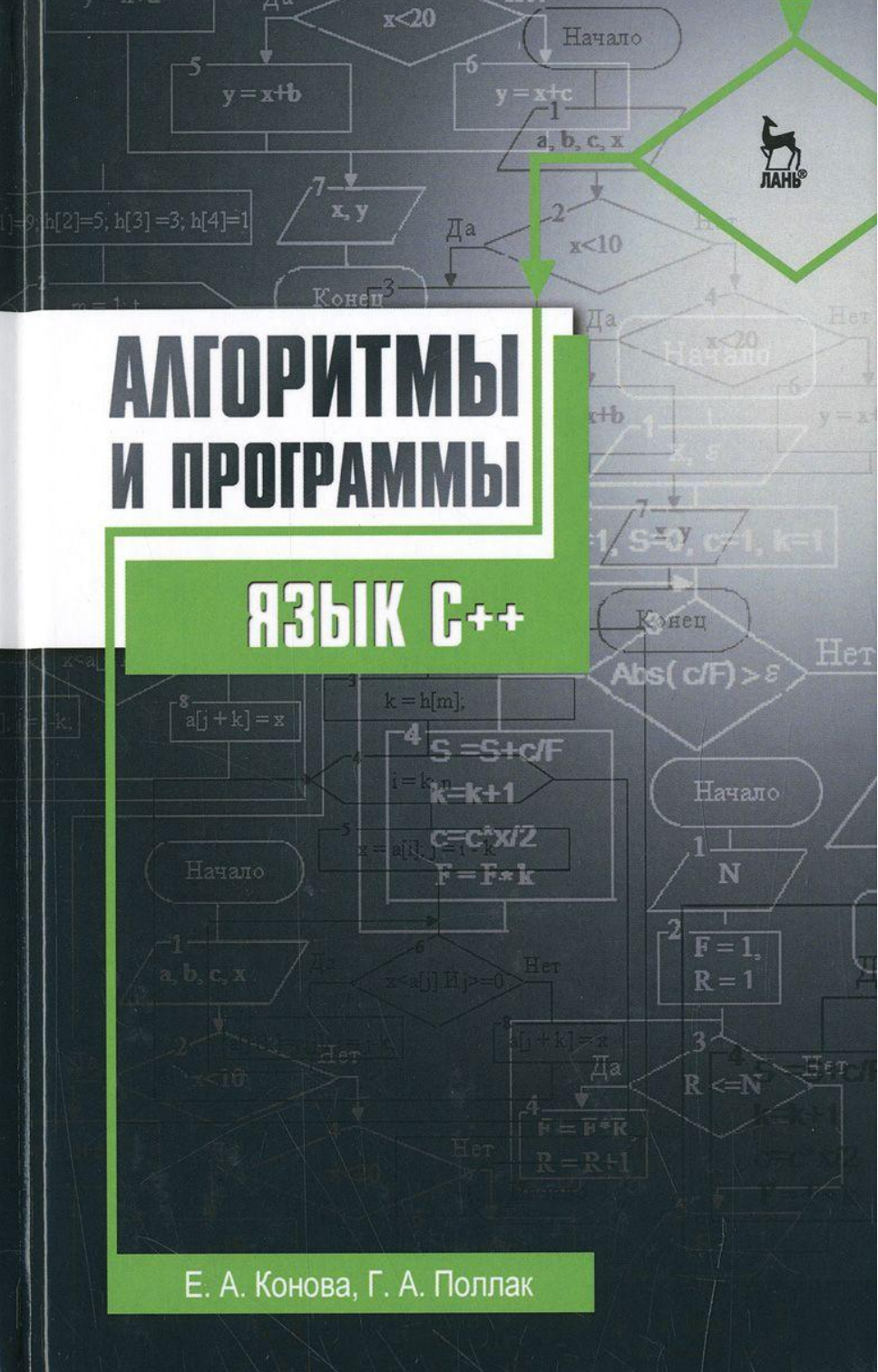


АЛГОРИТМЫ И ПРОГРАММЫ

ЯЗЫК С++

Е. А. Конова, Г. А. Поллак

Е. А. КОНОВА,
Г. А. ПОЛЛАК

АЛГОРИТМЫ И ПРОГРАММЫ. ЯЗЫК C++

Издание второе, стереотипное

ДОПУЩЕНО
УМО по образованию в области прикладной информатики
в качестве учебного пособия для студентов,
обучающихся по направлению
«Прикладная информатика»

САНКТ-ПЕТЕРБУРГ
МОСКВА · КРАСНОДАР
2017

ББК 32.973.26-018.1я73
К 64

Конова Е. А., Поллак Г. А.

К 64 Алгоритмы и программы. Язык С++: Учебное пособие. — 2-е изд., стер. — СПб.: Издательство «Лань», 2017. — 384 с.: ил. — (Учебники для вузов. Специальная литература).

ISBN 978-5-8114-2020-9

При изложении материала авторы используют методику обучения от алгоритмов к программам, поэтому вначале излагаются сведения об алгоритмах с примерами реализации типовых алгоритмов. Изучение основ языка программирования С++опирается на полученные знания. Примеры можно решать в любой среде разработчика, поддерживающей язык С++, но авторами примеры отлажены в Visual Studio 2013. Коды программ соответствуют стандарту С++11 (ISO/IEC 14882:2011), разработаны в консольных приложениях на основе шаблона «Пустой проект».

В задачах практикума предлагаются как задачи, использующие типовые алгоритмы, так и содержательные, для которых приведено только вербальное описание. Пособие предназначено для студентов направления подготовки «Прикладная информатика» и других, может быть рекомендовано для самостоятельного изучения, так как не требует предварительных знаний о языках программирования.

ББК 32.973.26-018.1я73

Рецензенты:

В. И. ШИРЯЕВ — доктор технических наук, профессор, зав. кафедрой «Системы управления» Южно-Уральского государственного университета; *И. В. САФРОНОВА* — кандидат технических наук, доцент, зав. кафедрой прикладной информатики и математики Уральского социально-экономического института (филиала) Академии труда и социальных отношений.

Обложка
Е. А. ВЛАСОВА

ВВЕДЕНИЕ

Авторы прекрасно понимают трудности, которые возникают у начинающего программиста, поэтому учебное пособие предназначено, в первую очередь, для тех, кто только начинает изучать программирование.

В разработке методики изложения авторы придерживались определения, данного Н. Виртом [1]: Программа = Алгоритмы + Данные.

Согласно этому определению программы представляют собой конкретные формулировки абстрактных алгоритмов, основанные на конкретных представлениях данных. Именно поэтому авторами уделяется большое внимание как разработке алгоритмов, так и концепции данных языка программирования.

Учебное пособие состоит из трех глав.

В первой главе приводятся сведения о типовых алгоритмах обработки данных безотносительно к языку программирования. Приведены примеры решения некоторых классов задач, где для каждой задачи разработан алгоритм в виде блок-схемы с пояснениями, набор тестовых данных и таблица исполнения алгоритма. Приводимые способы решения задач по возможности являются рациональными, но не претендуют на то, чтобы быть наилучшими.

В качестве языка программирования выбран классический C++. Краткое описание синтаксических правил C++ и механизмов реализации приведено во второй главе. Материал разбит на темы, позволяющие изучать язык по принципу «от простого к сложному». В изложении авторы опираются на материал первой главы, приводя программную реализацию всех алгоритмов. Добавлено мно-

го примеров, иллюстрирующих как особенности языка C++, так и некоторые алгоритмические приемы решения задач. В качестве методологии выбрано структурное программирование на основе функций.

Все приведенные программы проверены на работоспособность авторами в среде проектирования Visual Studio 2013. Коды программ соответствуют стандарту C++11 (ISO/IEC 14882:2011) и разработаны в консольных приложениях на основе шаблона «Пустой проект».

Третья глава — это практикум, который содержит задания для самостоятельного выполнения. В практикуме авторы опираются на материал второй главы. Третья глава состоит из десяти основных тем. В каждой теме есть краткое теоретическое введение, а также примеры программ решения некоторых задач. В каждом примере назван типовой алгоритм и приведен код программы с подробными комментариями.

Примеры и типовые решения помогут начинающим в освоении практических приемов программирования и выработке собственного стиля.

В каждой теме для самостоятельного решения предлагаются по 30 вариантов задач примерно одинакового уровня сложности. Особый интерес, по нашему мнению, представляют содержательные задачи, в которых постановка задачи выполнена на вербальном уровне, т. е. не формализована. Здесь студент должен самостоятельно осмыслить задачу, формализовать ее, предложить структуру данных и выбрать или разработать алгоритм решения. Опыт показывает, что часто именно этот этап в практическом программировании является наиболее трудным.

Материал пособия консолидирует многолетний опыт работы авторов в преподавании различных курсов программирования.

Соответствует ФГОС ВПО третьего поколения для направления «Прикладная информатика».

Не требуется каких-либо предварительных знаний о языках программирования, поэтому пособие может быть рекомендовано для самостоятельного изучения.

ГЛАВА 1

ОСНОВЫ АЛГОРИТМИЗАЦИИ

1.1. ОПРЕДЕЛЕНИЕ АЛГОРИТМА И ЕГО СВОЙСТВА

Под алгоритмом понимается точное предписание, задающее последовательность действий, которая ведет от произвольного исходного данного (или от некоторой совокупности возможных для данного алгоритма исходных данных) к достижению полностью определяемого этим исходным данным результата.

Алгоритм должен обладать определенными свойствами, наличие которых гарантирует получение решения задачи исполнителем.

Дискретность. Решение задачи должно быть разбито на элементарные действия. Запись отдельных действий реализуется в виде упорядоченной последовательности отдельных команд, образующих дискретную структуру алгоритма. Это свойство непосредственно отражено в определении алгоритма.

Понятность. На практике любой алгоритм предназначен для определенного исполнителя, и любую команду алгоритма исполнитель должен уметь выполнить.

Определенность (детерминированность). Каждая команда алгоритма должна определять однозначные действия исполнителя. Результат их исполнения не должен зависеть от факторов, не учтенных в алгоритме явно. При одних и тех же исходных данных алгоритм должен давать стабильный результат.

Массовость. Разработанный алгоритм должен давать возможность получения результата при различных исходных данных для однотипных задач.

Например, пользуясь алгоритмом решения квадратного уравнения, можно находить его корни при любых значениях коэффициентов.

Свойство массовости полезное, но не обязательное свойство алгоритма, так как интерес представляют и алгоритмы, пригодные для решения единственной задачи.

Результативность (конечность). Это свойство предполагает обязательное получение результата решения задачи за конечное число шагов. Под решением задачи понимается и сообщение о том, что при заданных значениях исходных данных задача решения не имеет.

Если решить задачу при заданных исходных данных за конечное число шагов не удается, то говорят, что алгоритм «зацикливается».

Смысл условий дискретности, понятности и определенности ясен: их нарушение ведет к невозможности выполнения алгоритма. Остальные условия не столь очевидны. Для сложных алгоритмов выполнить исчерпывающую проверку результативности и корректности невозможно. Это равносильно полному решению задачи, для которой создан алгоритм, вручную.

Можно сформулировать общие правила, руководствуясь которыми следует записывать алгоритм решения задачи.

1. Выделить величины, являющиеся исходными данными для задачи.

2. Разбить решение задачи на такие команды, каждую из которых исполнитель может выполнить однозначно.

3. Указать порядок выполнения команд.

4. Задать условие окончания процесса решения задачи.

5. Определить, что является результатом решения задачи в каждом из возможных случаев.

Хотя алгоритмы обычно предназначены для автоматического выполнения, они создаются и разрабатываются людьми. Поэтому первоначальная запись алгоритма обычно производится в форме, доступной для восприятия человеком.

Самой простой является *словесная* форма записи алгоритмов на естественном языке. В этом виде алгоритм представляет собой описание последовательности этапов обработки данных, изложенное в произвольной форме.

Словесная форма удобна для человеческого восприятия, но страдает многословностью и неоднозначностью.

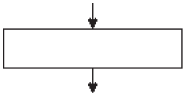
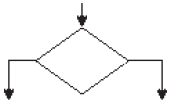
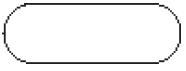
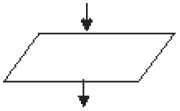
Когда запись алгоритма формализована частично, то используется *псевдокод*. Он содержит как элементы естественного языка, так и формальные конструкции, описывающие базовые алгоритмические структуры. Эти конструкции называются *служебными словами*. Формального определения псевдокода или строгих правил записи алгоритмов в таком формате не существует.

Графическая форма представления алгоритма является более компактной. Алгоритм изображается в виде последовательности связанных между собой блоков, каждый из которых соответствует выполнению одного или нескольких действий. Графическое представление алгоритма называется *блок-схемой*. Блок-схема определяет структуру алгоритма.

Графические обозначения блоков стандартизованы. Некоторые из часто используемых блоков представлены в таблице 1.1.

Таблица 1.1

Изображение основных блоков на блок-схеме

Обозначение блока	Пояснение
	Процесс (вычислительное действие, реализованное операцией присваивания)
	Решение (проверка условия, реализующая условный переход)
	Начало, конец алгоритма
	Ввод-вывод в общем виде
	Модификация (начало цикла с параметром)

Отдельные блоки соединяются линиями переходов, которые определяют очередность выполнения действий. Направление линий сверху вниз или слева направо принимается за основное.

Алгоритм, записанный на языке программирования, называется *программой*. При использовании этих языков запись алгоритма абсолютно формальна и пригодна для выполнения на ЭВМ. Отдельная конструкция языка программирования называется *оператором*. Программа — это упорядоченная последовательность операторов.

1.2. БАЗОВЫЕ АЛГОРИТМИЧЕСКИЕ КОНСТРУКЦИИ

Число реализованных конструкций конечно в любом языке программирования. Структурной элементарной единицей алгоритма является команда, обозначающая один элементарный шаг обработки или отображения информации. Простая команда на языке блок-схем изображается в виде функционального блока «процесс», который имеет один вход и один выход. Из команд проверки условий и простых команд образуются составные команды, имеющие более сложную структуру, но тоже один вход и один выход.

Алгоритм любой сложности может быть представлен комбинацией трех базовых структур:

- следование;
- ветвление (в полной и сокращенной форме);
- цикл (с предусловием или постусловием).

Характерной особенностью этих структур является наличие у них одного входа и одного выхода.

1.2.1. Линейные алгоритмы

Базовая структура «следование» означает, что несколько операторов выполняются последовательно друг за другом, и только один раз за время выполнения программы. Структура «следование» используется для реализации задач, имеющих *линейный* алгоритм решения. Это означает, что такой алгоритм не содержит проверок

условий и повторений, действия в нем выполняются последовательно, одно за другим.

Пример 1.1. Построить блок-схему алгоритма вычисления высот треугольника со сторонами a, b, c по формулам:

$$h_a = \left(\frac{2}{a}\right) \cdot \sqrt{p(p-a)(p-b)(p-c)};$$

$$h_a = \left(\frac{2}{b}\right) \cdot \sqrt{p(p-a)(p-b)(p-c)};$$

$$h_a = \left(\frac{2}{c}\right) \cdot \sqrt{p(p-a)(p-b)(p-c)},$$

где $p = \dfrac{(a+b+c)}{2}$ — полупериметр.

Для того чтобы не вычислять три раза одно и то же значение, введем вспомогательную величину:

$$t = 2\sqrt{p(p-a)(p-b)(p-c)}.$$

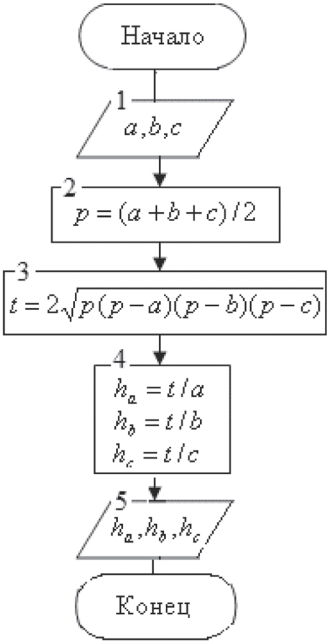
Блок 1. Ввод значений сторон треугольника.

Блок 2. Вычисление полупериметра.

Блок 3. Вычисление вспомогательной величины t.

Блок 4. Вычисление высот, опущенных на стороны a, b, c.

Блок 5. Вывод результатов.



1.2.2. Разветвляющиеся алгоритмы

Второй базовой структурой является «ветвление». Эта структура обеспечивает, в зависимости от результата проверки условия, выбор одного из альтернативных путей работы алгоритма, причем каждый из путей ведет к обще-

му выходу, так что работа алгоритма будет продолжаться независимо от того, какой путь будет выбран.

Существует структура с полным и неполным ветвлением.

Структура с полным ветвлением (*если — то — иначе*) записывается так:

Если < условие >
то <действия 1>
иначе <действия 2>
Все если

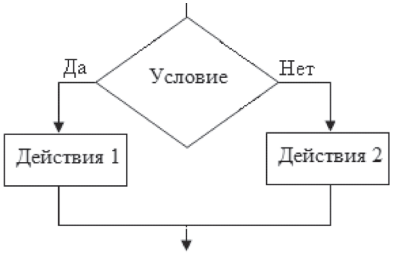
Команда выполняется так: если <условие> является истинным, то выполняются <действия 1>, записанные после ключевого слова *то*, если <условие> является ложным, то выполняются <действия 2>, записанные после слова *иначе*.

Структура с неполным ветвлением (*если — то*) не содержит части, начинающейся со слова *иначе*:

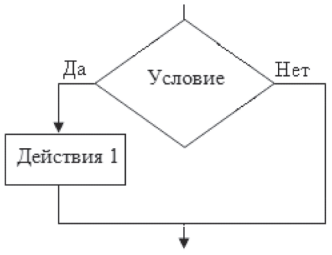
Если <условие>
то <действия 1>
Все если

Команда выполняется так: если <условие> является истинным, то выполняются <действия 1>, записанные после ключевого слова *то*.

Блок-схема алгоритма с ветвлением выглядит так:



Полное ветвление. Структура *Если — То — Иначе*



Неполное ветвление. Структура *Если — То*

Пример 1.2. Вычислить значение функции

$$y = \begin{cases} x + a, & \text{при } x < 10; \\ x + b, & \text{при } 10 \le x \le 20; \\ x + c, & \text{при } x > 20. \end{cases}$$

Дано: x, a, b, c — произвольные числа.

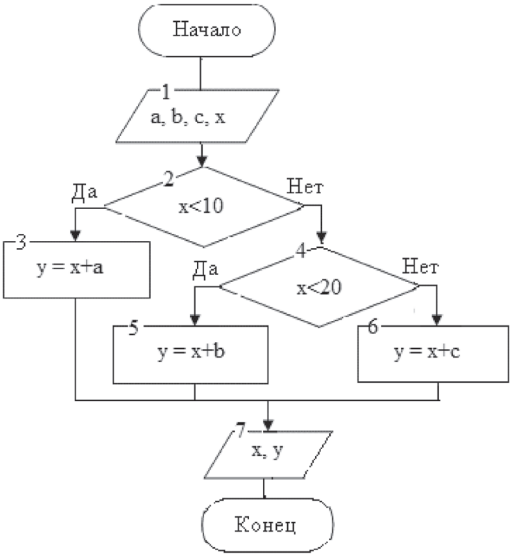
Найти: y.

Представим задачу графически на числовой оси

$x < 10$	$10 \le x \le 20$	$x > 20$
10		20
$y = x + a$	$y = x + b$	$y = x + c$
1 интервал	2 интервал	3 интервал

Так как значение переменной x вводится произвольно, то оно может оказаться в любом из трех интервалов.

Приведем блок-схему.



Блок 1. Ввод исходных данных.

Блок 2. Проверка введенного значения. Если $x < 10$ (выход «Да»), то точка находится в первом интервале. В противном случае $x \ge 10$ (выход «Нет»), и точка может оказаться во втором или третьем интервале.

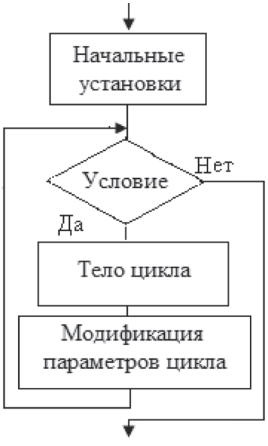
Блок 4. Проверка ограничения значения x справа ($x < 20$). Если условие выполняется (выход «Да»), то x находится во втором интервале, иначе $x \ge 20$, и точка находится в третьем интервале.

Блоки 3, 5 и 6. Вычисление значения y.

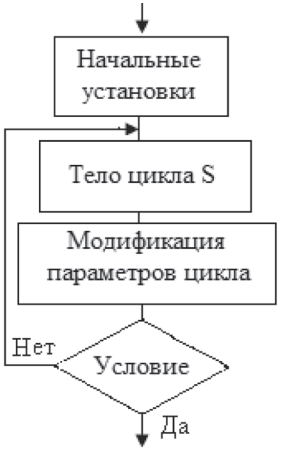
1.2.3. Циклические алгоритмы

При составлении алгоритмов решения большинства задач возникает необходимость в неоднократном повторении одних и тех же команд. Алгоритм, составленный с использованием многократных повторений одних и тех же действий (циклов), называется *циклическим*. Однако слово «неоднократно» не означает «до бесконечности». Организация циклов, никогда не приводящая к остановке в выполнении алгоритма («зацикливание» алгоритма), нарушает требование его результативности — получения результата за конечное число шагов.

Блок, для выполнения которого организуется цикл, называется *телом цикла*. Остальные операторы служат для управления процессом повторения вычислений: это начальные установки, проверка условия продолжения цикла и модификация параметра цикла. Один проход цикла называется *итерацией*.



Цикл с предусловием



Цикл с постусловием

Начальные установки служат для того, чтобы до входа в цикл задать значения переменных, которые в нем используются.

Проверка условия продолжения цикла выполняется на каждой итерации либо до тела цикла (*цикл с предусло-*

вием), либо после тела цикла (*цикл с постусловием*). Тело цикла с постусловием всегда выполняется хотя бы один раз. Проверка необходимости выполнения цикла с предусловием делается до начала цикла, поэтому возможно, что он не выполнится ни разу.

При конструировании циклов следует соблюдать обязательное условие результативности алгоритма (т. е. его окончания за конечное число шагов). Практически это означает, что в условии должна быть переменная, значение которой изменяется в теле цикла. Причем, изменяется таким образом, чтобы условие в конечном итоге перестало выполняться. Такая переменная называется управляющей переменной цикла или *параметром* цикла.

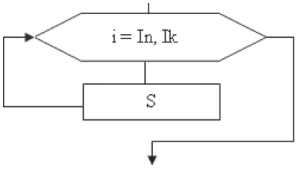
Еще один вид циклов — цикл с *параметром*, или *арифметический цикл*. Тело цикла выполняется, пока параметр цикла i пробегает множество значений от начального (I_n) до конечного (I_k).

Переменная i определяет количество повторений тела цикла S. Если шаг изменения значения параметра цикла обозначить через ΔI, то количество повторений тела цикла n можно вычислить по формуле

$$n = \frac{I_k - I_n}{\Delta I} + 1.$$

Если параметр цикла i изменяется с шагом 1, то шаг может не указываться.

Цикл выполняется так: начальное значение параметра цикла i равно I_n. Если $i \le I_k$, выполняется тело цикла S, после чего параметр цикла увеличивается на 1 с помощью оператора присваивания $i = i + 1$, и снова проверяется условие $i \le I_k$.



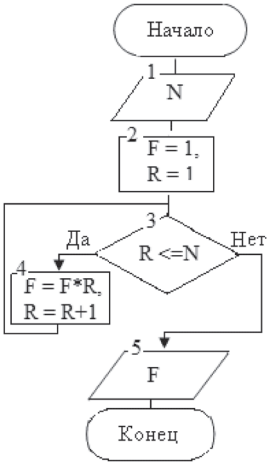
Пример 1.3. Дано целое положительное число n. Вычислить факториал этого числа. Известно, что факториал любого целого положительного числа n определяется как произведение чисел от 1 до заданного числа n:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$$

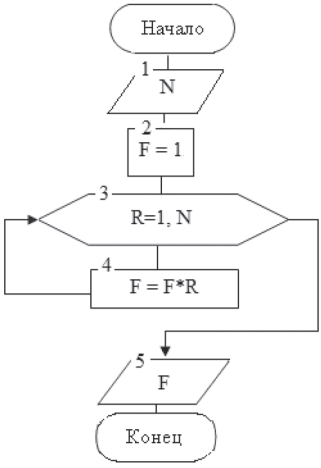
По определению $0! = 1$ и $1! = 1$.

Задача решается с помощью циклического алгоритма. Введем следующие обозначения: N — заданное число, F — факториал числа, R — параметр цикла. Составим два варианта алгоритма: с использованием цикла с предусловием и цикла с параметром.

Правильность алгоритма можно проверить, если выполнить его формально «вручную». Выполним алгоритм при $n = 4$.



Цикл с предусловием



Цикл с параметром

Цикл с предусловием

R	*R <= N*	*F*
1	Да	$1 \cdot 1 = 1$
2	Да	$1 \cdot 2 = 2$
3	Да	$2 \cdot 3 = 6$
4	Да	$6 \cdot 4 = 24$
5	Нет	— (кц)

Цикл с параметром

R	*R <= N*	*F*
1	Да	$1 \cdot 1 = 1$
2	Да	$1 \cdot 2 = 2$
3	Да	$2 \cdot 3 = 6$
4	Да	$6 \cdot 4 = 24$
5	Нет	— (кц)

Итак, при решении данной задачи выполнение цикла с предусловием ничем не отличается от выполнения цикла с параметром. При $R = 5$ произойдет выход из цикла и окончательное значение $4! = 24$.

В чем же отличие? Посмотрим на структуру этих циклов. В цикле *пока* начальное значение параметра цикла R задается перед входом в цикл, в теле цикла организовано изменение параметра цикла командой $R = R + 1$, условие $R \le N$ определяет условие продолжение цикла. В цикле с заданным числом повторений эти же команды неявно заданы в операторе заголовка цикла.

Пример 1.4. Пусть $a_0 = 1$; $a_k = k \cdot a_{k-1} + 1/k$, $k = 1, 2, \ldots$ Дано натуральное число n. Получить a_n.

Дано: a_0 — первый член последовательности; n — номер члена последовательности, значение которого требуется найти.

Найти: a_n — n-й член последовательности.

Математическая модель. Посмотрим, как изменяется значение члена последовательности при изменении значения k. При $k = 1$ $a_1 = 1 \cdot a_0 + 1$. При $k = 2$ $a_2 = 2 \cdot a_1 + 1/2$. При $k = 3 a_3 = 3 \cdot a_2 + 1/3$. Выполнив указанные вычисления n раз, получим искомое значение a_n. В задачах такого типа не требуется хранить результаты вычислений на каждом шаге. Поэтому можно использовать простые переменные.

Обозначим через a — произвольный член последовательности. Тогда формула для вычисления члена последовательности будет выглядеть так:

$$a = k \cdot a + 1/k.$$

В этой формуле значение a, стоящее справа от знака «=», определяется на предыдущем шаге вычисления, а значение a, стоящее в левой части выражения, определяется на данном шаге и заменяет в памяти предыдущее значение.

Переменная a — вещественного типа; переменные k и n — целого типа.

Приведем словесное описание и блок-схему алгоритма.

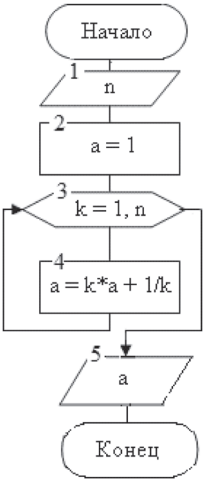
Блок 1. Ввод количества членов последовательности n.

Блок 2. Присваивание начального значения $a = 1$.

Блок 3. Арифметический цикл. Начальное значение параметра цикла k равно 1, конечное значение равно n, шаг изменения параметра — 1.

Блок 4. Выполнение тела цикла. Для каждого значения k вычисляется выражение $a = k * a + 1/k$.

Блок 5. Вывод значения a.



Пример 1.5. Даны натуральное число n, действительное число x. Вычислить сумму:

$$\sum_{i=1}^{n} \frac{x^i}{i!}.$$

Дано: n — количество слагаемых; x — действительное число.

Найти: S — сумму чисел.

Математическая модель. Обозначим c — слагаемое. Начальное значение суммы $S = 0$.

Количество слагаемых известно и равно n. Следовательно, можно использовать арифметический цикл.

Выведем рекуррентное соотношение, определяющее формулу, по которой вычисляется слагаемое на очередном шаге выполнения цикла.

Вспомним, что факториал числа n определяется как произведение чисел от 1 до n включительно. Следовательно, $i! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot i$, по определению $0! = 1$, $1! = 1$. Для вывода рекуррентного соотношения вычислим несколько значений слагаемых.

$i = 1,\ \ c = x^1/1! = x;$
$i = 2,\ \ c = x^2/2! = x^2/\ (1 \cdot 2) = c \cdot x/2 = c \cdot x/i;$
$i = 3,\ \ c = x^3/3! = x^3/\ (1 \cdot 2 \cdot 3) = c \cdot x/3 = c \cdot x/i;$
$i = 4,\ \ c = x^4/4! = x^4/\ (1 \cdot 2 \cdot 3 \cdot 4) = c \cdot x/4 = c \cdot x/i.$

В правой части всех формул значение c равно значению, вычисленному на предыдущем шаге. Итак, получаем следующее соотношение, которое будем использовать в алгоритме $c = c \cdot x/i$, где начальное значение $c = 1$, $i = 1, 2, ..., n$.

Сумму накапливаем, используя формулу $S = S + c$. Записанное соотношение определяет, что значение суммы увеличивается на величину слагаемого при каждом повторении цикла. Начальное значение суммы равно 0.

Переменные i, n — целого типа, переменные c, x, S — вещественного типа.

Приведем словесное описание и блок-схему алгоритма.

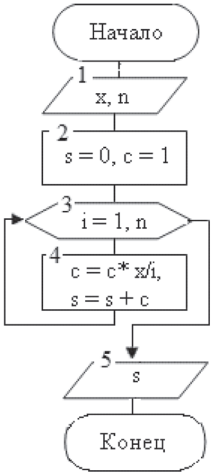
Блок 1. Ввод x, n.

Блок 2. Присваивание начальных значений $S = 0$, $c = 1$.

Блок 3. В цикле для всех значений параметра цикла i вычисляем тело цикла (блок 4):

$$c = c * x/i;$$
$$S = S + c.$$

Блок 5. Вывод суммы S.



1.2.4. Итерационные циклы

Все циклические алгоритмы можно разделить на две группы: циклы с заранее известным количеством повторений и циклы с заранее неизвестным количеством повторений. Циклы с заранее неизвестным количеством повторений называются *итерационными*. Итерационный процесс — это последовательное приближение к результату за некоторое количество шагов.

Рассмотрим блок-схемы итерационных циклов на примере.

Пример 1.6. Даны действительные числа x, ε ($x \neq 0$, $\varepsilon > 0$). Вычислить сумму с точностью ε:

$$\sum_{k=0}^{\infty} \frac{x^k}{2^k \cdot k!}$$

Вычисление бесконечной суммы с заданной точностью ε означает, что требуемая точность достигнута, когда вычислена сумма нескольких первых слагаемых и очередное слагаемое оказалось по модулю меньше, чем ε, — это и все последующие слагаемые можно не учитывать.

Пусть c — слагаемое, S — сумма, F — факториал числа k.

Математическая модель. Выведем рекуррентное соотношение, используя способ, показанный в примере 1.5.

Для вычисления факториала числа можно воспользоваться формулой $F = F \cdot k$, которую мы вывели ранее. Начальное значение $F = 1$. Проверьте правильность этой формулы, последовательно вычисляя значение $k!$.

Выведем общую формулу для вычисления одного слагаемого (без учета факториала числа k):

$$\begin{aligned}
&k = 0, \quad c = x^0/2^0 = 1;\\
&k = 1, \quad c = x^1/2^1 = x/2;\\
&k = 2, \quad c = x^2/2^2 = (x/2)\,(x/2) = c \cdot x/2;\\
&k = 3, \quad c = x^3/2^3 = (x^2/4)\,(x/2) = c \cdot x/2;\\
&k = 4, \quad c = x^4/2^4 = (x^3/8)\,(x/2) = c \cdot x/2.
\end{aligned}$$

Обобщая, для произвольного значения k можно записать $c = c \cdot x/2$, где значение переменной c в правой части

формулы вычисляется на предыдущем шаге. Начальное значение $c = 1$ при $k = 0$.

Сумму вычисляем по формуле $S = S + c / F$ (см. пример 1.5). Начальное значение $S = 0$.

В данной задаче следует использовать цикл с предусловием, потому что количество слагаемых будет зависеть от введенного значения x и требуемой точности вычислений ε.

Переменная k — целого типа, переменные x, ε, c, S — вещественного типа. Переменную F следует взять вещественного типа, так как диапазон целых чисел ограничен, а факториал быстро возрастает с ростом значения k.

Приведем словесное описание и блок-схему алгоритма.

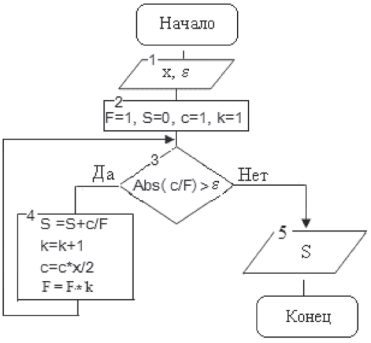
Блок 1. Ввод исходных данных x и ε.

Блок 2. Присваивание начальных значений $F = 1$, $S = 0$, $c = 1$. Параметр цикла $k = 1$.

Блок 3. Прежде чем вычислять сумму, определим выполнение условия продолжения цикла. По условию, когда очередное слагаемое окажется по модулю меньше заданной точности ε, вычисления следует прекратить. Слагаемым в нашем случае является дробь c/F. Так как может быть введено отрицательное значение x, то использована функция *Abs*(), определяющая модуль числа.

Блок 4. Если требуемая точность вычислений не достигнута (выход «Да» блока 3), выполняем тело цикла, определяем новые значения переменных и продолжаем цикл.

Блок 5. Как только требуемая точность вычисления достигнута (выход «Нет» блока 3), завершаем цикл и печатаем значение переменной S.



Цикл, который мы использовали при решении задачи, называется *итерационным*. Особенностью такого цикла является то, что число его повторений зависит от выполнения условия, записанного при входе в цикл. В итерационных алгоритмах необходимо обеспечить обязательное достижение условия выхода из цикла (сходимость итерационного процесса). В противном случае произойдет «зацикливание» алгоритма. Управление итерационным циклом организует программист, который должен позаботиться и об инициализации управляющей переменной, и об ее приращении, и об условии завершения цикла.

1.3. АЛГОРИТМЫ, ИСПОЛЬЗУЮЩИЕ ОДНОМЕРНЫЕ МАССИВЫ

Массив — это упорядоченный набор однотипных значений (элементов массива). Каждый массив имеет имя, что дает возможность различать массивы между собой и обращаться к ним по имени.

Каждый элемент массива имеет три характеристики:

1) *имя*, совпадающее с именем массива;

2) *индекс* — целое число или множество целых чисел, однозначно определяющее местоположение элемента в массиве. В качестве индекса может использоваться также переменная или арифметическое выражение целого типа. Примеры индексов: 3, 15, i, j, $i - 1$, $j + 2$;

3) *значение* — фактическое значение элемента, определенное его типом.

Элементы массива могут использоваться произвольно и являются одинаково доступными. Доступ к элементам массива производится по его индексу.

Массивы могут быть *одномерными* и *многомерными*. В этом подпараграфе рассмотрим некоторые алгоритмы на одномерных массивах.

1.3.1. Ввод и вывод элементов массива

Одномерный массив определяется именем и числом элементов (размером), и мы обозначим его $a[n]$, где a — имя массива; n — число элементов массива. Например,

$a[10]$, где a — имя массива; 10 — число элементов в массиве.

Каждый элемент одномерного массива имеет один индекс, равный порядковому номеру элемента в массиве. Например, массив из 10 элементов выглядит так:

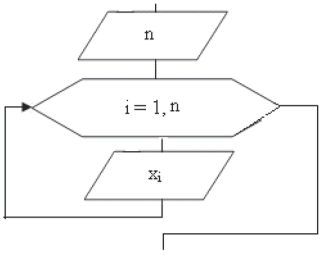
Индекс	1	2	3	4	5	6	7	8	9	10
Значение	3	0	15	4	6	−2	11	0	−9	7

$a[1] = 3;\ \ a[5] = 6;\ \ a[7] = 11;\ \ a[9] = -9;\ \ a[10] = 7.$

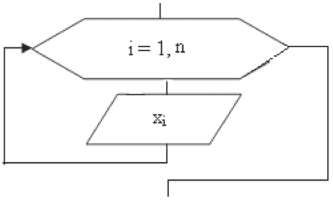
Так как всегда известно количество элементов в массиве, то для ввода и вывода его элементов используется цикл с заданным числом повторений.

Мы рассматриваем массив, состоящий из произвольного числа элементов. Поэтому, прежде чем задать значения элементов массива, требуется ввести количество элементов массива n.

Вывод элементов также производится с использованием цикла с заданным числом повторений.



Ввод элементов массива



Вывод элементов массива

Здесь и далее будем обозначать через i — текущий индекс элемента массива. Он же будет являться параметром цикла, так как количество повторений цикла зависит от количества элементов в массиве.

Далее во всех задачах будем считать, что элементы массива заданы тем или иным способом, и показывать только реализацию алгоритма решения задачи, опуская команды ввода данных и вывода результата.

Также не будем обозначать блоки начала и конца выполнения алгоритма. Подразумевается, что все пере-

численные блоки должны всегда присутствовать в алгоритме.

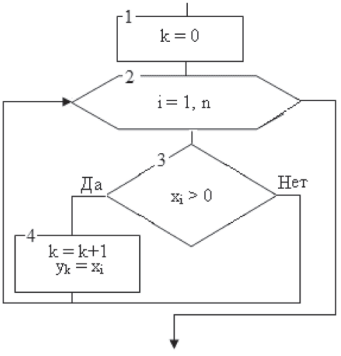
Пример 1.7. Сформировать новый одномерный массив из положительных элементов заданного массива.

Дано: n — размер массива произвольный; массив $X[n]$.

Найти: массив $Y[k]$.

Обозначим: i — текущий индекс элементов массива X, k — текущий индекс элементов массива Y.

Ясно, что для реализации алгоритма необходимо использовать цикл с заданным числом повторений, так как количество элементов в массиве X известно. Приведем фрагмент блок-схемы алгоритма и ее словесное описание.



Блок 1. $k = 0$.

Блок 2. В цикле для всех значений параметра i от 1 до n выполняется тело цикла.

Блок 3. Если условие $X[i] > 0$ выполняется (выход «Да»), то вычисляется группа операторов в блоке 4:

$$k = k + 1; \quad Y[k] = X[i].$$

Тест

Данные	Результат
$n = 5$ $X = (-1, 2, 0, 4, -3)$	$k = 2$ $Y = (2, 4)$

Выполнение алгоритма.

k	i	$x_i > 0$?	$y_k = x_i$
0	1	$x_1 > 0$? «Нет»	
	2	$x_2 > 0$? «Да»	
1			$y_1 = x_2 = 2$
	3	$x_3 > 0$? «Нет»	
	4	$x_4 > 0$? «Да»	
2			$y_2 = x_4 = 4$
	5	$x_5 > 0$? «Нет»	
	6	Конец цикла	

Мы используем цикл с заданным числом повторений. Напомним, что такой цикл (арифметический цикл) применяется, когда число повторений цикла известно к началу его выполнения.

1.3.2. Вычисление суммы и количества элементов массива

Пример 1.8. Вычислить сумму элементов одномерного массива.

Дано: n — размер массива; массив $A = (a_1, a_2, ..., a_n)$.

Найти: S — сумму элементов массива.

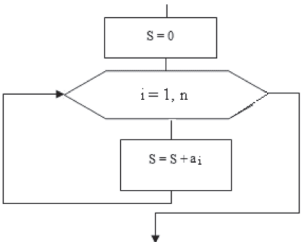
Начальное значение суммы равно нулю. В предыдущих подпараграфах мы говорили о том, что значение суммы накапливается с каждым шагом выполнения алгоритма. Вычисляем сумму по формуле $S = S + a_i$.

Тест

Данные		Результат
$N = 4$	$A = (3, 5, -2, 8)$	$S = 14$

Исполнение алгоритма

i	S
	0
1	$S + a_1 = 0 + 3 = 3$
2	$S + a_2 = a_1 + a_2 = 3 + 5 = 8$
3	$S + a_3 = a_1 + a_2 + a_3 = 8 - 2 = 6$
4	$S + a_4 = a_1 + a_2 + a_3 + a_4 = 6 + 8 = 14$



Фрагмент алгоритма

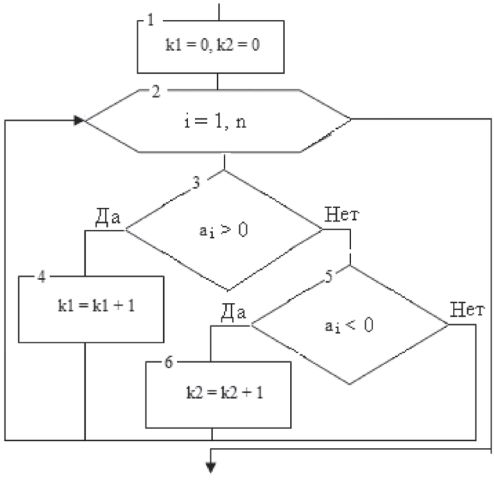
Пример 1.9. Найти количество положительных и отрицательных чисел в данном массиве.

Дано: n — размер массива; массив $A = (a_1, a_2, ..., a_n)$. Обозначим $k1$ — количество положительных чисел, $k2$ — количество отрицательных чисел.

Найти: $k1$, $k2$ — количество положительных и отрицательных чисел массива.

Математическая модель. Пусть $k1 = 0$ и $k2 = 0$. Если $a[i] > 0$, то $k1 = k1 + 1$. Если $a[i] < 0$, то $k2 = k2 + 1$. Процесс повторяется до окончания просмотра всех чисел массива.

Приведем фрагмент блок-схемы алгоритма и ее словесное описание.



Блок 1. Задание начальных значений переменным $k1$ и $k2$.

Блок 2. Заголовок арифметического цикла.

Блок 3. Проверка условия. Если очередное значение элемента массива положительное (выход «Да» блока 3), то увеличиваем количество положительных чисел (блок 4).

Блок 5. Если очередное значение элемента массива отрицательное (выход «Да» блока 5), то увеличиваем количество отрицательных чисел массива (блок 6).

Указанные вычисления выполняем для всех n чисел массива.

В примере нам понадобилось два условных оператора, так как в массиве могут встретиться нулевые элементы, количество которых нам считать не надо.

1.3.3. Определение наибольшего элемента массива

Пример 1.10. Дан массив произвольной длины. Найти наибольший элемент массива и определить его номер.

Дано: n — размер массива; массив $A = (a_1, a_2, ..., a_n)$.

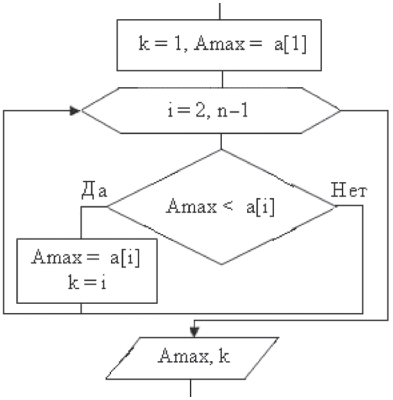
Найти: A_{max} — наибольший элемент массива, k — его номер.

Математическая модель. Пусть $A_{max} = a[1]$; $k = 1$. Если $A_{max} < a[i]$, то $A_{max} = a[i]$, $k = i$, для $i = 2, 3, ..., n$.

Тест

Данные		Результат	
$n = 5$	$A = (3, -1, 10, 1, 6)$	$A_{max} = 10$	$k = 3$

Приведем фрагмент блок-схемы алгоритма и его выполнение для тестового примера.



i	$i \le n$?	$A_{max} < A[i]$?	A_{max}	k
			3	1
2	2 ≤ 5? «Да»	–1 < 3? «Да»		
3	3 ≤ 5? «Да»	10 < 3? «Нет»	10	3
4	3 ≤ 5? «Да»	1 < 10? «Да»		
5	5 ≤ 5? «Да»	6 < 10? «Да»		
6	6 ≤ 5? «Нет» (кц)			

1.3.4. Удаление и вставка элементов массива

Пример 1.11. Удалить из массива, в котором все элементы различны, наибольший элемент.

Дано: n — размер массива; $A[n]$ — массив вещественных чисел.

Найти: новый массив A размера $n - 1$.

Для решения задачи необходимо:

1) найти номер k наибольшего элемента массива. Эта задача решена в примере 1.10;

2) сдвинуть все элементы массива, начиная с k-го, на один элемент влево.

Для того чтобы разработать алгоритм, рассмотрим конкретный пример. Пусть дан одномерный массив, состоящий из 7 элементов:

$$A = (6, 3, 7, 11, 2, 8, 1).$$

Номер наибольшего элемента равен 4 ($k = 4$), т. е. начиная с четвертого элемента будем сдвигать элементы на один влево: четвертому элементу присвоим значение пятого, пятому — значение шестого, а шестому — значение седьмого. На этом сдвиг заканчивается. Таким образом, сдвиг начинается с k-го элемента и заканчивается элементом с номером n, где n — количество элементов в массиве. После сдвига массив будет такой: $A = (6, 3, 7, 8, 1, 1)$. Уменьшим количество элементов в массиве, так как после удаления количество элементов в массиве станет на один элемент меньше. Массив примет вид $A = (6, 3, 7, 8, 1)$.

В примере 1.10 уже рассматривался алгоритм поиска наибольшего элемента в одномерном массиве. Так как в данной задаче нас не интересует конкретное значение наибольшего элемента, то модифицируем рассмотренный алгоритм и сохраняем только номер наибольшего элемента.

Приведем фрагмент блок-схемы алгоритма и ее словесное описание.

Блок 1. Пусть первый элемент максимальный ($k = 1$).

Блок 2. В цикле просматриваем все элементы массива, начиная со второго до последнего.

Блок 3. Сравниваем элемент с номером k с очередным элементом массива.

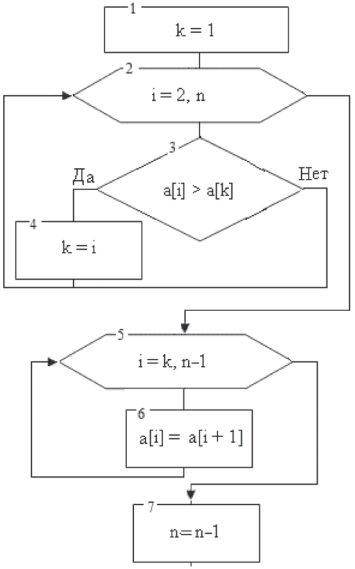
Блок 4. Запоминаем номер элемента, который на очередном шаге выполнения цикла больше k-го.

По окончании цикла (блок 2) переменная k сохранит номер наибольшего элемента.

Начинам сдвиг.

Блок 5. В цикле, начиная с k-го элемента по $n - 1$-й, выполняем присваивание (блок 6), тем самым заменяя элемент с номером i на элемент с номером $i + 1$.

Блок 7. Размер массива уменьшаем на 1, выполняя присваивание $n = n - 1$.



Мы уже рассматривали подробно выполнение алгоритма нахождения наибольшего элемента массива и его номера для тестового примера (см. пример 1.10). Покажем, как выполняется вторая часть (сдвиг элементов массива) для тестового примера.

Тест

Данные		Результат
$n = 7$	$A = (6, 3, 7, 11, 2, 8, 1)$	$A = (6, 3, 7, 2, 8, 1)$ $n = 6$

В массиве номер наибольшего элемента $k = 4$. Удаляем этот элемент.

i	$i \le n - 1$?	$a_i = a_{i+1}$	Результат
4	$4 \le 6$? «Да»	$a_4 = a_5$	$A = (6, 3, 7, 2, 2, 8, 1)$
5	$5 \le 6$? «Да»	$a_5 = a_6$	$A = (6, 3, 7, 2, 8, 8,1)$
6	$6 \le 6$? «Да»	$a_6 = a_7$	$A = (6, 3, 7, 2, 8, 1,1)$
7	$7 \le 6$? «Нет» (кц)		
$n = 6$			$A = (6, 3, 7, 2, 8,1)$

Пример 1.12. Вставить произвольное число в одномерный массив $A[n]$ после элемента с заданным номером.

Дано: n — размер массива; $A[n]$ — числовой вещественный массив; k — номер элемента, после которого вставляется число, равное m.

Найти: новый массив A размера $n + 1$.

Словесное описание алгоритма. Вставка нового элемента осуществляется следующим образом:

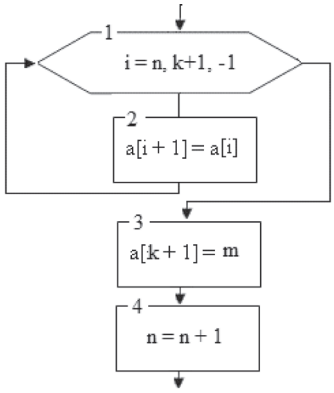
1) первые k элементов массива остаются без изменения;

2) все элементы, начиная с $(k + 1)$-го необходимо сдвинуть вправо, чтобы освободить место для вставляемого элемента;

3) элементу с номером $(k + 1)$ присвоить значение m. Количество элементов массива увеличить на 1.

Рассмотрим пример. Пусть дан массив $A = (3, -12, 5, 14, 27)$, где $n = 5$.

Вставим элемент со значением 10 после второго элемента массива. Получим массив $A = (3, -12, 10, 5, 14, 27)$. Количество элементов $n = 6$.



Приведем фрагмент блок-схемы алгоритма и ее словесное описание.

Блок 1. Начинаем просматривать массив с конца, с n-го элемента до $(k + 1)$-го.

Блок 2. В цикле сдвигаем все элементы массива вправо.

Блок 3. На освободившееся место вставляем новый элемент.

Блок 4. Увеличиваем количество элементов в массиве.

Тест

Данные		Результат
$n = 5$; $k = 2$; $m = 10$	$A = (3, -12, 5, 14, 27)$	$A = (3, -12, 10, 5, 14, 27)$; $n = 6$

Покажем выполнение алгоритма для тестового примера.

i	$a[i+1]=a[i]$	$a[k+1]=m$	n	Массив
5	$a[6]=a[5]$		5	$A=(3, -12, 5, 14, 27, 27)$
4	$a[5]=a[4]$			$A=(3, -12, 5, 14, 14, 27)$
3	$a[4]=a[3]$ (кц)			$A=(3, -12, 5, 5, 14, 27)$
		$a[3]=10$	6	$A=(3, -12, 10, 5, 14, 27)$

В рассмотренных примерах мы использовали арифметический цикл, количество повторений которого всегда известно до работы начала алгоритма.

1.3.5. Определение первого элемента массива, имеющего заданное свойство

Пример 1.13. Определить, является ли заданная последовательность различных чисел a_1, a_2, ..., a_n монотонно убывающей.

По определению последовательность монотонно убывает, если $a[i] \geq a[i+1]$. Если хотя бы для одной пары чисел это условие нарушается, то последовательность не является монотонно убывающей. Следовательно, при построении алгоритма мы должны зафиксировать именно этот факт.

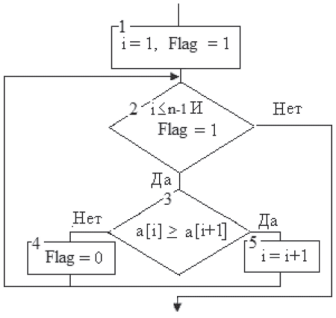
Пусть переменная *Flag* принимает значение равное 1, если последовательность является монотонно убывающей, и 0 — в противном случае.

Дано: n — размер массива; $A[n]$ — числовой вещественный массив.

Найти: $Flag = 1$, если последовательность монотонно убывает; $Flag = 0$ в противном случае.

Словесное описание алгоритма. Предположим, что последовательность монотонно убывает, и присвоим переменной *Flag* начальное значение, равное 1. В цикле последовательно сравниваем значения двух соседних элементов. Выход из цикла возможен в двух случаях:

- просмотрены все элементы заданной последовательности. Это означает, что условие $a[i] \geq a[i+1]$ выполняется для всех пар соединений элементов, и последовательность является монотонно убывающей. Тогда $Flag = 1$;
- условие $a[i] \geq a[i+1]$ не выполнилось для пары соседних элементов, тогда $Flag = 0$. Цикл прерывается. Сле-



довательно, последовательность не является монотонно убывающей.

Приведем фрагмент блок-схемы алгоритма и ее словесное описание.

Блок 1. Берем первое число последовательности $i = 1$. Предполагаем, что последовательность монотонно убывает ($Flag = 1$).

Блок 2. Цикл с предусловием. Пока не проверены все элементы и пока последовательность монотонно убывает (проверка условия в блоке 2), выполняется тело цикла (блоки 3–5).

Блок 3. Если последовательность не монотонная (выход «Нет»), то фиксируем это и $Flag = 0$ (блок 4).

Если условие не нарушается (выход «Да» блока 3), то присваиваем номеру элемента следующее значение (блок 5) и возвращаемся в цикл.

Система тестов

№ теста	Проверяемый случай	Данные		Результат
		n	Массив *A*	*Flag*
1	Является	3	(3, 2, 1)	1
2	Не является	3	(2, 3, 1)	0

Выполнение алгоритма.

№ теста	*i*	*Flag*	(*i* <= n – 1) и (*Flag* = 1)?	*a*[*i*] > *a*[*i* + 1]?
1	1	1	«Да»	*a*[1] > *a*[2]? «Да»
	2		«Да»	*a*[2] > *a*[3]? «Да»
	3		«Нет»	
			Последовательность монотонно убывает	
2	1	1	«Да»	*a*[1] > *a*[2]? «Нет»
		0	«Нет»	
			Последовательность не монотонно убывающая	

Пример 1.14. Определить, есть ли в одномерном массиве хотя бы один отрицательный элемент.

Пусть переменная $Flag = 1$, если в массиве есть отрицательный элемент, и $Flag = 0$ — в противном случае.

Дано: n — размер массива; $A[n]$ — числовой вещественный массив.

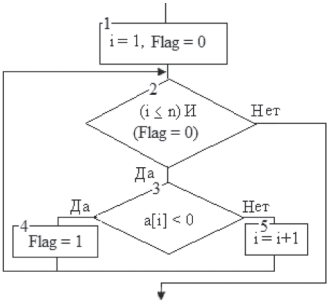
Найти: $Flag = 1$, если отрицательный элемент есть; $Flag = 0$ — в противном случае.

Словесное описание алгоритма. Начинаем просматривать массив с первого элемента ($i = 1$). Пока не просмотрен последний элемент ($i \le n$) и не найден отрицательный элемент ($a[i] >= 0$), будем переходить к следующему элементу ($i = i + 1$). Таким образом, мы закончим просмотр массива в одном из двух случаев:

1) просмотрели все элементы и не нашли отрицательного, тогда $Flag = 0$ и $i > n$;

2) нашли нужный элемент, при этом $Flag = 1$.

Приведем фрагмент блок-схемы алгоритма и ее словесное описание.



Блок 1. Берем первый элемент массива. Предполагаем, что в массиве нет отрицательных чисел и $Flag = 0$.

Блок 2. Пока не просмотрены все числа и пока не встретилось отрицательное число (выход «Да» блока 2) выполняем тело цикла (блоки 3–5).

Блок 3. Если встретилось отрицательное число (выход «Да» блока 3), запоминаем это и $Flag = 1$. Если очередное число положительное или равно нулю (выход «Нет» блока 3), то увеличиваем номер числа (блок 5) и переходим к следующему числу.

Система тестов

№ теста	Проверяемый случай	Данные		Результат
		n	Массив A	$Flag$
1	Есть	3	(3, –2, 1)	1
2	Нет	3	(2, 3, 1)	0

Исполнение алгоритма.

№ теста	i	*Flag*	$(i \le n)$ и $(Flag = 0)$?	$a[i] < 0$?
1	1	0	$(1 \le 3)$ и $(Flag = 0)$? «Да»	$a[1] < 0$? «Нет»
	2	0	$(2 \le 3)$ и $(Flag = 0)$? «Да»	$a[2] < 0$? «Да»
		1	Найдено отрицательное число (кц)	
2	1	0	$(1 \le 3)$ и $(Flag = 0)$? «Да»	$a[1] < 0$? «Нет»
	2	0	$(2 \le 3)$ и $(Flag = 0)$? «Да»	$a[2] < 0$? «Нет»
	3	0	$(3 \le 3)$ и $(Flag = 0)$? «Да»	$a[3] < 0$? «Нет»
	4	0	$(4 \le 3)$ и $(Flag = 0)$? «Нет» (кц)	
			Отрицательных чисел нет	

1.4. АЛГОРИТМЫ, ИСПОЛЬЗУЮЩИЕ ДВУМЕРНЫЕ МАССИВЫ

1.4.1. Понятие двумерного массива

Понятие «двумерный массив» определим на примере. Пусть имеются данные о продажах некоторого товара по месяцам:

Месяц	Объем продаж, пар	Цена продажи, руб.	Себестоимость, руб.
1	4500	100	50
2	3900	110	55
3	3100	120	60

Таблица представляет собой множество из двенадцати однородных величин — это массив. Ее элементы расположены в 3 строки по 4 столбца в каждой.

Подобного рода таблицы из нескольких строк с равным числом элементов в каждой называют в информатике двумерными массивами или матрицей.

Двумерный массив определяется именем, числом строк и столбцов и обозначается, например, так: $A[n, m]$, где A — произвольное имя массива; n — число строк; m — число столбцов. Обратите внимание на то, что сначала всегда указывается количество строк, а потом — количество столбцов массива.

Если имеются данные о продажах за 3 мес., то нашу таблицу можно обозначить так: $A[3, 4]$, т. е. массив состоит из 3 строк и 4 столбцов.

Строки двумерных массивов нумеруются по порядку сверху вниз, а столбцы — слева направо.

Элементы таблицы, представленной в примере, получат такие обозначения:

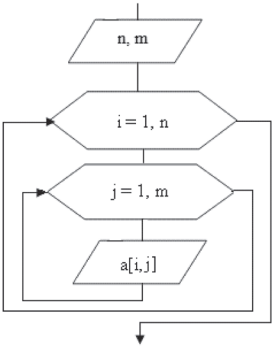
$$A[3,4]=\begin{pmatrix} A[1,1] & A[1,2] & A[1,3] & A[1,4] \\ A[2,1] & A[2,2] & A[2,3] & A[2,4] \\ A[3,1] & A[3,2] & A[3,3] & A[3,4] \end{pmatrix}=$$

$$=\begin{pmatrix} 1 & 4500 & 100 & 50 \\ 2 & 3900 & 110 & 55 \\ 3 & 3100 & 120 & 60 \end{pmatrix}.$$

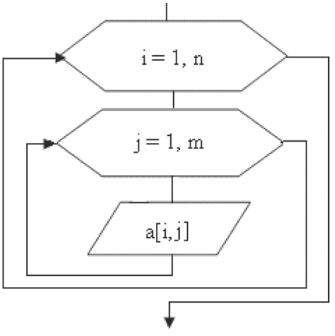
Каждый элемент двумерного массива определяется номерами строки и столбца, на пересечении которых он находится, и в соответствии с этим обозначается именем массива с двумя индексами: первый — номер строки, второй — номер столбца. Например, *A*[1, 3] — элемент находится в первой строке и третьем столбце.

Таким образом, *A*[1, 1] = 1, *A*[2, 3] = 110 и т. д. Произвольный элемент двумерного массива мы будем обозначать *A*[*i*, *j*], где *i* — номер строки, *j* — номер столбца.

Поскольку положение элемента в двумерном массиве описывается двумя индексами, алгоритмы для решения большинства задач с их использованием строятся на основе вложенных циклов. Обычно внешний цикл организуется по строкам массива, т. е. в нем выбирается требуемая строка, а внутренний — по столбцам, в котором выбирается элемент внутри строки.



Ввод двумерного массива



Вывод двумерного массива

В отличие от одномерных массивов для ввода и вывода данных в двумерные массивы необходимо использовать вложенные циклы. Циклы являются арифметическими, так как количество строк и столбцов известно.

Мы ввели и вывели элементы произвольного массива, имеющего n строк и m столбцов. Понятно, что при вводе необходимо задать количество строк и столбцов.

Напомним, что во всех задачах будем считать, что элементы массива заданы тем или иным способом, и показывать только реализацию алгоритма решения задачи, опуская команды ввода данных и вывода результата.

Также не будем обозначать блоки начала и конца выполнения алгоритма. Подразумевается, что все перечисленные блоки должны всегда присутствовать в алгоритме.

Следует понимать, что все алгоритмы, приведенные ранее для одномерных массивов, можно использовать и для двумерных массивов, применив вложенный цикл. Приведем алгоритмы решения некоторых задач.

1.4.2. Вычисление суммы элементов двумерного массива

Пример 1.15. Вычислить сумму элементов строк заданного двумерного массива.

Дано: n, m — количество строк и столбцов массива соответственно; массив $A[n, m]$.

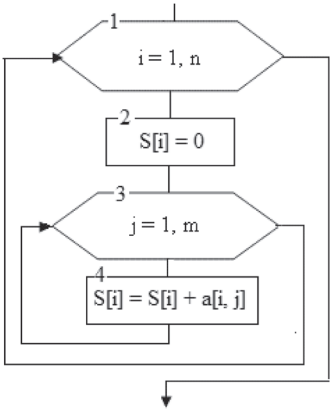
Найти: $S[n]$ — сумму элементов строк массива.

Тест

Данные			Результат
N	M	A	S
2	2	$\begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix}$	(3, 7)

Приведем фрагмент блок-схемы алгоритма и выполнение тестового примера.

Так как в массиве 2 строки, то сумма — это одномерный массив, состоящий из 2 элементов.



Исполнение алгоритма.

i	$i \le n$?	j	$j \le m$?	S
1	1 ≤ 2? «Да»			$S_1 = 0$
		1	1 ≤ 2? «Да»	$S_1 = S_1 + a_{1,1} = 0 + 2 = 2$
		2	2 ≤ 2? «Да»	$S_1 = S_1 + a_{1,2} = 2 + 1 = 3$
		3	3 ≤ 2? «Нет» (кц)	
2	2 ≤ 2? «Да»			$S_2 = 0$
		1	1 ≤ 2? «Да»	$S_2 = S_2 + a_{2,1} = 0 + 4 = 4$
		2	2 ≤ 2? «Да»	$S_2 = S_2 + a_{2,2} = 4 + 3 = 7$
		3	3 ≤ 2? «Нет» (кц)	
3	3 ≤ 2? «Нет» (кц)			

В этом примере мы нашли сумму элементов каждой строки матрицы. Количество таких сумм равно количеству строк матрицы, поэтому для сохранения значений сумм использован одномерный массив $S[n]$, количество элементов в котором равно количеству строк матрицы.

Обратите внимание на то, где присваивается начальное значение переменной S. Если находится сумма всех элементов матрицы, то команда $S = 0$ записывается перед началом внешнего цикла. В результате получается одно значение $S = 10$. Если ставится задача нахождения суммы элементов строк матрицы, то команда $S[i] = 0$ записывается внутри цикла по строкам матрицы, тогда результат представляет собой одномерный массив, в котором количество элементов равно количеству строк матрицы, и тогда $S = (3, 7)$. Если требуется найти сумму элементов столб-

цов матрицы, то команда $S[j] = 0$ записывается внутри цикла по столбцам матрицы, тогда результат представляет собой одномерный массив, в котором количество элементов равно количеству столбцов матрицы, и $S = (6, 4)$.

1.4.3. Вычисление наибольшего элемента двумерного массива

Пример 1.16. Найти наибольший элемент двумерного массива и его индексы.

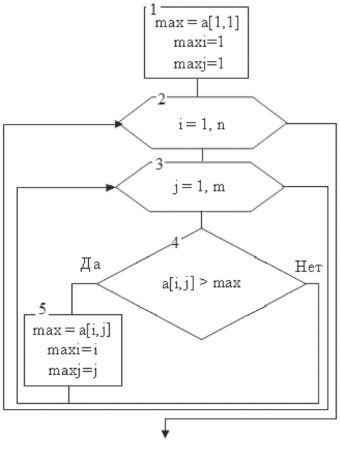
Дано: массив $A[n, m]$, где n, m — количество строк и столбцов массива соответственно.

Найти: max — наибольший элемент массива, а также max*i* и max*j* — номер строки и столбца соответственно, на пересечении которых находится искомый элемент.

Словесное описание алгоритма. Пусть первый элемент двумерного массива является наибольшим. Запоминаем его значение и индексы. Сравниваем наибольшее значение со всеми оставшимися элементами. Если запомненное значение меньше очередного элемента массива, то запоминаем новое значение и его индексы.

Так как значения элементов в массиве могут повторяться, то договоримся, что будем запоминать только индексы первого наибольшего элемента.

Приведем фрагмент блок-схемы алгоритма и ее описание.



Блок 1. Присваиваем начальные значения переменным max, max*i*, max*j*.

Блок 2. Берем очередную строку матрицы.

Блок 3. На *i*-й строке матрицы последовательно выбираем каждый элемент.

Блок 4. Сравниваем выбранный элемент со значением переменной max.

Блок 5. Если значение очередного элемента больше max (выход «Да» блока 4),

то запоминаем его значение и индексы. Возвращаемся в блок 3 и выбираем следующий элемент строки.

Блок 6. Если в строке нет элементов, то выбираем новую строку и повторяем вычисления (блоки 3–5).

Тест

Данные			Результат
n	m	A	
2	3	$\begin{pmatrix} 1 & 3 & 5 \\ 4 & 3 & 5 \end{pmatrix}$	$\max = 5$ $\max i = 1$ $\max j = 3$

Выполните тестирование алгоритма самостоятельно.

1.4.4. Вставка и удаление строк и столбцов двумерного массива

Мы уже рассматривали задачи вставки и удаления элементов в одномерном массиве. Обобщим эти алгоритмы на двумерный массив.

Пример 1.17. Вставить в двумерный массив строку, состоящую из нулей, после строки с номером k.

Для решения этой задачи необходимо:

1) первые k строк оставить без изменения;

2) все строки после k-й сдвинуть на одну вниз. Сдвиг лучше всего начать с последней строки и идти до $(k + 1)$-й строки;

3) присвоить новые значения элементам $(k + 1)$-й строки и увеличить количество строк в двумерном массиве.

Дано: массив $A[n, m]$, где n, m — количество строк и столбцов массива соответственно; k — номер строки, после которой вставляется новая строка.

Найти: новую матрицу A, содержащую $n + 1$ строку и m столбцов.

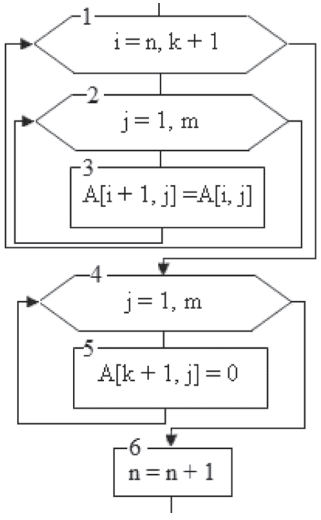
Приведем фрагмент блок-схемы алгоритма.

В блоке 3 в цикле выполняется сдвиг строк матрицы вниз, начиная со строки с номером k. В блоке 5 происходит вставка новой строки.

Тест

n	m	k	Исходная матрица	Результат
2	3	1	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 4 & 5 & 6 \end{pmatrix}$

Так как $k = 1$, то необходимо сдвинуть строку 2 вниз и вставить после первой строки новую строку, состоящую из нулей.



Исполнение алгоритма.

i	*j*	*A*
Сдвигаем строки матрицы вниз (блоки 1–3)		
2	1	$A[3, 1] = A[2, 1] = 4$
	2	$A[3, 2] = A[2, 2] = 5$
	3	$A[3, 3] = A[2, 3] = 6$
Промежуточный результат		
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 4 & 5 & 6 \end{pmatrix}$		
Вставка строки		
2	1	$A[2, 1] = 0$
	2	$A[2, 2] = 0$
	3	$A[2, 3] = 0$

Примечания

1. Если необходимо вставить новую строку после строки, удовлетворяющей некоторому условию, то надо найти номер этой строки — и задача сведется к решению уже рассмотренной.

2. Если надо вставить новые строки после всех строк с заданным условием, то надо учесть это при описании массива. Заметим, что удобнее рассматривать строки, начиная с последней строки, и после вставки очередной строки увеличивать количество строк.

3. Вставка новой строки перед строкой с номером k изменится только тем, что сдвигать назад надо не с $(k + 1)$-й строки, а с k-й.

4. Если надо вставлять столбцы, то размерность массива увеличивается по столбцам, а все остальное практически не меняется: надо сдвинуть столбцы вправо и на данное место записать новый столбец.

Пример 1.18. Удалить из двумерного массива строку с номером k.

Для удаления строки с номером k необходимо:

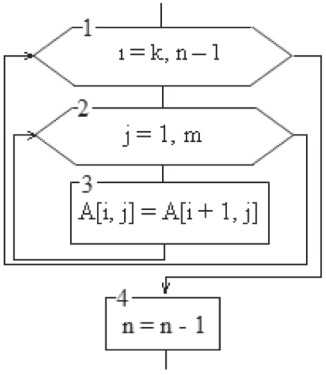
1) сдвинуть все строки, начиная со строки с номером k, вверх;

2) уменьшить количество строк в двумерном массиве.

Дано: массив $A[n, m]$, где n, m — количество строк и столбцов массива соответственно; k — номер удаляемой строки.

Найти: новую матрицу A, содержащую $(n - 1)$-ую строку и m столбцов.

Фрагмент блок-схемы алгоритма.



Тест

n	m	k	Исходная матрица	Результат
3	3	2	$\begin{pmatrix}1 & 2 & 3\\0 & 0 & 0\\4 & 5 & 6\end{pmatrix}$	$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\end{pmatrix}$

Исполнение алгоритма.

i	j	A
2	1	$A[2, 1] = A[3, 1] = 4$
	2	$A[2, 2] = A[3, 2] = 5$
	3	$A[2, 3] = A[3, 3] = 6$

После выполнения циклов (блоки 1 и 2) матрица имеет следующий вид:

$$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\4 & 5 & 6\end{pmatrix}.$$

Уменьшаем количество строк матрицы (блок 4), после чего она примет искомый вид.

1.4.5. Нахождение строк или столбцов двумерного массива, обладающих заданным свойством

Пример 1.19. Дан двумерный массив $A[n, m]$. Найти количество строк, содержащих хотя бы один нуль.

Обозначим: k — количество строк, содержащих хотя бы один нуль; *Flag* — признак наличия нулей в строке. *Flag* = 1 означает, что нули в строке есть; *Flag* = 0 — нулей в строке нет.

Словесное описание алгоритма. Начинаем просматривать массив с первой строки. Берем первый элемент столбца ($j = 1$). Пока не просмотрен последний элемент столбца ($j \le m$) и не найден отрицательный элемент ($a[i, j] >= 0$), будем переходить к следующему элементу, увеличивая индекс элемента j ($j = j + 1$). Таким образом, мы закончим просмотр строки массива в одном из двух случаев:

1) просмотрели все элементы и не нашли отрицательного, тогда $Flag = 0$;

2) нашли нужный элемент, при этом $Flag = 1$. Увеличиваем значение количества строк k, содержащих нули.

Фрагмент блок-схемы алгоритма.

Блок 1. Присваиваем переменной k начальное значение.

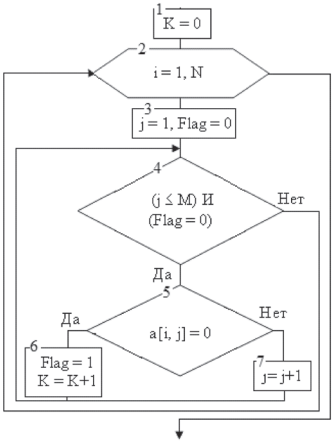
Блок 2. Берем очередную строку i.

Блок 3. Для выбранной строки задаем первый элемент строки и устанавливаем $Flag = 0$.

Блок 4. Выполняем итерационный цикл для всех элементов строки (блоки 5–7).

Блок 5. Сравниваем очередной элемент строки с нулем.

Блок 6. Если нулевой элемент найден (выход «Да» блока 5), то $Flag = 1$, увеличиваем значение k и заканчиваем просмотр строки (выход «Нет» блока 4).



Блок 7. Если нулевой элемент не найден (выход «Нет» блока 5), то продолжаем просмотр оставшихся элементов в строке, увеличивая индекс элемента j.

Покажем выполнение алгоритма на тестовом примере.

Тест

Данные			Результат
N	M	Матрица A	K
3	3	$\begin{pmatrix} 1 & 0 & 1 \\ 2 & 4 & 7 \\ 0 & 1 & 0 \end{pmatrix}$	2

Исполнение алгоритма.

i	*Flag*	*j*	(*j* <= *M*) и (*Flag* = 0)?	*A*[*i*, *j*] = 0?	K
1	0				0
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[1, 1] = 0? «Нет»	
		2	(2 <= 3) и (*Flag* = 0)? «Да»	*A*[1, 2] = 0? «Да»	
	1				1
			(2 <= 3) и (*Flag* = 0)? «Нет» (кц)		
2	0				
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 1] = 0? «Нет»	
		2	(2 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 2] = 0? «Нет»	
		3	(3 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 3] = 0? «Нет»	
	0	4	(4 <= 3) и (*Flag* = 0)? «Нет» (кц)		
3	0				
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[3, 1] = 0? «Да»	2
			(4 <= 3) и (*Flag* = 0)? «Нет» (кц)		

Итак, количество строк, в которых встречается хотя бы один нуль, равно 2, что соответствует условию задачи.

1.5. АЛГОРИТМЫ СОРТИРОВКИ

Сортировка — это процесс перестановки элементов некоторого заданного множества в определенном порядке. Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве.

В общем случае задача сортировки ставится следующим образом. Имеется массив, тип данных которого позволяет использовать операции сравнения («=», «>», «<», «>=» и «<=»). Задачей сортировки является преобразование исходного массива в массив, содержащий те же элементы, но в порядке возрастания (или убывания) значений.

Методы сортировки не должны требовать дополнительной памяти: все перестановки с целью упорядочения элементов массива должны производиться в пределах того же массива.

Будем выполнять сортировку по возрастанию.

Методы, сортирующие массив «на месте», делятся на три основных класса в зависимости от лежащего в их основе приема.

1. Сортировка включениями (вставкой).
2. Сортировка выбором.
3. Сортировка обменом.

1.5.1. Сортировки включениями (вставкой)

Сортировка простой вставкой

Идея алгоритма очень проста. Пусть имеется массив $a[1], a[2], \ldots, a[n]$. Пусть элементы $a[1], a[2], \ldots, a[i-1]$ уже отсортированы, и пусть имеем входную последовательность $a[i], a[i+1], \ldots, a[n]$. На каждом шаге, начиная с $i = 2$ и увеличивая i на единицу, берем i-й элемент входной последовательности и вставляем его на подходящее место в уже отсортированную часть последовательности.

Обозначим вставляемый элемент через x.

Пусть начальный массив 32 64 9 30 87 14 2 76.

$i = 2$ $x = 64$. Ищем для x подходящее место, считая, что $a[1] = 32$ — это уже отсортированная часть последовательности. Получаем *32 64* **9** 30 87 14 2 76.

$i = 3$ $x = 9$. Часть последовательности $a[1]$, $a[2]$ уже отсортирована. Ищем для x подходящее место: *9 32 63* **30** 87 14 2 76.

Аналогично выполняем последующие шаги сортировки.

$i = 4$ $x = 30$. Ищем для x подходящее место: *9 30 32 64* **87** 14 2 76.

$i = 5$ $x = 87$. Ищем для x подходящее место: *9 30 32 64 87* **14** 2 76.

$i = 6$ $x = 14$. Ищем для x подходящее место: *9 14 30 32 64 87* **2** 76.

$i = 7$ $x = 2$. Ищем для x подходящее место: *2 9 14 30 32 64 87* **76**.

$i = 8$ $x = 76$. Ищем для x подходящее место: *2 9 14 30 32 64 76* **87**.

При поиске подходящего места для элемента x мы чередовали сравнения и пересылки, т. е. как бы «просеивали» x, сравнивая его с очередным элементом $a[i]$ и либо

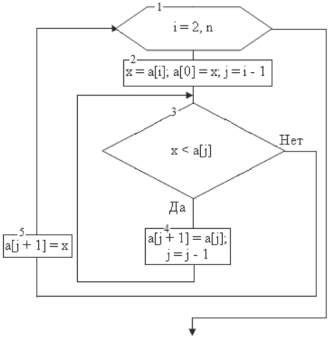
вставляя x, либо пересылая $a[i]$ направо и продвигаясь влево. Заметим, что «просеивание» может закончиться при двух различных условиях:

1) найден элемент, значение которого больше, чем x;

2) достигнут конец последовательности.

Это типичный пример цикла с двумя условиями окончания. При реализации алгоритма с целью окончания выполнения внутреннего цикла используют прием фиктивного элемента (барьера). Его можно установить, приняв $a[0] = x$.

Приведем фрагмент блок-схемы алгоритма и ее описание.



Блок 1. Начинаем цикл для перебора всех элементов исходного массива.

Блок 2. Присваиваем x значение очередного элемента массива. Устанавливаем барьер и продвигаемся назад по отсортированной части массива.

Блок 3. Начинам цикл для поиска подходящего места для значения x.

Блок 4. Пока x меньше очередного элемента (выход «Да» блока 3), выполняем сдвиг и продвигаемся к началу отсортированной части массива.

Блок 5. По выходу «Нет» блока 3 вставляем элемент x на нужное место.

Покажем исполнение алгоритма для рассмотренного примера.

i	x	$a[0]$	j	$x < a[j]$?	$a[j+1]$	Массив
2	64	64	1	64 < 32? «Нет»	$a[2] = x = 64$	32 64 9 30 87 14 2 76
3	9	9	2	9 < 64? «Да»	$a[3] = a[2] = 64$	32 64 64 30 87 14 2 76
			1	9 < 32? «Да»	$a[2] = a[1] = 32$	32 32 64 30 87 14 2 76
			0	9 < 9? «Нет»	$a[1] = x = 9$	9 32 64 30 87 14 2 76
4	30	30	3	30 < 64? «Да»	$a[4] = a[3] = 64$	9 32 64 64 87 14 2 76
			2	30 < 32? «Да»	$a[3] = a[2] = 32$	9 32 32 64 87 14 2 76
			1	30 < 9? «Нет»		$a[2] = x = 30$

После этого шага массив выглядит так: 9 30 32 64 87 14 2 76. Сортировка продолжается далее для оставшихся значений i.

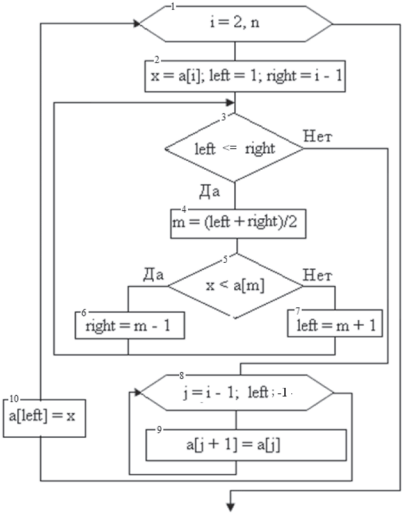
Сортировка бинарными включениями

Алгоритм сортировки простыми включениями имеет слабые места. Это, во-первых, необходимость перемещения данных, причем при вставке элементов, близких к концу массива, приходится перемещать почти весь массив. Второй недостаток — это необходимость поиска места для вставки, на что также тратится много ресурсов. Эту часть алгоритма можно улучшить, применив так называемый бинарный поиск. Этот метод на каждом шаге сравнивает x со средним (по положению) элементом отсортированной последовательности до тех пор, пока не будет найдено место включения.

Модифицированный алгоритм называется сортировкой бинарными включениями.

Обозначим *left* — левая граница отсортированного массива, *right* — его правая граница.

Приведем фрагмент блок-схемы алгоритма и ее описание.



Блок 1. Начинаем цикл для перебора всех элементов исходного массива.

Блок 2. Присваиваем x значение очередного элемента массива. Устанавливаем левую и правую границы отсортированной части массива.

Блок 3. Пока левая граница отсортированного массива не превосходит правую, выполняется тело цикла, состоящее из блоков 4–7.

Блок 4. Находится средний по положению элемент отсортированной части массива.

Блок 5. Значение x сравнивается с найденным элементом. По выходу «Да» блока 5 корректируется правая граница отсортированной части массива, по выходу «Нет» — левая.

При выходе из цикла с предусловием будет найдено положение вставляемого элемента.

Блоки 8–9 реализуют сдвиг элементов массива для вставки значения x.

Блок 10. Вставка значения в отсортированную часть массива.

Приведем выполнение алгоритма при сортировке массива 32 64 9 30 87 14 2 76.

i	*x*	*left*	*right*	*left* <= <= *right*?	*m*	*x* < *a*[*m*]?	*j*	*a*[*j* + 1]	*a*[*left*]
2	64	1	1	«Да»	1	32 < 64? «Да»			
			0	«Нет»			1	*a*[2] = 64	*a*[1] = 32
3	9	1	2	«Да»	1	9 < 64? «Да»			
			1	«Нет»			2	*a*[3] = *a*[2] = 64	
							1	*a*[2] = *a*[1] = 32	
									a[1] = 9
4	30	1	3	«Да»	2	30 < 32? «Да»			
			2	«Да»	1	30 < 9? «Нет»			
		2		«Нет»			3	*a*[4] = *a*[3] = 64	
							2	*a*[3] = *a*[2] = 32	*a*[2] = 30

После этого шага массив выглядит так: 9 30 32 64 87 14 2 76. Сортировка массива продолжается для следующих значений i.

Сортировка Шелла

Дальнейшим развитием метода сортировки включениями является сортировка методом Шелла, называемая по-другому сортировкой включениями с уменьшающимся расстоянием. Мы не будем описывать алгоритм в общем виде, ограничимся случаем, когда число элементов в сортируемом массиве является степенью числа 2. Для массива с 2^n элементами алгоритм работает следующим

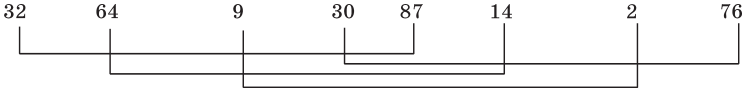
образом. На первом этапе производится сортировка включением всех пар элементов массива, расстояние между которыми равно 2^{n-1}. На втором этапе производится сортировка включением элементов полученного массива, расстояние между которыми равно 2^{n-2}. Алгоритм продолжается до тех пор, пока мы не дойдем до этапа с расстоянием между элементами, равным единице, и не выполним завершающую сортировку включениями.

Покажем сортировку Шелла на массиве, состоящем из 8 элементов.

На первом проходе отдельно группируются и сортируются все элементы, отстоящие друг от друга на четыре позиции. Этот процесс называется 4-сортировкой. В нашем примере из восьми элементов каждая группа содержит ровно два элемента. После этого элементы вновь объединяются в группы с элементами, отстоящими друг от друга на две позиции, и сортируются заново. Этот процесс называется 2-сортировкой. Наконец, на третьем проходе все элементы сортируются обычной сортировкой, или 1-сортировкой.

Начальное состояние 32 64 9 30 87 14 2 76.

$n = 3$.



На первом шаге сравниваются элементы, отстоящие друг от друга на 4 позиции: $a[1]$ и $a[5]$, $a[2]$ и $a[6]$, $a[3]$ и $a[7]$, $a[4]$ и $a[8]$.

На втором шаге сравниваются элементы, отстоящие друг от друга на 2 позиции: $a[1]$ и $a[3]$, $a[2]$ и $a[4]$, $a[3]$ и $a[5]$, $a[4]$ и $a[6]$, $a[5]$ и $a[7]$, $a[6]$ и $a[8]$.

На этом шаге сравниваются все соседние элементы.

Отсортированный массив:

2 9 14 30 32 64 76 87.

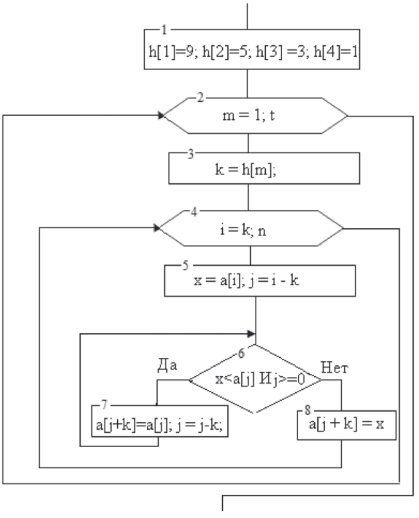
Таким образом, на каждом проходе либо участвуют сравнительно мало элементов, либо они уже довольно хорошо упорядочены и требуют относительно мало перестановок.

Очевидно, что этот метод в результате дает упорядоченный массив, и также совершенно ясно, что каждый проход будет использовать результаты предыдущего прохода, поскольку каждая i-сортировка объединяет две группы, рассортированные предыдущей 2^i-сортировкой. Также ясно, что приемлема любая последовательность приращений, лишь бы последнее было равно 1, так как в худшем случае вся работа будет выполняться на последнем проходе. Однако менее очевидно, что метод убывающего приращения дает даже лучшие результаты, когда приращения не являются степенями двойки [1].

Таким образом, алгоритм разрабатывается вне связи с конкретной последовательностью приращений. Все t приращения обозначаются через h_1, h_2, ..., h_t, с условиями $h_1 = 1$, $h_{i+1} < h_t$.

Каждая h-сортировка программируется как сортировка простыми включениями. Для того чтобы условие окончания поиска места включения было простым, используется барьер.

Ясно, что каждая h-сортировка требует собственного барьера, и программа должна определять его место как можно проще. Поэтому исходный массив нужно дополнить несколькими h-компонентами. Этот алгоритм называется сортировкой Шелла.



Приведем блок-схему алгоритма для $t = 4$. Примем $h[1] = 9$; $h[2] = 5$; $h[3] = 3$; $h[4] = 1$.

Исполнение алгоритма.

$$n = 8; \quad a = (32\ 64\ 9\ 30\ 87\ 14\ 2\ 76); \quad t = 4.$$
$$h[1] = 9; \quad h[2] = 5; \quad h[3] = 3; \quad h[4] = 1.$$

m	*k*	*i*	*x*	*j*	*j* <= 0Ux < *a*[*j*]	*a* [*j* + *k*]
1	9	9				
2	5	5	87	0	«Нет»	
		6	14	1	«Да»	*a*[6] = 32
				−4	«Нет»	*a*[1] = 14
		7	2	2	«Да»	*a*[7] = 64
				−1	«Нет»	*a*[2] = 2
		8	76	3	«Нет»	

После этой фазы массив выглядит так: 14 2 9 30 87 32 64 76.

В конечном счете, сортировка включениями оказывается не очень подходящим методом для ЭВМ, так как включение элемента с последующим сдвигом всего ряда элементов на одну позицию неэкономна.

1.5.2. Сортировка простым выбором

Сортировка простым выбором основана на следующем правиле.

В неупорядоченном массиве выбирается и отделяется от остальных элементов наименьший элемент. Наименьший элемент записывается на *i*-е место исходного массива, а элемент с *i*-го места — на место выбранного. Уже упорядоченные элементы (а они будут расположены начиная с первого места) исключаются из дальнейшей сортировки, поэтому длина оставшегося неупорядоченного массива должна быть на один элемент меньше предыдущего.

Улучшенный алгоритм — пирамидальная сортировка требует организации массива виде дерева. Желающих изучить этот алгоритм сортировки авторы отсылают к [1], где он прекрасно описан.

Сортировку простым выбором продемонстрируем на массиве 32 64 9 30 87 14 2 76.

$i = 1$, наименьшее значение 2, меняем местами 2 и 32. Массив после этого шага:

2 64 9 30 87 14 32 76.

$i = 2$, наименьшее значение в оставшейся части массива 9, меняем местами 9 и 64. Массив после этого шага:

2 9 64 30 87 14 32 76.

$i = 3$, наименьшее значение в оставшейся части массива 14, меняем местами 14 и 64. Массив после этого шага:

2 9 14 30 87 64 32 76.

$i = 4$, наименьшее значение в оставшейся части массива 30. Обмен не происходит, так как 30 стоит на своем месте.

$i = 5$, наименьшее значение в оставшейся части массива 32, меняем местами 32 и 87. Массив после этого шага:

2 9 14 30 32 64 87 76.

$i = 6$, наименьшее значение в оставшейся части массива 64. Обмен не происходит, так как 64 стоит на своем месте.

$i = 7$, наименьшее значение в оставшейся части массива 76, меняем местами 76 и 87. Массив после этого шага:

2 9 14 30 32 64 76 87.

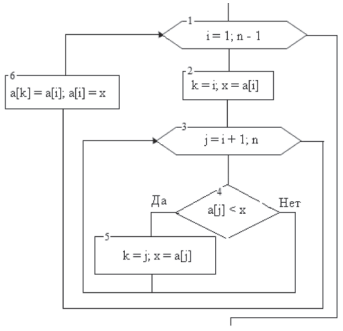
Массив отсортирован, но выполняется еще один шаг.

$i = 8$, наименьшее значение в оставшейся части массива 87. Обмен не происходит, так как 87 стоит на своем месте.

Таким образом, мы видим, что метод сортировки простым выбором основан на повторном выборе наименьшего элемента сначала среди n элементов, затем среди $(n - 1)$-го элемента и т. д. Возможна ситуация, когда обмен значений элементов не происходит.

Приведем блок-схему и описание алгоритма сортировки простым выбором.

Блок 1. Организует внешний цикл для просмотра элементов неупорядоченной части массива.



Блок 2. Запоминание индекса и значения i-го элемента, которые принимаем за начальные при поиске наименьшего элемента.

Блоки 3–5. Поиск значения наименьшего элемента и его номера в еще неупорядоченной части массива.

Блок 6. Обмен наименьшего и i-го элементов.

Приведем исполнение алгоритма для массива:

32 64 9 30 87 14 2 76; $n = 8$.

i	k	x	j	$a[j] < x$?	$a[k] = a[i]$; $a[i] = x$
1	1	32	2	a[2] < 32? «Нет»	
	3	9	3	a[3] < 32? «Да»	
			4	a[4] < 9? «Нет»	
			5	a[5] < 9? «Нет»	
			6	a[6] < 9? «Нет»	
	7	2	7	a[7] < 9? «Да»	A[7] = a[1] = 32; a[1] = 2
			кц		

После окончания внутреннего цикла массив имеет вид

2 64 9 30 87 14 32 76.

Сортировка продолжается для всех оставшихся значений параметра внешнего цикла i.

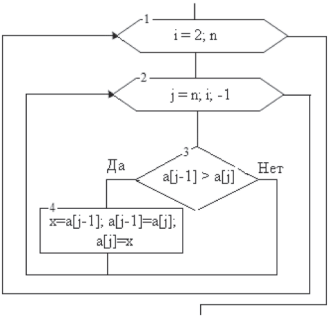
1.5.3. Обменные сортировки

Сортировка простым обменом

Простая обменная сортировка (называемая также «методом пузырька») для массива $a[1], a[2], \ldots, a[n]$ работает следующим образом. Начиная с конца массива, сравниваем два соседних элемента ($a[n]$ и $a[n-1]$). Если выполняется условие $a[n-1] > a[n]$, то значения элементов меняются местами. Процесс продолжается для $a[n-1]$ и $a[n-2]$ и т. д., пока не будет произведено сравнение $a[2]$

и $a[1]$. Понятно, что после этого на месте $a[1]$ окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но последними сравниваются $a[3]$ и $a[2]$. На последнем шаге будут сравниваться только значения $a[n]$ и $a[n-1]$. Понятна аналогия с пузырьком, поскольку наименьшие элементы (самые «легкие») постепенно «всплывают» к верхней границе массива.

Простая реализация алгоритма.



Блок 1. Арифметический цикл. Значение параметра цикла i определяет количество сравниваемых элементов во внутреннем цикле.

Блок 2. Задание номеров элементов для сравнения.

Блок 3. Сравнение двух соседних элементов.

Блок 4. Выполняется обмен элементов массива при выходе «Да» блока 3.

Сортируем массив 32 64 9 30 87 14 2.

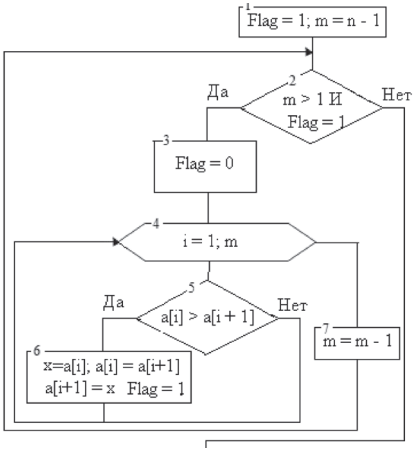
Выполнение алгоритма для $i = 2$ приведено ниже.

i	j	$a[j-1] > a[j]$?	$a[j-1]$, $a[j]$	Массив
2	8	2 > 76? «Нет»		32 64 9 30 87 14 2 76
	7	14 > 2? «Да»	$a[6] = 2$, $a[7] = 14$	32 64 9 30 87 **2** 14 76
	6	87 > 2? «Да»	$a[5] = 2$, $a[6] = 87$	32 64 9 30 **2** 87 14 76
	5	30 > 2? «Да»	$a[4] = 2$, $a[5] = 30$	32 64 9 **2** 30 87 14 76
	4	9 > 2? «Да»	$a[3] = 9$, $a[4] = 2$	32 64 **2** 9 30 87 14 76
	3	64 > 2? «Да»	$a[2] = 2$, $a[3] = 64$	32 **2** 64 9 30 87 14 76
	2	32 > 2? «Да»	$a[1] = 2$, $a[2] = 32$	**2** 32 64 9 30 87 14 76

После этого шага значение 2 встало на свое место, и алгоритм повторяется для $i = 3, 4, \ldots, n$.

Очевидный способ улучшить алгоритм — это запоминать, производился ли на данном проходе какой-либо обмен. Если нет, то алгоритм может закончить свою работу.

Используем переменную *Flag*, и пусть переменная $Flag = 1$, если на очередном шаге был выполнен обмен элементов, и $Flag = 0$, если обмена не было.



Приведем фрагмент блок-схемы и ее описание.

Блок 1. Задаем начальные значения $Flag = 1$ и $m = n - 1$.

Блок 2. Заголовок цикла. Пока не просмотрен весь массив и пока был обмен, выполняем внутренний цикл.

Блок 3. Предположим, что элементы массива уже отсортированы, тогда $Flag = 0$.

Блок 4. Организация арифметического цикла для сравнения и обмена соседних элементов массива.

Блоки 5–6. Тело внутреннего цикла. В блоке 5 выполняется сравнение соседних элементов.

При выходе «Да» блока 5 меняем местами элементы и фиксируем факт обмена в переменной *Flag*.

Блок 7. По окончании внутреннего цикла очередной элемент встал на свое место, уменьшается размер еще не отсортированной части массива.

Если был хотя бы один обмен, алгоритм повторяется.

Выполним алгоритм и отсортируем массив 32 64 9 30 87 14 2.

Flag	*m*	*m* > 1 и *Flag* = 1?	*i*	*a*[*i*] > *a*[*i* + 1]	*a*[*i*]	*a*[*i* + 1]
1	6	«Да»				
0			1	32 > 64? «Нет»		
1			2	64 > 9? «Да»	*a*[2] = 9	*a*[3] = 64
1			3	64 > 30? «Да»	*a*[3] = 30	*a*[4] = 64
			4	64 > 87? «Нет»		
1			5	87 > 14? «Да»	*a*[5] = 14	*a*[6] = 87
1			6	87 > 2? «Да»	*a*[6] = 2	*a*[7] = 87

В результате получим следующий массив: 32 9 30 64 14 2 87.

По выходе из внутреннего цикла $m = 6$ и $Flag = 1$, так как выполнялись обмены элементов массива. Начинаем внутренний цикл для нового значения переменной m.

Последний элемент находится на своем месте.

Продолжим сортировку для $m = 5$.

Flag	*m*	*m* > 1 и *Flag* = 1?	*i*	*a*[*i*] > *a*[*i* + 1]?	*a*[*i*]	a[i+ 1]
1		«Да»				
0			1	32 > 9? «Да»	*a*[1] = 9	a[2] = 32
1			2	32 > 30? «Да»	*a*[2] = 30	a[3] = 32
1	5		3	32 > 64? «Нет»		
			4	64 > 14? «Да»	*a*[4] = 14	a[5] = 64
1			5	64 > 2? «Да»	*a*[5] = 2	a[6] = 64
1			6	64 > 87? «Нет»		

Теперь массив имеет вид 9 30 32 14 2 64 87, и два последних элемента находятся на своих местах.

Так как были перестановки элементов, то алгоритм продолжается до полной сортировки массива.

Второе улучшение алгоритма связано с тем, что можно установить барьер: запоминать наименьшее значение индекса массива, для которого на текущем шаге выполнялись перестановки. Очевидно, что верхняя часть массива до элемента с этим индексом уже отсортирована, и на следующем шаге можно прекращать сравнения значений соседних элементов при достижении такого значения индекса.

В-третьих, метод пузырька работает неравноправно для так называемых «легких» и «тяжелых» значений. Один неправильно расположенный «пузырек» в «тяжелом» конце рассортированного массива всплывет на место за один проход, а неправильно расположенный элемент в «легком» конце будет опускаться на правильное место только за один шаг на каждом проходе. Например, массив

12 18 42 44 55 67 94 6

будет рассортирован за один проход, а сортировка массива

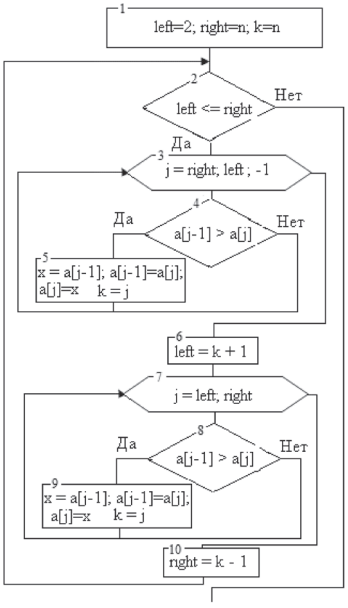
94 6 12 18 43 44 55 76

потребует семи проходов. Эта асимметрия подсказывает третье улучшение: менять местами направление следующих один за другим проходов. Этот улучшенный алгоритм называется шейкер-сортировкой.

Шейкер-сортировка

При ее применении на каждом следующем шаге меняется направление последовательного просмотра. В результате на одном шаге «всплывает» очередной наиболее легкий элемент, а на другом «тонет» очередной самый тяжелый.

Обозначим *left* — номер левой границы сортируемой части массива, *right* — номер его правой границы.



Блок 1. Установка номеров начальных границ сортируемого массива.

Блок 2. Вход в цикл. Пока левая граница не превосходит правую границу (выход «Да» блока 2) выполняем цикл.

Блок 3. Выполнение прохода массива вниз.

Блок 4. Сравнение соседних элементов.

Блок 5. Если $a[j-1] > a[j]$ (выход «Да» блока 4), производим обмен этих элементов и фиксируем номер элемента j, с которым производился обмен.

По окончании прохода вниз (выход блока 3) сдвигаем левую границу массива (блок 6) и выполняем проход вверх (блоки 7–9).

Покажем выполнение алгоритма для массива, состоящего из 7 элементов:

32 9 30 64 14 2 87.

Выполняем проход сверху вниз.

left	*right*	*k*	*left* <= *right*?	*j*	*j* > *left*?	*a*[*j* − 1] > *a*[*j*]?	Обмен
2	7	7	«Да»	7	«Да»	2 > 87 «Нет»	
		6		6	«Да»	14 > 2 «Да»	*x* = 14; *a*[5] = 2; *a*[6] = 14
		5		5	«Да»	64 > 2 «Да»	*x* = 64; *a*[4] = 2; *a*[5] = 64
		4		4	«Да»	30 > 2 «Да»	*x* = 30; *a*[3] = 2; *a*[4] = 30
		3		3	«Да»	9 > 2 «Да»	*x* = 9; *a*[2] = 2; *a*[3] = 9
		2		2	«Да»	32 > 2 «Да»	*x* = 32; *a*[1] = 2; *a*[2] = 32
				1	«Нет»		

Массив после этого прохода имеет вид 2 32 9 30 64 14 87.

Меняем направление движения и выполняем проход снизу вверх.

left	*right*	*k*	*j*	*j* <= *right*	*a*[*j* − 1] > *a*[*j*]?	
3	7	3	3	«Да»	32 > 9? «Да»	*x* = 32; *a*[2] = 9; *a*[3] = 32
		4	4	«Да»	32 > 30? «Да»	*x* = 32; *a*[3] = 30; *a*[4] = 32
			5	«Да»	32 > 64? «Нет»	
		6	6	«Да»	64 > 14? «Да»	*x* = 64; *a*[5] = 14; *a*[6] = 64
			7	«Да»	64 > 87? «Нет»	
			8	«Нет»		

Теперь массив имеет вид 2 9 30 32 14 64 87.

Опять меняем направление движения. Смена направления движения выполняется до тех пор, пока выполняется условие, записанное в блоке 2.

Анализ показывает [1], что сортировка обменом и ее небольшие улучшения хуже, чем сортировка включениями и выбором. Алгоритм шейкер-сортировки рекомендуется использовать в тех случаях, когда известно, что массив «почти упорядочен».

В 1962 г. Чарльз Хоар предложил метод сортировки разделением. Этот метод является развитием метода простого обмена и настолько эффективен, что его стали называть методом быстрой сортировки — QuickSort.

Метод требует использования рекурсивных функций и в рамках данного пособия не рассматривается.

1.5.4. Сравнение методов сортировки

На основании анализа, приводимого в [1], можно сделать следующие выводы об эффективности рассмотренных алгоритмов сортировки.

1. Преимущество сортировки бинарными включениями по сравнению с сортировкой простыми включениями мало, а в случае уже имеющегося порядка вообще отсутствует.

2. Сортировка методом «пузырька» является наихудшей среди сравниваемых методов. Ее улучшенная версия — шейкер-сортировка — все-таки хуже, чем сортировка простыми включениями и простым выбором.

3. Сортировка простым выбором является лучшим из простых методов.

1.6. АЛГОРИТМЫ ПОИСКА

Алгоритмы поиска занимают очень важное место среди прикладных алгоритмов, и это утверждение не нуждается в доказательстве.

Все алгоритмы поиска разбиваются на две большие группы в зависимости от того, упорядочен или нет массив данных, в котором проводится поиск. Рассмотрим простые алгоритмы поиска заданного элемента в одномерном массиве данных.

1.6.1. Последовательный поиск

Наиболее примитивный, а значит, наименее эффективный способ поиска — это обычный последовательный просмотр массива.

Пусть требуется найти элемент X в массиве из n элементов. Значение элемента X вводится с клавиатуры.

В данном случае известно только значение разыскиваемого элемента, никакой дополнительной информации о нем или о массиве, в котором его надо искать, нет. Поэтому для решения задачи разумно применить очевидный метод — последовательный перебор элементов массива и сравнение значения очередного элемента с заданным образцом.

Пусть $Flag = 1$, если значение, равное X, в массиве найдено, в противном случае $Flag = 0$. Обозначим k — индекс найденного элемента. Элементы массива — целого типа.

Элементы массива

Присвоим начальные значения переменным $Flag$ и k и возьмем первый элемент массива (блок 1).

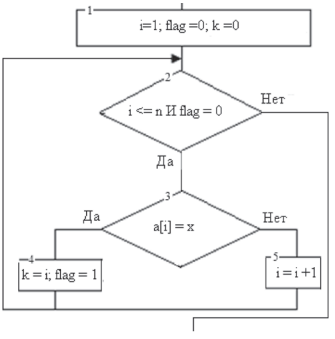
Пока не просмотрены все элементы и пока элемент, равный X, не найден (выход «Да» блока 2), выполняем цикл.

Сравниваем очередной элемент массива со значением X, и если они равны (выход «Да» блока 3), то запоминаем его индекс и поднимаем $Flag = 1$. Если элемент еще не найден (выход «Нет» блока 2), продолжаем просмотр элементов массива.

Выход из цикла (блок 2) возможен в двух случаях:

1) элемент найден, тогда $Flag = 1$, а в переменной k сохранен индекс найденного элемента;

2) элемент не найден, тогда $Flag = 0$, а k также равен 0.



1.6.2. Бинарный поиск

Если известно, что массив упорядочен, то можно использовать бинарный поиск.

Пусть даны целое число X и массив размера n, отсортированный в порядке неубывания, т. е. для любого k ($1 <= k < n$) выполняется условие $a[k - 1] <= a[k]$. Требуется найти такое i, что $a[i] = X$, или сообщить, что элемента X нет в массиве.

Идея бинарного поиска состоит в том, чтобы проверить, является ли X средним по положению элементом

массива. Если да, то ответ получен. Если нет, то возможны два случая.

1. X меньше среднего по положению элемента, следовательно, в силу упорядоченности массива можно исключить из рассмотрения все элементы массива, расположенные правее этого элемента, так как они заведомо больше его, который, в свою очередь, больше X, и применить этот метод к левой половине массива.

2. X больше среднего по положению элемента, следовательно, рассуждая аналогично, можно исключить из рассмотрения левую половину массива и применить этот метод к его правой части.

Средний по положению элемент и в том, и в другом случае в дальнейшем не рассматривается. Таким образом, на каждом шаге отсекается та часть массива, где заведомо не может быть обнаружен элемент X.

Пусть $X = 6$, а массив состоит из 10 элементов:

3 5 6 8 12 15 17 18 20 25.

1-й шаг. Найдем номер среднего по положению элемента: $m = [(1 + 10) / 2] = 5$. Так как $6 < a[5]$, то далее можем рассматривать только элементы, индексы которых меньше 5. Об остальных элементах можно сразу сказать, что они больше, чем X, вследствие упорядоченности массива и среди них искомого элемента нет.

3 5 6 8 **12** 15 17 18 20 25.

2-й шаг. Сравниваем лишь первые 4 элемента массива; значение $m = [(1 + 4) / 2] = 2$. $6 > a[2]$, следовательно, рассматриваем правую часть подмассива, а первый и второй элементы из рассмотрения исключаются:

3 **5** *6 8* 12 15 17 18 20 25.

3-й шаг. Сравниваем два элемента; значение $m = [(3 + 4) / 2] = 3$:

3 5 **6** *8* 12 15 17 18 20 25.

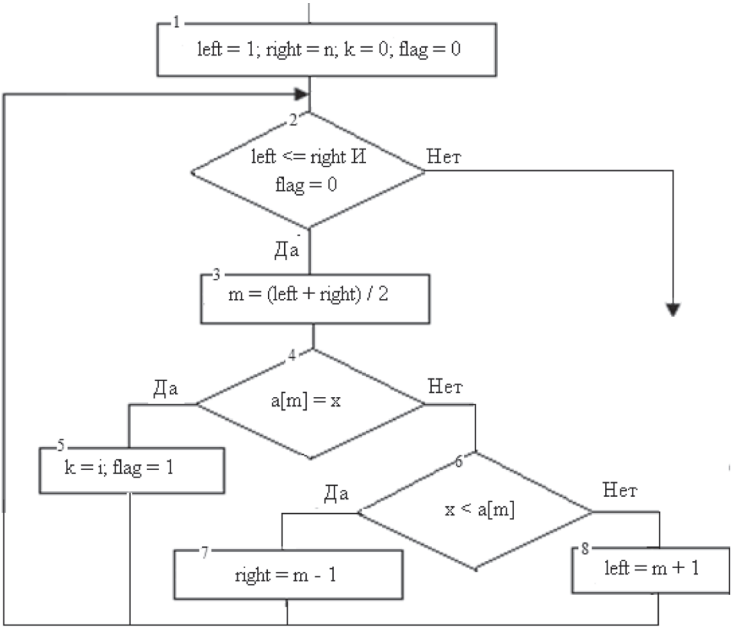
$a[3] = 6$. Элемент найден, его номер — 3.

В общем случае значение *m* равно целой части от дроби **[(*left* + *right*) / 2]**, где *left* — индекс первого, а *right* — индекс последнего элемента рассматриваемой части массива.

Если *Flag* = 1, то нужное значение в массиве найдено, а если *Flag* = 0, то значения, равного *X*, в массиве нет.

Алгоритм бинарного поиска можно применять только для упорядоченного массива. Это происходит потому, что данный алгоритм использует тот факт, что индексами элементов массива являются последовательные целые числа. По этой причине бинарный поиск практически бесполезен в ситуациях, когда массив постоянно меняется в процессе решения задачи. Алгоритм также используется при сортировке массивов методом бинарного включения (см. п. 1.5.1).

Ниже приведен фрагмент блок-схемы алгоритма бинарного поиска.



ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Дайте определение алгоритма. Перечислите свойства алгоритма.
2. Назовите отличия программного способа записи алгоритмов от других способов.
3. Назовите базовые алгоритмические структуры и дайте им краткую характеристику.
4. Дайте определение цикла с заданным числом повторений. Когда целесообразно применять циклы этого вида?
5. Что такое итерационные циклы? Когда возникает необходимость в их использовании?
6. Определите основные отличия между циклами с постусловием и предусловием. Как они выполняются?
7. Что называется рекуррентной формулой? Когда она применяется?
8. Дайте определение массива. Поясните, почему для хранения его элементов используется непрерывная память.
9. Можно ли при вводе или выводе элементов массива использовать цикл с предусловием или с постусловием?
10. Укажите, как изменится алгоритм нахождения наибольшего значения (пример 1.10), если все элементы массива — отрицательные числа.
11. Если известно, что в массиве обязательно имеется отрицательный элемент, то как изменится алгоритм решения задачи 1.14?
12. Дайте определение двумерного массива. Поясните особенности хранения элементов двумерного массива.
13. Почему при составлении алгоритмов для решения задач с использованием двумерного массива применяется вложенный цикл?
14. Можно ли при решении задачи 1.19 использовать цикл с постусловием? Ответ поясните.
15. Перечислите простые алгоритмы сортировки и укажите их основные отличия.
16. Почему сортировка включениями является неэкономным методом?
17. Какими характеристиками должен обладать массив, чтобы применение шейкер-сортировки было эффективным?
18. Чем сортировка Шелла отличается от сортировки простыми вставками?
19. Назовите метод сортировки, который является лучшим среди простых методов. Поясните, за счет чего это достигается.
20. Почему алгоритм бинарного поиска превосходит «слепой» поиск? Какими характеристиками должен обладать массив, в котором применяется алгоритм бинарного поиска?

ГЛАВА 2

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Программа — это алгоритм, записанный на каком-либо языке программирования (формальном языке).

Программа предназначена для обработки данных (управления данными). В процессе работы приложения сама программа (или ее часть) и данные (или их часть) находятся в оперативной памяти компьютера (ОЗУ).

Язык программирования является инструментом для написания программ. Языки программирования, в том числе C++, относятся к *формальным языкам*. Задачей формальных языков является описание алгоритмов и функций, которые выполняет компьютер. Формальные языки отличает небольшой набор изобразительных средств соответственно целям этих языков. В состав системы правил алгоритмического языка входят две группы.

1. *Синтаксис* — формальный набор правил, определяющий способ построения любых конструкций языка.

2. *Семантика* — множество правил, определяющих смысл синтаксических конструкций.

Алгоритм решения задачи записывается программистом согласно синтаксическим правилам языка, и это программа. Семантика реализована в компиляторе, который переводит код во внутреннее представление. Компилятор обнаруживает синтаксические ошибки, допущенные в коде программы.

Для разработки приложений в современном программировании используются интегрированные среды разработчика (Integrated Developer Environment — IDE, или

Rapid Application Development — RAD), которые позволяют программировать быстро и надежно. Современные среды разработчика являются многоязыковыми и позволяют в одном проекте кодировать на разных языках. Так, Visual Studio.Net поддерживает языки C++, C#, Basic и Fortran#.

Обязательными этапами обработки кода программы на языке C++ являются:

- препроцессорное преобразование;
- компиляция;
- компоновка.

Задачей *препроцессора* является преобразование текста программы до ее компиляции. Препроцессор сканирует текст, находит в нем команды, называемые директивами препроцессора, и выполняет их. В результате происходят изменения в исходном тексте: вставка фрагментов текста и замена некоторых его фрагментов. Полученный текст называется полным текстом программы.

На этапе *компиляции* полный код программы преобразуется во внутреннее машинное представление, некоторую последовательность команд, которая понятна компьютеру. Компилятор находит ошибки несоответствия кода программы синтаксическим правилам языка.

На этапе *компоновки* происходит редактирование связей и сборка исполнимого кода программы. Компоновщик обрабатывает все вызовы библиотечных функций и выполняет их подключение. Таким образом, к компилированному исходному коду добавляются необходимые функции стандартных библиотек. Готовый код является исполнимым и может быть выполнен компьютером.

2.1. НАЧАЛЬНЫЕ СВЕДЕНИЯ О ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C++

Чтобы писать хорошие программы, недостаточно знать правила записи синтаксических конструкций, необходимо отчетливо представлять себе механизмы, встроенные в реализацию компилятора, которые, собственно, и являются той самой семантикой.

2.1.1. Алфавит языка С++ и лексемы

Алфавит — набор символов, разрешенных для построения синтаксических конструкций. Алфавит языка С++ содержит четыре группы символов.

1. *Буквы*. Разрешается использовать буквы латинского алфавита, прописные (A–Z) и строчные (a–z). Русские буквы не входят в алфавит, но используются в комментариях и текстовых константах, там, где они не влияют на смысл программы.

2. *Цифры*. Используются арабские цифры 0, 1, …, 9.

3. *Специальные символы*. Они могут быть разделены на подгруппы:

- знаки препинания (разделители): , ; . : ;
- знаки операций: +, –, *, /, % , &, |, ?, !, <, =, >;
- парные скобки: [ ], { }, ( ), " ", ' ';
- прочие символы: _, #, ~, ^.

4. *Невидимые символы*. Они могут считаться разделителями, их особенность в том, что символы существуют (каждый имеет код), но в редакторе не видны. Это такие символы, как пробел, табуляция, разделитель строк. Их общее название — обобщенные пробельные символы.

Из символов алфавита строятся все конструкции языка.

Лексема — это единица текста (конструкция, слово), воспринимаемая компилятором как единое неделимое целое. Можно выделить пять классов лексем.

1. *Имена* (идентификаторы) для именования произвольных объектов программы, например, x1, Alpha, My_file.

2. *Служебные* (ключевые) слова, обозначающие конструкции языка (имена операторов), например, for, while, do.

3. *Константы*, например, 1, 12.5, “Василий”.

4. *Операции* (знаки операций), например, ++, >=, !=, >.

5. *Разделители* (знаки пунктуации), например, [ ], ( ), { }.

2.1.2. Концепция данных в языке C++

Данные — все, что подлежит обработке с помощью программы. Данные, используемые приложением, должны быть описаны в программном коде. Классификацию данных можно выполнять по нескольким категориям.

1. *По типу*. Каждое данное имеет тип. Типы данных можно разделить на *базовые*, т. е. такие, правила организации которых предопределены реализацией языка, и *конструируемые*, т. е. те, которые пользователь строит по определенным правилам для конкретной задачи.

2. *По способу организации*. Для каждого из простых (базовых) типов каждое данное может быть неизменяемым или изменяемым. По способу организации данные делятся на два класса:

- *константа* — данное, которое не меняет своего значения при выполнении программы и присутствует в тексте программы явным образом. Тип константы определен ее записью;
- *переменная* — данное, которое изменяется при выполнении программы и в тексте присутствует своим именем (идентификатор). Тип переменной величины должен быть объявлен в тексте программы.

Каждое данное, независимо от способа организации, обладает типом. Тип данного очень важен, так как он определяет:

- *механизм* выделения памяти для записи значений переменной;
- *диапазон* значений, которые может принять переменная;
- *список* операций, которые разрешены над данной переменной.

Механизм выделения памяти и внутреннего представления данного важнее всего, диапазон значений и операции являются следствием из первого.

Тип любого данного программы должен быть объявлен обязательно. Для констант тип определен явно записью константы в тексте программы. Для переменных тип должен быть задан в объявлении объекта. При объявле-

нии переменной происходит выделение области памяти, размер которой определен типом переменной и в которой хранится значение переменной.

Классификация типов данных:

- *простые типы* (базовые, скалярные, внутренние, предопределенные);
- *конструируемые типы* (массивы, строки, структуры, функции и др.).

Основные типы данных определены ключевыми словами, список которых приведен в таблице 2.1.

Таблица 2.1

Ключевые слова, определяющие основные типы данных

Имя типа	Значение типа	Размер	Диапазон значений
char	Символьный (целое)	1 байт	–128–127
int	Целый	4 байта	±2 147 483 649
float	С плавающей точкой (действительный)	4 байта	3.4e-38–3.4e38
double	Двойной точности (действительный)	8 байт	1.7e-308–1.7e308
void	Любой или никакой		Не ограничен

Замечания

1. В языке С++ для определения размера памяти, занимаемой объектом, есть операция sizeof(), например, sizeof(int); sizeof(double); sizeof(My_obj).

В качества аргумента операции можно указать имя типа (int, double) или имя объекта (My_obj).

2. Существует специальный тип данных void, который относится к базовым, но имеет существенные особенности. Про данное такого типа говорят, что оно «не имеет никакого типа». На самом деле тип void используется при работе с динамическими данными и может адресовать пространство произвольного размера, где можно разместить данное любого типа. Значит, можно считать, что данное типа void может представлять «данное любого типа».

3. Основные типы данных можно изменить (модифицировать) с использованием вспомогательных ключевых слов long и unsigned:

- long (длинный) увеличивает объем выделяемой памяти в 2 раза;
- unsigned (без знака) не использует знаковый бит, за счет чего хранимое значение может быть увеличено.

2.1.3. Константы в языке C++

Синтаксис языка выделяет пять типов констант: целые, действительные (вещественные), символьные, перечислимые, нулевой указатель.

Целые константы

Синтаксис языка позволяет использовать константы трех систем счисления: десятичные, восьмеричные, шестнадцатеричные. Основание определяется префиксом в записи константы. По умолчанию основание 10, префикс 0 предваряет восьмеричную константу, префикс 0x или 0X предваряет шестнадцатеричную константу. В остальном запись целых констант соответствует общепринятой.

Примеры записи целых констант приведены в таблице 2.2.

Таблица 2.2

Примеры записи целых констант

Десятичные	Восьмеричные	Шестнадцатеричные
127 –256	012 –014	0xA –0x10

Целые числа в памяти компьютера представлены в форме с фиксированной точкой. Эта форма позволяет представить значение с абсолютной точностью. На рисунке 2.1 показано представление целого числа в двухбайтовой ячейке памяти.

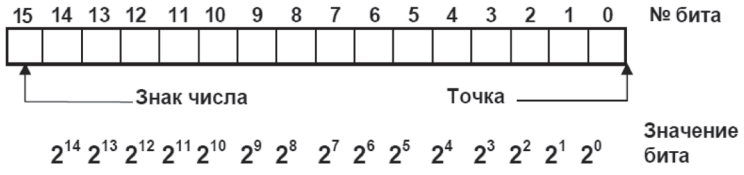


Рис. 2.1
Представление целых чисел

Действительные константы

Вещественные (действительные) числа представлены в памяти компьютера в форме с плавающей точкой в виде $M \cdot 10^p$, где M — мантисса вещественного числа; 10 — основание системы счисления; p — показатель десятичной степени, целое число со знаком. Основание системы счисления в языке С++ заменяется буквой *e* или *E*. Некоторые из составляющих могут быть опущены.

Примеры записи действительных констант:

```
44.  3.1415926  44.e0  .31459E1  0.0
```

На рисунке 2.2 показано представление вещественного числа в четырехбайтовой ячейке памяти.

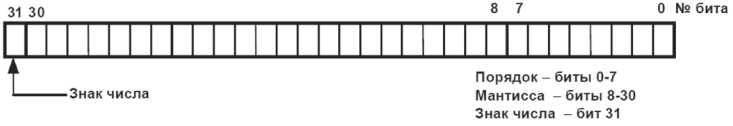


Рис. 2.2
Представление вещественных чисел

Символьные константы

Символьная константа — это лексема, которая содержит произвольный символ, заключенный в одинарные кавычки (апостроф). Хранится в одном байте памяти в виде целого числа, определяющего целочисленный код символа. Символьной константой может быть любой символ, изображаемый на экране в текстовом режиме, или не имеющий отображения (пробельный символ).

Примеры символьных констант: '!', '.', 's', 'A', '+', '2'.

Существуют также управляющие символы (*ESC*-последовательности), которые используются в строковых константах и переменных. Они имеют признак '\' (backslash или обратный слеш), который используется для указания управляющего воздействия. Вот их неполный список:

'\0' — нулевой символ;
'\n' — перевод строки;
'\r' — возврат каретки;

'\t' — табуляция;

'\' — апостроф;

'\\' — обратный слэш;

'\ddd' — восьмеричное представление символьной константы, где ddd — ее восьмеричный числовой код, например, '\017' или '\233';

'\xhh' или '\Xhh' — шестнадцатеричное представление символьной константы, где hh — ее числовой код, например, '\x0b' или '\x1F'.

Символьные константы имеют целый тип и могут использоваться в выражениях.

Строковые константы

Формально строки не относятся к константам языка C++, а представляют отдельный тип его лексем. Строковая константа определяется как набор символов, заключенных в двойные кавычки " ".

Например,

```
#define STR "Программа"
```

В конце строковой константы добавляется нулевой символ '\0'. В тексте строки могут встретиться *ESC*-последовательности, которые будут выполнять управляющее воздействие.

Например,

```
"\nПри выводе \nтекст может быть \nразбит на строки."
```

Здесь "\n" выполняет управляющее воздействие, и вывод будет таким:

```
При выводе
текст может быть
разбит на строки.
```

При хранении строки все символы, в том числе управляющие, хранятся последовательно, и каждый занимает ровно один байт.

Типы числовых констант и модификаторы типов

Для модификации типа числовых констант используются префиксы и суффиксы, добавляемые к значению константы. Суффикс l(L) от слова long увеличивает длину данного в два раза, суффикс u(U) от слова unsigned

используется для чисел, не содержащих знак, т. е. положительных, при этом увеличивается число разрядов, отводимых под запись числа, а значит, и его возможное значение.

Например, длинные константы 12345l, –54321L; константы без знака 123u, 123U; шестнадцатеричная длинная константа 0xb8000000l; восьмеричная длинная константа без знака 012345LU.

2.1.4. Переменные в языке С++

Одним из основных понятий в языках программирования является объект. Будем использовать это понятие как объект программного кода.

Объект — это некоторая сущность, обладающая определенным набором свойств и способная хранить свое состояние.

Объектами программного кода являются константы, переменные, функции, типы и т. д.

Переменная — это объект, для использования которого необходимо определить три характеристики:

- имя;
- тип;
- значение (не обязательно).

Имя переменной (идентификатор) обозначает в тексте программы величину, изменяющую свое значение. Для каждой переменной выделяется некоторая область памяти, способная хранить ее значение. Имя позволяет осуществить доступ к области памяти, хранящей значение переменной.

Тип переменной определяет размер выделяемой памяти и способ хранения значения (см. рис. 2.1, 2.2).

Значение переменной не определено вначале, изменяется при присваивании переменной значения.

Имя переменной (идентификатор) включает в себя латинские буквы, цифры, знак подчеркивания "_". Первым символом должна быть буква.

Примеры идентификаторов: x1, Price, My_file1, alpha, PI. В качестве имен рабочих переменных используются простые имена i, j, k1, k2 и им подобные.

Ограничения:

- длина имени содержит не более 32-х символов;
- в качестве имен нельзя использовать ключевые слова языка C++, например, названия операторов for, if, do;
- в качестве имен не рекомендуется использовать имена объектов стандартных библиотек, например, имена функций sin(), sqrt(), pow();
- имена, которые начинаются со знака подчеркивания, зарезервированы для использования в библиотеках и компиляторах, поэтому их не следует выбирать в качестве прикладных имен, например, _acm, _AX, _BX.

Перечень основных, не всех служебных слов приведен в таблице 2.3.

Таблица 2.3

Служебные слова в языке C++

Типы данных	char, double, enum, float, int, long, short, struct, signed, union, unsigned, void, typedef
Квалификаторы типа	const, volatile
Классы памяти	auto, extern, register, static
Названия операторов	break, continue, do, for, goto, if...else, return, switch, while, case, default, sizeof
Модификаторы	asm, far, interrupt, pascal и др.
Псевдопеременные	Названия регистров _AH _DS _DX и др.

Замечания

1. Рекомендуется использовать имена переменных, отражающие их первоначальный смысл, например, Count (количество), X_coord (координата по x), Old_value_of_long (предыдущее значение длины).

2. Заглавные и строчные буквы считаются различными. Так, разными объектами будут переменные, имеющие имена Alpha и alpha.

3. Идентификаторы, зарезервированные в языке, являются служебными (ключевыми) словами. Следовательно, их нельзя использовать как имена свободных объектов программы. К ним относятся типы данных, квалификаторы типа, классы памяти, названия операторов, модификаторы, псевдопеременные.

2.1.5. Объявление переменных

Каждая переменная должна быть объявлена в тексте программы перед первым ее использованием. Начинающим программистам рекомендуется объявлять переменные в начале тела программы перед первым исполнимым оператором. Синтаксис объявления переменной требует указать ее тип и имя:

Имя_типа Имя_переменной;

Можно одновременно объявить несколько переменных, тогда в списке имен они отделяются друг от друга запятой, например:

```
int a, b, c;              // целые со знаком
char ch, sh;              // однобайтовые символьные
long l, m, k;             // длинные целые
float x, y, z;            // вещественные
long double u, v          // длинные двойные
```

На этапе компиляции для объявленных переменных в ОЗУ будет выделена память для записи их значений. Имя переменной определяет адрес выделенной для нее памяти, тип переменной определяет способ ее хранения в памяти. Как следствие, тип определяет диапазон ее возможных значений и операции, разрешенные над этой величиной.

2.1.6. Инициализация переменных

Переменные, объявленные в программе, не имеют значения, точнее имеют неопределенные значения. Так как память, выделенная под запись переменных, не очищается, то значением переменной будет «мусор» из того, что находилось в этой памяти ранее.

Инициализация — это прием, который позволяет присвоить значения переменным при их объявлении. Синтаксис инициализации:

Имя_типа Имя_переменной = Начальное_значение;

Например,

```
float pi=3.1415;
unsigned Year=2014;
```

2.1.7. Именованные константы

В языке C++, кроме переменных, можно использовать *именованные* константы, т. е. константы, имеющие фиксированные названия (имена). Имена могут быть произвольными идентификаторами, не совпадающими с другими именами объектов. Принято использовать в качестве имени константы большие буквы и знаки подчеркивания, что визуально отличает имена переменных от имен констант. Способов определения именованных констант три.

Первый способ — это константы перечисляемого типа, имеющие синтаксис:

enum тип_перечисления {список_именованных_констант};

Здесь «тип_перечисления» — это его название, необязательный произвольный идентификатор; «список_именованных_констант» — это разделенная запятыми последовательность вида:

имя_константы = значение_константы

Примеры.

```
enum BOOL {FALSE, TRUE};
enum DAY {SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                    THURSDAY, FRIDAY, SATURDAY};
enum {ONE=1, TWO, THREE, FOUR, FIVE};
```

Если в списке нет элементов со знаком «=», то значения констант начинаются с 0 и увеличиваются на 1 слева направо. Так, FALSE равно 0, TRUE равно 1. SUNDAY равно 0 и так далее, SATURDAY равно 6. Если в списке есть элементы со знаком «=», то каждый такой элемент получает соответствующее значение, а следующие за ним увеличиваются на 1 слева направо, так, значение ONE = 1, TWO = 2, THREE = 3, FOUR = 4, FIVE = 5.

Второй способ — это именованные константы, заданные с ключевым словом const в объявлении типа:

const Имя_типа Имя_константы = Значение_константы;

Ключевое слово const имеет более широкий смысл и используется также в других целях, но здесь его суть в том, что значение константы не может быть изменено.

Примеры.

```
const float Pi=3.1415926;
const long M=999999999L;
```

Попытка присвоить значение любой из констант Pi, M будет неудачной. Визуализация имен констант с помощью использования больших букв необязательна.

Третий способ ввести именованную константу дает препроцессорная директива #define, подробный синтаксис и механизм выполнения которой будет рассмотрен далее.

2.2. ОПЕРАЦИИ И ВЫРАЖЕНИЯ ЯЗЫКА С++

Операция — это символ или лексема вычисления значения. Операции используются для формирования и вычисления значений выражений и для изменения значений данных.

2.2.1. Классификация операций

Классифицировать операции можно по нескольким признакам. Приведем два варианта: по типу возвращаемого значения и по числу операндов.

1. *Тип возвращаемого значения* является важным свойством любой операции. В таблице 2.4 приведена классификация операций по типу возвращаемого значения.

Таблица 2.4

Классификация операций по типу возвращаемого значения

Арифметические	Логические
+ сложение	> больше
– вычитание	< меньше
* умножение	== равно
/ деление (для целых операндов — целая часть от деления)	!= не равно
% остаток от деления (только для целых)	>= больше или равно
	<= меньше или равно
	&& логическое «И»
	‖ логическое «ИЛИ»
	! логическое «НЕ»

Арифметические операции имеют обычный математический смысл. Для целого типа данных существуют две операции деления: целочисленное деление / и остаток от деления %. Каждая из этих операций, примененная к данным целого типа, возвращает целое значение.

Пример 2.1.

```
int x=5, y=3;
x/y              // результат = 1
x%y              // результат = 2
y/x              // результат = 0
y%x              // результат =3
x=–5             // знак сохраняется
x/y              // результат=–1
x%y              // результат= –2
y/x              // результат=0
y%x              // результат=3
```

Логические операции предназначены для проверки условий и для вычисления логических значений.

Замечание. Логического типа данных в стандарте языка C++ нет, его имитирует тип int (условно принято, что 0 — это значение «Ложь», а все отличные от нуля значения — это «Истина»). Компилятор Visual Studio 2013 в качестве логического типа позволяет использовать тип bool, например:

```
bool yes;
yes=x>0;
```

Логическими являются операции, возвращающие логическое значение. Их можно, в свою очередь, разбить на две группы.

Операции отношения. Связывают данные числовых или символьных типов и возвращают логическое значение, например:

```
3>1              // истина, не равно 0
'a'>'b'          // ложь равна 0
x>=0             // зависит от x
y!=x             // зависит от x и от y
```

Значение операции отношения может быть присвоено целой переменной или переменной логического типа, например:

```
int a=5, b=2;
bool c;
c=a>b;           // c = 1
c=a<b;           // c = 0
c=a==b;          // c =0
c=a!=b;          // c = 1
```

Операции отношения могут участвовать в записи выражения наряду с арифметическими операциями, тогда при вычислении значения выражения важен порядок вычисления, например:

```
c=10*(a>b);       // a > b = 1, 10 * 1 = 10, значит, c = 10
c=10*(10*a<b); // 10 * a = 50, 50 < b = 0, 10 * 0 = 0, c =0
c=1<a&&a<10;  // 1 < a = 1, a < 10 = 1, значит, c = 1
```

Собственно логические операции. Связывают данные логического типа (целые) и возвращают логическое значение. Это логические операции конъюнкция &&, дизъюнкция || и отрицание !, которые применяются к значениям операндов, имеющих логическое значение. Если операндом является отношение, то предварительно вычисляется его значение, например:

```
// оба операнда (x>0 и y>0) одновременно истинны
                                              x>0 && y>0;
// хотя бы один операнд (x % 2==0 или y % 2==0) истинен
x%2==0||y%2==0;
// значение, обратное значению операнда
// (x*x+y*y<=r*r)
!(x*x+y*y<=r*r);
```

2. Второй вид классификации операций языка С++ — классификация по *числу операндов*.

1) *Унарные* операции имеют только один операнд, например:

```
+10, –a, !(x>0), ~a.
```

2) *Бинарные* операции имеют два операнда, например:

```
x+5, x–y, b&&c.
```

3) *Тернарная* операция имеет три операнда, это операция *условия*, приведем ее синтаксис здесь:

```
Логическое_выражение ? Выражение1 : Выражение2
```

Операция *условия* выполняется в два этапа. Сначала вычисляется значение логического выражения. Затем, если оно истинно, вычисляется значение второго операнда, который назван «Выражение1», оно и будет результатом, иначе вычисляется значение третьего операнда, который назван «Выражение2», и результатом будет его значение.

Например, вычисление абсолютного значения некоторого *i*:

```
int abs;
abs=(i<=0)?–i:i;
```

2.2.2. Операции изменения данных

Операции изменения данных имеют особое значение, позволяя выполнить присваивание значения переменной.

Список операций изменения данных приведен в таблице 2.5.

Таблица 2.5

Операции изменения данных

Основные операции	Арифметические операции с присваиванием
= операция присваивания	+= сложение с присваиванием
++ увеличение на единицу	−= вычитание с присваиванием
−− уменьшение на единицу	*= умножение с присваиванием
	/= деление с присваиванием
	%= остаток от деления с присваиванием

Операция присваивания в C++ — это обычная операция, которая является частью выражения, но имеет очень низкий приоритет и поэтому выполняется в последнюю очередь. Семантика этой операции заключается в том, что вычисляется выражение правой части и присваивается левому операнду. Это означает, что слева может быть только переменная величина, которая способна хранить и изменять свое значение (левостороннее выражение). В правой части, как правило, записывается вычисляемое выражение.

Пример 2.2.

```
x=2;                  // x принимает значение константы
cond=(x<=2);          // cond принимает логическое
                      // значение
3=5;                  // ошибка, слева константа
x=x+1;                // x принимает значение,
                      // увеличенное на 1
```

Расширением операции присваивания являются операции вычисления с присваиванием, которые еще называются «присваивание после» и позволяют сократить запись выражений, например:

```
x+=5;                    // аналог записи x = x + 5
y*=3                     // аналог записи y = y * 3
```

Пример 2.3.

Пусть c = 5, a = 3; тогда

```
c+=a;                    // c = 8
c–=a;                    // c = 2
c*=2;                    // c = 10
c/=4;                    // c = 1
```

Необычными операциями являются операции *увеличения* и *уменьшения* на *единицу*, имеющие самостоятельные названия *инкремент* и *декремент*, например:

```
x++;                     // аналог записи x = x + 1 или x += 1
– –x;                    // аналог записи x = x – 1 или x –= 1
```

Каждая из этих операций имеет две формы.

1. Префиксная ++x или – –x. При ее выполнении x изменяется прежде, чем выполняются другие операции, входящие в выражение.

2. Постфиксная x++ или x– –. При ее выполнении сначала выполняются все операции выражения, и только потом изменяется x.

Пример 2.4.

```
int q, a;
a=2;
q=2*++a;   // сначала a увеличивается на 1, т. е. a = 3,
           // затем a умножается на 2 и результат
           // присваивается q
q=2*a++;   // сначала a умножается на 2, результат
           // присваивается
           // q, затем a увеличивается на 1,
           // после чего a = 5
```

2.2.3. Выражения в зыке С++

Выражение — это правило вычисления одного значения.

Формально выражение — это несколько операндов, объединенных знаками операций. В качестве операндов выступают константы, имена переменных, вызовы операций-функций. Кроме того, в запись выражения могут входить технические символы, такие как скобки ().

Как правило, смыслом выражения является не только вычисление значения, но и присваивание его переменной, поэтому в записи выражения присутствует операция присваивания «=» или ее клоны.

Поскольку все данные в C++ (переменные, константы и другие объекты) имеют тип, то значение, вычисленное выражением, также будет иметь тип, который определяется типом операндов, например, как в таблице 2.6.

Таблица 2.6

Примеры определения типа выражения

Операнды	Операции	Результат
Целые	+ – * / %	Целый
Вещественные	+ – * /	Вещественный
Числовые или символьные	> < != == >= <=	Логический (int, bool)
Логические	&& ! ||	Логический

Если тип вычисленного выражением значения не совпадает с типом переменной левой части, то происходит *преобразование* (*приведение*) типов.

Порядок вычисления выражений определяется *рангом* (*приоритетом*) входящих в него операций и правилами ассоциативности, которые применяются при вычислении значений выражений, если несколько операций одного приоритета стоят подряд. Большинство операций левоассоциативны (→), т. е. вычисляются слева направо естественным образом, например, бинарные операции. Однако есть правоассоциативные операции (←), которые вычисляются справа налево.

Например, операция присваивания позволяет выполнить цепочку присваиваний: x = y = z = 10;

Здесь цепочка раскрывается по порядку z = 10, затем y = z, затем x = y.

В таблице 2.7 приведены приоритеты операций C++.

Из таблицы 2.7 видно, что принятый ранг операций наиболее близок к математическому, также как и принятый порядок их вычисления. Так, умножение и деление (мультипликативные операции) старше сложения и вычитания (аддитивные операции). Унарные операции + и – старше бинарных, стало быть, знак операнда вычисляет-

Таблица 2.7

Полная таблица приоритетов операций С++

Ранг (приоритет)	Операции	Ассоциативность
1	() [] –> (операции разыменования)	→
2	! ~ + – ++ –– & * (тип) sizeof	←
3	* / % (бинарные)	→
4	+ – (бинарные)	→
5	<< >> (сдвиг)	→
6	< <= > >= (отношения)	→
7	== != (отношения)	→
8	&	→
9	^ (поразрядное исключающее или)	→
10	|	→
11	&&	→
12	||	→
13	? : (условная операция)	←
14	= *= /= %= += –= &= ^= |= <<= >>=	←
15	, (операция запятая)	→

ся в первую очередь. Операция присваивания и ее клоны младше прочих, что позволяет выполнить присваивание только после того, как значение выражения вычислено полностью. Операции отношения младше арифметических операций, что позволяет использовать естественную запись логических выражений, например, x>0 && y>0. Здесь в первую очередь вычисляются значения отношений, которые затем являются операндами конъюнкции.

Пример 2.5.

```
int c, a=5, b=3;
c=a++;                  // c=5, a=6
c=a++*3;                // c=a*3=15, потом a=7
c=--a/2;                // --a, a=6, потом c=a/2=3
c=a+++b;                // сначала c=a+b=9, потом a++, a=7
c=a++ + ++b;            // ++b=4, потом c=a+b=11, потом a++8
```

Приведение типов

Под *приведением* типов понимаются действия, которые компилятор выполняет в случае, когда в выражении смешаны операнды различных типов, чего, строго говоря, не следует допускать. Механизм приведения типов в С++ заключается в том, что при вычислении значения выраже-

ния компилятор, не изменяя внутреннего представления данных, преобразует данное «меньшего» типа к «большему», где «величину» типа определяет размер выделенной для него памяти. Такое преобразование выполняется без ведома программиста и может привести к потере данных.

Пример 2.6. Известно, что целое значение представлено в памяти точно, а вещественное приближенно.

```
int a=5, b=2;
float c;
c=a*1.5;          // в выражении операнды разных типов
                  // автоматически включаются механизмы
                  // приведения типов, c=7.5
```

Преобразование типов

Под преобразованием типов в выражениях понимаются действия, выполняемые при присваивании.

Пример 2.7.

```
int a=5, b= 2;
float c;
c=a/b;
```

Хочется думать, что значение c будет равно 2.5, ведь оно вещественное, но порядок операций таков, что деление старше присваивания и оно выполняется с операндами целого типа, и его результат c = 2, т. е. приведение типа будет выполнено только при присваивании.

Для того чтобы избежать потери информации, в выражениях следует применять операцию явного преобразования типа, синтаксис которой:

```
(Имя_типа) Имя_переменной;
```

Эта запись включается в те выражения, в которых необходимо выполнить преобразование перед вычислением. В этом случае программист явно указывает компилятору, что тот должен сделать.

Пример 2.8.

Перепишем пример 2.7.

```
int a=5, b=2;
float c;
c=(float)a/2.0;   // c=5.0/2.0=2.5
b=(int)c/b;       // b=2/2=1
```

При выполнении присваивания, если типы левой и правой частей выражения не совпадают, происходит неявное преобразование. Компилятор С++ всегда пытается это сделать, и упрощенно можно считать, что преобразование происходит без потери данных от меньшего типа к большему, например от int к float или от char к int, и с потерей данных из большего типа к меньшему, например от float к int. Это легко понять, если вспомнить, что тип данного — это объем занятой им памяти.

Рекомендуется строго относиться к типам данных, не смешивать типы в выражениях, следить, чтобы тип левого операнда присваивания соответствовал типу выражения правой части.

2.3. СТРУКТУРА И КОМПОНЕНТЫ ПРОСТОЙ ПРОГРАММЫ НА ЯЗЫКЕ С++

2.3.1. Функция main()

В современных средах программирования программа — это «*проект*». В состав проекта входят один или нескольких файлов с расширением «.c» или «.cpp». Все исходные файлы компилируются и собираются совместно. Простейшие программы содержатся в одном текстовом файле.

Программа может состоять из одной или нескольких функций. Функция — это самостоятельный именованный алгоритм решения некоторой законченной задачи. Одна из функций должна иметь имя main(). С нее всегда начинается выполнение программы. Любая функция, кроме main(), может быть вызвана из другой функции. При вызове функции могут быть переданы параметры (данные). По окончании выполнения функции в вызывающую функцию может быть возвращено значение (результат), а может не быть возвращено ничего. Примером обращения к функциям являются библиотечные функции, например, sin(x), fabs(a).

Функция обладает типом, соответствующим возвращаемому значению. Если функция ничего не возвращает,

ее тип void. Если функция не имеет аргументов, вместо них в списке параметров записывается слово void.

Пример 2.9.

```
void main(void)  // не имеет типа и не имеет параметров
main()           // тип функции по умолчанию будет int,
                 // аргументов нет, значит, их может быть
                 // произвольное число
```

Пример функций, не возвращающих значения — printf(), scanf().

2.3.2. Комментарии

Комментарии предназначены для записи пояснений и примечаний к тексту программы и не влияют на выполнение программы. Они записываются на родном языке программиста.

В языке C++ существуют два вида комментариев.

1. *Многострочный* комментарий записывается в любом месте текста программы в скобках вида /* */. Переводит в разряд примечаний весь текст, заключенный между ними. Такие комментарии удобны при отладке программы, чтобы исключить фрагмент кода, или для пояснений.

2. *Однострочный* комментарий записывается в любой строке программы после сочетания символов //. Комментирует весь текст до окончания строки. Используется для пояснений к строкам.

Использование комментариев является признаком хорошего стиля программирования. Так, необходимы пояснения к смыслу объявленных объектов программного кода, к используемым алгоритмам, к структуре программного кода.

2.3.3. Структура файла программы из одной функции. Блок операторов

Код программы, которая состоит из одной функции main(), обычно содержит следующие составляющие, хотя почти все они могут отсутствовать:

```
#Директивы препроцессора     // начинаются с символа # и
                             // записываются в одну строку
Тип_функции main(параметры) // заголовок функции
```

```
{ // блок тела функции;
        определения объектов;
        исполняемые операторы;
        return выражение;        // если функция возвращает
                                                значение
}
```

В С++ объявление переменной (объекта) возможно не только в начале программы, но и в любом месте текста до первого обращения к ней. Областью действия объекта является только непосредственно охватывающий его блок. Как правило, так объявляют рабочие переменные.

Блок (операторов) — это произвольная последовательность определений и операторов, заключенная в фигурные скобки:

```
{
  // блок;
}
```

Блок используется для укрупнения структуры программы. Точка с запятой в конце блока не ставится.

Текст программы на языке С++ обладает структурой. Существует система правил корректной записи текста программ, которые просты, но улучшают зрительное восприятие и понимание кода.

1. Каждый оператор заканчивается знаком «;». Обычная ошибка начинающего — это знак «;», завершающий заголовки функций или операторов цикла. В первом случае синтаксическая ошибка распознается как отсутствие тела функции, во втором случае телом цикла является пустой оператор, что синтаксической ошибкой не является, и программа выполняется.

2. Каждый оператор записывается в одну строку. Такая запись позволяет структурировать текст программы, наглядно видеть ее алгоритм, и облегчает отладку при пошаговом исполнении.

3. Блок, т. е. произвольный фрагмент текста, заключенный в фигурные скобки {}, размещается в любом месте программы. Использование блоков позволяет укрупнить структуру алгоритма.

4. Структура программы подчеркивается отступами. Это позволяет визуально показать блоки, составляющие структуру алгоритма.

5. Необходимо использовать разумное количество комментариев.

6. Имена объектов программы выбираются осмысленно. Каждое имя подчеркивает назначение и логику объекта, например, имена библиотечных функций sin(), abs(), printf() и прочих говорят сами за себя. Имена объектов, введенные программистом, подчеркивают их абстрактный смысл, например, Count, Square, Point.x, Point.y и т. д.

7. Пробелы в тексте являются значащими только в составе текстовых констант. В коде программы пробелы отделяют друг от друга элементы текста. В остальных случаях их использование произвольно, например, лишние пробелы улучшают читабельность программы.

2.3.4. Директивы препроцессорной обработки

Директивы начинаются со знака # и записываются в одной строке. Являются командами (директивами), выполняемыми препроцессором на стадии предварительной обработки текста программы, т. е. до ее компиляции. Директив препроцессора достаточно много, на начальном этапе достаточно ознакомиться с двумя из них.

Директива #define

Используется для задания именованных констант и для задания строк подстановки.

Синтаксис.

#define Имя Выражение

Механизм действия директивы — макроподстановки, т. е. препроцессор сканирует весь текст программы и выполняет замены в тексте программы, везде вместо «Имени» подставляя «Выражение».

Замечание. Имена define определенных констант записываются большими буквами, чтобы визуально отличить их от имен переменных и других объектов.

Пример 2.10.

```
#define N 10              // по всему тексту вместо N число10
#define PI 3.1416926      // вместо PI его числовое значение
#define STR "Строковая константа, подставляется в текст\n"
```

Директива #define может не только определить именованную константу, но и выполнить макроподстановки.

Пример 2.11.

```
#define N1 N+1            // вместо N1 текст N+1
#define int long          // в тексте все описания int
                          // заменятся на long
```

Если в замещаемом имени есть скобки, следующие за именем без пробела, то это макроопределение с параметрами:

```
#define Имя(список_параметров) Выражение
```

Например,

```
#define Cube(x) x*x*x
```

Имя здесь играет роль имени макроопределения, а параметров может быть несколько, тогда они отделяются запятыми. Между именем и списком параметров не должно быть пробела. Такое макроопределение может использоваться как функция, хотя макроподстановки не заменяют функции, и иногда могут привести к ошибкам.

Пример 2.12.

```
#define Cube(x)  x*x*x
// макроопределение возведения в степень
#include <stdio.h>
void main(void)
{
  int a=2;
  printf("%d %d", a, Cube(a));    // выведено 2 8
}
```

Директива #include

Используется для замены в тексте путем добавления текста из других файлов в точку нахождения #include.

Синтаксис.

```
#include “filename”
#include <filename>
```

Механизм действия — включение текста указанного файла в текущее место в программе. Включаемые файлы называются заголовочными и содержат информацию, которая для программы глобальна.

Директива #include "filename" осуществляет поиск файла сначала в текущем каталоге, а затем в системных каталогах. Так подключаются личные файлы программиста, содержащие произвольные тексты, например, определения констант, объявления или описания функций.

Директива #include <filename> осуществляет поиск файла только в системных каталогах. Так подключаются стандартные заголовочные файлы, поставляемые в комплекте со стандартными библиотеками функций.

Каждая библиотечная функция имеет свое описание (прототип). Кроме того, в заголовочных файлах описаны многие константы, определения типов и макроподстановки. Например, стандартный заголовочный файл <stdio.h> содержит описания библиотечных функций ввода и вывода, <math.h> подключается при использовании математических функций. Их описание содержится в справочной системе. Следует понимать, что использование #include не подключает к программе соответствующую библиотеку, а только включает в текст программы на глобальном уровне все нужные описания и объявления. Сами библиотечные функции подключаются к программе в виде объектного кода на этапе компоновки, когда компоновщик обрабатывает все вызовы функций, определяет, какие потребуются программе, и собирает исполнимый файл, включая в него объектные (компилированные) коды тех функций, к которым выполняется обращение.

2.3.5. Ввод и вывод данных. Начальные сведения

В этом подпараграфе описаны инструменты ввода-вывода данных в классическом языке C с использованием библиотеки stdio.h (standard input output library).

Ввести данное — означает присвоить произвольное значение переменной во время выполнения программы.

Вывести данное — означает напечатать на экране значение переменной при выполнении программы.

Простейший из способов обмена данных — это форматированный ввод-вывод с определением правил размещения данных во входном/выходном потоке. Для реализации такого обмена к программе директивой #include <stdio.h> подключается библиотека stdio.h.

Для ввода значения данного с клавиатуры с эхо повтором на экране используется функция scanf(), синтаксис которой:

scanf("форматная строка", список_ввода);

Здесь «список_ввода» — имена переменных, значения которых будут введены с клавиатуры при выполнении функции scanf(). Имена переменных предваряются символом «&», который является признаком адресной операции и означает, что введенное значение пересылается по адресу, определенному именем переменной. При вводе данные отделяются пробелами или нажатием клавиши [Enter].

Для вывода значения данного на экран используется функция printf(), синтаксис которой:

printf("форматная строка", список_вывода);

Здесь «список_вывода» — список имен переменных и выражений (в том числе констант), значения которых появятся на экране при выполнении функции printf().

Форматная (управляющая) строка — это строка символов внутри двойных кавычек, содержащая управляющие символы и текст. При вводе данных функция scanf() читает посимвольно текст из входного потока, распознает лексемы и преобразует их в машинное представление в соответствии с признаком формата, сопоставленного переменной, ожидающей данное. При выводе функция printf() берет машинное представление значения переменной, соответственно признаку формата преобразует в текстовое представление и выводит на экран.

Число управляющих символов равно числу объектов в списке ввода-вывода. Управляющий символ имеет признак % и может иметь одно из следующих значений:

%d — ввод-вывод целого десятичного числа (int),
%u — ввод-вывод целого без знака (unsigned),
%f — ввод-вывод числа с плавающей точкой (float),

%lf — ввод-вывод числа с плавающей точкой (double),
%e — ввод-вывод числа в экспоненциальной форме (double и float),
%c — ввод-вывод символа (char),
%l — ввод-вывод длинного значения (long),
и др.

При вводе и выводе необходимо строгое соответствие типа данного управляющему символу формата.

Пример 2.13.

Приведем пример форматированного ввода и вывода:

```
#include <stdio.h>
void main(void)
{
  int my_int;
  float my_float;
  printf("\nВведите целое и дробное число\n");
  scanf("%d", &my_int);
  scanf("%f", &my_float);
  printf("%d %f", my_int, my_float);
}
```

При запуске программы функция printf() выведет на экран строку — приглашение к вводу данных, затем при выполнении каждой функции scanf() будет ожидать ввода данных. Пользователь должен ввести требуемое количество данных, отделяя их пробелами или нажатием клавиши [Enter]. При завершении ввода данные тут же будут выведены на экран самым примитивным образом. Так, если ввести целое 5 и дробное 9.9, то строка вывода будет иметь вид:

```
59.900000
```

Поскольку при вводе данного функция scanf() находится в состоянии ожидания ввода, рекомендуется каждый ввод предварять строкой, выводящей на экран приглашение к вводу данных, в котором пользователю подробно объясняют, что и как он должен сделать, чтобы правильно ввести данные, например:

```
printf("Введите координаты центра и радиус круга\n");
scanf("%f%f%f", &x, &y, &R);
```

При выводе данных для улучшения вывода рекомендуется использовать некоторые приемы.

1. Управляющие символы, например:

"\n" для перевода строки при выводе;

"\t" для выполнения табуляции.

2. Произвольный текст в форматной строке для приглашения на ввод данного и для пояснений при выводе, например, функция вывода может быть записана так:

printf("Целое = %d, Вещественное = %f\n", my_int, my_float);

Пробелы в строке текста являются значащими. Теперь, если ввести целое 5 и дробное 9.9, то строка вывода будет иметь вид:

Целое = 5, Вещественное = 9.900000

3. Модификаторы форматов. Они используются для оформления вывода. По умолчанию (без модификаторов) данные выводятся в поле минимальной ширины с точностью 6 знаков после запятой, число прижимается к правому краю поля. Этим выводом можно управлять:

- можно задать ширину поля как строку цифр, определяющую наименьший размер поля вывода, которое будет игнорировано, если число не входит в поле;
- можно задать точность вывода для вещественных чисел: это две цифры, определяющие общий размер поля вывода и число знаков после запятой.

В следующих примерах обозначим знаком □ пробелы, которые будут значимыми в строке вывода.

Пример 2.14.

printf("Целое=%4d, Вещественное=%5.2f\n", my_int, my_float);

Если ввести значения 10 и 2.3, то строка вывода будет иметь вид:

Целое = □□10, Вещественное = □9.90

Если ввести значения 19951 и 12.9999, то строка вывода будет иметь вид:

Целое = 19951, Вещественное = 13.00

Можно сделать вывод, что число округляется.

4. Знак минус используется для выравнивания числа влево внутри поля вывода:

printf("Целое=%–4d, Вещественное=%–5.2f\n", my_int, my_float);

Если ввести значения 2 и 2.36666, то строка вывода будет иметь вид:

Целое = 2□□□, Вещественное = 2.37□

Если ввести значения –1999 и 12.9999, то строка вывода будет иметь вид:

Целое = –1999, Вещественное = 13.00

Пример 2.15. Приведем пример использования форматированного ввода-вывода.

В комментариях к строкам приведен вид выводимого данного и пояснения.

```
#include <stdio.h>
#define STR "Программа"   // для иллюстрации вывода строк
void main (void)
{
// вывод целого числа 336
   printf("%d\n",    336);   // 336
   printf("%2d\n",   336);   // 336 формат 2d игнорируется
   printf("%8d\n",   336);   // □□□□□336 ширина поля 8
   printf("%-8d\n",  336);   // 336□□□□□ прижато влево
   printf("\n");               // пропуск строки при выводе

// вывод вещественного числа 12.345
   printf("%f\n",          12.345);  // 12.345000
   printf("%e\n",          12.345);  // 1.234500e+01
   printf("%10.1f\n",      12.345);  // □□□□□□12.3
   printf("%–12.1f\n",     12.345);  // 12.3□□□□□□
   printf("\n");

// вывод строки символов по формату s
   printf("%s\n",          STR);     // Программа
   printf("%12s\n",        STR);     // □□□Программа
   printf("%12.5s\n",      STR);     // □□□□□□□Прогр
   printf("%-12.5s\n",     STR);     // Прогр□□□□□□□
   printf("\n");
}
```

2.4. УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ ЯЗЫКА С++

Оператор — это предложение, описывающее одно действие по обработке данных или действия программы на очередном шаге ее исполнения.

2.4.1. Классификация операторов

Все операторы языка С++ можно разделить на две группы.

1. Операторы преобразования данных.
2. Операторы управления ходом выполнения программы.

Это разделение соотносится с определением алгоритма.

Операторами *преобразования* данных являются выражения. Оператора присваивания в языке С++ нет, его заменяет выражение, в составе которого есть операция присваивания.

Примеры операторов изменения данных:

```
y+4;          // выражение
x=y+4;        // выражение присваивания
x++;          // выражение – оператор
```

Операция присваивания правоассоциативна, поэтому допускается запись цепочек присваиваний, например:

```
x=y=z=1;      // каждая переменная будет равна 1
```

К операторам *преобразования* данных можно отнести оператор обращения к функции (вызов функции). Его операцией являются круглые скобки, а операндами — имя функции и список параметров, например:

```
scanf("%d%f", &a, &b);
printf("a=%d, b=%f", &a, &b);
```

При выполнении вызова функции scanf() значения переменных a, b будут изменены. При выполнении вызова функции printf() выполняются действия по преобразованию данных из внутреннего представления в текстовое, пригодное для вывода на экран.

Операторы *управления* предназначены для управления ходом выполнения программы. Обычно операторы программы выполняются в том порядке, в котором записаны. Поскольку каждый оператор выполняет одно действие по обработке данных, тем самым управляя работой компьютера, то порядок выполнения выражений называют потоком управления. Поток управления редко бывает линейным. Изменить направление потока управления позволяют операторы управления, перечисленные в таблице 2.8.

Таблица 2.8

Операторы управления C++

Название оператора	Ключевое слово
Составной оператор	{…}
Условный оператор	if
Оператор цикла с проверкой условия до выполнения	while
Оператор цикла с проверкой условия после выполнения	do… while
Оператор цикла типа «прогрессия»	for
Оператор прерывания	break
Оператор продолжения	continue
Оператор переключатель	switch
Оператор перехода	goto

При описании каждого оператора языка C++ будем придерживаться следующей схемы:

- назначение;
- синтаксис;
- семантика, механизм исполнения;
- пример;
- особенности.

2.4.2. Оператор присваивания

Назначение. Изменение значения данного в соответствии с выражением правой части.

Синтаксис.

```
Имя_переменной = Выражение;                    // форма 1
Имя_переменной = (тип) Выражение;              // форма 2
```

Выполнение. В первую очередь вычисляется выражение правой части, во вторую вычисленное значение присваивается переменной левой части.

Примеры:

```
x=y=0;
x++;
y=sin(2*PI*x);
y+=0.2;
```

Особенности. Если в выражении встречаются операнды разных типов, то при вычислении значения выражения происходит неявное *преобразование* типов, как правило, к большему типу. При несоответствии типа вы-

ражения правой части типу переменной в левой части происходит *приведение* типа выражения к типу переменной, при этом неизбежна потеря данных при приведении от большего типа к меньшему.

Приведем пример реализации линейного алгоритма вычисления высот треугольника, блок-схема которого приведена в примере 1.1 (глава 1).

Пример 2.16.

```
// вычислить высоты треугольника со сторонами a, b, c
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main(void)
{
  double a,b,c;
  printf("Введите стороны треугольника");
  scanf("%lf%lf%lf", &a, &b, &c);
  double p,t;
  // объявить переменные можно там, где они
  // понадобились
  double Ha,Hb,Hc;
  p=0.5*(a+b+c);
  // нет проверки условия существования треугольника
  t=2*sqrt(p*(p-a)*(p-b)*(p-c));
  Ha=t/a;
  Hb=t/b;
  Hc=t/c;
  printf("Значения высот треугольника:\n
        Ha=%6.2lf,Hb=%6.2lf, Hc=%6.2lf\n",Ha, Hb, Hc);
}
```

2.4.3. Составной оператор

К составным операторам относятся собственно составные операторы и блоки. В обоих случаях это последовательность операторов, заключенная в фигурные скобки. Блок отличается тем, что в его состав входят описания каких-либо объектов программы.

Блоком является тело любой функции, в том числе функция main().

Назначение. Составной оператор используется для объединения нескольких операторов в один. Составной оператор, например, формирует ветвь условного оператора или тело цикла в операторах цикла.

Пример 2.17.

{// это составной оператор
n++;
S+=n;
}
	{// это блок
	int n=0;
	n++;
	S+=n;
	}

2.4.4. Условный оператор if

В п. 1.2.2 (глава 1) приведены блок-схемы структуры *если — то — иначе* с полным и неполным ветвлением. В языке C++ эта структура реализуется условным оператором *if*.

Назначение. Выбор одного из двух возможных путей исполнения программы в зависимости от условий, сложившихся при выполнении.

Для проверки условия формулируется некоторое утверждение, которое может быть истинным или ложным. Это логическое выражение, которое может принять одно из значений *True* либо *False*. В языке C++ они имеют форму выражений целого типа, принимая значения «не 0», что соответствует истине, либо «0», что соответствует лжи, либо типа bool.

Синтаксис. Синтаксис условного оператора имеет две формы.

Сокращенная форма.

if (Логическое_выражение)
Оператор;

Здесь «Логическое_выражение» — любое выражение типа int или bool.

«Оператор» — это один или несколько операторов, в общем случае составной оператор. Логическое выражение записывается в скобках.

Полная форма.

if(Логическое_выражение)

```
  Оператор1;
else
  Оператор2;
```

Здесь «Логическое_выражение» имеет тот же смысл, что и ранее, а «Оператор1" или «Оператор2" — это, в общем случае, блок.

Выполнение.

1. Вычисляется значение логического выражения.

2. Если оно не равно 0, выполняется «Оператор1", а если оно равно 0, выполняется «Оператор2" (или ничего в первой форме).

Пример 2.18. Первая форма условного оператора.

Стоимость товара равна произведению цены на количество. Если есть скидка, то стоимость уменьшается на величину discount. Вычислить стоимость.

```
pay=cost*count;          // общая формула
if(discount!=0)          // если есть скидка, то стоимость
                         // уменьшается
pay=pay–(pay*discount)/100;
// вывод pay в любом случае
printf("Стоимость = %6.2f\n", pay);
```

Пример 2.19. Вторая форма условного оператора.

Оплата труда работника — это произведение количества отработанных часов на стоимость часа. Если отработано более 40 ч, то за каждый час работодатель платит в полтора раза больше.

```
if(hour<40)
  pay=rate*hour;                    // обычная оплата,
                                    // hour < 40
else
  pay=rate*40+(hour–40)*rate*1.5; // повышенная оплата
                                    // hour >= 40
// печать одинакова
printf("К оплате %6.2f рублей.\n", pay);
```

Пример 2.20. Использование блоков в составе условного оператора.

Если необходимо вывести на экран число оплаченных часов и значение суммы к оплате, тогда для каждой ветви нужна своя собственная печать.

```
if(hour<40)
{
   pay=rate*hour;                          // обычная оплата
   printf("Оплачено %d часов, к оплате %6.2f руб.\n", hour, pay);
}
else
{
   pay=rate*40+(hour–40)*rate*1.5;   // повышенная оплата
   printf("Оплачено %d часов, %d сверхурочно.\n",
   hour, hour–40);
   printf("К оплате %6.2f руб.\n", pay);
}
```

В состав операторов, формирующих ветви условного оператора, могут входить любые операторы, в том числе условные. В этом случае говорят о вложенном условном операторе. Каждая из ветвей, в свою очередь, может содержать условный оператор или несколько. Число уровней вложения не ограничено, однако чем их больше, тем сложнее восприятие текста.

Пример 2.21. Вычислить значение функции

$$y = \begin{cases} x + a, & \text{при } x < 10; \\ x + b, & \text{при } 10 \le x \le 20; \\ x + c, & \text{при } x > 20. \end{cases}$$

Блок-схема алгоритма решения задачи приведена в примере 1.2 (глава 1).

```
void main(void)
{
   float a,b,c;
   float x,y;
   printf("Введите параметры функции");
   scanf("%f%f%f",&a,&b,&c);
   printf("Введите аргумент функции");
   scanf("%f",&x);
   if(x<10)
      y=x+a;
   else
      if(x<20)
```

```
        y=x+b;
      else
        y=x+c;
   printf("x=%6.2f,y=%6.2f\n",x,y);
   getch();
}
```

Логическое выражение в условном операторе может быть как угодно сложным. Для записи сложных логических выражений используются специальные логические операции:

&& — логическое «и»;

|| — логическое «или»;

! — логическое отрицание «не».

Например, для обозначения категории «подростковый возраст» нужна условная градация, скажем, от 12 до 17 лет. Если переменная Age обозначает возраст некоего круга людей, то подростками из них будут только те, для которых справедливо утверждение Age>=12 && Age<=17. Логическое выражение будет истинно, только когда оба условия выполняются одновременно. Соответственно, если Age<12||Age>17, то человек не подросток (это ребенок, если Age<12, или взрослый человек, если Age>17). Это логическое выражение будет истинно, когда хотя бы одно или оба условия выполнены. Для того чтобы проиллюстрировать применение операции отрицания, запишем условие «не подросток» инверсией выражения «подростковый возраст» !(Age>=12 && Age<=17).

Приоритеты операций приведены ранее в таблице 2.7.

Наиболее распространены следующие ошибки записи логических выражений.

1. Знаки логических операций похожи на знаки поразрядных операций сравнения & (и) и | (или). Выполнение этих операций происходит различным образом, поэтому не всегда результат поразрядного сложения равен истинной дизъюнкции операндов, а результат поразрядного умножения равен истинной конъюнкции (хотя иногда равен).

2. Знак операции = отличен от знака ==. Знаки операций = и == похожи внешним образом, но радикально отличаются по механизму выполнения.

Каждый программист хотя бы раз допустил ошибку вида:

```
if(Key=27)          // ошибка здесь
{
  printf("Завершение работы.\n");
  return;
}
else
  printf("Продолжение работы.\n");
```

Здесь по смыслу алгоритма значение переменной Key определяет момент завершения или продолжения работы программы. Однако в этом тексте, независимо от первоначального значения Key, будет выполнено присваивание Key=27, и значение выражения станет равно 1. Значит, результат проверки значения выражения равен 1, всегда выполняется первая ветка, и программа благополучно всегда завершит свою работу.

3. Пропуск знаков логических операций. Математическая запись $3 < x < 6$ должна быть записана на языке C++ обязательно с использованием логической операции &&: if(–3<x && x<6), хотя компилятор согласится с текстом вида if(–3 < x < 6).

При вычислении выражения –3 < x < 6 сначала вычисляется (–3 < x). Это выражение имеет логическое значение (0 или 1). Затем полученное логическое значение участвует в вычислении выражения (–3 < x) < 6, результат которого всегда истинен.

4. Операции сравнения для вещественных типов. Операции сравнения могут применяться к данным любого из базовых типов C++. Для данных целого и символьного типов точное соответствие возможно. Для данных вещественных типов всегда существует ошибка представления данных, которая может накапливаться. Поэтому проверки на точное равенство вещественных чисел следует избегать.

Пример 2.22.

```
#include <stdio.h>
void main(void)
{
  float x=1./3.;
```

```
    float y;
    y=x+x+x;
    if(y==3*x)
       printf("Равны\n");
    else
       printf("Не равны\n");
}
```

Казалось бы, y должен быть равен 3*x, но в силу ошибки представления данных вещественного типа этого не произойдет.

Проверка вещественных чисел выполняется на приближенное равенство с использованием приближенного значения модуля разности сравниваемых значений:

```
(fabs(y–3*x)<0.001);
// вычисление, например с точностью 0.001
```

Примеры записи логических выражений приведены в таблице 2.9.

Таблица 2.9

Примеры записи логических выражений

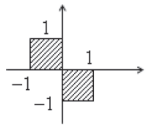
№ п/п	Условие	Математическая формулировка	Синтаксис записи на С++
1	Число X принадлежит отрезку [–2, +2]	$-2 <= x <= +2$ Или $\|x\| <= 2$	–2.<=x && x<=2. или fabs(x)<=2.
2	Число X принадлежит отрезку [–2, –4] или отрезку [+2, +4]	$-4 <= x <= -2$; $+2 <= x <= +4$	–4.<=x && x<=2. || 2.<=x && x<=4.
3	Точка с координатами (x, y) находится в первой четверти	$\begin{cases} x > 0 \\ y > 0 \end{cases}$	x>0 && y>0
4	Точка с координатами (x, y) находится в первой или в третьей четверти	$\begin{cases} x > 0 \\ y > 0 \end{cases} \cdot \begin{cases} x < 0 \\ y < 0 \end{cases}$	x>0&&y>0||x<0&&y<0
5	Числа a, b, c одновременно четны	Каждое число делится на 2 без остатка	a%2==0&&b%2==0&& c%2==0
6	Хотя бы одно из чисел a, b, c четно	Одно или два, или все три числа делятся на 2 без остатка	a%2==0||b%2==0 ||c%2==0

7	Символ является знаком препинания		char d; // объявление
			d=='.'||d==','
			||d=='?'||d=='!'

Продолжение табл. 2.9

№ п/п	Условие	Математическая формулировка	Синтаксис записи на C++

8	Символ является буквой латинского алфавита		char d; // объявление
			d>='A'&&d<='Z'||
			d>='a'&&d<='z'

9	Точка с координатами (x, y) принадлежит указанной области  	Во второй четверти $\begin{cases} x > -1 \\ y < 1 \end{cases}$ В четвертой четверти $\begin{cases} x < 1 \\ y > -1 \end{cases}$	x>−1&&x<0&&y>0 &&y<1||x>0&& x<1&&y<0&&y>−1
10	Точка с координатами (x, y) находится выше прямой $y = ax + b$	При подстановке в уравнение прямой $y_1 = a\,x + b$: если $y_1 > y$, то выше, иначе ниже.	// Значения a, b, x, y известны. a*x+b>y

2.4.5. Оператор цикла while

В п. 1.2.3 (глава 1) приведены блок-схемы циклических алгоритмов. Рассмотрим реализацию циклов в языке C++.

Для реализации цикла с предусловием используется оператор цикла while.

Назначение. Организация многократного повторения фрагмента программы.

Синтаксис.

```
while (Логическое_выражение)
{   // Тело цикла;
}
```

Здесь «Тело цикла» — один или несколько операторов, в общем случае составной оператор. «Логическое_выражение» — это выражение условия завершения цикла.

Выполнение. Тело цикла выполняется, пока «Логическое_выражение» имеет значение «истинно» (отлично от 0). Когда «Логическое_выражение» ложно (равно 0), управление передается оператору, стоящему за циклом.

Пример 2.23. Найти сумму n чисел натурального ряда $S = \sum_{i=1}^{n} i$. Параметр цикла $i \in [1, n]$, шаг изменения параметра $\Delta i = 1$. Тело цикла $S = S + i$.

```
void main(void)
{
  int n, Sum=0;
  int i;
  printf("\nВведите количество элементов ряда\n");
  scanf("%d", &n);
  i=1;
  while(i<=n)
  {
    Sum+=i;      // очередное сложение
    i++;         // увеличение слагаемого
  }
  printf("Сумма чисел натурального ряда=%d", Sum);
}
```

Поскольку значение суммы не зависит от порядка суммирования, то в стиле языка С++ алгоритм запишется так:

```
i=n;
Sum=0;
while(i)          // пока i отлично от 0
  Sum+=i--;
```

Особенности. Проверка условия происходит до входа в цикл, поэтому для заведомо ложного выражения тело цикла не будет выполнено ни разу.

2.4.6. Оператор цикла do... while

Оператор цикла do ... while используется для реализации цикла с предусловием, блок-схема которого приведена в п. 1.2.3 (глава 1).

Назначение. Организация многократного повторения фрагмента программы.

Синтаксис.

```
do
{
// Тело цикла;
}
while(Логическое_выражение);
```

Здесь «Тело цикла» — один или несколько операторов, в общем случае составной оператор. «Логическое_выражение» — это выражение условия завершения цикла.

Выполнение. Тело цикла (в общем случае блок) выполняется многократно, пока «Логическое_выражение» не равно 0. Как только выражение станет ложно (т. е. равно 0), цикл заканчивается, управление передается следующему по порядку оператору.

Пример 2.24. Решим задачу примера 2.23 с использованием оператора цикла do.

```
void main(void)
{
  int n, Sum=0;
  int i;
  printf("\nВведите количество элементов ряда \n");
  scanf("%d", &n);
  i=n;
  do
    Sum+=i--;
  while(i);                          // пока i отлично от 0
  printf("Сумма чисел натурального ряда=%d", Sum);
}
```

Особенности. Проверка условия происходит после выполнения тела цикла. Значит, как бы ни было задано «Логическое_выражение», оператор тела цикла выполнится не менее чем один раз.

2.4.7. Оператор цикла типа прогрессия for

Назначение. Организация многократного повторения произвольного фрагмента кода программы. Как правило, цикл используется, когда число повторений тела цикла известно заранее и явно есть переменная величина, которая, изменяясь, составляет прогрессию. Она является параметром цикла. Иногда ее называют счетчик цикла.

Синтаксис.

```
for(Выражение1; Выражение2; Выражение3)
  // Тело цикла;
```

Здесь параметр цикла составляет прогрессию и явно присутствует в записи заголовка цикла (так называется строка for(...)).

«Выражение1" задает начальное значение параметра цикла, «Выражение2" задает условие завершения выпол-

нения тела цикла и является логическим, «Выражение3" задает приращение параметра цикла. «Тело цикла» — это произвольная последовательность операторов, как правило, составной оператор. Первое и третье выражения могут состоять из нескольких операторов, отделенных друг от друга запятой.

Блок-схема, реализующая оператор этого типа, приведена в п. 1.2.3 (глава 1).

Выполнение. Перед входом в цикл, т. е. до первого выполнения тела цикла, однократно выполняется «Выражение1". Тело цикла выполняется многократно, пока «Выражение2" (условие) не равно 0. Как только его значение становится равным 0, управление передается следующему по порядку оператору. Проверка условия завершения цикла происходит до выполнения тела цикла.

Пример 2.25. Решим задачу примера 2.23 с использованием цикла for.

```
void main(void)
{
  int n, Sum=0;
  int i;
  printf("\nВведите количество элементов ряда \n");
  scanf("%d",&n);
  for(i=1; i<=n; i++)
    Sum+=i;
  printf("Сумма членов натурального ряда=%d", Sum);
}
```

Особенности.

1. Управляющая переменная цикла for не обязательно целого типа, она может быть вещественной или символьной.

Приведем пример цикла, вычисляющего сумму геометрической прогрессии.

Пример 2.26. Вычислить сумму геометрической прогрессии по формуле $S = \sum_{i=1}^{2} 1{,}1 \cdot i$, где i изменяется по закону $i = i \cdot 1{,}1$.

```
void main(void)
```

```
{
  float Sum=1, i;
  for(i=1.; i<=2.; i*=1.1)
    Sum+=i;
  printf("\nСумма геометрической прогрессии %f", Sum);
}
```

2. Любое из трех выражений, стоящих в заголовке цикла, любые два или все могут отсутствовать, но разделяющие их символы «;» опускать нельзя. Если «Выражение2" отсутствует, то считается, что оно истинно, и цикл превращается в бесконечный, для выхода из которого необходимы специальные средства. Бессмысленно использовать несколько «Выражений2", так как управление циклом выполняется по первому условию.

3. Оператор цикла for удобен тем, что интегрирует в заголовке описание всего процесса управления. Телом цикла является, в общем случае, блок. Поскольку начальных присваиваний и выражений приращения может быть несколько, весь цикл вместе с телом может быть записан одной строкой. Например, тот же цикл в стиле языка C++ может выглядеть так:

```
for(Sum=0,i=1;i<=n; Sum+=i++);
```

Здесь два начальных присваивания, а тело цикла Sum+=i++ вместе с приращением параметра цикла записано в его заголовке.

Приведем код программы вычисления факториала функции с применением циклов while и for. Блок-схема алгоритма приведена в примере 1.3 (глава 1).

Пример 2.27. Дано целое положительное число n. Вычислить факториал этого числа.

```
// первый вариант
void main(void)
{
int N;
long int F;
int R;
printf("Введите число\n");
scanf("%d",&N);
F=R=1; // присваивание начальных значений
```

```
while (R<=N)
   {
    F=F*R;
    R++;
   }
printf("Факториал %d равен %ld\n",N,F);
}

// второй вариант
void main(void)
{
   int N;
   long int F;
   int R;
   printf("Введи число\n");
   scanf("%d",&N);
   F=1;                          // начальное значение факториала
   for(R=1; R<=N; R++)   // здесь R – параметр цикла
      F=F*R;
   printf("Факториал %d равен %ld\n",N,F);
}
```

При использовании цикла while программист должен явно указать начальное значение параметра цикла (в программе это переменная R), а в теле цикла изменить его значение. При использовании цикла for эти действия выполняются в заголовке цикла.

Пример 2.28. Пусть $a_0 = 1$; $a_k = k \cdot a_{k-1} + 1/k$, $k = 1, 2, \ldots$ Дано натуральное число n. Получить a_n.

В задаче требуется определить член последовательности. Особенности этой задачи и блок-схема ее решения приведены в примере 1.4 (глава 1).

```
void main(void)
{
// вычисление элемента последовательности
   int n;
   int k;
   double a=1.;              // инициализация данного
   printf("Введите число элементов последовательности.\n");
   scanf("%d",&n);
```

```
  for(k=1; k<=n; k++)
     a=k*a+1./(double)k; // вещественное 1., целое k
  printf("%d-элемент последовательности равен
%8.4lf",n,a);
}
```

Пример 2.29. Даны натуральное число n, действительное число x. Вычислить сумму:

$$\sum_{i=1}^{n} \frac{x^i}{i!}.$$

Прежде чем написать программный код для решения этой задачи, необходимо вывести рекуррентное соотношение, которое определит способ вычисления слагаемого на каждом шаге выполнения цикла. Способ вывода этой зависимости и блок-схема алгоритма приведены в примере 1.5 (глава 1).

```
void main(void)
{
  int i,n;
  double c,x,S;
  printf("Введите число слагаемых.\n");
  scanf("%d",&n);
  printf("Введите x\n");
  scanf("%lf",&x);
  S=0;
  c=1;
  for(i=1; i<=n; i++)
     {
      c=c*x/i;              // неявное приведение типов
      S+=c;
     }
  printf("Сумма равна %8.3lf", S);
}
```

Пример 2.30. Даны действительные числа x, ε ($x \neq 0$, $\varepsilon > 0$). Вычислить сумму с точностью ε:

$$\sum_{k=0}^{\infty} \frac{x^k}{2^k \cdot k!}$$

Блок-схема алгоритма решения задачи приведена в примере 1.6 (глава 1). Там же показан вывод рекуррентного соотношения, определяющего вычисление слагаемого на каждом шаге выполнения цикла.

```
void main(void)
{
   double x,eps;
   double c,S;
   double F;        // или long double, значение может быть
                    // большим.
   int k;
   printf("Введите точность вычислений.\n");
   scanf("%lf",&eps);
   printf("Введите x\n");
   scanf("%lf",&x);
   F=1;
   S=0;
   c=1;
   k=1;
   while(fabs(c/F)>eps) // итерационный цикл
    {
        S+=c/F;
        k+=1;
        c=c*x*0.5;
        F*=k;
   }
   printf("Сумма равна %lf\n",S);
}
```

2.4.8. Правила организации циклических алгоритмов

Циклические алгоритмы разделяются на две группы.

1. *Арифметический цикл* (управляемый счетчиком). Цикл повторяется заранее известное число раз. Например, спортсмен должен пробежать 10 кругов или 40 км.

2. *Итерационный цикл* (управляемый событием). Обычно число повторений цикла заранее неизвестно. Например, спортсмен должен бежать, пока не устанет, или пока суммарный путь пробега не составит 42 км.

Программа должна организовать правильное управление процессом выполнения циклического алгоритма. Для управления в программе используется некая величина, которая называется «параметр цикла» (управляющая переменная). Это одна из переменных программы, которая изменяется в теле цикла, определяет число повторений цикла и позволяет завершить его работу.

При организации цикла всегда должны быть пройдены следующие этапы.

1. *Подготовка цикла*. Подготовка включает действия, которые не относятся непосредственно к логической схеме цикла, но позволяют правильно его выполнить. Обычно на этом этапе выполняется присваивание начальных значений переменным, в том числе параметру цикла.

2. *Точка входа в цикл* определяет момент передачи управления первому оператору тела цикла.

3. *Итерация*. Итерацией называется очередное выполнение тела цикла, т. е. фрагмента, который должен быть повторен многократно. Как правило, включает в себя изменение параметра цикла.

4. *Точка проверки условия*. Она определяет момент проверки условия, при котором решается, делать ли новую итерацию или перейти к оператору, стоящему за циклом. В проверке условия явно или нет присутствует параметр цикла.

5. *Выход из цикла* определяет передачу управления оператору, стоящему за циклом.

Не всегда эти составляющие присутствуют явным образом.

В п. 2.4.5–2.4.7 (глава 2) описаны три вида операторов цикла языка C++, которые служат инструментом для организации циклических процессов.

С помощью каждого из этих операторов можно организовать циклический алгоритм любого типа. В примере 2.27 (глава 2) показана реализация вычисления факториала числа с использованием циклов while и for.

При проектировании цикла необходимо решить две задачи.

1. *Разработать поток управления*. Здесь наиболее важный шаг — это выбор параметра цикла. Для параметра цикла должно быть известно:

- условие завершения цикла;
- начальное значение параметра;
- способ обновления параметра цикла.

2. *Спланировать действия внутри цикла*. Здесь важно решить, что представляет собой отдельная итерация, и точно определить:

- способ инициализации повторяющегося процесса;
- действия, которые входят в него;
- способ его обновления.

Пример 2.31. Приведем пример проектирования циклического алгоритма, в котором использованы различные операторы цикла для решения одной и той же задачи.

Условие задачи: снаряд выпущен под углом λ к горизонту со скоростью V (рис. 2.3). Требуется определить высоту и дальность полета в течение промежутка времени от t =1 с до t = 10 с с интервалом, равным 1 с.

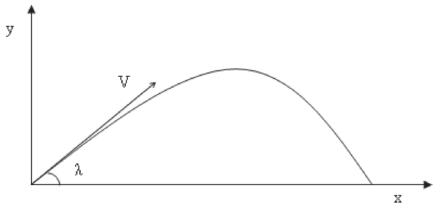


Рис. 2.3
Траектория полета снаряда

Циклом управляет переменная t, для которой известен закон изменения. Содержанием тела цикла является вычисление очередного значения высоты и дальности полета и вывод этих значений на печать. Число повторений заранее известно и равно 10, значит, можно использовать цикл арифметического типа. Арифметический цикл можно организовать с применением любого из операторов цикла языка С++, что и показано в полном тексте программы.

```
#include <stdio.h>
#include <math.h>
#define G 9.8                    // константа тяготения
void main(void)
{
  // переменные, которые являются начальными значениями
```

```
   float V;                          // скорость
   int A_Grad;                       // угол в градусах
   // переменные, участвующие в вычислениях
   float Alpha;                      // угол в радианах
   float t;                          // время полета
   float Sy, Sx;                     // высота и дальность
   printf("Введите скорость и угол в градусах\n");
   scanf("%f%d",&V,&A_Grad);
   Alpha=(float)A_Grad*M_PI/360.;
   // арифметический цикл. Управляющая переменная
t=[1.0; 10.0], Δt=1.
   // 1. Оператор for
   printf("---------------------------------------\n");
   printf("--Время--Высота--Дальность\n");
   // подготовка цикла
   printf("---------------------------------------\n");
   for(t=1.;t<=10.;t+=1.)
   {
      Sx=V*t*cos(Alpha);
      Sy=V*t*sin(Alpha)–0.5*G*t*t;
      printf("%6.2f %6.2f %6.2f\n",t,Sy,Sx);
   }

// 2. Оператор while
printf("---------------------------------------\n");
printf("Время Высота Дальность\n");
printf("---------------------------------------\n");
// подготовка цикла. Начальное значение t=1
t=1.;
while(t<=10.)
{
   Sx=V*t*cos(Alpha);
   Sy=V*t*sin(Alpha)–0.5*G*t*t;
   Printf("%6.2f %6.2f %6.2f\n",t,Sy,Sx);
   t+=1.;
}

// 3. Оператор do
printf("-------------------------------\n");
```

```
printf("--Время--Высота--Дальность\n"); // подготовка цикла
printf("-----------------------------\n");
t=1.;
do
{
  Sx=V*t*cos(Alpha);
  Sy=V*t*sin(Alpha)–0.5*G*t*t;
  Printf("%6.2f %6.2f %6.2f\n",t,Sy,Sx);
  t+=1.;
}
while(t<=10.);
}
```

В этом примере код программы приведен в трех вариантах, чтобы показать отличия в организации циклических алгоритмов при использовании разных операторов цикла. Решение задачи выявляет одну ее особенность. Приведем одну из распечаток решения, полученную при скорости 100 м/с и угле, равном 30°.

```
-----------------------------------------------------------
Время          Высота          Дальность
-----------------------------------------------------------
 1.00           20.98            96.59
 2.00           32.16           193.19
 3.00           33.55           289.78
 4.00           25.13           386.37
 5.00            6.91           482.96
 6.00          –21.11           579.56
 7.00          –58.93           676.15
 8.00         –106.54           772.74
 9.00         –163.96           869.33
10.00         –231.18           965.93
```

Как видим, результат решения зависит от входных данных. Наибольшую дальность полета можно достичь за время, меньше чем 10 с, как было объявлено в постановке задачи. При этом координата, определяющая высоту, становится отрицательной, как будто снаряд пробил землю и продолжает лететь под землей. Этот факт требует улучшения постановки задачи, а именно нужно отслеживать траекторию полета, пока снаряд находит-

ся в движении, т. е. пока его вертикальная координата больше нуля.

Следовательно, от арифметического цикла следует отказаться и выбрать итерационный, в котором условием завершения вычислений будет условие «пока снаряд находится в полете», т. е. его вертикальная координата больше 0. Итерационный цикл также можно организовать с использованием любого из операторов цикла.

Приведем основную часть программы, отвечающую за организацию циклов. Условие завершения цикла
V*t*sin(Alpha)–0.5*G*t*t>0
управляет выполнением циклического алгоритма независимо от используемого оператора цикла посредством параметра цикла t, входящего в запись условия.

```
// 1. Оператор for
printf("--------------------------------------\n");
printf("Время Высота Дальность\n");
printf("--------------------------------------\n");
for(t=1.;V*t*sin(Alpha)–0.5*G*t*t>0.;t+=1.)
{
  Sx=V*t*cos(Alpha);
  Sy=V*t*sin(Alpha)–0.5*G*t*t;
  printf("%6.2f %6.2f %6.2f\n",t,Sy,Sx);
}

// 2. Оператор while
printf("--------------------------------------\n");
printf("Время Высота Дальность\n");
printf("--------------------------------------\n");
t=1.;
while(V*t*sin(Alpha)–0.5*G*t*t>0.)
{
  Sx=V*t*cos(Alpha);
  Sy=V*t*sin(Alpha)–0.5*G*t*t;
  printf("%6.2f %6.2f %6.2f\n",t,Sy,Sx);
  t+=1.;
}
```

```
// 3. Оператор do
printf("-------------------------------------\n");
printf("Время Высота Дальность\n");
printf("-------------------------------------\n");
t=1.;
do
{
  Sx=V*t*cos(Alpha);
  Sy=V*t*sin(Alpha)–0.5*G*t*t;
  printf("%6.2f %6.2f %6.2f\n",t,Sy,Sx);
  t+=1.;
}
while(V*t*sin(Alpha)–0.5*G*t*t>0.);
```

При выполнении этой программы процесс завершается по событию, число строк результирующей таблицы зависит от входных данных, и при тех же начальных значениях распечатка будет такой:

```
-------------------------------------------------------
Время          Высота          Дальность
-------------------------------------------------------
1.00           20.98            96.59
2.00           32.16           193.19
3.00           33.55           289.78
4.00           25.13           386.37
5.00            6.91           482.96
```

Пример 2.32. Изменим постановку задачи, чтобы получить чисто итерационный процесс. Например, пусть требуется определить время полета и его дальность при заданных начальных значениях скорости и угла. В этой постановке задачи итоговыми значениями являются t и Sx, но для проверки условия завершения нужно вычислять значение вертикальной координаты. Переменная Sy не нужна, так как формула проверки условия записана в виде логического выражения. Вывод заголовка не входит в подготовку цикла, а вывод на печать вынесен за тело цикла.

```
// 1. Оператор for
for(t=1.;V*t*sin(Alpha)–0.5*G*t*t>0.;t+=1.)
  Sx=V*t*cos(Alpha);
```

```
printf("--Время--Дальность\n");
printf("%6.2f %6.2f \n",t,Sx);

// 2. Оператор while
t=1.;
while(V*t*sin(Alpha)–0.5*G*t*t>0.)
{
   Sx=V*t*cos(Alpha);
   t+=1.;
}
printf("--Время--Дальность\n");
printf("%6.2f %6.2f \n",t,Sx);

// 3. Оператор do
t=1.;
do
{
   Sx=V*t*cos(Alpha);
   t+=1.;
}
while(V*t*sin(Alpha)–0.5*G*t*t>0.);
printf("--Время--Дальность\n");
printf("%6.2f %6.2f \n",t,Sx);
```

Еще более интересной задачей является задача определения попадания в цель. Если известны координата цели и ее размер, то после выполнения цикла значение Sx можно сравнить с координатой цели. Например, пусть координата центра мишени Sm, а ее линейный размер L, тогда условие попадания в цель запишется:

```
if(Sx<Sm+0.5*L&&Sx>Sm–0.5*L)
   printf("Цель поражена\n");
else
   if(Sx<Sm–0.5*L)
      printf("Недолет\n");
   else
      printf("Перелет\n");
```

Надо заметить, что и в последнем случае задача решена некорректно. Значение времени t выбрано дискрет-

ным, и изменение t равно 1 с. Следовательно, точность вычисления неизвестна, но весьма невелика.

Еще более интересной задачей является задача подбора входных данных t и V таким образом, чтобы обеспечить попадание в цель. Это задача моделирования, имеющая более сложное решение.

2.4.9. Оператор прерывания break для циклов do, while, for

Назначение. Прекращение выполнения цикла с передачей управления следующему за циклом оператору.

Синтаксис.

```
break;
```

Выполнение. Оператор break осуществляет выход из тела цикла к оператору, следующему по порядку за циклом. Используется для прерывания при организации бесконечных циклов, позволяя корректно завершить алгоритм.

Пример 2.33. Найти сумму арифметической прогрессии всех чисел натурального ряда, которая не превышает некоторого наперед заданного значения, например, n.

```
void main(void)
{
  int n, Sum=0, i;                    // i — числа натурального
ряда
  printf("\nВведите наибольшее значение\n");
  scanf("%d", &n);
  for(i=1;;i++)              // бесконечный цикл
  {                          // суммирует числа i
    Sum+=i;
    if(Sum>n)                // условие завершения
      break;                 // прерывание цикла
  }
  printf("Количество элементов, включенных в сумму
%d\n",i);
}
```

Особенности. В случае вложения циклов оператор break прерывает только непосредственно охватывающий цикл.

2.4.10. Оператор продолжения continue для циклов do, while, for

Назначение. Переход к следующей итерации тела цикла без прерывания выполнения цикла. Противоположен break.

Синтаксис.

continue;

Выполнение. В любой точке цикла continue прервет текущую итерацию и перейдет к проверке условия завершения цикла.

Пример 2.34. Найти сумму дробей вида $1/x$, если $x \in [-1; +1]$, шаг изменения $\Delta x = 0{,}1$. При вычислении суммы $\frac{1}{-1}+\frac{1}{-0{,}9}+\ldots+\frac{1}{0{,}1}+\frac{1}{0}+\frac{1}{0{,}1}+\ldots+\frac{1}{1}$ точка $x = 0$ является особой точкой, в которой знаменатель дроби равен 0, и, следовательно, значение слагаемого не определено. Можно организовать два цикла сложения с управлением по $x \in [-1; -0{,}9]$ и $x \in [0{,}1; +1]$, а можно в одном цикле предусмотреть особую точку и не вычислять очередное слагаемое.

```
void main(void)
{
  float Sum=0, x;
  for(x=-1;x<=1.05; x+=0.1)
  {
    if(fabs(x)<0.0001)
      continue; // если в знаменателе 0
    Sum+=1/x;
  }
  printf("Сумма = %8.2f\n", Sum);
}
```

2.4.11. Оператор выбора switch

Назначение. Организация множественного ветвления, когда при выполнении алгоритма возможна передача управления на несколько ветвей в зависимости от условий, сложившихся при выполнении программы.

Синтаксис.

switch (Выражение)

```
{
  case  // Значение1:
  {
    // Оператор_1;
    break;
  }
  case  // Значение2:
  {
    // Оператор_2;
    break;
  }
  // и так далее
  case  // ЗначениеN:
  {
    // Оператор_N;
    break;
  }
  default:
  {
    // Оператор_N+1;
  }
}
```

Слово switch — это название оператора. Здесь «Выражение» — это целочисленное или символьное выражение, которое может принять одно из нескольких прогнозируемых значений. Каждая метка case связана с константой, в тексте они названы «Значение_НОМЕР». Все значения констант должны быть различны. Слово default также обозначает метку.

Выполнение. Вначале вычисляется значение выражения, затем вычисленное значение последовательно сравнивается с каждой константой «Значение_НОМЕР». При первом же совпадении выполняются операторы, помеченные данной меткой. Обычно это составной оператор, который завершается оператором прерывания break, return или exit(), как показано в описании синтаксиса. В случае использования оператора break происходит прерывание с выходом из switch к следующему по порядку оператору. Если в составном операторе нет оператора прерывания, то

после выполнения операторов, найденных по совпадению значения, выполняются операторы всех последующих вариантов по порядку до тех пор, пока не будет встречен оператор прерывания или не закончится оператор выбора. Операторы, стоящие за меткой default, выполняются тогда, когда значение выражения не совпало ни с одним значением константы «Значение_НОМЕР». Метка default может отсутствовать.

Пример 2.35. Для управления используется символьное значение знака операции. При вычислении переменной S, если sign='+', то значение x прибавляется к S, если sign='–', вычитается. Если переменная sign отлична от символов операций '+' или '–', то S=x.

```
char sign;
// здесь sign каким-то образом получает значение
S=0;
switch(sign)
{
   case '+':
   {
      S+=x;
      break;
   }
   case '–':
   {
      S–=x;
      break;
   }
   default:
      S=x;
}
```

Особенности. Оператор switch способен работать как переключатель только в том случае, если в каждой ветви case есть оператор прерывания break (или другие, например, return). В случае использования оператора break управление из switch передается к следующему по порядку оператору. Если в ветви case нет оператора прерывания, то управление передается на следующую ветку case, и выполняются операторы всех последующих вариантов

по порядку до тех пор, пока не будет встречен оператор прерывания или не закончится оператор выбора.

Пример 2.36. Найти сумму N слагаемых вида

$$S = \sin(x) + \cos(x) + \sin(x) + \cos(x) + \ldots$$

Приведем фрагмент программы, решающий эту задачу.

```
for(n=1,S=0; n<=N; n++)
{
  switch(n%2==0)
  {
     case 0:
     { S+=sin(x); break; }
     case 1:
     { S+=cos(x); break; }
  }
}
```

Пример 2.37. Если в примере 2.36 убрать оператор прерывания, то будет найдена сумма N слагаемых вида

$$S = \cos(x) + \sin(x) + \cos(x) + \cos(x) + \sin(x) + \cos(x)\ldots$$

```
for(n=1,S=0;n<=N;n++)
{
  switch(n%2==0)
  {
     case 0:
        S+=sin(x); // при n=2, 4 и т.д. S=S+sin(x)+cos(x)
     case 1:
        S+=cos(x); // при n=1, 3 и т.д. S=S+cos(x)
  }
}
```

2.5. МЕХАНИЗМ ФУНКЦИЙ ЯЗЫКА С++

Язык С++ является процедурно-ориентированным языком, и это означает, что принципы программирования С++ основаны на понятии функции.

Как правило, прикладная программа на С++ содержит несколько файлов кода, где каждый файл — это со-

вокупность функций. Функции объединяются в модули, модули — в проект. В состав языка входят также библиотеки стандартных функций, которые содержат функции обработки данных, например, библиотека <math.h> содержит математические функции и константы, библиотека <stdio.h> — функции обмена с внешними устройствами.

2.5.1. Введение в модульное программирование

Модуль — отдельный файл, в котором группируются функции и связанные с ними данные. Модуль решает некоторую задачу обработки данных.

Для начинающих программистов рекомендуется использование модульного стиля программирования, достоинства которого очевидны.

1. Алгоритмы отделены от данных. Как правило, данные имеют структуру, определенную логикой прикладной задачи. Алгоритмы обработки данных определяются составом и структурой данных.

2. Высокая степень абстрагирования проекта. Достигается при использовании функциональной декомпозиции.

3. Модули кодируются и отлаживаются отдельно друг от друга. Чем более они независимы, тем легче процесс отладки. Функции, входящие в состав каждого модуля, представляют его интерфейс. Для использования модуля достаточно знать только его интерфейс, не вдаваясь в подробности реализации.

Функциональная декомпозиция — это метод разработки программ, при котором задача разбивается на ряд легко решаемых подзадач, решения которых в совокупном виде дают решение исходной задачи в целом.

Проектирование приложения строится от абстрактного описания основной задачи — это высший уровень абстракции. Основная задача может быть разбита на ряд более простых подзадач — это второй уровень абстракции. Каждая из подзадач, в свою очередь, разбивается на ряд еще более простых. Процесс детализации заканчивается, когда очередная подзадача не может быть больше разде-

лена на более простые составляющие или когда решение очередной задачи становится очевидным.

В процессе детализации создается иерархическое дерево решения задачи, где каждый уровень дерева является решением более детализированной задачи, чем предшествующий уровень.

Каждый блок представляет собой программный модуль. Каждый модуль — это законченный алгоритм решения некоторой конкретной задачи.

Процесс кодирования выполняется снизу вверх, от написания и полной отладки кода небольших подзадач с их последующей сборкой на верхнем уровне, при этом каждый модуль безошибочно решает одну задачу. Объем задачи нижнего уровня достаточно небольшой — одна или две страницы кода (не более 50 строк).

Приведем пример применения метода функциональной декомпозиции к решению задачи, поставленной неформально.

Пример 2.38. Жители Средиземья организовали курсы информатики. Список дисциплин, по которым проводятся курсы, пока содержит три дисциплины: «Прикладные пакеты Office», «Офисное программирование» и «Защита информации», но в дальнейшем предполагается его изменять и расширять. Занятия ведут приглашенные преподаватели и специалисты. Количество часов по каждой дисциплине определено. Для слушателей определена оплата одного курса, а для преподавателей, в зависимости от квалификации, — оплата одного часа занятий. По окончании курсов всем выдаются сертификаты. Лучшим слушателям вручаются похвальные листы.

Статическая модель приложения моделирует структуры данных, которые предназначены для хранения всей информации, связанной с задачей. Выделим все сущности, данные о которых необходимо хранить:

- курсы (дисциплины);
- слушатели;
- преподаватели;
- аудитории;
- оплата и т. д.

Для хранения данных используются таблицы, сохраняющие полные данные о сущностях информационных процессов.

Динамическая модель поддерживает сценарий работы:

- кто-то организует занятия, проводит рекламу и обеспечивает материальную базу;
- кто-то отвечает за дидактическую наполненность курсов;
- слушатели записываются на курсы, этот процесс занимает определенное время;
- по мере комплектации групп приглашается преподаватель и начинаются занятия;
- для проведения занятий составляется расписание;
- параллельно или последовательно взимается плата за обучение, по окончании курса оплачивается труд преподавателя;
- по окончании курса подводятся итоги, выдаются сертификаты.

Функциональную декомпозицию задачи можно выполнить в два этапа. На первом этапе изобразим модульную структуру информационной системы (ИС), определив потоки данных для всех участников задачи (рис. 2.4).

В состав ИС входят следующие модули (блоки).

1. Модуль «Слушатели» управляет информацией о слушателях курсов.

2. Модуль «Преподаватели» управляет информацией о приглашенных преподавателях.

Рис. 2.4
Первый уровень модульной структуры ИС

3. Модуль «Бухгалтер» управляет информацией о денежных потоках.

4. Модуль «Методист» управляет информацией о составе курсов и их дидактическом содержании, а также может управлять расписанием.

5. Модуль «Администратор» имеет доступ к функциональности отдельных модулей ИС.

На примере модуля «Слушатели» проведем функциональную декомпозицию. Логика задачи требует реализации следующих функций (рис. 2.5).

Рис. 2.5
Декомпозиция модуля «Слушатели»

Обычные операции добавления, удаления, редактирования позволяют вносить изменения в таблицы данных о слушателях курсов.

Функция «Оформить сертификат» рассчитывает количество часов, проверяет выполнение контрольных мероприятий, вносит данные в итоговый документ.

Функция «Сведения об оплате» контролирует оплату курса слушателем.

Кроме того, модуль должен иметь функции поиска информации о слушателе по многим критериям: по фамилии, по успеваемости, по посещаемости, по оплате, и др.

Подобным образом выполняется декомпозиция для всех модулей ИС.

Так, в модуле «Преподаватели» должны быть реализованы функции: добавить, редактировать, вывести информацию, рассчитать количество проведенных часов, рассчитать сумму к оплате, а также многие функции поиска по различными критериям.

Обратим внимание на то, что различные модули ИС могут иметь сходную функциональность, например, функции поиска одинаковы в нескольких модулях.

Среди функций модулей можно увидеть типичные алгоритмы обработки данных: добавление, удаление, поиск, суммирование, сортировка и др.

2.5.2. Назначение и виды функций

Функция — это самостоятельный именованный алгоритм решения некоторой законченной задачи.

Фактически, функция является одним из конструируемых пользователем типов данных, и как объект программы:

- имеет имя;
- имеет тип;
- может иметь параметры (аргументы функции).

Назначение функций можно рассматривать двояко. Во-первых, утилитарное назначение, при котором функция используется только для того, чтобы сократить код программы. Тогда в виде функции записывается выражение, которое повторяется в тексте программы несколько раз. Во-вторых, как требуют принципы модульного подхода, использование функций позволяет получать хорошо структурированные программы.

Язык C++ имеет библиотеки стандартных функций. Библиотечные функции C++ хранятся в компилированном виде и присоединяются на этапе сборки. Объявления библиотечных функций содержатся в заголовочных файлах с именами <имя.h>. Чтобы объявление стало доступно программе, в тексте программы записывается директива препроцессора #include <имя.h>. При этом текст заголовочного файла <имя.h>, в котором есть объявление функции, включается в код программы на этапе препроцессорной обработки. В этом смысле объявление функции — это аналог объявлению переменных. Подробное описание всех библиотек языка C++ можно найти в справочной системе.

Пользователь также может определять собственные функции и объединять их в библиотеки.

Создание простой функции

Любую функцию следует описать. *Описание* функции вводит в употребление собственно функцию и содержит абстрактное описание внешних данных функции и испол-

няемого ею алгоритма. В языке С++ все функции описываются на одном уровне, вложений не допускается.

Структура описания простой функции не отличается от структуры функции main() и в общем виде такова:

```
Тип_функции Имя_функции
(Тип_параметров Имена_параметров)
{
// описания локальных переменных;
// описание алгоритма;
return возвращаемое_значение;
// отсутствует, если функция типа void
}
```

Отдельные составляющие могут быть опущены, кроме имени и блока тела функции. Первая строка описания функции называется *заголовком* функции, в ней указаны все внешние характеристики функции.

«Тип функции» — это тип возвращаемого функцией значения или void для функций, которые не возвращают значения. Типом функции может быть имя базового типа или указатель.

«Имя функции» — это имя main для главной функции программы или любое, не совпадающее с ключевыми словами и именами других объектов программы.

«Имена_параметров» — это формальные параметры функции: перечисленные через запятую имена аргументов функции вместе с их типами. Как правило, это входные данные для функции, т. е. те, которые функция получает извне. Они могут быть и результатом работы функции, но об этом позже. В списке параметров указывается void, если параметров нет. Пустой список параметров означает, что список параметров может быть произвольной длины.

Пример описания простой функции.

```
// функция возведения в квадрат
// описана перед функцией main()
float sqr(float x)          // тип, имя, один параметр
{
  return x*x;               // сразу возвращает значение
}
```

```
// теперь функция main() может к ней обратиться
void main(void)
{
   float a, b;
   a=2;
   b=sqr(a);
   printf("\n%f", b);          // здесь a — фактический параметр
}
```

Замечания о типе функции

Поскольку функция возвращает значение, она обладает типом. В этом смысле функция рассматривается как данное, тип которого конструируется пользователем. Данное должно иметь значение. Для функции значением является значение выражения, записанного в операторе return:

- если возвращается простой объект, то тип функции — какой-нибудь базовый тип;
- если возвращается объект сложного конструируемого типа (массив, структура и подобные), то тип функции — это указатель (адрес) объекта;
- если функция не возвращает значения, ее тип void и в теле функции может не быть оператора return;
- если тип вообще не указан, это не ошибка, тип функции по умолчанию целый (int).

Если главная функция программы main начинается void main (void), то это означает, что функция не возвращает значения и не имеет параметров.

Если заголовок функции main(), то это означает, что тип функции int, а число параметров произвольно. В этом случае в теле main() должен быть оператор return, возвращающий значение функции в окружающую среду, например, return 1; или return 0;.

Если функция main() имеет параметры, то они передаются как параметры командной строки при вызове программы.

Тело любой функции содержит:

- описания локальных переменных;
- описание алгоритма;
- возврат в точку вызова по оператору return.

Описания объектов в теле функции используются, чтобы объявить *локальные* переменные функции — те, которыми функция пользуется для решения своих частных задач. Областью действия локальных переменных является тело функции.

Описание алгоритма не представляет особенностей. Это должен быть общий алгоритм решения некоторой самостоятельной задачи. Он замкнут в себе, абстрактен. Единственное, что связывает его с внешним миром, это входные данные, приходящие извне, и результат работы алгоритма, который он должен передать в вызывающую функцию (вернуть во внешний мир).

Возврат из функции в точку вызова выполняет оператор return, который передает управление в вызывающую программу и возвращает одно значение. Может иметь две формы:

```
return выражение;  // выход из функции с передачей значения
return;            // выход из функции без передачи значения
```

Таким образом, возвращаемое значение может быть только одно. На самом деле это не так, далее мы рассмотрим механизм указателей, который позволит решать задачи передачи данных.

Оператор return в тексте функции может быть не один, если алгоритм функции требует, чтобы вариантов выхода из нее было несколько.

Обращение к функции

Под обращением к функции понимается реальное выполнение ее алгоритма, каждый раз с различными входными данными. Выполняется из любой другой функции программы с использованием механизма вызова функции через операцию обращения. Обращение к функции зависит от типа функции и может иметь две формы.

1. Первая форма обращения — *оператор-выражение*.

Используется для функций, возвращающих значение. Значение функции — это одно значение базового типа или указатель, которое может быть использовано в выражениях или печати в виде обращения к функции.

Синтаксис оператора-выражения:
имя_функции (фактические параметры)

Это обращение можно использовать везде, где значение функции должно быть вычислено, например, в выражении присваивания:
переменная = имя_функции(фактические параметры);

в списке вывода при печати:
printf("форматная_строка", имя_функции(фактические_ параметры));

Покажем на примере обращения к библиотечной функции sin().

```
y=sin(x);                 // значение функции вычислено
                          // и присвоено
printf("%6.2f",sin(x));   // значение функции напечатано
sin(x);                   // значение функции вычислено, но
                          // что происходит с вычисленным
                          // значением?
```

2. Вторая форма обращения — *оператор-функция*.

Если функция не возвращает значения (функция типа void), то обращение к ней выглядит как обычный оператор программы и имеет специальное название «оператор-функция», синтаксис которого:
имя_функции (фактические параметры);

Внешне оператор-функция выглядит так же, как обычный оператор программы. Например, обращение к библиотечной функции printf():
printf("%d, %d",a, b);

Формальные и фактические параметры функции

В описании функции ее внешние данные названы *формальными* параметрами. Это название подчеркивает, что данные описания формальны, абстрактны, не участвуют в реальных действиях, а только описывают взаимосвязь данных в теле функции. Формальные параметры функции — это всегда имена переменных. В заголовке функции они объявляются, поэтому и должны быть указаны вместе с типом.

При обращении к функции ей передаются *фактические* параметры. Это параметры, значения которых из-

вестны на момент обращения к функции, и с которыми функция отрабатывает очередной вызов. Фактические параметры подставляются на место формальных при каждом обращении к функции. Они могут быть в общем случае константами, переменными, или выражениями.

Тип параметров, их количество и порядок следования называются совместно «*сигнатура параметров*». Есть непреложное правило С++: в описании функции и в обращении сигнатуры параметры должны строго совпадать. Это означает, что формальные и фактические параметры должны соответствовать друг другу по количеству, типу и порядку следования.

Пример 2.39.

Пример иллюстрирует разные варианты обращения к функции, возвращающей среднее арифметическое значение трех чисел.

```
float Avg(float a, float b, float c)
{
  float S;                  // локальная переменная
  S=(a+b+c)/3.;
  return S;                 // тип совпадает с типом функции
}
void main(void)
{
  float x1=2.5, x2=7, x3=3.5;
  float y;
  // фактические параметры – переменные
  y=Avg(x1, x2, x3);
  // обращение в присваивании
  printf("x1=%f, x2=%f, x3=%f y=%f\n", x1, x2, x3, y);
  // фактические параметры – константы вещественного типа
  y=Avg(2., 4., 7.);
  printf("x1=%f, x2=%f, x3=%f y=%f\n", 2., 4., 7., y);
  // фактические параметры – выражения
  y=Avg(x1*2., x2+4., sin(PI/2.));
  printf("x1=%f,x2=%f,x3=%f,y=%f\n",2*x1,x2+4.,sin(PI/2.),y);
  // обращение в функции вывода, фактические параметры
  // произвольные, то есть константы, переменные,
  // выражения
```

```
  printf("x1=%f,x2=%f,x3=%f,y=%f\n",
  2.,x2,x3+0.7,Avg(2.,x3+0.7));
  // оператор-обращение может входить в другие
  // выражения
  y=(Avg(0.5, 1.7, 2.9)+Avg(x1,x1+2,x1+2.))*0.5;
  printf("y=%f\n",y);
}
```

2.5.3. Описание и объявление функции. Прототип функции

С точки зрения структуры программы, в C++ все функции описываются на одном уровне. Поясним различие в терминах описание и объявление.

Описание функции предоставляет программисту всю информацию о ней, как внешнюю (тип, имя, сигнатура параметров), так и внутреннюю (тонкости реализации алгоритма). Если рассматривать функцию как данное конструируемого типа, то описание функции как раз и конструирует данный тип с указанным именем.

Объявление функции предоставляет только внешнюю (интерфейсную часть), для которой важно только умение правильно использовать функцию, т. е. обратиться к ней. Заголовок (первая строка описания) функции как раз и предоставляет такую информацию.

Любой объект программы должен быть объявлен перед первым его использованием. Это непреложное правило относится и к функциям. «Объявление» или «описание» функции в программе должно появиться до первого обращения к ней, и здесь возможны варианты.

1. Описание функции фактически расположено перед текстом вызывающей программы. В этом случае описание функции есть одновременно и ее объявление.

Пример 2.40.

```
// функции без параметров,
// описание опережает обращение
void f1(void)       // описание функции f1
{
  printf("Функция 1\n");
}
```

```
void f2(void)      // описание функции f2
{
  printf("Функция 2\n");
}
void main(void)  // функции известны до момента обращения
{
  printf("\nГлавная функция main\n");
  f1();            // обращение к функции f1
  f2();            // обращение к функции f2
}
```

В этом случае описание функций играет роль их объявления.

2. Описание функции фактически расположено после текста вызывающей программы. Значит, обращение к функции появляется перед ее описанием. Чтобы функция стала известной вызывающей программе, кроме описания, нужно еще и объявление функции. Объявление функции (в С++ это еще называется *прототип*) — это ее заголовок, за которым стоит знак завершения оператора «;». Прототип может быть записан в начале программы, если функция глобальна, или в теле main(), если функция локализована в main(), или в теле любой функции, где функция известна локально. Прототип функции — это аналог описания переменных.

Пример 2.41.

Использованы те же самые функции без параметров, что и в примере 2.40. Прототип объявляет функцию.

```
void f1(void);     // прототип функции f1
void f2(void);     // прототип функции f2
// обращение опережает описание
void main(void)
{
  printf("\nГлавная функция main\n");
  f1();            // обращение к функции f1
  f2();            // обращение к функции f2
}
// описание функций после вызывающей программы
void f1(void)      // описание функции f1
{
```

```
   printf("Функция 1\n");
}
void f2(void)       // описание функции f2
{
   printf("Функция 2\n");
}
```

3. Наилучший способ организации проекта, это вынесение описаний в отдельные файлы.

Все описания функций выносятся в отдельный файл, который называется заголовочным файлом. Он подключается к любой программе, которая их использует. Это обычные текстовые файлы, содержанием которых является описание собственных функций пользователя, а также констант и других данных, глобальных по отношению к проекту в целом. Заголовочные файлы, как правило, имеют расширение .h или .hpp (от head — голова), легко добавляются к любому проекту, и включаются в программный код директивой препроцессора
#include "Имя_файла.h".

2.5.4. Передача параметров в функцию. Изменяемые значения параметров

В языке C++ существуют два способа передачи параметров в функцию.

1. *По значению*. До сих пор рассматривался синтаксис именно этого способа передачи данных. Его механизм — это создание локальной копии параметра в теле функции. В таком случае значения фактических параметров нельзя изменить в результате обращения. Это позволяет защитить входные данные от нежелательного изменения функцией.

2. *По ссылке*. Изменение механизма передачи данного в функцию формально выглядит добавлением признака адресной операции к имени параметра в описании функции:

```
Тип_ функции Имя_функции
(Тип_параметра &Имя_параметра)
```

Механизм такого способа передачи данных заключается в том, что функция и вызывающая программа ра-

ботают с адресом объекта в памяти (фактически, с одной и той же областью данных). Как результат, функция может изменить значения фактических параметров. Как следствие, фактическим параметром, соответствующим формальному параметру-ссылке, может быть только имя переменной.

Пример 2.42. Функция находит площадь и периметр треугольника, заданного длинами сторон. Возвращаемых значений два — площадь и периметр. Эти значения функция вернет через параметры. Кроме того, нужна проверка условия существования треугольника. Это логическое значение функция вернет через оператор return (1, если треугольник существует, и 0, если не существует).

```
#include <stdio.h>
#include <math.h>
int Triangle(float a, float b, float c, float &p, float &s)
{
  // функция имеет два варианта выхода
  // параметры a, b, c передаются по значению
  // (только входные данные),
  // параметры p, s, по ссылке
  // (как входные данные, так и результат)
  float pp;                          // полупериметр
  if(a+b<=c||a+c<=b||b+c<=a) // треугольник не существует
    return 0;
  else
  {                                  // треугольник существует
    p=a+b+c;
    pp=0.5*p;
    s=sqrt(pp*(pp–a)*(pp–b)*(pp–c));
    return 1;
  }
}
```

При обращении фактическими параметрами, подставляемыми на место формальных параметров-значений, могут быть выражения, а фактическими параметрами, подставляемыми на место формальных параметров-адресов, могут быть только переменные.

```
void main(void)
{
float A, B, C;
// длины сторон – фактические параметры
float Perim, Square;
// периметр и площадь – фактические параметры
// пример обращения
printf("Введите длины сторон треугольника\n");
scanf("%f%f%f", &A, &B, &C);
if(Triangle(A, B, C, Perim, Square)==1)
     printf("Периметр = %6.2f, площадь = %6.2f\n", Perim,
                                                    Square);
  else
     printf("Треугольник не существует\n");
}
```

При обращении к функции происходит следующая цепочка событий (рекомендуется наблюдать данный процесс в отладчике).

1. Управление передается в функцию.

2. Выделяется память для параметров функции, вычисляются значения формальных параметров и копируются в локальную память. Так передаются внешние данные, необходимые для работы функции.

3. Создаются локальные переменные функции (те, которые объявлены в теле функции).

4. Выполняется алгоритм функции.

5. По оператору return управление передается в точку вызова, при этом в вызывающую программу передаются новые данные (результат функции).

6. Локальные переменные, в том числе формальные параметры, умирают, память высвобождается.

2.5.5. Перегруженные функции

Согласно правилам описания функций принято, что имя функции отражает ее внутреннее содержание, так или иначе сообщает пользователю о том, какой алгоритм реализует эта функция. Например, имена математических функций sin(), sqrt(), fabs() говорят сами за себя, имена функций ввода-вывода printf(), scanf() под-

черкивают, что ввод-вывод является форматированным (суффикс f) и т. д. Вместе с тем список возможных имен, описывающий абстрактный смысл алгоритма, является ограниченным.

Перегрузка функций широко используется в современных компиляторах С++. Принцип перегрузки заключается в том, что одно имя может реализовать разные алгоритмы. Это означает, что можно описать несколько функций с одним и тем же именем. Они должны отличаться друг от друга либо количеством параметров, либо типом параметров. Использование одного имени в разных целях называется *перегрузкой*.

Возможна перегрузка:

- по числу параметров;
- по типу параметров.

Таким образом, перегруженные функции различаются по сигнатуре параметров.

Механизм работы перегруженных функций основан на обработке вызовов, в которых показана сигнатура фактических параметров. Их типы или их количество сообщают компилятору, какой именно функции сопоставить данный вызов.

Пример 2.43. Пример перегрузки функции по числу параметров.

Пусть требуется функция, которая вернет наибольшее из своих параметров. Именем функции должно быть абстрактное имя, соответствующее смыслу алгоритма, например, имя max.

Число параметров может быть различным, так, прототипы функций могут выглядеть:

```
int max(int, int);          // функция получает два параметра
int max(int, int, int);     // функция получает три параметра
```

Для каждого алгоритма требуется своя реализация, поэтому для каждой реализации пишется функция.

```
int max(int x,int y)
{
  return x>y?x:y;
}
int max(int x,int y,int z)
```

```
{
   return x>y?(x>z?x:z):(y>z?y:z);
}
```

В зависимости от сигнатуры фактических параметров, переданных при вызове, будет вызван и отработает нужный алгоритм, например:

```
int Angle1, Angle2, Angle3;
// вызов функции с двумя параметрами
int Angle_max=max(Angle1, Angle2);
// вызов функции с тремя параметрами
Angle_max=max(Angle1, Angle2, Angle3);
```

Пример 2.44. Пример перегрузки функции по типу параметров.

Функцию max() можно перегрузить еще раз, так, чтобы она находила наибольшее значение из элементов массива. Тогда имя будет прежнее, а тип параметров совершенно иной. Прототип функции будет таким:

```
int max(int *,int);          // функция получает два параметра
```

Первый параметр — это указатель на массив, а второй — число элементов массива. Это будет перегрузкой функции по типу параметров.

Реализация алгоритма для массива:

```
int max(int x[],int n)
{
   int m=x[0];
   for(int i=0;i<n;i++)
      if(x[i]>m) m=x[i];
         return m;
}
```

При обработке вызова компилятор сопоставит сигнатуру фактических параметров требуемой реализации алгоритма, и безошибочно передаст управление функции, которая может это сделать.

```
int a[10];
int Angle_max;
Angle_max=max(Angle1,Angle2); // найдет наибольший угол
int Max_value;
Max_value=max(a, 10);          // найдет наибольший
                               // элемент массива
```

2.6. ЛОКАЛЬНЫЕ И ГЛОБАЛЬНЫЕ ДАННЫЕ. ВРЕМЯ ЖИЗНИ И ОБЛАСТЬ ДЕЙСТВИЯ ОБЪЕКТОВ

В тексте программы термином «объект» называют переменные, именованные константы, функции, в целом имена данных различных типов. Объект программного кода присутствует в тексте своим именем. Каждый объект должен быть объявлен.

Область действия объекта — это область программного кода, в которой объект известен (т. е. действует его объявление). Если объект объявлен в начале программы, вне тела всех функций, то он известен везде внутри того файла, где объявлен.

Время жизни объекта — это понятие, связанное с областью действия: период времени в процессе выполнения программы, когда объект фактически занимает память (память выделяется при объявлении).

Локальные (внутренние) объекты объявлены внутри тела блока. Локальные объекты функции объявлены внутри тела функции.

Область действия локального объекта — блок, в котором он описан. Описание действует от точки описания до конца блока.

Время жизни локального объекта — только время выполнения блока. При входе в блок память выделяется, при выходе память освобождается.

Глобальные (внешние) объекты объявлены вне тела функции на внешнем уровне.

Область действия глобального объекта — от точки объявления до конца файла с кодом программы, в котором объявлен объект.

Время жизни глобального объекта — время выполнения программы.

Параметры функций по механизму действия тоже можно отнести к локальным или глобальным.

Параметр, передаваемый по ссылке, действует по механизму глобального объекта. Это единый объект, так как функция и вызывающая программа работают с одним адресом объекта. Это удобное средство для того, чтобы сделать открытым адресное пространство.

Параметр, передаваемый по значению, действует по механизму локального объекта. Это объект, который создается и живет в течение времени работы функции. Это надежный механизм защиты внешних данных от случайного их изменения функцией.

2.6.1. Принцип локализации имен

Этот термин имеет много синонимов, например, пространство имен, приоритет имен, принцип сокрытия имен. Применяется в случае, когда локальное имя какого-нибудь объекта в теле функции совпадает с именем глобального объекта. Что касается выбора имен, то имя любого объекта в любом блоке уникально, но в разных блоках имена могут повторяться, неся различную смысловую нагрузку. Особенно это касается имен рабочих переменных.

Пример 2.45.

```
int A=90;
void F(int x)
{
  int A=0;   // каким же будет значение A в теле функции?
}
```

Принцип локализации имен заключается в том, что на время действия локальной переменной глобальная переменная с тем же именем временно прекращает свое существование. Она возобновляет свое значение при завершении работы функции, как только заканчивается время жизни локальной переменной.

Пример 2.46. Чтобы увидеть механизмы локализации данных и принцип локализации имен, этот пример следует выполнять в отладчике.

```
#include <stdio.h>
#include <conio.h>
// глобальные данные программы
int a,n=1;
void f1(void);  // объявление функций глобально
void f2(void);
void f3(int);
void main(void)
{
```

```
int i;          // локальная переменная функции main
a=10;           // функция main изменяет глобальную
                // переменную
                // все функции имеют одинаковые права на
                // изменение глобального данного
for(i=1;i<=3;i++)
{
  n++;          // main изменяет значение n, нумеруя
                // последовательность вызовов функции f1
printf("\nMain. Вход № %d", n);
f1();           // f1 тоже меняет значение n, потому что main
}               // и f1 имеют одинаковые права на изменение n
for(i=1;i<=3;i++)
{
  a++;          // main изменяет значение переменной a
  printf("\nMain. Глобальная a=%d",a);
  f2();         // в функции f2 значение переменной a свое
}
                // как передача данного в функцию защищает
                // его от изменения?
f3(a);
printf("\nГлобальное a после обращения к f3=%d",a);
}

// f1 печатает номер обращения
void f1()
{
n++;            // функция меняет глобальную переменную n
printf("\nФункция f1. Вход %d",n);
}

// f2 печатает значение локальной переменной
void f2()
{
  int a=90;  // a — локальное данное функции f2
  printf("\nФункция f2. Локальная a=%d", a);
}

// f3 печатает значение локальной переменной
```

```
  void f3(int a) // глобальная переменная передается
  {              // как параметр, защищена от изменения
  a++;           // локальное данное функции f3
  printf("\nФункция f3. Не меняет глобальное a=%d", a);
}
```

2.6.2. Классы памяти

Существуют три класса памяти, к которым может быть отнесен объект программного кода: auto, static, register. Они классифицируют переменные по схеме выделения памяти. Приписываются перед объявлением типа переменной, например,

```
auto     int i;
static   int n;
register int k;
```

Класс auto называется автоматическим классом памяти, он действует по умолчанию. Регистровый класс памяти (register) раньше использовался для того, чтобы явно указать компилятору, что этот объект должен непосредственно размещаться в регистрах процессора, что позволяло повысить быстродействие программ. Компиляторы современного уровня являются оптимизирующими, и сами определяют механизмы выделения памяти, поэтому объявление register носит рекомендательный характер, и объявленный объект будет, скорее всего, класса auto. Класс static называется статическим классом памяти, используется для того, чтобы вынести на глобальный уровень объявление локального объекта.

Механизм действия классов памяти зависит от того, как локализован объект.

Объекты auto существуют только внутри того блока, где они определены. Память для объекта выделяется при входе в блок, а при выходе освобождается, т. е. объекты перестают существовать. При повторном входе в блок для тех же объектов снова выделяется память, значения переменных не сохраняются.

Объекты static существуют в течение всего времени выполнения программы. Память выделяется один раз при старте программы и при этом обнуляется. При повторном

входе в блок, где объявлен статический объект, его значение сохраняется. Вне блока, где объявлен статический объект, его значения недоступны.

Пример 2.47.

```
void f_auto(void)
{
  // переменная K по умолчанию имеет класс auto
  int K=1;          // локальный объект автоматической памяти
  printf("K=%3d ",K);
  K++;
  return;
}
void main(void)
{
  for(int i=1;i<=3;i++)
    f_auto();
}
```

Результат выполнения программы обусловлен локализацией объекта K:

```
K=1 K=1 K=1
```

В том же примере покажем действие статического объекта:

```
void f_stat(void)
{
// переменная K статическая
  static int K=;   // локальный объект статической памяти
  printf("K=%3d ",K);
  K++;
  return;
}
void main(void)
{
  for(int i=1;i<=3;i++)
  f_stat();
}
```

Результат выполнения программы обусловлен «глобализацией» объекта K с изменением класса памяти:

```
K=1 K=2 K=3
```

Вне тела функции f_stat объект недоступен.

2.7. ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ

В языке C++ есть две группы типов: базовые типы и производные типы.

Синонимами названия «базовый тип» являются названия «стандартные», «предопределенные». Реализация этих типов заложена в стандарте языка. *Производным* типом является такой тип, который описывается пользователем перед употреблением. Полезно напомнить, что тип данного определяет его размещение в памяти и набор операций.

К производным типам относятся функции, массивы, указатели, структуры и объединения.

2.7.1. Одномерные массивы

Определение массива

Массив — это упорядоченное множество данных одного типа, объединенных общим именем.

Тип элементов массива может быть почти любым типом C++, не обязательно это базовый тип.

Обязательные атрибуты массива — это тип элементов, число элементов и размерность.

В главе 1 пособия приведены блок-схемы алгоритмов, использующих как одномерные, так и двумерные массивы.

Чтобы ввести в употребление массив, его необходимо *описать* (конструировать). Можно одновременно выполнить *инициализацию* элементов массива. Смысл описания и объявления для массивов идентичен. Описание массива выполняется, как и для обычных переменных, в разделе описаний.

Синтаксис.

Тип_массива Имя_массива [количество_элементов];

Здесь «количество_элементов» — это константа, заданная явно или именованная (константное выражение).

Пример 2.48.

```
#define N 10
int mas[3];        // одномерный массив mas содержит 3
                   // элемента целого типа
```

```
int matr[2][10];  // одномерный массив matr содержит два
                  // одномерных массива по 10 элементов
                  // в каждом
float w[N];       // одномерный массив w содержит N
                  // вещественных элементов
char c[5][80];    // массив из 5 строк, по 80 символов в строке
```

Примечание. Элементы массива нумеруются с 0, т. е. для массива, описанного как int mas[3] имеются элементы mas[0], mas[1], mas[2].

При объявлении массива имя массива сопоставляется всей совокупности значений. Элементы массива размещаются в памяти подряд в соответствии с ростом индекса (номера элемента внутри массива). Размер выделяемой памяти такой, чтобы поместить значения всех элементов. Такие массивы называются *статическими*, место в памяти выделяется на этапе компиляции, именно поэтому размер массива определен константой.

Чтобы определить общий размер памяти, занимаемой массивом, можно использовать операцию sizeof(). Как известно, ее аргументом может быть имя объекта или имя типа, в том числе объекта сконструированного типа.

```
sizeof(имя_типа)          // в байтах для любого данного
                          // этого типа
sizeof(имя_переменной)    // в байтах для этой переменной
sizeof(имя_массива)       // в байтах для переменной
                          // с учетом длины
```

Операции над массивами

Над массивом как единой структурой никакие операции не определены. Над данными, входящими в массив, набор операций определен их типом. При работе с массивом можно обращаться только к отдельным его элементам. Операция обращения к одному элементу массива называется операцией *разыменования*. Это бинарная операция [], синтаксис которой:

Имя_массива[индекс]

Первый операнд «Имя_массива» показывает, что происходит обращение к данному, в составе которого много значений, т. е. ко всем элементам массива.

Второй операнд «индекс» дает возможность выделить один элемент из группы, и может быть только целочисленным. В общем случае, индекс — это выражение целого типа, определяющее номер элемента внутри массива (счет с нуля), например:

```
mas[0];      // обращение к первому элементу массива
             // (его номер равен 0)
matr[1][3];  // обращение к элементу матрицы, стоящему
             // на пересечении 1-ой строки и 3-го столбца
char c[i][2]; // обращение ко 2-му символу i-ой строки текста
```

Замечание. Контроль выхода индекса за границы массива не существует.

Пусть имеем массив int mas[4];

На рисунке 2.6 показано размещение элементов этого массива в памяти.

	Имя *mas* сопоставлено всей совокупности данных				
…	*mas*[0]	*mas*[1]	*mas*[2]	*mas*[3]	…
	Индекс = 0	Индекс = 1	Индекс = 2	Индекс = 3	

Рис. 2.6
Размещение в памяти элементов одномерного массива

Для каждого элемента массива выделено по 4 байта, как для данного типа int. В целом для всего массива выделено $4 \times 4 = 16$ байт. Значения элементов массива неизвестны.

Начальная инициализация элементов массива

При описании массива можно выполнить начальное присваивание значений его элементам, для этого нужно задать список инициализирующих значений. В списке инициализации перечисляются через запятую значения элементов массива.

Например, в году всегда 12 месяцев, известно значение числа дней в каждом месяце, значит, такая структура может быть задана массивом:

```
int month[12]={31,28,31,30,31,30,31,31,30,31,30,31};
```

Эквивалентом инициализации является простое присваивание вида:

```
month[0]=31;    // январь
    ...и т. д.
month[11]=31;  // декабрь
```

Механизм инициализации прост, но имеет некоторые особенности.

1. Если значение размера массива опущено, то число элементов массива равно числу значений в списке инициализации:

```
int month[]={31,28,31};  // количество элементов равно 3
```

Чтобы программно определить число элементов в таком массиве, используется операция sizeof:

```
int len;              // число элементов
len=sizeof(month)/sizeof(int);
```

2. Если число элементов в списке инициализации меньше, чем объявлено в описании, то число элементов массива равно количеству объявленных элементов, а остальные значения будут равны нулю:

```
int month[12]={31,28,31,30};
```

Количество элементов равно 12, их значения 31, 28, 31, 30, 0, 0,...

3. Если число элементов в списке инициализации больше, чем объявлено в описании, то это синтаксическая ошибка:

```
int month[2]={31,28,31,30};        // ошибка
```

Массивы переменной длины

Одним из недостатков статических массивов, что обусловлено реализацией, является то, что длина такого массива в тексте программы определена константным выражением, следовательно, жестко задана в программе. Часто бывает необходимо, чтобы число элементов массива изменялось при работе программы. Существуют приемы, которые используются, чтобы работать с массивом, длина которого может изменяться:

- #define определенные константы;
- массив условно переменной длины;
- динамические массивы.

Механизмы работы с динамической памятью изложены в п. 2.7.4 (глава 2).

Рассмотрим два первых приема. Механизм #define определенных констант заключается в изменении текста программы перед ее компиляцией, значит, препроцессор может редактировать код программы, внеся в него любое количество изменений. Длина массива записывается в директиве #define, например:

```
#define N 20
```

Имя N теперь константа, и в тексте программы для управления алгоритмами обработки массива следует использовать ее имя. Числовое значение константы (в этом примере 20) записывается в тексте один раз в самом начале, и перед очередным запуском программы может быть изменено один раз, остальные изменения выполнит препроцессор. После этого программа нуждается в повторной компиляции и сборке.

Пример 2.49. Выполним ввод данных массива и вывод на печать.

```
#define N  10
// статический массив. Длина массива равна 10
// значение константы записано один раз
void main(void)
{
   int a[N];
   int i;
   printf("\nВведите %d значений\n",N);
   for(i=0;i<N;i++)
      scanf("%d",&a[i]);
   for(i=0;i<N;i++)
      printf("%5d",a[i]);
   printf("\n");
}
```

В данном примере в массиве 20 значений, но если нужно больше или меньше, в коде изменяется только одна строка:

```
#define N 50
```

или

```
#define N 10
```

После этого код должен компилироваться заново.

Смысл второго приема заключается в том, что длина массива, даже если она изменяется, может быть

оценена заранее. Например, в студенческой группе не более 30 человек, в телефонной книге не более 1 млн записей и т. д. Если знать, какова возможная наибольшая длина массива, то именно это значение и следует выбрать для описания массива. Для того чтобы знать реальную длину массива, вводится специальная переменная, которая принимает значение при выполнении программы, а затем использует его для управления алгоритмами обработки массива. Оба приема можно совместить.

Пример 2.50.

```
#define N 20      // наибольшая длина массива равна 20
void main(void)
{
int a[N];         // память выделена для 20-ти элементов
int i;
int Len;          // фактическая длина массива меньше 20
printf("\nВведите длину массива < %d\n",N);
// ограничение длины
scanf("%d", &Len);
printf("\nВведите %d значений\n", Len);
// переменная Len управляет циклами
for(i=0;i<Len;i++)
   scanf("%d", &a[i]);
for(i=0;i<Len;i++)
   printf("%5d", a[i]);
printf ("\n");
}
```

Наибольшая длина массива равна N, фактическая длина равна Len, где Len < N. В пределах диапазона [0..N] длина массива может изменяться.

Алгоритмы работы с одномерными массивами

Алгоритмы работы с одномерными массивами основаны на использовании циклических алгоритмов, в которых выполняется последовательное обращение к элементам массива. Управляющей переменной в таких алгоритмах должен быть индекс массива.

Приведем фрагменты алгоритмов решения некоторых задач для массивов. Будем ориентироваться на абстрактное решение задачи для произвольного массива произвольной длины. Такую возможность дает только использование функций обработки массивов. В функцию следует передать массив (передается адрес) и длину массива. Если функция не изменяет длину массива, она передается по значению, если изменяет, то по адресу. Объявление фактических массивов, т. е. тех, с которыми фактически будет работать функция, и присваивание значений их элементам происходит в вызывающей программе.

Приведем программный код задач, алгоритмы которых даны в примерах 1.7–1.14 (глава 1). Решение каждой задачи оформлено в виде функции.

Пример 2.51. Сформировать новый одномерный массив из положительных элементов заданного массива.

```
int New_Arr(int x[],int n,int y[],int &k)
{
  k=0;
  for(int i=0;i<n;i++)
    if(x[i]>0)
    {
      y[k]=x[i];
      k++;
    }
  return k;        // равенство нулю значения функции
                   // свидетельствует о неуспехе
}
```

Пример 2.52. Вычислить сумму элементов одномерного массива.

```
int Sum(int a[],int Len)
{
  int S=0;
  for(int i=0;i<Len;i++)
    S+=a[i];
  return S;
}
```

Пример 2.53. Найти количество положительных и отрицательных чисел в данном массиве.

```
void Counts(int a[],int Len, int &K1,int &K2)
{
  K1=K2=0;
  for(int i=0;i<Len;i++)
  {
    if(a[i]>0)
    K1++;
  else
    if(a[i]<0)
  K2++;
  }
}
```

Пример 2.54. Дан массив произвольной длины. Найти наибольший элемент массива и определить его номер.

Практически, если известен номер элемента, то известно и его значение. Поэтому в этом коде отыскивается не «значение и номер», а только номер.

```
int N_max(int a[],int Len)
{
  int K=0;          // номер наибольшего элемента
  for(int i=1;i<Len;i++)
    if(a[i]>a[K])
      K=i;
  return K;
}
```

Пример 2.55. Удалить из массива, в котором все элементы различны, наибольший элемент.

```
void Del_max(int a[],int &Len)
{
  // номер наибольшего элемента находит функция N_max,
  // к ней нужно обратиться
  int k;
  k=N_max(a,Len);
  // сдвиг от k-го элемента до конца массива
  for(int i=k;i<Len-1;i++)
    a[i]=a[i+1];
  Len--;            // параметр передается по адресу,
                    // длина массива изменится
}
```

Пример 2.56. Вставить произвольное число **m** в одномерный массив $A[n]$ после элемента с заданным номером k.

```
void Insert(int a[],int &Len,int k,int m)
{
  // сдвиг от конца массива до k+1-го элемента
  for(int i=Len-1;i>=k;i--)
    a[i+1]=a[i];
  // место свободно
  a[k]=m;
  Len++;          // параметр передается по адресу, длина
массива изменится
}
```

Пример 2.57. Определить, является ли заданная последовательность различных чисел a_1, a_2, ..., a_n монотонно убывающей.

```
int Decrease(int a[],int Len)
{
  int flag=1;       // flag=1 – последовательность убывает
                    // flag=0 – последовательность не убывает
  int i;
  i=1;
  while(i<Len&&flag==1)
    if(a[i]>a[i+1])
      i++;
    else
      flag=0;
  return flag;
}
```

Пример 2.58. Определить, есть ли в одномерном массиве хотя бы один отрицательный элемент.

```
int Negative(int a[],int Len)
{
  int flag=0;
  int i;
  i=0;
  while(i<Len&&flag==0)
    if(a[i]<0)
      flag=1;
```

```
    else
        i++;
  return flag;
}
```

2.7.2. Многомерные массивы

В синтаксисе С++ определены только одномерные массивы. Массивы большей размерности трактуются как массивы массивов. Все особенности использования статических многомерных массивов можно рассмотреть на примере двумерных массивов (матриц).

Описание двумерного массива

Описание двумерного массива требует, чтобы были указаны два размера: число строк и число столбцов матрицы, например:

```
int a[2][5];        // массив из 2-х одномерных массивов,
                    // по 5 элементов в каждом
```

В памяти элементы двумерного массива размещаются построчно подряд по возрастанию индексов. Индексы, как и раньше, нумеруются с 0.

Пример инициализации двумерных массивов:

```
int a[3][4]={
  {1,2,3,4},        // значения элементов нулевой строки
  {2,4,6,8},        // значения элементов первой строки
  {9,8,7,6}         // значения элементов второй строки
  };
```

При размещении в памяти будет выделено место под запись $3 \times 4 = 12$ элементов, и они будут размещены линейно, как на рисунке 2.7, в следующем порядке.

1	2	3	4	2	4	6	8	9	8	7	6
Нулевая строка				Первая строка				Вторая строка			

Рис. 2.7
Размещение элементов двумерного массива

Операции над двумерным массивом не разрешены, кроме операции доступа к его элементам (разыменования). Обращение к произвольному элементу массива выполняется по индексу: a[i][j] — адресуется элемент

массива, стоящий на пересечении *i*-го столбца и *j*-й строки.

В п. 1.4 (глава 1) приведены блок-схемы алгоритмов, использующие двумерные массива.

Алгоритмы работы с двумерными массивами

Алгоритмы работы с двумерными массивами похожи во многом на алгоритмы работы с одномерными массивами. В большинстве случаев требуется обращение ко всем элементам матрицы, тогда алгоритм просмотра содержит вложенный цикл, у которого во внешнем цикле управляющей переменной служит номер строки матрицы, а во внутреннем — номер столбца. Тогда просмотр элементов происходит построчно.

Пусть есть описание матрицы:

```
#define N 5
#define  M 7
int a[N][M];
```

Здесь размер матрицы определяют define-константы. Для матрицы условно переменной длины следует описать размер по максимуму и ввести переменные, реально определяющие размер матрицы.

```
int a[25][25];
// максимальное число строк, столбцов
int n, m;                    // реальное число строк, столбцов
printf("Введите размер матрицы (строк, столбцов <25)\n");
scanf("%d%d",&n,&m);  // теперь это наибольшие значения
                             // параметров циклов управления
```

Для обращения к элементам матрицы по строкам цикл записывается так:

```
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
  {
    // в теле цикла обращение к переменной a[i][j]
  }
```

Если переменить эти циклы местами, просмотр будет происходить по столбцам:

```
for(j=0;j<m;j++)
  for(i=0;i<n; ++)
```

```
{
  // в теле цикла обращение к переменной a[i][j]
}
```

Использование функций при работе с двумерными массивами

Использование функций при обработке матриц предполагает два кардинально различных подхода. В первом случае матрица существует или рассматривается как самостоятельная структура данных, к которой необходимо применить какой-либо алгоритм обработки. В качестве параметра такой функции будет использована вся матрица целиком. Для решения задачи обработки в общем виде, в функцию следует передать имя матрицы и ее размеры (как параметры функции), при этом функция получает матрицу как двумерный массив. Компилятор C++ должен знать, каков способ разбиения этой структуры на строки, поэтому число строк матрицы можно опустить, а число данных в каждой строке опускать нельзя. Оно должно быть указано обязательно как константное выражение в квадратных скобках при передаче матрицы. Так, прототип функции, получающей матрицу в качестве входного данного, может выглядеть так:

```
int function(int a[][M],int n,int m);
// здесь M – константное выражение
```

В качестве примера рассмотрим функции для ввода и вывода матрицы на экран. Предполагается, что матрица условно переменного размера, поэтому число строк и столбцов матрицы по описанию определено define-константами N = 5 и M = 5. Реальный размер матрицы определяют переменные n, m, значения которых вводятся.

Блок-схемы алгоритмов ввода и вывода элементов матрицы приведены в п. 1.4.1 (глава 1).

```
#include <stdio.h>
#define N 5
#define M 5
void input_matr(int a[][M],int &n,int &m);
```

```
// важно, что длина строки – это константа
void print_matr(int a[][M],int n,int m);
// n – число строк, m – число столбцов
void main(void)
{
   // главная программа описывает входные данные
   int n, m;                  // реальные размеры матрицы
   int matr[N][M];            // описан двумерный массив
   input_matr(matr,n,m); // передан в функцию ввода
   print_matr(matr,n,m);  // передан в функцию вывода
}
// описание функции ввода
// параметры функции – имя и размеры массива
void input_matr(int a[][M],int &n,int &m)
// n, m возвращаются по ссылке
{
   int i, j;
   printf("Введите размер матрицы не более %d на %d\n", N,
M);
   scanf("%d%d", &n,&m);
   printf("Введите матрицу.\n");
   for(i=0;i<n;i++)
      for(j=0;j<m;j++)
         scanf("%d", &matr[i][j]);
}
// описание функции вывода
// параметры функции – имя и размеры массива
void print_matr(int a[][M],int n,int m)
{
   int i, j;
   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
         printf("%5d",mas[i][j]);
         printf("\n");          // разбиение вывода на строки
   }
}
```

Во втором случае матрица рассматривается как массив из одномерных массивов, при этом для обработки от-

дельных строк матрицы можно использовать функции обработки одномерных массивов. Зная, что элементы матрицы — это одномерные массивы, каждый из них можно по очереди передавать в функцию, которая работает с одномерным массивом.

Приведем код программ, реализующих примеры 1.15–1.19 (глава 1). Алгоритм решения каждой задачи реализован с помощью функции.

Пример 2.59. Вычислить сумму элементов строк заданного двумерного массива.

```
void Sum_Rows(int a[][N],int n,int m,int S[])
{
  int i,j;
  for(i=0;i<n;i++)
  {
    S[i]=0;
    for(j=0;j<m;j++)
      S[i]=S[i]+a[i][j];
  }
}
```

Пример 2.60. Найти наибольший элемент двумерного массива и его индексы.

```
int Maxx(int a[][N],int n,int m,int &Maxi,int &Maxj)
{
  int Max=a[0][0];
  int i,j;
  Maxi=Maxj=0;
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      if(a[i][j]>Max)
      {
        Max=a[i][j];
        Maxi=i;
        Maxj=j;
          }
    return Max;
}
```

Пример 2.61. Вставить в двумерный массив строку, состоящую из нулей, после строки с номером k.

```
int Insert_Row(int a[][N],int &n,int m,int k)
{
   int i,j;
   if(k>m)
      return 0;                  // если неверно задано значение k
   for(i=n-1;i>=k;i--)           // раздвигаем строки
      for(j=0;j<m;j++)
         a[i+1][j]=a[i][j];
   n++;
   // заполняем нулями освободившуюся строку
   for(j=0;j<m;j++)
      a[k][j]=0;
   return 1;
}
```

Пример 2.62. Удалить из двумерного массива строку с номером k.

```
int Del_Row(int a[][N],int &n,int m,int k)
{
   int i,j;
   if(k>m)
      return 0;      // если неверно задано значение k
   // сдвигаем строки
   for(i=k;i<n-1;i++)
      for(j=0;j<m;j++)
         a[i][j]=a[i+1][j];
   n--;
   return 1;
}
```

Пример 2.63. Дана матрица $A[n, m]$. Найти количество строк, содержащих хотя бы один нуль.

```
int Cou_Rows(int a[][N],int n,int m)
{
   int i,j;
   int k=0;
   int flag;
   for(i=0;i<n;i++)
   {
      j=0;
      flag=0;
```

```
    while(j<m && flag==0)
    {
      if(a[i][j]==0)
      {
        flag=1;
        k++;
      }
      else j++;
      }
  }
  return k;
}
```

2.7.3. Указатели

Указатели — такие объекты (переменные), значением которых являются адреса других объектов (или какой-либо области памяти).

Кроме того, значением указателя может быть пустое значение, не равное никакому адресу. В некоторых библиотеках, например, <stdlib.h>, эта константа определена значением NULL.

Объявление указателей

Описание переменных типа указатель обязательно должно показывать тип переменной, которую адресует указатель. Сама переменная типа указатель имеет размер 2 или 4 байта в зависимости от модели памяти.

Признаком того, что объявляется указатель, является символ «*».

```
int *ip;      // указатель на ячейку, содержащую целое число
float *fp;
// указатель на ячейку, содержащую действительное число
char *cp;  // указатель на ячейку, содержащую символ
void *pp;  // указатель на ячейку неизвестного типа
```

Пример 2.64.

Пусть объявлены указатели:

```
char *pa;
int *pb;
double *pc;
```

Тогда размещение переменных и указателей в памяти будет таким, как на рисунке 2.8.

	pa	*pb*	*pc*	
	адрес char 4 байта значение DS:FF12 адресует 1 байт	адрес int 4 байта значение DS:FF08 адресует 4 байта	адрес double 4 байта значение DS:FF00 адресует 8 байт	

Рис. 2.8
Размещение переменных и указателей в памяти

Определение операций над указателями. Операция получения адреса &

Это унарная операция, которая может быть применена к операнду любого типа. Возвращает 16-ричный адрес объекта, определяющий его размещение в памяти.

Пример 2.65.

```
int x=3;  // выделены 2 байта по какому-то адресу
          // и присвоено значение
int *px;  // выделены 2 байта для хранения адреса типа int
```

Операция &x получит адрес переменной x, который можно присвоить переменной, предназначенной для хранения адреса, это px.

```
px = &x; // адрес можно присвоить указателю
```

Операция применяется только к именованным объектам, размещенным в памяти. Бессмысленно ее применение к константам, выражениям, внешним объектам.

Операция получения значения по указанному адресу *

Синонимы — операция разыменования, раскрытия ссылки или обращения по адресу. Это унарная операция, операндом может быть только указатель. Возвращаемое значение имеет тип той переменной, на которую показывает указатель, и возвращает значение, размещенное в той области памяти, на которую ссылается указатель.

Пример 2.66.

```
int x=3;  // выделены 2 байта по какому-то адресу
          // и присвоено значение
int *px;  // выделены 2 байта для хранения адреса типа int
          // операция &x получит адрес переменной x
px = &x; // адрес переменной x можно присвоить указателю
```

```
int new_x; // новая переменная new_x
new_x = *px;
```

Переменная new_x получила значение той переменной, которая хранится по адресу, имеющему значение px, следовательно px знает адрес x. Поэтому переменная new_x получит значение, равное 3.

Тип данного void может адресовать объект любого типа, но к нему нельзя применить операцию «*».

Примеры недопустимого использования типа void.

```
int x=3;
float y=2.5;
void *p_k;          // указатель на void
void *p_y;          // указатель на void
p_k = &k;
p_y = &y;
```

Операция присваивания для указателей

Операция присваивания в С++ имеет особенности, связанные с преобразованием типов.

При присваивании указателей возможные ошибки несоответствия типов могут вызвать не только потерю данных, но и привести к трагическим последствиям, точно так же, как если адрес перепутал почтальон. Тем не менее, преобразование типов указателей может быть выполнено так же, как и для простых типов, но с использованием операции «*», например:

```
int *x;
float *y;
y=(int *)x;
x=(float *)y;
```

Механизмы, связанные с таким преобразованием, очень тонкие. Попытаемся показать это на примере.

Пример 2.67.

Объявим переменную L типа unsigned long и запишем в нее 16-ричную long константу 0x12345678L.

Объявим три указателя (на char, int и long), и каждому из них присвоим адрес этой константы (&L).

При присваивании будем выполнять приведение типов.

```
unsigned long    L=0x12345678L;          // длинное целое
  char *c=(char *)&L;
  int *i=(int *)&L;
  long *l=(long *)&L;
  printf("\nЗначение L=%#lx",L);
  printf "\nАдрес L (то, что &L)=%p",&L);
  printf("\nАдрес c=%p *c=%#x",c,*c);
  printf("\nАдрес i=%p *i=%#x",i,*i);
  printf("\nАдрес l=%p *l=%#lx",l,*l);
```

Вывод будет таким:

```
Значение L = 0x12345678
Адрес L (то, что &L)=0016FBC8
Адрес c=0016FBC8 *c=0x78
Адрес i=0016FBC8 *i=0x5678
Адрес l=0016FBC8 *l=0x12345678
```

Здесь отчетливо видно, что адреса всех объектов c, i, l одинаковы, и это означает, что все они адресуют одну и ту же область, выделенную переменной L. Однако же значения, извлекаемые из одной и той же области различны, и зависят от типа указателя.

Унарные операции ++ и – –

Унарные операции ++ и – – изменяют значение адреса в зависимости от типа данных, с которым связан указатель, а именно:

- для char на 1 байт;
- для int на 4 байта;
- для double на 8 байт.

Эта операция называется смещением указателя. В следующем примере показано, что выделение адресов в сегменте данных происходит не справа налево, а от старших байт вниз.

Пример 2.68.

```
int a=0;
int b=20;
int c=30;
int *pa=&a;
int *pb=&b;
int *pc=&c;
```

```
printf("\nАдрес %p Значение %d", a,*pa);
// FFF4, значение 10
printf("\nАдрес %p Значение %d", pb,*pb);
// FFF2, значение 20
printf "\nАдрес %p Значение %d", pc, pc);
// FFF0, значение 30
// операция смещения указателя приведет к просмотру
объектов
// a, b, c, если использовать pa –– или pc++
for(int k=1; <=3;k++)
   printf("\nАдрес %p Значение %d",pa ––, *pa);
// выводится то же, что и в трех предыдущих операторах
вывода
```

Сложение и вычитание

Складывать адреса бессмысленно, но можно прибавить к адресу целое число, например, pa+число. Будет получено новое значение адреса, смещенное относительно старого указателя в зависимости от типа указателя, на число*sizeof(тип_указателя) в байтах.

Вычитание адресов можно применить к указателю и числу или к двум указателям. В первом случае будет получено новое значение адреса, смещенное относительно старого указателя, как и при сложении. Во втором случае разность (со знаком), это расстояние в единицах, кратных размеру одного объекта указанного типа. Например, смещение pb на +1 — это адрес a, смещение pb на 1 — это адрес c.

Пример 2.69.

```
printf("\nАдрес %p Значение %d",pb+1,*(pb+1));
// значение 10
printf("\nАдрес %p Значение %d",pb–1,*(pb–1));
// значение 30
printf("\nРасстояние между c и a=%d",pc–pa);
// расстояние = 1
```

Операции отношения

Все операции отношения могут использоваться для сравнения значений адресов. Разрешается сравнивать только указатели одинакового типа или указатель сравнить

с константой NULL. Многие функции стандартных библиотек C++ возвращают указатели. Если функция не может вернуть адрес, то возвращаемым значением будет NULL.

Приоритеты операций, разрешенных над указателями:

- унарные операции косвенной адресации «*» и получения адреса «&»;
- аддитивные операции;
- операции сравнения;
- операция присваивания.

Указатели и массивы

Синтаксис языка C++ определяет имя массива как адрес его первого элемента (с нулевым значением индекса). Все следующие элементы располагаются подряд в порядке возрастания адресов. Обращение к элементам массива может быть выполнено с помощью операции [ ] (прямая адресация) или с помощью операции «*» (косвенная адресация).

В первом случае для нумерации элементов массива используются целочисленные индексы, нумерующие элементы массива.

Во втором случае для определения номера элемента внутри массива используется смещение указателя от первого элемента.

Например, если объявлен массив:

```
int mas[]={10,20,30,40};
```

то mas — это адрес первого элемента массива (адрес mas[0]).

Адрес следующего элемента mas[1] или mas+1, следующего mas[2] или mas+2, и т. д.

Косвенная адресация в массивах

Пусть объявлен массив

```
int mas[]={10,20,30,40};
```

Пусть его длина определена как

```
int n=sizeof(mas)/sizeof(int);
```

Для прямой адресации в массиве используется рабочая переменная, которая определяет целочисленное значение индекса элемента массива:

```
int i;
```

```
for(i=0;i<n;i++)
  // обращение к mas[i]
```

Для косвенной адресации в массиве используется рабочая переменная, которая определяет значение адреса элемента массива:

```
int *pti;  // тип этого указателя, как у элементов массива
           // (синоним массива)
for(pti=mas; pti<mas+n;pti++)
// обращение к *pti
```

Обращение к элементам массива при изменении индекса и обращение к элементам массива при смещении указателя показано на рисунке 2.9.

…	10	20	30	40
	Выделение памяти для указателя *pti*			
	4 байта *mas*[0] при i = 0 **pti* при *pti* = *mas*	4 байта *mas*[1] при *i* = 1 *(*pti* + 1) при *pti* = *mas* + 1	4 байта *mas*[2] при *i* = 2 *(*pti* + 2) при *pti* = *mas* + 2	4 байта *mas*[3] при *i* = 3 *(*pti* + 3) при *pti* = *mas* + 3

Рис. 2.9
Прямая и косвенная адресация в массивах

```
int i;
int *pti;
pti=mas;
```

После присваивания pti=mas, указателю доступен адрес массива:

```
printf("\n Первый элемент массива=%d", *pti;)
printf("\n Второй элемент массива=%d", *++pti);
// *(pti+1)
printf("\n Третий элемент массива=%d", *++pti);
// *(pti+2)
printf("\n Последний элемент массива=%d", *(pti+3));
```

Приведем пример различной адресации в массиве при вычислении суммы элементов массива. Это обычный алгоритм обработки массива. Для решения задачи необходим последовательный просмотр всех элементов массива.

Пример 2.70.

```
#define N 5
void main(void)
{
```

```
int mas[N]={1,2,3,4,5};
int Sum;
int i;
// прямая адресация
// обращение к элементам массива через операцию []
for(i=0,Sum=0;i<N;i++)
   Sum+=mas[i];
printf("\nСумма=%d", Sum);
// косвенная адресация
// обращение к элементам массива через операцию *
int *pti;
for(pti=mas,Sum=0;pti<mas+N;pti++)
   Sum+=*pti;
printf("\nСумма=%d", Sum);
// косвенная адресация
// обращение к элементам массива смещением указателя
// относительно начала массива
for(pti=mas,i=0,Sum=0;i<N;i++)
   Sum+=*(mas+i);
printf("\nСумма=%d", Sum);
// напечатаем номера и адреса элементов этого массива
// спецификатор формата для вывода адресов %p
int k;
for(pti=mas,k=0;pti<mas+N;pti++)
   printf("\nНомер=%d, Адрес=%p",k++,pti);
}
```

2.7.4. Динамические массивы (массивы динамической памяти)

При статическом распределении памяти для элементов массива их общее количество должно быть известно при компиляции, когда происходит распределение памяти. Часто это значение неизвестно заранее, и его необходимо определить в процессе выполнения программы.

Динамическое выделение памяти в языке C++

Для динамического выделения памяти в C++ введены специальные операции выделения динамической памяти new и высвобождения delete. Операция new выделяет

в куче (*heap*) память требуемого размера для объекта, операция delete разрушает объект, возвращая память в кучу.

Синтаксис.

Указатель=new Имя_объекта[количество_элементов];

Операция new пытается открыть объект с именем «Имя_объекта» путем выделения sizeof(Имя_объекта) байт в куче.

Синтаксис.

delete имя_объекта;

Операция delete удаляет объект с указанным именем, возвращая столько же байт в кучу.

Объект существует с момента создания до момента разрушения или до конца программы.

Память для размещения статических объектов в С++ распределяется во время компиляции, для динамических — во время выполнения программы. Этот механизм требует пояснений.

При выполнении приложения операционная система выделяет процессу виртуальное адресное пространство, которым процесс владеет единолично. Данные и коды их обработки в процессе работы приложения должны находиться в оперативной памяти. Память, выделенная процессу, распределяется так, как показано на рисунке 2.10.

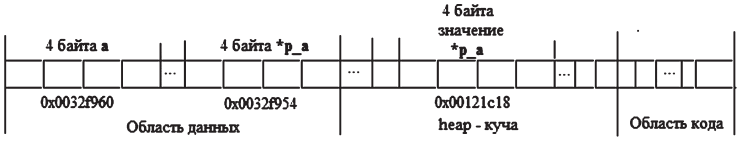


Рис. 2.10
Распределение памяти в «куче»

Данные размещаются в младших адресах, коды — в старших. Между ними остается свободная область адресов, называемая кучей (*heap*). Управление памятью в куче выполняет специальная утилита операционной системы, называемая *диспетчером кучи*. Выделение и высвобождение динамической памяти в неуправляемых приложениях С++ не контролируется операционной системой, и на-

ходится целиком в ведении программиста. Динамическая память может быть выделена для простого объекта, что не имеет смысла, или для сложного объекта, занимающего много места в памяти, например, массива.

Покажем механизмы выделения памяти для простого объекта.

Пусть в коде программы объявлена переменная int, и указатель на int. При объявлении память будет выделена статически, в отладчике можно узнать адреса. Например, память распределена так, как в примере:

```
int a;          // 4 байта в статической памяти,
                // адрес  &a=0x0032f960
int *p_a;       // 4 байта в статической памяти,
                // адрес &p_a=0x0032f954
```

Только после выполнения операции

```
*p_a=new int;
// 4 байта в динамической памяти, адрес 0x00121c18
```

объекту p_a выделена память. Ее адрес становится значением переменной p_a:

```
p_a = 0x00121c18
```

По значению адреса видно, что он находится в другой области памяти. Собственного имени эта область не имеет, обращение к ней выполняется через синоним имени:

```
*p_a=99;
p_a=&a;// a=99
```

При высвобождении адреса операцией delete:

```
delete p_a;
```

указатель p_a, оставаясь на прежнем месте, перестает быть связанным с областью памяти, адресованной 0x00121c18, и его можно использовать для других целей.

Пример 2.71. Приведем пример использования операций new и delete.

```
name *ptr;              // здесь name любой тип, кроме функции
if(!(ptr=new name))  // может оказаться равным NULL
{
  printf("Нет места в памяти.\n");
  exit();
}
                        // использование объекта по назначению
delete ptr;             // разрушение объекта
```

Пример 2.72. Приведем пример создания динамического одномерного массива.

```
int *mas;                    // указатель на массив
printf("Введите количество элементов в массиве");
scanf("%d",&N);
mas=new int[N];              // выделение памяти для массива
if(mas==NULL)
{
  printf("\nНет памяти");
  return;
}
// массив существует, адресация массива
// выполняется через имя указателя
delete mas[];                // массив разрушен
```

Примечание. При выделении памяти переменной (массиву) через указатель, переменная (массив) не имеет собственного имени. Обращение к переменной (массиву) выполняется только посредством указателя, который играет роль синонима имени.

Объект динамической памяти разрушается при использовании операции delete или при завершении работы программы, если не был разрушен ранее.

Динамические двумерные массивы

Динамические двумерные массивы объявляются как указатель на указатель. Память выделяется для каждой строки отдельно, так и высвобождается тоже для каждой строки. Адресация не зависит от способа выделения памяти и может быть как прямой, так и косвенной. Размещение в памяти для каждого массива линейное, примерная схема выделения памяти показана на рисунке 2.11.

Пример 2.73. Приведем фрагмент кода программы, которая выделяет память для динамического двумерного массива.

```
int **a;  // адрес динамической памяти
int i,j;
printf("Введите количество строк и столбцов\n");
scanf("%d%d",&n,&m);
// первый этап: выделение памяти для массива указателей
```

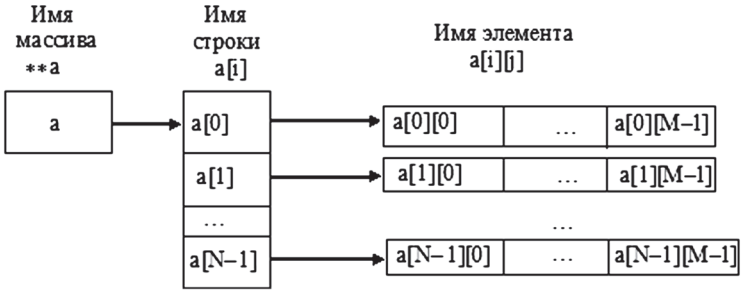


Рис. 2.11
Схема выделения памяти для динамического двумерного массива

```
// для строк матрицы нужно n элементов размером int
a=new int *[n];
if(a==NULL)
{
   printf("\n Нет памяти\n");
   return NULL;                  // Выход из функции
}
// второй этап: выделение памяти для каждой строки
// матрицы
// выполняется n раз, выделяется по m элементов
for(i=0;i<n;i++)
{
   a[i]=new int[m];
   if(a[i]==NULL)
   {
      printf("\n Нет памяти");
      return NULL;                        // выход из функции
}
}
```

Пример 2.74. Приведем фрагмент кода в качестве примера работы с элементами массива.

```
printf("Введите матрицу размером %d строк и %d
столбцов\n", n, m);
   for(i=0;i<n;i++)
      for(j=0; <m; ++)
scanf("%d", &a[i][j]);
```

```
printf("\nРезультат");
for(i=0;i<n;i++)
{
  for(j=0;j<m;j++)
    printf("%d ",a[i][j]);
printf("\n");
}
```

Высвобождение памяти, занятой динамическим двумерным массивом, осуществляется тоже в два этапа: сначала высвобождается память, занятая строками, затем высвобождается двойной указатель на массив.

```
for(i=0; i<n; i++)
  delete a[i];     // разрушаются строки
  delete a;        // высвобождается указатель на матрицу
```

Массивы указателей

Массивы указателей могут использоваться для хранения адресов объектов и управления ими. Эта возможность применяется для объектов большого размера или для динамических объектов.

Синтаксис определения массива указателей:

```
Тип_массива * Имя_массива[количество_элементов];
```

Определение с одновременной инициализацией:

```
Тип_массива * Имя_массива[] ={инициализатор};
Тип_массива * Имя_массива[количество_элементов]=
{инициализатор};
```

Здесь «Тип_массива» может быть базовым или производным, а «инициализатор» – это список значений адресов.

Например:

```
int Data[6];        // массив целочисленных значений
int *pD[6];         // массив указателей
int *pI[]={&Data[0],&Data[1],&Data[2]};   // с инициализацией
```

Здесь каждый элемент массивов pD и pI является указателем на объекты типа int, следовательно, значением каждого элемента pD[i] и pI[i] может быть адрес объекта типа int. Элементы массива pD не инициализированы, значит, имеют неопределенные адреса. В массиве pI три элемента, значения которых — это адреса конкретных элементов массива Data.

В качестве простого примера рассмотрим программу сортировки по возрастанию одномерного массива без перестановки его элементов. Здесь элементы исходного массива не меняются местами, т. е. остаются в прежнем порядке, а сортируется массив указателей на элементы массива *pA. С его помощью определяется последовательность просмотра элементов массива A[N] в порядке возрастания их значений.

Рисунок 2.12 иллюстрирует исходную и результирующую адресации элементов массива A элементами массива указателей *pA.

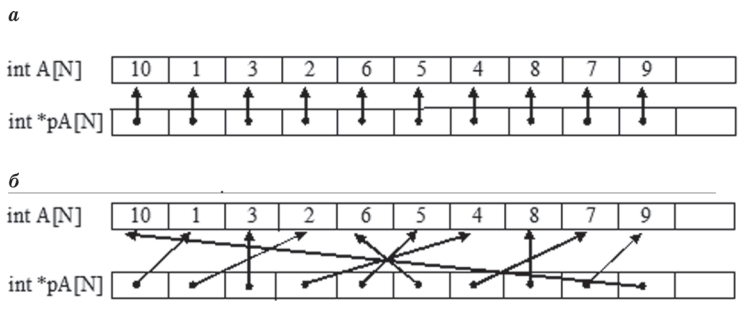


Рис. 2.12
Массивы в задаче сортировки:
а — массивы до сортировки; *б* — массивы после сортировки.

Приведем код примера.

Пример 2.75.

```
#define N 10
void main(void)
{
   int A[N]={10,1,3,2,6,5,4,8,7,9};
   int *pA[N];
   int *buf;          // буфер для перемены адресов
   int i,j;
   for(i=0;i<N;i++)
      pA[i]=&A[i]; // копирование адресов (см. рис. 2.12а)
   for(i=0;i<N-1;i++)
      for(j=i+1;j<N;j++)
      {
```

```
        if(*pA[i]>*pA[j])
        {
        // перестановка в массиве адресов (см. рис. 2.12б)
           buf=pA[i];
           pA[i]=pA[j] ;
           pA[j]=buf;
        }
   }
   printf("\n Массив, отсортированный по возрастанию:\n");
   for(i=0;i<N;i++)
      printf("%d ",*pA[i]); // вывод по возрастанию
   printf("\n Массив в исходном порядке:\n");
for(i=0;i<N;i++)
   printf("%d", A[i]);        // вывод в исходном порядке
}
```

2.7.5. Указатели и функции

Механизмы указателей при работе с функциями используются для решения многих практических задач.

Передача параметров по адресу

Ранее рассматривался механизм передачи параметров в функцию по значению (см. п. 2.5). При обращении к функции и передаче ей параметров создается локальная копия объекта в теле функции. Значение объекта вызывающей программы, передаваемое в функцию как фактический параметр, не может быть изменено в теле функции.

Чтобы определить функцию, возвращающую более одного значения, нужно дать ей возможность изменить значения параметров. В качестве параметра следует передавать не значение объекта вызывающей программы, а его адрес. При этом используется механизм указателей. Адресная операция (&) применяется к имени параметра. Синтаксис этого описания:

Тип_функции Имя_функции(Тип_параметра &Имя_параметра)

Механизм этого способа передачи данных заключается в том, что функция и вызывающая программа работают с адресом объекта в памяти, следовательно, обе могут

изменить значение этого данного. При этом и в вызывающей программе и в функции этот параметр должен быть именованным объектом.

В классическом языке С этот механизм отсутствует. В С++ признак передачи параметра по ссылке указывается в списке формальных параметров при описании функции и в прототипе.

```
// передача параметра по адресу в стиле С++
void change_5(int &ptr)
// формальный параметр – адрес объекта
{
    ptr+=5;
  }
  void main(void)
  {
    int x=5;
    printf("x=%d\n",x);
    change_5(x);
    printf("А теперь x=%d\n",x);
}
```

Массивы как параметры функций

Особенность массива в том, что он является указателем. Имя массива — это адрес его нулевого байта. Когда параметром функции является массив, то, согласно синтаксису С++, в функцию передается его адрес, следовательно, функция будет изменять элементы массива, так как включается механизм передачи данных по адресу.

Способов записи формальных параметров массивов в заголовке функции может быть два:

```
тип_функции имя_функции(тип_массива Имя[])
тип_функции имя_функции(тип_массива *Имя)
```

Как видим, длина массива отсутствует в описании формального параметра. Конструкции «Имя[]" и "*Имя» различны внешне, но одинаковы по выполняемым механизмам, если «Имя» — это имя массива. Длина массива, как правило, определена его описанием или передается в функцию как параметр.

В теле функции элементы массива адресуются любым способом независимо от синтаксиса формального параметра. При обращении к функции указывается только имя массива.

Несомненным достоинством этого механизма является возможность простой и надежной реализации алгоритмов обработки массивов переменной длины. При этом задачи обработки массивов решаются в общем виде. В качестве параметров в функцию следует передать имя массива и его длину. Если содержимое массива будет изменено функцией, это изменение доступно и вызывающей программе. Если функция не изменяет длину массива, ее следует передать как параметр-значение, если изменяет, то как параметр-адрес, например:

```
int function(int a[],int n);
// a – имя массива, n – длина массива
int function (int a[],int &n);
// a – имя массива, n – длина массива
```

Независимо от способа передачи, фактическим параметром при обращении к функции должно быть имя массива.

Для двумерных массивов задача не может быть решена так просто, потому что двумерный массив при передаче в функцию разбивается на строки. Функция должна знать способ разбиения на строки, поэтому число строк данных можно опустить, а число данных в каждой строке опускать нельзя. Оно должно быть указано обязательно как константное выражение в квадратных скобках при передаче матрицы. Так, прототип функции, получающей матрицу в качестве входного данного, может выглядеть так:

```
int function(int a[][M],int n,int m);
// здесь M – константное выражение
```

Примером использования массивов в качестве параметров функций могут служить функции ввода и вывода одномерных массивов. Эти функции решают алгоритмические задачи в общем виде.

Пример 2.76.

```
#define N 40
```

```
// функция вывода массива произвольной длины
void print_mas(int mas[],int len)   // параметры – имя и длина
{
   int i;
   printf("Массив:\n");
   for(i=0;i<len; ++)
      printf("%5d ", mas[i]);
   printf("\n");
}
// функция ввода массива произвольной длины
void input_mas(int *mas,int &len)// параметры – имя и длина
// длина передается по ссылке
{
   int *ip;
   printf("Введите количество элементов массива <
%d\n",N);
   // N – наибольшая длина
   scanf("%d",&len);
   printf("Введите элементы массива\n");
   for(ip=mas;ip<mas+len;ip++)
      scanf("%5d",ip);                    // признак & не нужен
}
// функция преобразования массива
void transform_mas(int *mas,int len)
// параметры – имя и длина
{
   int i;
   for(i=0;i<len;i++)
   mas[i]=mas[i]*2;
}
```

В вызывающей программе массив должен быть объявлен, может получить значения. В обращении к функции, чтобы передать массив, нужно указать только его имя:

```
void main(void)
{
   int a1[N],a2[N];          // объявлены два массива
   int n1,n2;                // их длины
   // каждый массив введен, преобразован, распечатан
   input_mas(a1,n1);
```

```
   transform_mas(a1,n1);
   print_mas(a1,n1);
   input_mas(a2, 2);
   transform_mas(a2,n2);
   print_mas(a2,n2);
}
```

Следует обратить внимание на то, что главная программа теперь занимается только управлением и передачей данных. Логика ее работы прозрачна, текст не перегружен подробностями алгоритмов.

Функции, возвращающие адреса объектов

Такие функции используются при работе с динамическими данными, т. е. массивами и строками. Тип функции (возвращаемого ею значения) может быть одним из базовых типов или указателем. Во втором случае возвращаемое значение — это адрес какого-либо объекта в памяти. Сам объект может иметь произвольный тип и может быть объектом базового, но чаще конструируемого типов. В случае динамических данных функция может породить новый объект и вернуть его адрес.

Возвращаемое функцией значение должно быть присвоено переменной типа указатель.

Такими функциями являются многие библиотечные функции, которые требуют выделения памяти под вновь созданный объект, и, как правило, возвращают вновь созданное значение. Например, многие функции работы со строками библиотеки <string.h>, многие функции ввода данных библиотек <stdio.h> и <stdlib.h> и др.

Пример 2.77. Приведем пример функции, возвращающей указатель.

```
int *func(int n)
{
   int *a;          // объявлен указатель
   *a=n;            // его значение равно значению параметра n
   return a;        // возвращается адрес
}
void main(void)
{
```

```
   int b;
   b=*func(12);  // простая переменная b получает значение
                 // возвращенного функцией адреса
   printf("%d",b);
   int *d;
   d=func(12);   // указатель d получает значение адреса,
                 // который вернула функция
   printf("%d",*d);
}
```

Более полезным будет пример, в котором функция возвращает массив динамической памяти. Количество данных, для которых нужно выделить память и присвоить значения, является внешним параметром функции, и передается из вызывающей программы.

```
int *new_mas(int n)
{
   int *mas;
// область оперативной памяти, связанная с указателем
   int *ip;
   // выделена память: mas – это реальный адрес массива
   mas=new int[n];
   printf("Введите %d элементов.", n);
   for(ip=mas; ip<mas+n; ip++)
      scanf("%d", ip);
   return mas;    // вернуть объект
}
```

Обращение к функции требует, чтобы ее значение было присвоено указателю, адресующему массив.

```
int *My_Arr;
My_Arr=*new_mas(5);
```

Имя массива в функции main() – My_Arr, его длина равна 5, для массива выделена память и введены значения.

2.7.6. Символьные строки

Для объявления объектов символьного типа используются ключевые слова char или unsigned char.

Данные символьного типа занимают один байт. Внутреннее представление символьного данного — это его

целочисленный код. Такое представление позволяет обращаться к символам как к числовым величинам, что дает некоторые преимущества.

1. Можно использовать все операции отношений для того, чтобы сравнивать символы друг с другом. При этом сравниваются коды символов, например:

```
char c1, c2;
c1=getchar();
// функция getchar получает символ с клавиатуры
c2=getchar();
```

Любую операцию отношения можно использовать для сравнения символов.

```
int k=(c1==c2);   // k=1, если символы равны, k=0, если нет
k=(c1>c2);        // k=1, если код c1 больше кода c2, иначе 0
```

Для проверки условия «символ c1 является цифрой», можно использовать упорядоченность кодов цифр по возрастанию

```
k=(c1>='0'&& c1<='9')
```

2. При выполнении арифметических операций над символами, операции выполняются над значениями внутренних кодов символов: можно применять арифметические операции, например, +, –, ++, – –, чтобы получить код символа.

3. Можно использовать символьные переменные как управляющие переменные, чтобы организовать циклы обработки символьных или строковых данных, например, перебор в алфавитном порядке:

```
char c;
for(c='a';c<='z';c++)
{ // например, сравнение символа со значениями c
}
```

Здесь символьная переменная c является операндом арифметической операции ++, выполняемой над значением ее внутреннего кода.

Коды символов иногда знать необходимо. Для определения кода символа используется функция getch() библиотеки <conio.h>. Эта функция читает с клавиатуры символ, и если присвоить результат переменной целого типа, то будет получен код символа.

Пример 2.78.

```
#include <conio.h>        // для geth ()
#define ESC 0x11b
// шестнадцатеричный код клавиши [Esc]
void main(void)
{
   int key;
   do
   {
      key =getch();          // код символа
      printf("%d\n", key);
   } while(p!=27);           // обрабатывается событие
"Нажатие Esc"
}
```

Строка символов рассматривается как массив символов (типа char[]). Каждый символ хранится в отдельном байте, включая специальные символы. В конце строки должен быть нулевой байт '\0', как признак конца строки. Он добавляется автоматически при инициализации строки и при вводе строки. Общее число символов в строке определяется с учетом нулевого символа, и равно «длина строки +1». При формировании строки вручную необходимо заботиться о том, чтобы этот символ был добавлен. Так, длина константы 'W' равна одному байту, а длина строки "W" равна двум байтам.

Операции с массивами в языке C++ не допускаются, следовательно, и для строк в целом операций нет. Символьная строка в выражении — это адрес массива символов. Указатель на строку может быть использован везде, где можно использовать указатель.

Объявить строковую переменную можно двумя способами.

1. Как статический массив. Считается, что это плохой стиль, так как длина строки может изменяться ограниченно. На самом деле при обработке текстов, так или иначе, ограничение на длину строки существует, и оно может быть достаточно расплывчатым.

```
char Str[80];                  // длина не более 79 символов
```

2. Как указатель. При этом создается динамическая строка данных типа char, для хранения которой необходимо выделение памяти.

```
char *Str;
Str=new char[80];          // длина строки и здесь
                           // определена числом
```

При инициализации, независимо от способа объявления строки, всегда происходит выделение памяти под запись значений строки.

```
char *Str1="строка 1";     // длина строки 9 байт
char Str2[9]="строка 2";   // длина строки 9 байт,
                           // как объявлено
char Str1[]="строка 3";    // специальный инициализатор
char Err[4]="ошибка";      // длина строки больше,
                           // чем объявлено
char Err[10]="не ошибка"; // длина строки меньше
                           // объявленной,
                           // остальные символы нули
```

Ввод и вывод символов и строк

Для ввода и вывода текстовой информации инструментов довольно много, это функции библиотеки <stdio.h>.

1. Форматированный ввод-вывод символов и строк.

Используются функции форматированного ввода printf(), scanf(). Управляющие форматы для символов — %c, для строк — %s. При вводе строка вводится только до первого пробела.

2. Ввод-вывод символов.

Используются функции getchar(), putchar(). Функция getchar() читает из входного потока по одному символу за обращение. Чтение данных начинается после нажатия клавиши [Enter]. Функция putchar(символ) выводит символьное значение в стандартный выходной поток, обычно это экран.

3. Ввод-вывод строк.

Используются функции gets(), puts(). Функция gets(строка) читает из входного потока последовательность символов до нажатия клавиши [Enter]. Чтение

начинается после нажатия клавиши [Enter]. Функция puts(строка) выводит строку в стандартный выходной поток.

Прототипы функций:

```
char *gets(char *Str);              // вернет NULL при ошибке
int puts(const char *Str)           // вернет EOF при ошибке
```

Пример 2.79. Пример использования функций ввода и вывода строк.

```
#include <stdio.h>
void main(void)
{
  char Str[80];
  printf("Введите строку\n");
  gets(Str);
  printf("Введена строка:\n");
  puts(Str);
}
```

Следующий пример покажет, как можно вводить строки при работе с текстом как с массивом строк.

Пример 2.80.

```
#include <stdio.h>
void main(void){
  char text[3][20];
  for(int i=0;i<3;i++)
    gets(text[i]);
}
```

Строки и указатели

Особенности строк связаны с представлением их в памяти как массивов однобайтовых переменных. Соотношение строк и указателей такое же, как соотношение указателей и массивов, но в конце строки есть нулевой байт '\0'. Для статических строк память выделяется на этапе компиляции программы, для динамических строк при выполнении программы, но в любом случае массив должен где-то разместиться.

Пусть есть два описания:

```
char Str1[20] ;   // массив, в который можно записать строку
char *Str2;       // указатель, с которым можно связать строку
```

Для статического массива Str1 память будет выделена автоматически при обработке описания компилятором. Строке, с которой можно связать указатель *Str2, память не выделяется, поэтому, если далее следуют операторы:

```
gets(Str1);                // ошибки нет
gets(Str2);                // ошибка
```

то первый из них допустим, а второй приведет к ошибке во время выполнения программы, так как gets() вводит строку в участок памяти с неизвестным адресом. Перед использованием строки следует связать адрес области памяти, в котором может разместиться строка, с указателем Str2. Во-первых, переменной Str2 можно присвоить адрес уже определенного символьного массива. Во-вторых, можно выделить динамическую память в требуемом размере с помощью операции new, а затем полученный адрес связать с указателем Str2. Например:

```
Str2=Str1;         // указатель Str2 адресует строку Str1
Str2=new char[80];
```

Во втором случае выделен блок памяти 80 байт и связан с указателем Str2. Строка не имеет собственного имени, а имя указателя Str2 является его синонимом.

Символьная строка, встреченная в выражении (например, Str1), — это адрес массива символов. Может встречаться везде, где можно использовать указатель. Для доступа к отдельным символам строк применяются обычные механизмы прямой или косвенной адресации.

Для прямой адресации используются индексы элементов массива. Выделим динамическую память для строки Str2, и выполним копирование в нее исходной строки Str1 в цикле do, управляемом индексом. Реальное число символов в исходной строке, чаще всего, неизвестно, поэтому используется цикл с выходом по событию «достигнут конец строки». Нулевой символ должен быть перенесен в новую строку:

```
char Str1[80]="Первая строка.";
char *Str2;
Str2=new char[80];
// оператор do выполняет посимвольное копирование,
// перенесет нулевой байт в новую строку
```

```
i=0;
do
  Str2[i]=Str1[i];
while(Str1[i++]!='\0');      // проверка достижения конца строки
puts(Str2);
```

Для косвенной адресации используются указатели типа char * в качестве переменных, управляющих обработкой строки. Текущий указатель будет адресовать ровно один символ строки, но в отладчике будет видна оставшаяся часть строки до нулевого байта '\0'. Покажем этот механизм на примере.

Пример 2.81.

```
char     *pts1, *pts2;
```

Это рабочие переменные для косвенной адресации. Управляет циклом переменная pts1. Ее начальное значение равно адресу исходной строки Str1. *pts1 — это значение очередного символа строки, pts2++ смещает указатель. Выход из цикла происходит, когда найден признак конца строки:

```
char Str1[80]="Первая строка.";
char *Str2=new char[80];
char *pts1, *pts2;
pts1=Str1;
// подготовка цикла, это настройка указателей
pts2=Str2;
do
  *pts2++=*pts1;// сначала присваивание, затем смещение
while(*pts1++!='\0');
puts(Str2);
```

В стиле C++ этот цикл выглядит так:

```
pts1=Str1;
pts2=Str2;
while((*pts2++=*pts1++));
puts(Str2);
```

Большинство ошибок при работе со строками связаны с механизмами выделения памяти для динамических строк. Если при объявлении указателя текстовая строка проинициализирована, то выделена память, равная длине этой строки, и нельзя пытаться записать в эту строку больше символов, чем выделено. Если же при объявлении

указателя текстовая строка не проинициализирована, то память под запись строки вообще не выделена, и этот указатель можно использовать только как рабочую переменную для косвенной адресации при работе с какой-либо строкой.

В примере 2.82 описан и проинициализирован одномерный массив Rainbow[] указателей типа char*, каждый элемент которого является строкой. Для вывода строк используется функция puts(), но и функция printf() со спецификатором преобразования %s допускает использование указателя на строку в качестве параметра. В выходной поток выводится не значение указателя Rainbow[i], а содержимое адресуемой им строки.

Пример 2.82.

```
#include <stdio.h>
void main(void)
{
// объявлен массив строк,
// каждое слово – название цвета радуги
// память выделена при инициализации
char *Rainbow[]={"Красный","Оранжевый","Желтый",
"Зеленый", "Голубой","Синий","Фиолетовый"};
   int i, len;
   len=sizeof(Rainbow)/sizeof(Rainbow[0]);
   // определена длина массива строк
   for(i=0;i<len;i++)
      puts(Rainbow[i]);
}
```

В результате выполнения программы мы увидим на экране названия цветов радуги:

```
Красный
Оранжевый
```

и т. д.

Строки как параметры функций

Когда строка является параметром функции, то передача строки в функцию выполняется так же, как и для массивов. Строка в качестве формального параметра в заголовке функции описывается как одномерный массив типа char:

```
char Имя_строки[Константа];
```

или как указатель типа char:

```
char *Имя_строки;
```

Строка в качестве фактического параметра в вызывающей программе может быть статической или динамической. Всегда через параметр в функцию передается адрес начала символьного массива, содержащего строку. И функция, и вызывающая программа имеют одинаковые права на изменение содержимого строки.

Отдельно передавать длину строки нет необходимости, так как, во-первых, строка имеет признак конца '\0', который позволяет управлять циклами перебора символов строки, не выходя за ее пределы, а во-вторых, ее длину всегда можно определить, используя функцию strlen() библиотеки <string.h>.

Рассмотрим пример функции преобразования строки, которая не изменяет размер строки, а только инвертирует ее содержимое.

Пример 2.83.

```
void invert(char Str[])
{
  char Buf;
  int i,j,m;
  for(m=0;Str[m]!='\0';m++);
  // переменная m найдет положение '\0'
  for(i=0, j=m–l;i<j;i++,j––)
  {
    Buf=Str[i];
    Str[i]=Str[j];
    Str[j]=Buf;
  }
}
```

Функция не возвращает значения, так как она изменит содержимое строки, при этом символ '\0' остается на своем месте в конце строки. Пример использования функции invert():

```
void main(void)
{
  char S[]="0123456789";
  // вызов функции
```

```
  invert(S);
  puts(S);
}
```

Результат выполнения программы:

```
9876543210
```

Для многих функций библиотеки <string.h>, а также некоторых других, компилятор выдает предупреждения о том, что функция не является безопасной, например:

```
warning C4996: 'strcpy': This function or variable may be unsafe.
```

Предупреждение относится к улучшениям Microsoft по безопасности функций стандартных библиотек. Данные предупреждения можно игнорировать по ряду причин, например, безопасные функции при выполнении требуют дополнительного времени на обеспечение безопасности, что в общем случае не требуется.

Для подавления этого предупреждения достаточно директивой #define определить макрос _CRT_SECURE_NO_WARNINGS:

```
#define _CRT_SECURE_NO_WARNINGS
```

2.7.7. Структуры

Структура — это множество именованных данных, объединенное в единое целое.

Определение. Описание структуры

Как и массив, структура предназначена для логического объединения данных в более крупные единицы, чем данные простых типов. Массив всегда состоит из однотипных элементов, и все они имеют общее имя. Элементы (компоненты, поля) структуры могут быть разных типов, и все имеют различные имена.

Описание структуры конструирует новый тип данного, объединяющий в себе разнотипные разноименные компоненты, имеет синтаксис:

```
struct Имя_структурного_типа
{
// Определения элементов;
};          // ";" завершает описание
```

Здесь struct — ключевое слово, обозначающее структурный тип. «Имя_структурного_типа» — произвольное имя, которое идентифицирует сконструированный тип; «Определения элементов» (компонентов, полей) — описания объектов, входящих в состав структуры, каждый из которых есть прототип одного из данных, входящих в состав вводимого структурного типа.

Описание структурного типа заканчивается точкой с запятой.

Пример структуры «товар на складе»:

```
struct Goods
{
  char *name;   // наименование
  float price;    // оптовая цена
  float percent; // торговая наценка в %
  int vol;        // объем партии товара
  char date[9]; // дата поставки партии товара
};
```

Полное имя структурного типа «struct Goods». Это имя является именем типа данного, который в общем виде описывает объект «товар на складе». Наименование товара будет связано с указателем типа char* и именем name, оптовая цена единицы товара будет значением типа float с названием price и т. д.

Введенное описание структурного типа именует абстрактный тип данного, сконструированного программистом, и далее может использоваться для объявления переменных этого типа, так же как для базовых типов используются спецификаторы, например, double или int.

Синтаксис такого объявления:

```
struct Имя_структурного_типа Имя_переменной; // переменная
struct Имя_структурного_типа *Имя_переменной; // указатель
```

Так вводятся фактические объекты указанного типа или указатели на них.

Например,

```
struct Goods    food;
struct Goods    *p_food;
```

Служебное слово typedef позволяет ввести собственное обозначение для любого определения типа. Для структурного типа синтаксис таков:

```
typedef struct
{
  // определения элементов;
}
// обозначение_структурного_типа;
```

«обозначение_структурного_типа» — это абстрактное имя типа, произвольно введенное программистом.

Пример для того же объекта:

```
typedef struct
{
  char *name;          // наименование
  float price;         // оптовая цена
  float percent;       // торговая наценка в %
  int vol;             // объем партии товара
  char date[9];        // дата поставки партии товара
} Goods;
```

Это описание вводит структурный тип struct{определения элементов}, и присваивает ему «обозначение_структурного_типа» Goods. Обозначение имеет тот же смысл, что и ранее (название, имя), и так же используется для объявления объектов структурного типа, но без использования слова struct, например:

```
Goods food1, food2, food3;
```

Объявлены три структуры: food1, food2, food3, каждая из которых представляет реальный объект.

Инструкция typedef назначает имя структурному типу, который может в то же время иметь второе имя, вводимое стандартным образом после служебного слова struct. Это имя может быть синонимом первому.

```
typedef struct Merchandise
{
  char *name;          // наименование
  float price;         // оптовая цена
  float percent;       // торговая наценка в %
  int vol;             // объем партии товара
  char date[9];        // дата поставки партии товара
```

```
} Goods;
```

Здесь Goods — обозначение структурного типа, введенное с помощью typedef, а имя Merchandise введено для того же типа стандартным способом. После такого определения структуры объекты этого типа могут вводиться как с помощью названия struct Merchandise, так и с помощью обозначения того же типа Goods.

Изложенная последовательность определения структур (определить структурный тип, затем ввести переменные этого типа) является самой логичной из всех, но в языке C++ имеются еще две схемы определения данных типа структура.

Во-первых, структуры можно ввести в употребление одновременно с определением структурного типа:

```
struct Имя_структурного_типа
{
// определения_элементов;
} Список_структур; // это имена переменных структурного типа
```

Пример одновременного определения структурного типа и объектов этого типа:

```
struct Student
{
  char name[15];              // имя
  char surname[20];           // фамилия
  int group;                  // группа
} student_1, student_2, student_3;
```

Здесь определен тип с именем Student и три объекта этого типа student_1, student_2, student_3, которые являются полноправными объектами. В каждую из этих трех структур входят элементы, позволяющие представить имя (name), фамилию (surname), группу (group).

После приведенного определения в той же программе можно определять любое количество структур, используя структурный тип Student:

```
struct Student   leader, freshman;
```

Во-вторых, можно определять структуры, не вводя названия типа. Безымянный структурный тип обычно используется в программе для однократного определения структур:

```
struct
{
  // определения элементов;
} Список_структур;   // имена переменных структурного типа
```

Пример безымянного структурного типа, описывающего конфигурацию персонального компьютера:

```
struct
{
  char processor[10];    // тип процессора
  int frequency;         // тактовая частота в МГц
  int memory;            // объем основной памяти в Мбайт
  int disk;              // объем жесткого диска в Гбайт
} IBM_586, IBM_486, Compaq;
```

Введены три объекта с именами IBM_586, IBM_486, Compaq. В каждую из них входят элементы, в которые можно занести сведения о характеристиках конкретных компьютеров. Структурный тип «компьютер» не именован, поэтому если в программе потребуется определять другие структуры с таким же составом элементов, то придется полностью повторить приведенное выше определение.

Для работы с объектами структурного типа следует объявлять переменные этого типа, например:

```
struct Merchandise dress, footwear, toy;
```

Если имя структурного типа введено с помощью typedef, то определение не содержит слова struct, например:

```
Goods *food, *drink;
```

Описание структурного типа не связано с выделением памяти. Только при объявлении объекта типа структуры, ему выделяется память в таком количестве, чтобы могли разместиться данные всех элементов структуры. Реальный размер памяти в байтах, выделяемый для структуры, можно определить с помощью операции sizeof:

```
sizeof(Имя_структурного типа);
```

или

```
sizeof(Имя_объекта_структурного_типа);
```

Инициализация структур похожа на инициализацию массивов. При объявлении объекта структуры в фигур-

ных скобках после имени и знака присваивания размещается список начальных значений элементов, например:

```
struct Goods coat=
{
  "Черная шляпа",1000.00,15.0,100,"12.01.04"
};
```

При этом элементы объекта coat получают соответствующие начальные значения.

Операции над структурами. Доступ к элементам структур

Для структур стандарт языка C++ разрешает присваивание, если операнды являются однотипными. Для массива в целом операция присваивания не допускается.

```
struct Goods dress;
```

Допустимо следующее присваивание:

```
dress=coat;
```

Никакие операции сравнения для структур не определены. Сравнивать структуры следует только поэлементно.

Для доступа к элементам структур используется операция разыменования «точка» или уточненные имена полей структуры.

Синтаксис.

```
Имя_структуры.Имя_элемента
```

Здесь «Имя_структуры» — это имя объекта структурного типа, а «Имя_элемента» — это имя одного из элементов в соответствии с определением структурного типа.

Например, в примере с инициализацией структуры типа struct goods:

coat.name — указатель типа char* на строку «пиджак черный».

Перед точкой стоит имя конкретной структуры, для которой при ее объявлении выделена память.

Структуры, массивы и указатели

Рассматривая соотношение между этими типами, можно выделить возможные варианты.

Массивы и структуры в качестве элементов структур

Массив, как и любые другие данные, может быть элементом структуры, тогда для обращения к каждому элементу такого массива требуется операция разыменования элемента массива [].

Элементом структуры может быть другая структура, это называется вложение структур. Например, структурный тип для представления сведений о студенте, дополнительно к тем элементам, которые уже есть (имя, фамилия, группа), может содержать данное «дата рождения». Это может быть структура с элементами «число», «месяц», «год».

В тексте описания структурных типов должны быть размещены в такой последовательности, чтобы использованный в описании структурный тип был уже определен ранее:

```
struct Date                 // тип "дата"
{
  int day;                  // число
  int month;                // месяц
  int year;                 // год
};
```

При обращении к такому элементу операция «точка» используется дважды, так как имя поля нуждается в уточнении дважды.

Пример 2.84.

```
void main(void)
{
  // конкретная структура
  struct Student stud_1=
  {
    "Павел","Колесников",145,22,04, 988
  };
  printf("\n Введите группу:");
  scanf("%d",&stud_1.group);
  printf("Сведения о студенте:");
  printf("\n Фамилия: %s", stud_1.surname);
  printf("\n Имя:%s", stud_1.name);
printf("\n Группа: %d\n", stud_1.group);
printf("\n Дата рождения: %d.%d.%d", stud_1.date.day,
```

```
        stud_1.date.month, stud_1.date.year);
}
```

Объект stud_1 типа struct Student получает значение элементов при инициализации. Затем с помощью ввода изменяется элемент stud.group, и содержимое структуры выводится на экран. Для доступа к элементам вложенных структур используются дважды уточненные имена.

Массивы структур

Массивы структур вводятся в употребление так же, как и массивы других типов данных, с указанием имени структурного типа, например:

```
struct Goods    List[MAX];
```

Определен массив List, состоящий из MAX элементов, каждый из которых является структурой типа Goods. Имя List, это имя массива, элементами которого являются структуры. List[0], List[I] и т. д., все они структуры типа Goods.

Для доступа к полям структур, входящих в массив структур, используются уточненные имена с индексированием первого имени (имени массива). Индекс записывается непосредственно после имени массива структур, например: List[0].price — элемент с именем price структуры типа Goods, входящей в качестве элемента с нулевым индексом в массив структур List[].

При размещении в памяти массива структур элементы массива размещаются подряд в порядке возрастания индекса. Каждый элемент массива занимает столько места, сколько необходимо для размещения структуры.

Как и массивы других типов, массив структур при определении может быть инициализирован. Инициализатор массива структур должен содержать в фигурных скобках список начальных значений структур массива. В свою очередь, каждое начальное значение для структуры — это список значений ее компонентов (также в фигурных скобках).

Указатели на структуры

Указатели на структуры вводятся в употребление так же, как и указатели на данные других типов.

Например, для безымянных структурных типов:

```
struct Goods *p_goods;
struct
{                          // безымянный структурный тип
  char processor[10];      // тип процессора
  int frequency;           // частота в МГц
  int memory;              // память в Мбайт
  int disk;                // емкость диска в Мбайт
} *point_IBM, *point_2;
```

Если название структурного типа введено с помощью typedef, то объявление указателя выглядит как обычное объявление указателя.

При определении указателя на структуру он может быть инициализирован. Наиболее корректно в качестве инициализирующего значения применять адрес структурного объекта того же типа, что и тип определяемого указателя.

Пример 2.85.

```
struct Particle
{
  double mass;
  float coord[3];
} dot[3], point, *p_point;
// инициализация указателей
struct Particle *p_d=&dot[0];
```

Значение указателя на структуру может быть определено и с помощью обычного присваивания:

```
p_point=&point;
```

Если используется указатель на структуру, адресующий конкретный объект, то доступ к ее элементам обеспечивается с использованием операции косвенного разыменования «стрелка (–>)», синтаксис которой:

```
Указатель_на_структуру–>Имя_элемента
```

Операция имеет два операнда: «Указатель на структуру», который адресует левый операнд, и «Имя элемента». Тип результата операции «стрелка» совпадает с типом правого операнда, т. е. того элемента структуры, на который она нацелена. Имеет самый высший ранг, наряду со скобками и операцией «точка» обеспечивает доступ к эле-

менту структуры через адресующий ее указатель того же структурного типа.

Пример 2.86.

```
p_point–>mass          // эквивалентно (*p_point).mass
p_point–>coord[0]      // эквивалентно (*p_point).coord[0]
```

Изменить значения элементов структуры можно, используя присваивание:

```
p_point–>mass=18.4;
for(int i=0;i<n;i++)
  p_point–>coord[i]=0.1*i;
```

Структуры и функции

Возможны всего два варианта: структура может быть передана функции как параметр, или структура может быть значением, возвращаемым функцией. И в том, и в другом случае могут использоваться указатели на объекты структурных типов. Рассмотрим возможные варианты. Пусть есть структура:

```
struct Person
{
  char *name;
  int age;
};
```

Покажем обязательные синтаксические конструкции в прототипах функций, которые работают со структурами:

1) параметр функции — объект структурного типа:

```
void F1(struct Person Имя_объекта);
```

2) параметр функции — указатель на объект структурного типа:

```
void F2(struct Person *Имя_объекта);
```

3) функция возвращает структуру:

```
struct Person F3(Список_параметров);
```

4) функция возвращает указатель на структуру:

```
struct Person *F4(Список_параметров);
```

Механизмы передачи параметров и возвращения значения функцией подробно рассмотрены в параграфе 2.5.

2.8. ФАЙЛЫ

В языке С++ нет средств ввода-вывода. Этим обеспечивается аппаратная независимость языка. Ввод-вывод реализуется посредством библиотек стандартных функций. Библиотека stdio.h содержит средства ввода-вывода (обмена с устройствами), в том числе с файлами на диске. С точки зрения языка нет разницы, с устройством или файлом происходит обмен.

Существует три уровня ввода-вывода:

- верхнего уровня (потоковый);
- записями (низкоуровневый);
- для консоли и портов — системно-зависимый обмен.

В данном пособии рассмотрим только работу с файлами.

2.8.1. Определение и типы файлов

Файл — именованная область внешней памяти, в которой содержится некоторая информация. Например, тексты программ хранятся в виде текстовых файлов. Файлы также могут хранить данные, а программы могут обращаться к файлам данных для чтения информации или записи.

Файлы удобно использовать, когда программа обрабатывает большой объем информации, или когда данные должны храниться на внешних устройствах, или когда данные могут быть исходными для нескольких программ обработки.

Программы, которые работают с данными, хранящимися в файле, временно размещают их в оперативной памяти. Этот прием позволяет увеличить скорость обработки данных и снизить сложность программ. Данные из файлов, будучи прочитаны программой, становятся значениями объектов программы, например, массивами, матрицами и др.

По механизму хранения данных и обращения к ним файлы разделяются на файлы последовательного доступа и файлы прямого доступа.

Файлами *последовательного* доступа являются текстовые файлы. Такие файлы подготавливаются в текстовом редакторе и хранят данные в символьном представлении. Их можно легко просматривать и редактировать, но невозможно получить доступ к определенному данному, пока не прочитаны все данные, находящиеся перед ним.

Файл *прямого* доступа — это двоичный файл. Хранит данные одного типа, не обязательно базового. Каждое данное хранится во внутреннем представлении, и его размер определен типом данного. Чтобы получить любое данное, можно переместить указатель файла непосредственно на это данное, и выполнить операцию обмена.

Программа, использующая файл данных, должна выполнить следующие действия.

1. Открыть (закрыть) поток — объявить указатель на поток и связать его с физически существующим файлом.

2. Передать данные в файл (из файла) — для этого существуют функции ввода-вывода.

Кроме того, можно (и нужно).

3. Отсекать ошибки обмена данными. Для этого предназначены функции обработки ошибок.

4. Управлять буфером обмена. Это исполняют функции буферизации потока.

5. Для файлов прямого доступа существуют функции перемещения указателя потока.

6. И многое другое.

Объявление файловой переменной

Для объявления логического имени файла используется тип FILE, описание которого находится в библиотеке <stdio.h>. FILE — это структура данных, которая хранит данные о файле, такие как размер буфера и пр.

Объявление файла:

```
FILE *Имя_файла;
```

Здесь «Имя_файла», это имя переменной, связанной с файлом, указатель на стандартную структуру данных типа FILE.

Это объявление аналогично объявлению обычных переменных:

Например:

```
FILE *in, *out;     // файл in для ввода, out для вывода
FILE *my_file; *my_other_file;
```

Этот указатель можно назвать логическим именем файла, под которым файл будет известен программе, и будет использоваться во всех последующих операциях.

2.8.2. Открытие и закрытие файлов

Перед использованием файл следует открыть. Смысл этой операции в том, чтобы связать логическое имя файла с файлом, физически существующим на диске. Логическому имени файла присваивается значение, возвращаемое функцией fopen(). В случае ошибки открытия файла, fopen() возвращает NULL. Функция имеет два параметра строкового типа.

Формат обращения к функции:

```
имя_файла=fopen("Имя_физического_файла","Режим_
открытия_файла");
```

Пример 2.87.

```
in=fopen("input.txt","r");
out=fopen("output.txt","wb");
```

«Имя_физического_файла» — это строка, значение которой определяет имя файла, возможно, с указанием полного пути к нему. «Режим_открытия_файла» — это строка, определяющая режим доступа к потоку и тип файла, содержит один или два символа. Первый символ определяет тип операций обмена.

r — для чтения. Файл должен существовать на диске. Логическое имя файла связывается с физическим именем. Позиция чтения устанавливается перед первым байтом файла.

w — для записи. Если файл существует на диске, то он будет открыт для записи. Позиция записи устанавливается перед первым байтом файла. Если в файле была информация, она будет утеряна. Если файл не существует, он будет создан.

a — для добавления в конец. Если файл существовал на диске, он будет открыт для записи. Позиция записи устанавливается перед признаком конца файла. Если файла не было, он будет создан.

Признак + после типа файла, r+, w+, a+, расширяет возможности операций обмена, модифицируя тип файла. С таким типом файл открывается как для чтения, так и для записи.

r+ — существующий файл открывается как для чтения, так и для записи в любом месте файла. Запись в ко-

нец недопустима, так как недопустимо увеличение размера файла.

w+ — новый файл открывается для записи и последующих изменений. Если файл существует, прежнее содержимое стирается. Последующие операции записи и чтения допустимы в любом месте файла, в том числе в конец, т. е. размер файла может увеличиваться.

a+ — файл открывается или создается и становится доступным для изменений, т. е. для записи и чтения в любом месте файла. В отличие от w+, при открытии существующего файла его содержимое не уничтожается, в отличие от r+ можно добавить запись в конец файла.

Второй символ определяет тип файла, следовательно, тип хранения данных.

t — текстовый файл, действует по умолчанию.

b — двоичный файл.

Физический файл может находиться где угодно. Имя файла можно указывать с полным путем. Если имя краткое, файл будет отыскиваться только в текущем каталоге.

```
fopen("c:\\work\\f.txt","wt");
```

Имя физического файла может изменяться, тогда параметр функции fopen() должен быть переменной строкового типа:

```
char *name_of_file=new char[50];
puts("Введите имя файла\n");
gets(name_of_file);
fopen(name_of_file,"wt");  // запишет в файл, который укажет
                           // пользователь при работе
```

Как пример, приведем функции вывода одномерного массива. Вывод массива на экран описан функцией Print_mas, параметрами которой являются имя массива и его длина.

Пример 2.88.

```
#include <stdio.h>
// вывод массива на консоль
void Print_mas(int mas[],int len) // len – длина массива
{
  int i;
  printf("Массив:\n");
```

```
  for(i=0; i<len;i++)
     printf("%5d", mas[i]);
  printf("\n");
}
```

Если массив должен быть выведен в файл, то функция принимает еще один параметр — имя файловой переменной FILE *out, а операция вывода printf заменяется на fprintf.

```
// вывод массива в файл
void Print_mas(FILE *out,int mas[],int len)
{
  int i;
  fprintf(out, "Массив:\n");
  for(i=0;i<len; i++)
     fprintf(out, "%5d", mas[i]);
  fprintf out, "\n");
}
```

Файловая переменная объявляется в вызывающей программе, связывается с файлом на диске и передается функции как фактический параметр.

Существуют некоторые обычные ошибки открытия файла, например, файл не найден, диск заполнен, недостаточно динамической памяти для выполнения операции и пр. При любой ошибке открытия файла fopen() возвращает NULL. Это используется для того, чтобы обрабатывать возможные ошибки ввода-вывода, используя условный оператор. В случае возникновения одной из штатных ошибок ввода-вывода, ее код errno распознает и обрабатывает функция perror().

Функция perror() возвращает сообщение об ошибке, а fprintf() переназначает вывод строки сообщения об ошибке в стандартный поток stderr.

Пример 2.89.

```
FILE *my_file;
if((my_file=fopen("f.txt","wt"))==NULL)
// текстовый файл для записи
{
  perror("Ошибка открытия файла");
  // если есть ошибка, ее код = errno,
```

```
    // perror анализирует номер ошибки
    // и выводит поясняющий текст
    exit();
}
```

Прототип функции perror() находится в директиве препроцессора <stdio.h>. Там же определена переменная errno типа int. Ею пользуются многие функции, в том числе функции ввода-вывода. Функция fopen(), обнаружив ошибку, заносит ее код в переменную errno. Функция perror() выводит на экран текстовую строку (свой аргумент), затем двоеточие, пробел и сообщение об ошибке, содержимое и формат которого определены реализацией.

Если файл не найден, а ошибка не анализируется, поток ввода перенаправляется на стандартный поток ввода-вывода (консоль), с попыткой чтения из буфера обмена.

Файл необходимо после использования закрыть. *Закрытие файла* — это высвобождение логического имени файла. Используется, чтобы отвязать логическое имя от физического файла. Логическое имя не перестает существовать, и может быть использовано повторно для других целей, например, для изменения режима работы с файлом.

Синтаксис.

```
fclose(имя_файла);
```

Пример 2.90.

```
fclose(in);
fclose(out);
```

При завершении работы программы все открытые в ней файлы будут закрыты, даже если операция закрытия не была выполнена программистом. Если файл закрывает программист, то все данные из буфера выводятся в файл перед его закрытием. Если же файл закрывается при завершении программы, то некоторые данные могут быть потеряны вследствие механизма буферизации обмена.

Признак конца файла (end-of-file) присутствует в конце файла всегда. Чтобы его распознать в потоке, используется макроопределение feof. С его помощью можно проверить, найден ли в потоке признак конца файла.

Объявление

```
int feof(FILE *stream);
```

Как видим, тип возвращаемого значения int, следовательно, данная функция возвращает целое значение (точнее, логическое). Функция feof() проверяет данный поток на наличие end-of-file в текущем положении указателя потока. Возвращаемое значение отлично от нуля, если при выполнении последнего оператора ввода для потока найден end-of-file, и равно нулю, если end-of-file не обнаружен.

Файл может хранить произвольное количество данных. При вводе данных из файла в массив можно определить это число, тем самым узнать длину массива. Фактическим признаком конца файла является сочетание клавиш [Ctrl + Z]. Если данные вводятся в текстовом редакторе, этот признак невидим. При вводе данных с клавиатуры его можно ввести фактически.

Пример 2.91.

Пусть длина массива объявлена:

```
#define LEN_MAX          100
```

Это ограничение на длину необходимо учесть. Если входных данных больше 100, длина массива будет ограничена значением LEN_MAX. Если данных меньше, признак конца файла ограничит ввод, и длина массива будет равна числу данных во входном потоке. В этом примере данные вводятся из входного потока stdin.

```
void Input_mas(FILE *in,int *mas,int &len)
{
  len=0;
  do
  {
    scanf("%d", &mas[len]); // размер массива
    len++;
  }
// в цикле выполняется ввод пока не встречен конец файла,
// или длина массива < 100
  while(!feof(stdin)&&len<LEN_MAX);
  len--;
}
```

2.8.3. Чтение и запись для текстовых файлов

Все функции ввода-вывода для консоли, которые были рассмотрены ранее, работают по принципам потокового ввода-вывода, используя стандартные текстовые потоки stdin, stdout, srderr.

При работе с текстовыми файлами используются функции библиотеки <stdio.h>, отличительным признаком которых является буква f в начале имени функции. Каждая из них имеет дополнительный параметр — логическое имя файла, для которого выполняется данная операция.

Для чтения символов из файла:

```
int fgetc(FILE *stream);
int fputc(int C, FILE *stream);
```

Функция fgetc() читает символ (int) из входного потока stream.

Функция fputc() выводит символ C в выходной поток stream.

Для чтения строк из файла:

```
char *fgets(char *s,int n,FILE *stream);
int fputs (const char *s,FILE *stream);
```

Функция fgets() читает не более n–1 символа в строку s (char *) из входного потока stream. Константа n определят наибольшее количество считываемых символов. Чтение происходит до обнаружения признака новой строки или конца файла. Если символов меньше, будет прочитано их реальное количество.

Функция fputs() выводит строку s в выходной поток stream. Возвращает число записанных символов.

Для форматированных данных:

```
fprintf(Имя_файла,"форматная_строка",список_вывода);
fscanf(Имя_файла,"форматная_строка",список_ввода);
```

Здесь «Имя_файла» — логическое имя файла.

Пример 2.92.

В качестве примера приведем функции чтения из файла двумерного массива, и вывода матрицы в файл.

```
// чтение матрицы из файла
// в первой строке файла первые два числа
// обозначают размер матрицы
void Read_matr(FILE *in,int matr[][10],int &n,int &m)
```

```
{
   fscanf(in,"%d",&n);     // Количество строк
   fscanf(in,"%d",&m);     // Количество столбцов
   int i, j;
   for(i=0;i<n;i++)
      for(j=0;j<m;j++)
         fscanf(in,"%d",&matr[i][j]);
}
// вывод матрицы в файл
void Print_matr(FILE *out,int matr[][10],int n,int m)
{
   int i, j;
   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
         fprintf(out,"%6d", matr[i][j]);
   fprintf(out,"\n");
   }
}
```

Некоторые функции библиотеки <stdio.h> для работы с файлами в новых версиях языка относятся к небезопасным. Компилятор выдает предупреждение, в котором рекомендует использовать безопасную версию, как правило, имеющую префикс _s, например, вместо fopen() использовать fopen_s(). Можно использовать безопасные функции, или подавлять предупреждения директивой #define _CRT_SECURE_NO_WARNINGS

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Определите назначение и правила записи имен в языке С++.
2. Дайте определение и назначение констант. Как определяется тип константы?
3. Перечислите типы данных. Что такое явное и неявное преобразование типов? Какова роль объявления типов в программе?
4. Поясните роль определения переменной в программе.
5. Почему происходит преобразование типов при присваивании? Поясните механизм преобразования типов.
6. Какова структура программы на языке С++?
7. Что такое директивы препроцессора? Поясните назначение и механизм действия директив #define и #include.

8. Каково назначение операторов языка программирования? Приведите примерную классификацию операторов языка C++.
9. Поясните цель использования функций при составлении программного кода. Почему функции имеют тип, и какого типа может быть возвращаемое функцией значение? Как функция возвращает значение?
10. Чем описание функции отличается от ее объявления? Когда функцию можно только описать, а когда функцию следует объявить?
11. Чем передача параметров по ссылке отличается от передачи параметров по значению?
12. Что такое время жизни имени? Как область действия имени зависит от времени ее жизни?
13. Как передаются в функцию массивы?
14. Укажите особенности передачи в функцию строк.
15. Дайте определение массива как производного типа данных. Какие характеристики массива должен указать программист при его использовании в программе?
16. Поясните отличия между статическим и динамическим массивами. Как выделяется память для хранения статических и динамических массивов?
17. Что такое указатель? Когда он используется? Поясните особенности переменной «указатель».
18. Какие операции выполняются над указателями? Почему существуют ограничения?
19. Как применяются указатели для работы с массивами? Что такое косвенная адресация в массивах?
20. Что такое строка? Каковы особенности их представления?
21. Как создаются динамические строки?
22. Что такое структура? Чем структуры как тип данных отличаются от массивов? Как структуры используются в качестве параметров функций?
23. Что называется файлом? Когда они используются?
24. Какие действия следует выполнить при использовании файла? Что такое логическое имя файла и как оно связано с физическим именем?
25. Какие существуют типы операций обмена с файлом? Как тип операции обмена задается в программе?
26. Всегда ли необходимо закрывать файл после его использования? Если файл не закрыт, то что с ним происходит?

ГЛАВА 3

ЗАДАЧИ И УПРАЖНЕНИЯ

ТЕМА 1. ПРОСТЫЕ ПРОГРАММЫ НА ЯЗЫКЕ С++

Простыми можно считать такие задачи, у которых алгоритм решения линейный, т. е. все действия выполняются в порядке записи описывающих их операторов. Простые задачи сводятся к линейному процессу со следующими основными пунктами:

- ввести исходные данные;
- выполнить вычисления по формулам;
- вывести результат.

Последовательность решения должна складываться из следующих этапов.

1. Анализ условия задачи.

2. Определение необходимых входных данных: сколько их, какого типа каждое данное, какое имя следует присвоить переменной, обозначающей данное. Тип переменной должен быть определен в объявлении, а имя должно быть значимое, т. е. подчеркивать назначение переменной, ее логический смысл.

3. Определение выходных данных программы: сколько их, какие типы, как назвать переменные для обозначения результатов.

4. Определение списка рабочих переменных, если необходимо.

5. Определение необходимых формул вычисления, их последовательности и правил записи.

6. Кодирование алгоритма.

При кодировании необходимо соблюдать синтаксические правила и следить за стилем программирования. Эти требования можно сформулировать следующим образом.

1. Объявить все переменные программы.

2. Правильно оформить ввод данных. Перед каждым оператором ввода необходимо записать инструкцию, которая подскажет пользователю, что он должен сделать в данный момент времени.

3. Правильно записать формулы вычислений.

4. Грамотно оформить вывод результата. Вывод следует дополнить необходимыми заголовками и пояснениями, числовые данные форматировать.

Пример 1.1. Пусть требуется вычислить длину окружности, площадь круга и объем шара одного и того же радиуса.

Опишем последовательность решения.

1. Анализ алгоритма показывает, что он линейный.

2. Входная переменная одна, это значение радиуса, в общем случае величина вещественная. Обозначим его именем r.

3. Выходных данных три. Обозначим их именами l, s, v. Все переменные имеют вещественный тип.

4. Формулы вычислений общеизвестны.

```
#include <stdio.h>
// библиотека стандартного ввода-вывода
#include <math.h>
// библиотека математических функций
#define PI 3.1415926     // значение константы π
void main(void)
{
  float r;                    // объявление всех переменных
  float l, s, v;
  printf("\nДлина окружности, площадь круга и объем шара
:\n");
  // вывод заголовка
  printf("\nВведите значение радиуса:\n");
  // приглашение для ввода данных
  scanf("%f", &r);          // ввод исходного данного
  // вычисления по формулам:
  l=2.*PI*r;
  s=PI*r*r;
  v=4.*PI*r*r*r/3.;
```

```
  // Вывод результатов
  printf("r=%6.2f, l=%6.2f, s=%6.2f, v=%6.2f\n ", r, l, s, v);
} // End of main
```

Многие ошибки начинающих программистов вызваны незнанием информации о типах данных. Объявление переменной в программе указывает ее имя и тип. Тип переменной определяет способ хранения переменной, а именно сколько памяти выделено для переменной и как ее значение размещено в этом объеме. Тип также определяет диапазон значений, которые может принять переменная, и операции, которые можно к ней применить. Способ хранения переменных вещественного типа (double, float) радикально отличается от способа хранения данных целых типов (int, char). Тип констант, если они есть в тексте программы, определяется их записью. Так константа 2 будет целочисленной, а константы 2.0 или 2. будут вещественными. При вычислении значений выражений, в которых смешаны данные разных типов, выполняется преобразование (приведение) типов. Например, если в программе объявлены целочисленные переменные:

```
int a=1, b=2;
float f;
```

и выполнено деление, а значение присвоено переменной вещественного типа:

```
f=a/b;
```

то результат будет равен 0.0, потому что сначала выполняется целочисленное деление, а потом присваивание. Чтобы результат был равен 0.5, нужна запись с использованием явного преобразования типов:

```
f=(float)a/float(b);
```

Пример 1.2. Особенности, связанные с преобразованием типов, рассмотрим на следующем примере. Дано значение угла, измеренное в градусах, минутах и секундах. Требуется перевести значение угла в радианную меру. Формула вычисления:

$$\lambda_{\text{рад}} = \frac{\lambda_{\text{град}} \cdot 360}{2 \cdot \pi}.$$

```
// программа имеет скрытую ошибку
```

```
#include <stdio.h>
#include <math.h>
void main(void)
{
int Grad, Min, Sec;      // исходные данные всегда целые
float GRAD;              // новая переменная для
                         // вычисления полного значения угла
float Rad;               // итоговое значение
printf("Введите значение угла в градусах, минутах
и секундах:\n");
scanf("%d%d%d", &Grad, &Min, &Sec);
// полное значение угла:
GRAD=Grad+Min/60+Sec/3600;
// перевод в радианы:
Rad=Grad*M_PI/360;  // константа M_PI объявлена в
<math.h>.
printf("Значение угла в радианах: %12.6f\n",Rad);
} // End of main
```

Для проверки стоит выполнить два тестовых примера, вот их результаты:

Введите значение угла в градусах, минутах и секундах:
2 0 0
Значение угла в радианах: **0.017453**
Введите значение угла в градусах, минутах и секундах:
1 59 59
Значение угла в радианах: **0.008727**

Жирным начертанием выделены результаты решения задачи. Как видим, почти одинаковые входные данные дают разные результаты. Источником ошибки величиной в один градус является строка, вычисляющая полное значение угла. При делении чисел целого типа результат равен нулю, поэтому значения минут и тем более секунд отброшены при вычислении GRAD.

Для устранения ошибки используется явное преобразование типов.

```
#include <stdio.h>
#include <math.h>
void main(void)
```

```
{
   int Grad, Min, Sec;
   float Rad;
   float GRAD;
   printf("Введите значение угла в градусах, минутах
и секундах\n");
   scanf("%d%d%d", &Grad, &Min, &Sec);
   // полное значение угла:
   GRAD=(float)Grad+(float)Min/60.+(float)Sec/3600.;
   // перевод в радианы
   Rad=GRAD*M_PI/360;
   printf("Значение угла в радианах=%12.6f\n",Rad);
} // End of main
```

Пример 1.3. Рассмотрим особенности вычислений с использованием данных целого типа. Сформулируем содержательную задачу. Пусть для покупки подарков выделена некоторая сумма в рублях. Требуется узнать, сколько можно купить подарков, если известна цена одного подарка, и определить, сколько денег останется. Наоборот, если нужно купить N подарков, то найти наибольшую стоимость одного подарка и количество оставшихся денег.

Приведем решение задачи в целых числах.

```
#include <stdio.h>
void main(void)
{
   int Sum;          // выделена сумма денег
   int Cost;         // стоит один подарок
   int Cou;          // количество, которое можно купить на
эту сумму
   int N;            // количество, которое хочется купить
   int Rest;         // останется денег
   printf("Введите сумму и цену одного подарка\n");
   scanf("%d%d", &Sum, &Cost);
   // найдем количество и остаток:
   Cou=Sum/Cost;     // целочисленное деление
   Rest=Sum%Cost;    // остаток от деления
   printf("Можно купить %d подарков, останется %d рублей.
                                                \n",Cou,Rest);
   // найдем стоимость и остаток
```

```
  printf("Сколько подарков хочется купить?\n");
  scanf("%d", &N);
  Cost=Sum/N;
  Rest=Sum % N);
  printf("Наибольшая стоимость подарка %d\n",Cost);
  printf("Останется %d рублей.\n", Rest);
} // End of main
```

Замечание. Операции целочисленного деления и вычисления остатка от деления можно применять только к данным целых типов. Если такие операции нужно выполнить для данных вещественных типов, используются функции ceil, floor, fmod библиотеки <math.h>. Функция ceil округляет вещественное число вверх, функция floor округляет вниз, а функция fmod находит остаток от деления по модулю. Все функции имеют аргументы типа double и возвращают значение типа double. Приведение типа при переходе от float к double необязательно, так как не приведет к потере данных.

Рассмотрим тот же пример с использованием вещественного типа данных. Все денежные суммы должны вычисляться в рублях и копейках, следовательно, переменные Sum, Cost и Rest имеют вещественный тип. При вычислениях необходимо округление вниз для получения целочисленного значения и округление с точностью до копеек. Деление должно быть по модулю.

Теперь покажем решение этой же задачи в вещественных числах.

```
#include <stdio.h>
#include <math.h>
void main(void)
{
  float Sum; // выделена сумма денег
  float Cost; // стоит один подарок
  int Cou;    // количество, которое можно купить на эту сумму
  int N;      // количество, которое хочется купить
  float Rest; // останется денег
  printf("Введите сумму и цену одного подарка\n");
  scanf("%f%f", &Sum, &Cost);
  // найдем количество и остаток
```

```
  Cou=(int)floor(Sum/Cost);
  // целое число получено округлением вниз
  Rest=fmod(Sum, Cost); // остаток от деления.
  printf("Купили %d подарков, останется %6.2f рублей.\n",
        Cou, Rest);
  // найдем стоимость и остаток
  printf("Введите число подарков, которое хочется
        купить\n");
  scanf("%d", &N);
  Cost=Sum/(float)N;
  // точность вычислений должна быть 0,01
  Cost=floor(Cost*100)/100.;
  Rest=fmod(Sum, Cost);
  printf("Наибольшая стоимость подарка %6.2f руб.\n",
        Cost);
  printf("Останется %6.2f руб.\n", Rest);
} // End of main
```

Важное замечание. Если этот пример получить как точную копию первого, то хорошо будет работать только первая формула вычислений. Вторая даст ошибку, вызванную наличием погрешности вычислений, которая должна быть менее одной копейки. Для устранения погрешности добавлена строка

```
Cost=floor(Cost*100)/100.;
```

О смысле этой строки предлагается подумать читателю.

Варианты заданий

Задание 1. Даны длины катетов прямоугольного треугольника. Найти его острые углы, вывести в градусной мере с точностью до минут. Указание: функция arctg объявлена в библиотеке <math.h>.

Задание 2. Даны длины катетов прямоугольного треугольника. Найти его периметр и площадь. При выводе округлить до двух знаков.

Задание 3. Дан положительный угол в радианах. Найти, сколько градусов и минут содержит данный угол.

Задание 4. Дано вещественное число. Найти и напечатать его целую, дробную части и округленное значение. Использовать функции округления.

Задание 5. Камень бросили вверх со скоростью V. Определить расстояние от земли в некоторые моменты времени t_1 и t_2. Вывести с точностью 2 знака.

Задание 6. Даны координаты точек $A(x_1, y_1)$ и $B(x_2, y_2)$ гиперболы $y = k/x + b$. Найти и напечатать значения k, b.

Задание 7. От полуночи минутная стрелка описала угол K градусов (целочисленное значение). Определить, какое время (ч, мин) показывают часы. Указание: использовать операции целочисленного деления.

Задание 8. Даны значения переменных A и B. Найти частное от деления A на B и частное от деления B на A. Найти остаток от деления A на B и остаток от деления B на A. Найти вещественные значения дробей A/B и B/A.

Задание 9. Даны координаты левой верхней и правой нижней вершин квадрата. Найти длину стороны квадрата, найти длину диагонали, найти координаты точки пересечения диагоналей квадрата.

Задание 10. Две прямые на плоскости проходят через начало координат и точки $A(x_1, y_1)$ и $B(x_2, y_2)$. Найти в градусной мере с точностью до градусов угол наклона биссектрисы угла, образованного прямыми.

Задание 11. Дано натуральное четырехзначное число. Найти сумму его цифр. Указание: использовать операции целочисленного деления.

Задание 12. Определить, как наименьшим числом купюр можно выдать сумму K руб. ($K < 9999$), если есть купюры достоинством 1000 руб., 500 руб., 100 руб., 50 руб. и 10 руб. Оставшуюся мелочь выдать рублями.

Задание 13. Камень бросили в колодец, через t сек послышался всплеск. Определить расстояние от сруба до воды.

Задание 14. Две прямые на плоскости заданы своими уравнениями. Известно, что они пересекаются. Найти координаты точки пересечения заданных прямых.

Задание 15. На плоскости задан треугольник координатами своих вершин. Найти длины сторон треугольника.

Задание 16. На плоскости заданы два луча, выходящие из начала координат под углами λ и β к оси $0x$. Найти в градусной мере с точностью до градусов углы наклона

биссектрисы угла, образованного лучами, а также угол между лучами.

Задание 17. Зарплата сотрудника складывается из ставки минимальной зарплаты, умноженной на разрядный коэффициент (вещественное значение в диапазоне от 1 до 3), уральского коэффициента (15%) и премии (в %). Из зарплаты вычитается подоходный налог (12%). Вводя необходимые исходные данные, вычислить и вывести зарплату сотрудника в рублях.

Задание 18. Известно, что прямая, заданная уравнением $y = Kx + b$, пересекает оси координат в точках A и B. Найти координаты точек пересечения, вывести с точностью 2 знака после запятой.

Задание 19. Участок земли треугольной формы имеет длины сторон L_1, L_2, L_3. Найти площадь этого участка, вывести с точностью 2 знака после запятой.

Задание 20. Прямоугольной формы строительный блок имеет ширину w, высоту h и глубину l. Найти объем и площадь поверхности строительного блока.

Задание 21. Прямоугольной формы строительный блок имеет ширину w, высоту h и глубину l. Найти, сколько блоков войдет в квадратный контейнер с длиной стороны L.

Задание 22. В первый день тренировок спортсмен пробежал 10 км. Каждый день он пробегает на 10% больше, чем в предыдущий день. Найти и вывести на экран путь, пройденный спортсменом во второй, третий и четвертый дни тренировок.

Задание 23. В пачке N листов бумаги размера А4 плотностью P г/см^2. Найти вес пачки бумаги с точностью 2 знака после запятой.

Задание 24. Известны стоимость конфет, печенья и яблок. Найти общую стоимость покупки, если куплено X кг конфет, Y кг печенья и Z кг яблок.

Задание 25. Известны стоимость системного блока, монитора, клавиатуры и мыши. Найти, во сколько обойдется покупка N компьютеров.

Задание 26. Известно, что для кормления одного лабораторного животного нужно K_1 г корма для мыши, K_2 г

корма для крысы, K_3 г корма для морской свинки. Найти, сколько съедят в день X мышей, Y крыс, Z морских свинок. Найти, сколько корма нужно в месяц.

Задание 27. В подарке 2 шоколадки по цене K_1 руб., 2 яблока по K_2 руб. и коробка печенья стоимостью K_3 руб. Найти, сколько будут стоить N подарков.

Задание 28. В очереди к врачу N человек. Врач беседует с пациентом примерно 15 мин. Если сейчас 12 ч, а до бассейна бежать 30 мин, стоит ли занимать очередь, если надо успеть на сеанс в X ч (сеансы начинаются каждый час)?

Задание 29. Одна жемчужина диаметром d стоит $Cost$ руб. Сколько будет стоить ожерелье на шею, обхват которой D см?

Задание 30. Летит стая гусей, а навстречу ворона. Ворона говорит: «Как много вас, гусей, наверное, N?», на что вожак отвечает: «Если взять нас столько, и еще пол столько, и четверть столько, то будет N». Сколько же гусей в стае? Иметь в виду, что гуси и вороны считают не очень хорошо, и N не может быть любым.

ТЕМА 2. ИСПОЛЬЗОВАНИЕ УСЛОВНОГО ОПЕРАТОРА IF И ПЕРЕКЛЮЧАТЕЛЯ SWITCH

Условный оператор if изменяет последовательность выполнения операторов программы в зависимости от условий, сложившихся при ее выполнении. Классическая схема оператора if в C++ предполагает разветвление на две ветки, одна из которых может быть пустой. В более сложных случаях, когда проверяется множество условий, используются сложные (вложенные) условные операторы или сложные логические выражения.

Оператор переключатель switch используется для тех же целей, но позволяет организовать множественное ветвление. Если оператор if может передать управление только на две ветви, то switch на произвольное число ветвей.

Пример 2.1. Требуется найти корни квадратного уравнения вида $ax^2 + bx + c = 0$. Выполним этапы решения.

1. Анализ условия задачи. Квадратное уравнение может иметь одно или два решения или не иметь их вообще. Результат решения зависит от определителя уравнения, вычисляемого по формуле $d = b^2 - 4ac$.

2. Входными данными являются коэффициенты уравнения a, b, c. Чтобы уравнение осталось квадратным, первый коэффициент должен быть отличен от нуля. Имена переменных, обозначающих коэффициенты, могут совпадать с их математической записью, т. е. a, b, c. Тип коэффициентов вещественный.

3. Выходными данными программы будут значения корней уравнения, если их удастся найти по формулам $x_{1,2} = \frac{b^2 \pm \sqrt{d}}{2 \cdot a}$. Если же корней нет, об этом следует напечатать сообщение.

4. Для вычисления и хранения значения определителя требуется ввести новую переменную, которая не является результатом решения задачи, но необходима для ее решения. Такие переменные называются рабочими переменными программы. Обозначим ее буквой d, как и в постановке задачи.

5. Определение необходимых формул вычисления очевидно.

6. Кодирование алгоритма. При выполнении алгоритма можно выполнить предварительное словесное описание:

- ввести значения коэффициентов (названы a, b, c);
- вычислить дискриминант (назван d);
- если дискриминант отрицательный, решения нет, вывести сообщение;
- если дискриминант больше или равен 0, решение есть, вычислить значения корней и вывести на печать.

В первой ветви условного оператора нужно выполнить не одно, а три действия, поэтому используется блок {...}, который показывает компилятору, что эти действия следует воспринимать как единое целое.

```
#include <stdio.h>
#include <math.h>
void main(void)
```

```
{
   float a, b, c, d;
   float x1, x2;
   // ввод входных данных
   printf("\nВведите коэффициенты квадратного
уравнения.\n");
   scanf("%f%f%f", &a, &b, &c);
   d=b*b–4*a*c;                    // вычисление дискриминанта d
   if(d>=0.0)
      {
         x1=(–b+sqrt(d))/(2.*a); // вычисление корней
         x2=(–b–sqrt(d))/(2.*a);
         printf("Корни равны x1=%6.2f x2=%6.2f\n", x1, x2);
      }
   else
         printf("Корней нет\n"); // печать сообщения
} // End of main
```

В этом примере не предусмотрен случай равных корней, например, при $a = 2$, $b = 4$, $c = 2$. Для того чтобы отследить этот случай, потребуется внести изменение в текст программы.

Пример 2.2. Когда для принятия решения требуется проверка более одного условия, появляется множественное ветвление. В данном примере следует предусмотреть случай равных корней, при этом в условном операторе появляется второе условие $d = 0$, а значит, и второй оператор проверки условия.

Схема алгоритма условно может быть изображена так:

```
if(d>0)
{
   // блок операторов, чтобы вычислить два корня
}
else
if(d==0)
{
   // блок операторов, чтобы вычислить один корень
}
   else
```

```
  {
    // вывод сообщения о том, что решения нет
  }
```

Структура ветвления рассматривает только взаимоисключающие варианты. Выполнение оператора if при использовании многих условий, если они не выделены блоком, разворачивается по принципу матрешки, т. е. каждый else принадлежит ближайшему сверху if. Отступы в тексте программы позволяют лучше увидеть логику выбора решения.

```
#include <stdio.h>
#include <math.h>
void main(void)
{
  float a, b, c, d;
  float x1, x2;
  printf("\nВведите коэффициенты квадратного уравнения\n");
  scanf("%f%f%f", &a, &b, &c);
  d=b*b–4*a*c;
  if(d>0.0)
  {
      x1=(–b+sqrt(d))/(2.*a);
      x2=(–b–sqrt(d))/(2.*a);
      printf("Корни равны x1=%6.2f x2=%6.2f\n", x1, x2);
  }
    else
    // d имеет вещественное значение,
    // сравнивать его с нулем можно только
    // с некоторой степенью точности, например, так:
    // |d|<0.00001
    if(fabs(d)<=0.00001) // приблизительно равен нулю
    {
      x1=(–b+sqrt(d))/(2.*a);
      printf("Корни одинаковы x1=x2=%6.2f\n", x1);
    }
    else
      printf("Корней нет\n");
} // End of main
```

Пример 2.3. Диалог с программой в общепринятом виде. Для полного тестирования программы, алгоритм которой

проверяет множество условий, нужно выполнить столько примеров, сколько вариантов ветвления возможно. В программе примера 2.2 нужно выполнить три тестовых примера. Кроме того, пользователь с помощью программы может решать несколько уравнений. Поэтому необходимо уметь организовать многократное обращение к программе.

Для организации повторного выполнения программы или ее фрагмента используется циклический алгоритм. Этот цикл будет управляться внешним событием. Событие порождает пользователь, нажимая клавишу на клавиатуре. Общепринято использовать клавишу [Esc] для завершения процесса, а для продолжения клавишу [Enter] или любую другую.

Кроме того, при вводе данных необходимо предусмотреть, чтобы введенное значение коэффициента a было бы отлично от 0. Проверка выполняется в цикле сразу же после ввода данных. Цикл будет повторен всегда, когда введенное значение не соответствует правилам, и только при правильных данных программа продолжит работу.

```
// пример диалога в общепринятом виде
// для управления используется оператор do...while
// с выходом по условию "нажата клавиша [Esc]".
#include <stdio.h>
#include <math.h>
#include <conio.h> // библиотека консольного ввода-вывода
#define ESC 27      // код клавиши [Esc] в символьном виде
void main(void)
{
  float a, b, c, d;
  float x1, x2;
  char key;          // переменная key для обработки события
  // первое выполнение алгоритма обязательно,
  // поэтому необходим цикл do...while
  do
  {
    // проверка корректности вводимых данных
    // если значение a близко к нулю, цикл ввода повторяется
    do
    {
```

```
        printf("\nВведите коэффициенты квадратного
        уравнения\n");
        scanf("%f%f%f", &a, &b, &c);
    } while(fabs(a)<0.001)
    // ввод выполнен правильно
    d=b*b–4*a*c;
    if(fabs(d)<>0.00001)
    {
        x1=(–b+sqrt(d))/(2.*a);
        printf("Корни одинаковы x1=%6.2f x2=%6.2f\n", x1, x2);
    }
    else
        if(d>0.0)
        {
            x1=(–b+sqrt(d))/(2.*a);
            x2=(–b–sqrt(d))/(2.*a);
            printf("Корни равны x1=%6.2f x2=%6.2f\n", x1, x2);
        }
    else
        printf("Корней нет\n");
    // обработка события "нажатие клавиши"
    printf("Клавиша [ESC] – завершение работы,
            Any Key – продолжение…");
    // ожидание события "нажатие клавиши"
    key=getch();     // функция getch() читает код клавиши
  } while(key!=ESC )
  // код клавиши Esc прописан в директиве define
} // End of main
```

Пример 2.4. Использование вложенных условных операторов в программах с проверкой многих условий. Примером сложного выбора является оценка знаний ученика по результатам тестирования. Пусть известен результат ЕГЭ (от 0 до 100 баллов). Оценка по пятибалльной шкале должна быть выставлена по правилам:

$$\text{Оценка} = \begin{cases} 2, \text{если} & \text{ЕГЭ} \le 40; \\ 3, \text{если} & 40 < \text{ЕГЭ} \le 60; \\ 4, \text{если} & 60 < \text{ЕГЭ} \le 80; \\ 5, \text{если} & 80 < \text{ЕГЭ} \le 100. \end{cases}$$

В записи этих правил есть избыточность с точки зрения механизмов выполнения условного оператора. Ветвление — это взаимоисключающие варианты, значит, в первой ветви истинность условия ЕГЭ <= 40 означает получение результата. Если же это условие ложно, то к проверке второго условия 40 < ЕГЭ <= 60 приходят только те значения ЕГЭ, которые не удовлетворяют первому условию, т. е. только ЕГЭ > 40, а значит их не нужно включать в запись условия. Так же и в записи всех последующих условий. Логика вложенных операторов условия присоединяет каждый else к ближайшему сверху if.

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
  int Test;                  // ЕГЭ
  int Ball;                  // оценка
  do
  {
    printf("Введите результат ЕГЭ испытуемого\n");
    scanf("%d", &Test);
    if(Test<=40)
      Ball=2;
    else
      if(Test<=60)
          Ball=3;
      else
        if(Test<=80)
            Ball=4;
      else
              Ball=5;
    printf("Оценка в аттестате: %d\n", Ball);
    printf("Если больше нет испытуемых, нажмите ESC\n");
  } while(getch()!=27);
} // End of main
```

Обратите внимание, что цикл по-прежнему управляется событием, но в этом примере вспомогательная переменная key не используется.

Пример 2.5. Использование сложных логических выражений в программах с проверкой многих условий.

Использование множественных вложений операторов условия затрудняет «читабельность» программы. Зачастую его можно избежать, объединяя простые условия в логические выражения везде, где для поиска решения требуется проверка более одного условия. Пусть требуется проверить, принадлежит ли точка с произвольными координатами (x, y) некоторой области, например, заштрихованной области, изображенной на рисунке 3.1.

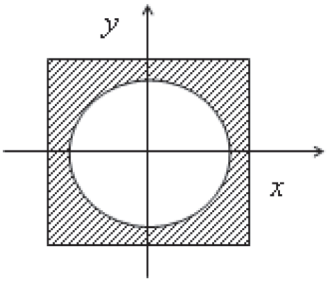


Рис. 3.1
Иллюстрация к примеру 2.5

Известны длина стороны квадрата (обозначим L) и радиус окружности (обозначим R), где не обязательно L>R, т. е. область может быть пустой.

Запишем условие математически с помощью системы неравенств:

$$x^2 + y^2 \geq R^2$$

— условие «точка находится вне окружности»;

$$\begin{cases} |x| < \dfrac{L}{2}, \\ |y| < \dfrac{L}{2} \end{cases}$$

— условие «точка находится внутри квадрата».

Для соблюдения условия нужно, чтобы все три неравенства выполнялись одновременно, поэтому при объединении их в одно требуется использовать логическую операцию «И» (конъюнкция) следующим образом:

```
x*x+y*y>=R*R && fabs(x)<0.5*L && fabs(y)<0.5*L
```

Тогда условный оператор запишется:

```
if(x*x+y*y>=R*R && fabs(x)<0.5*L && fabs(y)<0.5*L)
  printf("Точка принадлежит области \n");
```

```
else
  printf("Точка не принадлежит области \n");
```

Если не использовать сложное выражение, то приходится применять вложенные операторы условия следующим образом:

```
if(x*x+y*y>=R*R)
  if(fabs(x)<0.5*L)
    if(fabs(y)<0.5*L)
      printf("Точка принадлежит области \n");
    else
      printf("Точка не принадлежит области \n");
```

Квалифицированный программист отдаст предпочтение первому способу.

Пример 2.6. Использование оператора переключателя switch для организации множественного ветвления. Оператор switch использует для управления значение выражения целочисленного (символьного) типа, и позволяет выбрать один из нескольких возможных вариантов ветвления алгоритма. Покажем, как можно использовать этот оператор для управления алгоритмом с консоли, например, при организации меню пользователя. Значение кода нажатой пользователем клавиши управляет выбором ветви алгоритма. Код клавиши возвращает функция getch(). Если нажата одна из функциональных (управляющих) клавиш, то функция возвращает 0 (0xE0). Ее повторный вызов получает расширенный код клавиши.

```
void main(void)
{
  int key;
  do
  {
    printf("Выберите действие\n");
    key=getch();
    if(key==0)        // нажата управляющая клавиша
    {
      key=getch();
      // ввод не повторяется, символ получен
      // из буфера ввода
      switch key    // значение key управляет ветвлением
```

```
        {
          case 77: {printf("Стрелка вправо\n"; break;)}
          case 75: {printf("Стрелка влево\n"; break;)}
          case 72: {printf("Стрелка вверх\n"; break;)}
          case 80: {printf("Стрелка вниз\n"; break;)}
          case 27: {printf("Esc\n"; break;)}
          default: {printf("Не стрелка\n"; }
        }
      }
    } while(key!=27);  // выход из цикла по нажатию Esc.
} // End of main
```

Оператор break в каждой ветви передает управление на оператор, следующий за switch. Если break опущен, то после ветви, на которую пал выбор, выполняются все операторы, стоящие далее в тексте switch.

Варианты заданий

Задание 1. Прямая L, заданная на плоскости координатами двух точек (a, b) и (c, d), разбивает координатную плоскость на две полуплоскости. Известны также координаты двух точек (x_1, y_1) и (x_2, y_2), не лежащих на данной прямой.

Определить, находятся ли точки в разных полуплоскостях или в одной, и напечатать сообщение. Иметь возможность повторного обращения в диалоге.

Задание 2. Составить программу для определения корней системы двух линейных алгебраических уравнений по правилу Крамера. Решение вывести в виде системы уравнений. Иметь возможность повторного обращения в диалоге.

Задание 3. Дано натуральное число N, в записи которого ровно пять знаков.

Определить, является ли это число палиндромом или нет, напечатать сообщение текстом. Значение числа вводить в диалоге, иметь возможность повторного обращения.

Задание 4. Дано натуральное число N, определяющее возраст человека в годах.

Дать для этого числа наименование «*год*», «*года*» или «*лет*». Например, «*Вам* 21 *год*» или «*Вам* 43 *года*». Иметь возможность повторного обращения в диалоге.

Задание 5. Окружность на плоскости с центром в начале координат имеет радиус R. Известны также координаты концов некоторого отрезка (x_1, y_1) и (x_2, y_2).

Определить, пересекает ли отрезок окружность и сколько раз. Значения координат вводить в диалоге, иметь возможность повторного обращения.

Задание 6. Даны длины трех отрезков.

Определить, можно ли построить треугольник с такими длинами сторон. Если да, то определить, какой это треугольник: прямоугольный, остроугольный или тупоугольный. Значения длин вводить в диалоге, иметь возможность повторного обращения.

Задание 7. Заданы декартовы координаты точки на плоскости.

Перевести в полярные координаты с учетом номера четверти, где находится точка. Угол перевести в градусную меру с точностью до минут. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 8. По введенным координатам точки (x, y) определить номер четверти координатной плоскости, где находится точка.

Значения координат вводить в диалоге, иметь возможность повторного обращения.

Задание 9. На плоскости заданы три точки своими координатами.

Определить расстояния от точек до начала координат и напечатать, какая из точек расположена ближе к началу координат. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 10. Дано натуральное число N, в записи которого ровно пять знаков.

Определить, имеет ли это число одинаковые цифры или нет. Значение числа вводить в диалоге, иметь возможность повторного обращения.

Задание 11. Окружность с центром в начале координат имеет радиус 1.

Определить, пересекает ли прямая $y = kx + b$ окружность или хотя бы касается один раз. Значения k и b вводить в диалоге, иметь возможность повторного обращения.

Задание 12. На плоскости заданы два квадрата координатами левого верхнего угла и длинами сторон.

Определить, пересекаются ли они. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 13. На плоскости задано кольцо с центром в начале координат и радиусами r_1 и r_2, где $r_1 < r_2$. Дана точка своими координатами (x, y).

Определить, находится ли точка внутри кольца. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 14. Прямоугольной формы кирпич имеет стороны A, B, C.

Определить, пройдет ли кирпич в прямоугольное отверстие размером 5×8. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 15. Даны четыре числа.

Определить, являются ли они элементами арифметической прогрессии. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 16. На плоскости задан треугольник длинами своих сторон.

Вычислить его медианы:

$$ma = 0.5 \times \sqrt{2 \times b^2 + 2 \times c^2 - a^2};$$
$$mb = 0.5 \times \sqrt{2 \times a^2 + 2 \times c^2 - b^2};$$
$$mc = 0.5 \times \sqrt{2 \times b^2 + 2 \times a^2 - c^2}$$

и найти наибольшую. Значения длин сторон вводить в диалоге, иметь возможность повторного обращения.

Задание 17. Прямоугольной формы контейнер имеет размеры 8×8×12, где 12 — высота контейнера. Дано K прямоугольных блоков размером $n \times n \times 2n$.

Определить, войдут ли эти блоки в контейнер. Если не войдут, то определить, сколько останется. Размер блоков вводить в диалоге, иметь возможность повторного обращения.

Задание 18. На плоскости заданы три точки своими координатами.

Определить длины сторон охватывающего их прямоугольника наименьшего размера. Пусть его стороны парал-

лельны координатным осям. Значения координат вводить в диалоге, иметь возможность повторного обращения.

Задание 19. На плоскости задан квадрат координатами левого верхнего угла и длиной стороны. Задан также отрезок координатами концов.

Определить, находится ли отрезок полностью внутри квадрата или один его конец внутри, или отрезок полностью снаружи. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 20. На плоскости задан треугольник координатами своих вершин.

Найти наибольшую из сторон треугольника. Значения координат вводить в диалоге, иметь возможность повторного обращения.

Задание 21. Дано цилиндрическое ведро радиусом r и высотой h. Требуется с его помощью переместить K л жидкости из одной емкости в другую.

Определить, можно ли это сделать за одно действие. Если нет, то определить, сколько раз нужно воспользоваться ведром. Размеры вводить в диалоге, иметь возможность повторного обращения.

Задание 22. На плоскости заданы три точки своими координатами.

Определить, сколько из них и какие находятся внутри окружности, для которой известны радиус и координаты центра. Все значения вводить в диалоге, иметь возможность повторного обращения.

Задание 23. Даны прямоугольная коробка размером $w \times w \times h_1$ и цилиндрическое ведерко радиуса основания r и высотой h_2.

Найти, какая емкость больше по объему. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 24. Дано числовое значение денежной суммы не более 100 рублей.

Вывести значение числа прописью, например, «2 рубля», «12 рублей», «51 рубль». Анализировать остаток от деления на 10. Так, для остатка равного 1 наименование «рубль», для остатка равного 2, 3, 4 наименование

«рубля», для остатка, равного 5, 6, 7, 8, 9, 0 наименование «рублей». Числа второго десятка — исключение. Значение денежной суммы вводить в диалоге, иметь возможность повторного обращения.

Задание 25. У каждого из трех братьев есть некоторая сумма денег. Они решили поделить их поровну.

Определить, кто из них и кому должен передать какую сумму. Значение сумм вводить в диалоге, иметь возможность повторного обращения.

Задание 26. В продаже есть два вида системных блоков по цене K_1 и K_2 руб., и три вида мониторов по цене M_1, M_2 и M_3 руб. Известны также стоимость клавиатуры и мыши.

Найти, сколько будет стоить самый дешевый компьютер и сколько самый дорогой. Значения стоимостей вводить в диалоге, иметь возможность повторного обращения.

Задание 27. При кормлении одного лабораторного животного нужно в день K_1 калорий для мыши, K_2 калорий для крысы. Есть корма калорийностью P_1 по цене X_1 руб., P_2 по цене X_2 руб. и P_3 по цене X_3 руб. за кг.

Подобрать самый дешевый дневной рацион для X мышей и Y крыс. Значения данных вводить в диалоге, иметь возможность повторного обращения.

Задание 28. Для одного подарка выделено K руб. В подарок можно положить яблоки по цене K_1 руб., апельсины по цене K_2 руб. и печенье по цене K_3 руб. за штуку. В подарок каждый из предметов должен войти хотя бы один раз. Если получается, то по два одинаковых или по три.

Найти состав подарка. Найти сумму, которая останется от покупки. Значения данных вводить в диалоге, иметь возможность повторного обращения.

Задание 29. Даны два интервала числовой оси $[a_1, b_1]$ и $[a_2, b_2]$.

Найти соотношение интервалов: пересекаются или нет, первый принадлежит второму или наоборот. Значения вводить в диалоге, иметь возможность повторного обращения.

Задание 30. Окружность на плоскости с центром в начале координат имеет радиус 1. Известны координаты левого верхнего угла квадрата со стороной 1.

Определить, принадлежит ли окружности хотя бы одна вершина квадрата. Значения вводить в диалоге, иметь возможность повторного обращения.

ТЕМА 3. ИНСТРУМЕНТЫ C++ ДЛЯ РЕАЛИЗАЦИИ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ

Если какой-либо фрагмент алгоритма должен быть выполнен многократно, то это циклический алгоритм (цикл).

Циклические алгоритмы можно условно разделить на две группы.

1. Арифметический цикл, у которого заранее известно число повторений.

2. Итерационный цикл, у которого заранее неизвестно число повторений.

Управление циклом выполняет некоторая переменная величина, которая называется «параметр цикла» или «управляющая переменная». Это переменная программы, которая, как правило, изменяется в теле цикла, определяет число повторений цикла и позволяет вовремя завершить его работу.

Можно выделить четыре составные части цикла.

1. Подготовка цикла: присваивание начальных значений переменным, в том числе параметру цикла.

2. Тело цикла: фрагмент, который должен быть повторен многократно.

3. Изменение параметра цикла: как правило, выполняется в теле цикла.

4. Проверка условия завершения цикла: в проверке условия, явно или нет, используется параметр цикла.

Не всегда эти составляющие присутствуют явным образом.

В C++ существуют три вида операторов цикла, которые одинаково можно применять для организации любого циклического алгоритма.

1. while:
```
while (условие)
{   // количество повторений любое
   Тело цикла
}
```

2. do...while:
```
do
{   // количество повторений любое
   Тело цикла
}
while (условие);
```

3. for:
```
(объявление параметра цикла)
{   // количество повторений фиксировано
   Тело цикла
}
```

Пример 3.1. Алгоритм построения таблиц значений различных функций. Это чаще всего арифметический цикл. Обычно параметром цикла является аргумент функции. Для функции задана формула вычисления значения $y(x) = F(x)$. Известны диапазон изменения аргумента $x \in [x_0, x_n]$ и шаг изменения Δx.

Общая схема этого алгоритма на основе цикла while... do выглядит так:

```
// печать заголовка таблицы
x = x0;              // подготовка цикла
while(x<=xn)         // проверка условия завершения
{
   y=F(x);           // алгоритм вычисления значения
                     // вывод строки таблицы
   x+=Δx;            // приращение управляющей переменной
}                    // выход из цикла
```

Общая схема этого алгоритма на основе цикла do... while выглядит так:

```
// печать заголовка таблицы
x = x0;              // подготовка цикла
do
```

```
{
   y=F(x);          // алгоритм вычисления значения
                    // вывод строки таблицы
   x += Δx;         // приращение управляющей переменной
}
while(x<=x_n);      // проверка условия завершения
```

Общая схема этого алгоритма на основе цикла for выглядит так:

```
// печать заголовка
for(x=x_0; x<=x_n; x+=Δx)
// все составляющие цикла в заголовке
{
   y=F(x);                  // алгоритм вычисления значения
                            // вывод строки таблицы
}
```

Пример 3.2. Функция $F(x)$ может быть достаточно сложной, тогда в теле цикла нужно позаботиться о правильной записи блока, вычисляющего функцию. Пусть для $x \in [-\pi/2; +\pi/2]$ требуется вычислить таблицу значений функции, имеющей разрыв в точках $|x| = \pi/4$, по формуле

$$y(x) = \begin{cases} \sin x & \text{для } |x| < \dfrac{\pi}{4}; \\ \cos x & \text{для } |x| >= \dfrac{\pi}{4}. \end{cases}$$

```
#include <stdio.h>
#include <math.h>
// в библиотеке math.h определена константа
// M_PI – значение числа π
void main(void)
{
   float x, y;
   // аргумент и значение функции
   // вывод шапки таблицы в текстовом режиме экрана
   printf("\n Таблица значений функции\n");
   printf("-----------------------------------\n");
   printf("                x          y          \n");
```

```
  printf("-----------------------------------\n");
  // x – параметр цикла, в заголовке цикла for
  // задано полное управление
  for(x=–M_PI/2.; x<=M_PI/2.; x+=0.2)
  // цикл вычисления таблицы
  {
    // блок вычисления значения функции использует
оператор if
    // if(fabs(x)<=M_PI/4.)
      // логическая запись формулы вычисления
      y=sin(x);
    else
      y=cos(x);
    printf("%11.2f %11.2f\n" ,x, y);
    // вывод строки таблицы
  }
} // End of main
```

Пример 3.3. Итерационный цикл. Если в формулировке задачи явно или неявно присутствуют условия «пока не будет выполнено» или «до тех пор, пока не выполнено», то число повторений заранее неизвестно, и это итерационный процесс. Содержание тела цикла может быть произвольным. Для организации таких алгоритмов используются, как правило, циклы while или do... while.

Пример использования цикла, управляемого событием, уже был ранее показан в примерах 2.3 и 2.6 (глава 2). Еще раз покажем, как использовать цикл do... while для проверки корректности ввода. Если программа дружественна пользователю, то при вводе данных она сначала подскажет, что и как следует вводить, затем выполнит проверку введенных значений и в случае ошибки будет возвращать пользователя к этапу ввода данных.

Требуется найти площадь треугольника, построенного по значениям длин трех заданных отрезков. Известно, что треугольник существует только тогда, когда длина каждой его стороны меньше, чем сумма длин двух других сторон. Если введенные данные не удовлетворяют этому условию, ввод повторяется.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main(void)
{
  float a, b, c;              // длины сторон треугольника
  float S;                    // площадь треугольника
  // цикл проверки корректности введенных данных
  do
  //до получения корректных данных
  {
    printf("Введите длины сторон треугольника\n");
    scanf("%f%f%f", &a, &b, &c);
    // тут может появиться ошибка ввода данных
  } while(!(a<b+c&&b<a+c&&c<a+b));
// данные введены корректно
  float pp;
  pp=(a+b+c)/2.;
  S=sqrt(pp*(pp–a)*(pp–b)*(pp–c));
  printf("Площадь треугольника равна: %6.2f\n",S);
} // End of main
```

Пример 3.4. Программист, организуя управление итерационным циклом, должен позаботиться и об инициализации управляющей переменной, и об ее приращении, и об условии завершения цикла.

Сформулируем условие задачи. Мяч брошен вертикально вверх со скоростью V. Требуется построить таблицу зависимости высоты Y от времени t, начиная с момента броска до момента падения мяча на землю, если

$$Y(t) = V \cdot t - \frac{g \cdot t^2}{2},$$

где g — константа тяготения, равная 9,8.

При движении мяч сначала взлетает вверх, затем падает. Высота подъема и время полета мяча зависят от начальной скорости броска. Очевидно, что циклом вычисления должна управлять переменная t. Для нее известно начальное значение ($t = 0$). Шаг изменения можно оценить из физического смысла задачи и выбрать произ-

вольно, например, $t = 0{,}1$ с. Момент завершения вычислений неизвестен в числовом выражении, но известно, что значение высоты сначала будет возрастать, затем убывать и при падении мяча на землю станет равно нулю: $Y <= 0$. Величина Y — прямая функция t, значит, в этом условии переменная t присутствует, но неявно. Поскольку в постановке задачи условие завершения звучит «до момента падения», кажется естественным выбрать цикл do.

```
#include <stdio.h>
#include <conio.h>
#define G 9.8
void main(void)
{
float V, y, t;
printf("Введите значение скорости броска\n");
scanf("%f", &V);
printf("Таблица зависимости высоты от времени Y(t)\n");
printf("-----------------------------------\n");
printf("   t          y(t)       \n");
printf("-----------------------------------\n");
// подготовка цикла. Момент времени t = 0.1
t=0.1;
do                    // выбран цикл do
{
   y=V*t–0.5*G*t*t;
   printf("\t %6.2f \t %6.2f \n", t, y);
   t+=0.1;            // переход на новую итерацию
} while(y>=0);     // в условии t присутствует неявно
getch();              // для задержки таблицы на экране
} // End of main
```

Замечание. Этот вариант имеет недостаток. Последнее значение таблицы будет отрицательным, потому что точного совпадения координаты y с нулем не будет никогда, а цикл do проверяет условие завершения *после* печати строки таблицы. Чтобы избежать этой некорректности, следует выбрать цикл while, в котором отчетливо прописать условие завершения цикла: while(V*t–0.5*G*t*t>=0).

В рассмотренных примерах циклические алгоритмы являются простыми. Сложным (вложенным) циклом

называется такой, у которого содержанием циклически выполняемого фрагмента является также циклический алгоритм. Типы циклических алгоритмов внешнего и внутреннего циклов при этом могут быть любыми. Внутренний цикл, от подготовки до завершения, должен находиться внутри внешнего, точнее, быть его телом.

Пример 3.5. Сложный арифметический цикл. Функция двух переменных $F(x, y)$ или функция с параметром $F(A, x)$ порождает необходимость использования сложного арифметического цикла тогда, когда оба аргумента изменяются. Здесь решается задача полного перебора, т. е. нахождения всех возможных значений функции при всех возможных значениях ее аргументов.

При решении таких задач необходимо выделить обе управляющие переменные, описать закон их изменения и выбрать, какая из них по логике задачи главная или условно главная. Главная переменная будет параметром внешнего цикла, другая переменная будет параметром внутреннего цикла. Для каждого зафиксированного значения главной переменной другая переменная должна пробегать все значения своего диапазона. В сложном арифметическом цикле можно строить таблицу значений функции вида $F(x, y)$ для всех возможных значений x и y. Чтобы значение параметра внешнего цикла не повторялось во многих строках таблицы, его следует выводить однократно во внешнем цикле как заголовок таблицы.

Пусть требуется вычислить объемы нескольких цилиндрических емкостей:

$$V = H \cdot \pi \cdot \left(\frac{d}{2}\right)^2,$$

где H — высота; d — диаметр основания цилиндра.

Высота может изменяться в диапазоне $H \in [-1; 5]$, $\Delta H = 1$, а диаметр в диапазоне $d \in [0.5; 3.5]$, $\Delta d = 0.5$. Переменные равноправны, поэтому внешним циклом может быть любой. Пусть это высота, тогда для одного фиксированного значения высоты нужно выводить таблицу значений функции $F(d)$.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main(void)
{
  float d, h;       // диаметр основания и высота емкости
  float V;          // объем емкости
  // управление внешним циклом, параметр h
  for(h=1.; h<=5.; h+=1.)
  {
  printf("Таблица зависимости объема от диаметра
        основания\n");
  printf("----------------------------\n");
    printf(" Высота = %6.2f\n", h);
    printf("      d          V          \n");
    printf("----------------------------\n");
    // управление внутренним циклом, параметр d
    for(d=0.5; d<=3.5; d+=0.5)
    {
      V=M_PI*pow(0.5*d, 2.);
      printf("%6.2f \t %6.2f \n", d, V);
    }
    getch();      // удержание экрана после вывода таблицы
    }
} // End of main
```

Очистка экрана перед выводом очередной таблицы функцией clrscr() не является необходимой. В этом варианте на очередном экране мы будем видеть одну таблицу для одного значения высоты. Если убрать эту строку, то таблицы будут выводиться подряд одна за другой. Удержание экрана после вывода очередной таблицы обязательно, так как программа выводит данных больше, чем вмещается на один экран.

Пример 3.6. Сложный (вложенный) цикл, управляемый событием. Изменим текст примера 3.3, добавив к нему цикл многократного повторения. Новый внешний цикл просто охватывает весь текст старой программы и позволяет в диалоге управлять ее многократным повторением.

```
#include        <stdio.h>
#include        <conio.h>
#include        <math.h>
void main(void)
{
float a, b, c;      // длины сторон треугольника
float S, pp;        // S – площадь, pp – полупериметр
// цикл, управляющий процедурой
// многократных вычислений
do
{
  // цикл проверки корректности введенных данных
  do
    {
      printf("Введите длины сторон треугольника\n");
      scanf("%f%f%f", &a, &b, &c);
    } while(!(a<b+c&&b<a+c&&c<a+b));
  pp=(a+b+c)/2.;
  S=sqrt(pp*(pp–a)*(pp–b)*(pp–c));
  printf("Площадь треугольника равна: %6.2f\n",S);
  // управление внешним циклом
  printf("Если больше нет треугольников, нажмите ESC\n");
  } while(getch()!=27);
} // End of main
```

Пример 3.7. В сложном цикле типы алгоритмов могут быть произвольными. Оба цикла могут быть одного типа, или внешний цикл может быть итерационным, а внутренний арифметическим, или наоборот. Изменим пример 3.4, использующий итерационный цикл, добавив необходимость исследования решения задачи для различных значений стартовой скорости V. Пусть V может изменяться в диапазоне $V \in [-1; 7]$ с шагом 1. Как видно из постановки задачи, ее решение должно быть повторено многократно по правилам арифметического цикла. Переменная V из простой превращается в управляющую. Текст примера нужно изменить добавлением к нему управления по переменной V во внешнем цикле, при этом запись внутреннего цикла не изменится.

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

```
#define G 9.8
void main(void)
{
   float V, y, t;
   // управление внешним циклом, параметр V
   for(V=1; V<=7.; V+=1.)
   {
   printf("Таблица зависимости высоты от времени
         при V=%6.2f\n", V);
      printf("----------------------------\n");
      printf("       t          y(t)       \n");
      printf("----------------------------\n");
      // управление внутренним циклом, параметр t
      // подготовка цикла. Момент времени t=0.1
      t=0.1;
      while(V*t–0.5*G*t*t>=0)
      {
         y=V*t–0.5*G*t*t;
         printf("\t %6.2f \t %6.2f \n", t, y);
         t+=0.1;    // переход на новую итерацию
      }
   getch();          // для удержания таблицы на экране
   }
} // End of main
```

Варианты заданий

Задание 1. Составить программу, вычисляющую таблицу значений функции

$$y(x)=\begin{cases}\sin x+a, & \text{при } x<0;\\ \cos\pi\cdot x, & \text{при } 0<=x<=2;\\ a^2+x^2, & \text{при } x>2,\end{cases}$$

если $x\in[-4;4]$ с шагом $0{,}3$. Параметр a принимает значения 1.2, 1.3, 1.4, 1.5, 1.6. Для каждого значения параметра построить отдельную таблицу $y(x)$.

Задание 2. Составить программу, вычисляющую таблицу значений функции

$$y(x)=\begin{cases}\dfrac{2\cdot x^3}{x^2+1}, & \text{при } |x|<3;\\ 1.5\cdot\left|\, tg\dfrac{\pi}{x}\,\right|, & \text{для остальных } x,\end{cases}$$

если $x \in [-6; +6]$ с шагом 0.5.

Задание 3. Напечатать в возрастающем порядке все трехзначные числа, в десятичной записи которых нет одинаковых цифр. Иметь возможность повторного обращения в диалоге.

Задание 4. Составить программу, которая находит наибольший общий делитель двух натуральных чисел a и b по следующему алгоритму: до тех пор, пока a и b не сравняются, вычитать $a = a - b$ при $a > b$, или $b = b - a$ при $b > a$. Исходные числа вводить в диалоге, иметь возможность повторного обращения.

Задание 5. Натуральное число является простым, если оно делится на 1 и на самого себя. Натуральное число является совершенным, если оно равно сумме своих делителей, включая 1, например:

$$6 = 1 + 2 + 3;$$
$$28 = 1 + 2 + 4 + 7 + 14.$$

Составить программу, которая определит, является ли некоторое число N простым, а если нет, то является ли оно совершенным. Иметь возможность повторного обращения в диалоге.

Задание 6. Для любого действительного числа x вычислить значение $f(x)$, где f — периодическая функция с периодом $T = 2$, совпадающая на отрезке [0, 1] с функцией $y(x) = x^2 - 2{,}25x$, а на отрезке [1, 2] с функцией $y(x) = x - 1.25$.

Проверить в цикле на интервале $x \in [-4; 4]$ для четного числа точек.

Задание 7. Натуральное число является простым, если оно делится только на 1 и на самого себя. Составить программу, которая найдет все простые числа от N_1 до N_2 включительно. Иметь возможность повторного обращения в диалоге.

Задание 8. Составить программу для проверки знания таблицы умножения для столбца N, где $N = 2, 3, \ldots, 10$ (выбирать в диалоге).

Подсчитать правильные ответы, выставить оценки: «*отлично*», «*хорошо*», «*удовлетворительно*» или «*неудовлетворительно*» за 10, 9, 8, 7 и менее ответов соответственно. Иметь возможность повторного обращения в диалоге.

Задание 9. Составить программу для проверки навыков сложения и вычитания. Программа в диалоге генерирует необходимую процедуру «+» или «−» и два операнда, затем спрашивает сумму или разность. При правильном ответе печатать поощрительный текст и предлагать повторить ввод, иначе печатать сообщение об ошибке и предлагать повторить ввод результата до получения правильного ответа.

Задание 10. Для любого действительного x вычислить значение $f(x)$, где f —периодическая функция с периодом $T = 2.5$, совпадающая на отрезке $[0, 1]$ с функцией $y(x) = x^2$, а на отрезке $[1, 1.5]$ с функцией $y(x) = -x$.

Проверить на интервале $x \in [-3; 5]$, взять не менее 10 точек.

Задание 11. Для любого натурального числа N найти все такие натуральные числа x и y, для которых выполняется $N = x^2 + y^2$. Иметь возможность повторного обращения в диалоге.

Задание 12. Найти все целочисленные координаты точек, попадающих в круг радиуса R с центром в точке (A, B). Иметь возможность повторного обращения в диалоге.

Задание 13. Составить программу для вычисления таблицы значений функции

$$U(y) = 16 + y + y^2,$$

где

$$y(x) = \begin{cases} 5 \cdot x^2 - 0{,}5, & \text{при } x <= 0; \\ 3 \cdot x^2 + 4 \cdot x, & \text{при } 0 < x < 3; \\ x^2 - \dfrac{1}{x}, & \text{при } x >= 3, \end{cases}$$

если $x \in [-2; 4]$, шаг 0.25.

Задание 14. Найти все натуральные числа от N_1 до N_2, запись которых есть палиндром. Иметь возможность повторного обращения в диалоге.

Задание 15. Найти все натуральные числа от N_1 до N_2, запись которых совпадает с последними цифрами записи их квадрата, например: $6^2 = 36$, $25^2 = 625$ и т. д. Иметь возможность повторного обращения в диалоге.

Задание 16. Напечатать в возрастающем порядке все трехзначные числа, в десятичной записи которых есть одинаковые цифры. Иметь возможность повторного обращения в диалоге.

Задание 17. Составить программу, вычисляющую таблицу значений функции

$$Y(x) = \begin{cases} x + a, & \text{при } x < -1; \\ a, & \text{при } -1 <= x <= 1; \\ -x + a, & \text{при } x > 1, \end{cases}$$

если $x \in [-4; +4]$, шаг 0.5; $a \in [1; 5]$, шаг 1. Считать a параметром и строить отдельные таблицы $Y(x)$ для каждого a, печатая его в заголовке таблицы.

Задание 18. Найти все целочисленные степени произвольного числа K от 1 до N включительно. Иметь возможность повторного обращения в диалоге.

Задание 19. На клетчатой бумаге нарисовали окружность целого радиуса R с центром на пересечении линий. Найти количество клеток, целиком лежащих в этой окружности. Например, если $R = 5$, то $K = 60$.

Задание 20. Найти все делители некоторых натуральных чисел в диапазоне от N_1 до N_2. Иметь возможность повторного обращения в диалоге.

Задание 21. Найти разложение произвольного натурального числа на простые множители. Иметь возможность повторного обращения в диалоге.

Задание 22. Найти все числа в диапазоне от N_1 до N_2, которые одновременно кратны m и не кратны n. Иметь возможность повторного обращения в диалоге.

Задание 23. Составить программу для вычисления таблицы значений функции

$$F(x,y)=\begin{cases}1-e^{-(x+y)}, & \text{при } x>0, y>0;\\ x+y, & \text{при } x>0, y<0;\\ \sin^2(x+y), & \text{в остальных случаях.}\end{cases}$$

Здесь $y \in [0; 2.5]$ с шагом 0.5 и $x \in [0; 2.5]$ с шагом 0.5. Построить таблицу на решетке из полного перебора значений x, y, где y выводить в заголовке.

Задание 24. Поле шахматной доски определяется парой значений (вертикаль, горизонталь), где первое значение — буква, а второе — цифра, например, $e2$, $f5$. По двум заданным значениям полей определить, угрожает ли ферзь, стоящий на первом поле, второму полю. Иметь возможность повторного обращения в диалоге.

Задание 25. Числа Фибоначчи образуются по закону

$$f_1 = 1,\ f_2 = 1,\ \ldots,\ f_{k+1} = f_{k-1} + f_k.$$

Найти все числа Фибоначчи, не превышающие N. Иметь возможность повторного обращения в диалоге.

Задание 26. Написать программу, работающую как простой калькулятор, выполняющий операции «+», «−», «*», «/» для двух операндов. Реализовать в диалоге ввод операндов и выбор операции.

Задание 27. Составить программу для вычисления таблицы значений функции

$$F(x,y)=\begin{cases}x^2+y^2, & \text{при } x<0, y<0;\\ \sqrt{x+y}, & \text{при } x>0, y>0;\\ \sqrt{x^2+y^2}, & \text{в остальных случаях.}\end{cases}$$

Здесь $y \in [0; 2.5]$ с шагом 0.5 и $x \in [0; 2.5]$ с шагом 0.5. Построить таблицу на решетке из полного перебора значений x, y, где y выводить в заголовке.

Задание 28. Напечатать в возрастающем порядке все числа в диапазоне от 1 до N, в десятичной записи которых нет одинаковых цифр. Иметь возможность повторного обращения в диалоге.

Задание 29. Найти все натуральные числа в диапазоне от N_1 до N_2, равные сумме кубов своих цифр. Иметь возможность повторного обращения в диалоге.

Задание 30. Составить программу, вычисляющую таблицу значений функции

$$f(x)=\begin{cases}\sin\dfrac{\pi}{x}, & \text{при } |x|>=1;\\ \sin\left((x-1)\cdot\dfrac{\pi}{2}\right), & \text{при } -1<=x<=1,\end{cases}$$

если $x \in [-2; +2]$, шаг 0.2.

ТЕМА 4. АЛГОРИТМЫ ВЫЧИСЛЕНИЯ СУММ, ПРОИЗВЕДЕНИЙ, КОЛИЧЕСТВ, ПРЕДЕЛОВ, ПОСЛЕДОВАТЕЛЬНОСТЕЙ. СЛОЖНЫЕ ЦИКЛЫ

В алгоритмах вычисления сумм, произведений, количеств, пределов, последовательностей особенностью является содержание тела цикла. При вычислении суммы к значению суммы многократно прибавляются новые значения слагаемых. При вычислении произведения значение многократно умножается на очередной сомножитель. При вычислении количеств значение счетчика увеличивается на 1. При вычислении предела или последовательности значение многократно вычисляется на базе предыдущего значения. Итоговое значение, кроме вычисления последовательностей, чаще единственное, так как все остальные вычисленные значения являются промежуточными.

Управление циклами этого вида выполняется также с использованием управляющих переменных, которыми фактически служит номер вычисляемого значения (слагаемого, множителя, элемента последовательности). Если число повторений известно, цикл должен быть арифметическим. В задачах вычисления с указанной точностью цикл должен быть итерационным, так как заранее не может быть известно число повторений, которое понадобится, чтобы достичь заданной точности.

Пример 4.1. Вычисление суммы известного числа слагаемых. Пусть требуется вычислить сумму N чисел натурального ряда

$$S = 1 + 2 + 3 + 4 + \ldots + N,$$

где N — любое наперед заданное число.

Это арифметический цикл, у которого параметром является номер слагаемого, который также определяет и значение очередного слагаемого, включаемого в сумму. Обозначим его буквой n, тогда общая формула тела цикла запишется так $S = S + n$.

Смысл цикличности в том, что к значению суммы многократно прибавляются новые значения слагаемых, обновляя ее. Число повторений цикла равно числу действий сложения, которое нужно выполнить для достижения результата.

Номер слагаемого (и его значение) n меняется в диапазоне от 1 до N с шагом, равным 1.

```
void main (void)
{
  int n;     // управляющая переменная
  int S;     // сумма ряда
  int N;     // число слагаемых, включенное в сумму
  printf("Вычисление суммы чисел натурального ряда.\n");
  printf("Введите число слагаемых>\n ");
  scanf("%d", &N);
  S=0;       // инициализация переменной S нулем обязательна
  n=1;       // к нулю готовимся прибавить первое слагаемое
  do
  {
    S+=n; // тело цикла
    n++;   // приращение параметра цикла
  } while(n<=N);
// печать результата вне цикла
  printf("При %d слагаемых сумма = %d", N, S);
} // End of main
```

Поскольку данный цикл арифметический, использование оператора цикла do... while не является необходимым, но подчеркивает, что любой тип цикла в C++ можно реализовать с помощью любого оператора цикла.

Пример 4.2. Организация итерационного цикла на примере алгоритма суммирования. Пусть требуется найти сумму прогрессии

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} + \ldots$$

с точностью ε (например, $\varepsilon = 0.001$).

Количество слагаемых, которое нужно включить в сумму для достижения заданной точности, неизвестно, но известно условие, определяющее точность вычислений. Предел значения очередного слагаемого стремится к нулю:

$$\lim_{n \to \infty} \frac{1}{n} = 0,$$

поэтому можно считать, что именно это значение определяет требуемую точность вычислений, и можно закончить вычисления, когда очередное слагаемое настолько мало, что им можно пренебречь. Все переменные должны иметь вещественный тип, так как участвуют в вычислении вещественного значения.

```
void main(viod)
{
   float S;
   float eps;     // значение точности вычислений
   float n;       // номер слагаемого, определяет также
                  // его значение, изменяется от 1 с шагом 1
   printf("Вычисление суммы ряда.\n");
   printf("Введите точность вычислений <1\n");
   scanf("%f", &eps);
   n=1;
   S=0;           // входит в подготовку цикла
   do
   {
      S+= 1/n;
      n+= 1;
   } while(1./n>eps);       // eps достаточно мало
   printf("Слагаемых %5.0f, Сумма %f8.5", n, S);
} // End of main
```

В рассмотренных примерах циклические алгоритмы суммирования являются простыми. При организации сложных циклов внутренний цикл полностью, от подготовки до завершения, должен находиться внутри внешнего.

Пример 4.3. Пусть требуется вычислить сумму ряда по формуле

$$S = x + \frac{x}{2} + \frac{x}{3} + \frac{x}{4} + \ldots + \frac{x}{n} + \ldots$$

Очевидно, что значение суммы будет зависеть от значения x. Добавить сложность к этому алгоритму может одно из следующих условий.

1. Вычислить суммы для различного числа слагаемых ($n1$, $n2$, $n3$ и т. д.) в арифметическом цикле или цикле, управляемом событием.

2. Вычислить суммы для различных значений x, например, для $x \in [x_0, x_n]$ с шагом Δx, или вводимых с клавиатуры.

3. Вычислить суммы для различных степеней точности, например, для $\varepsilon \in [\varepsilon_0, \varepsilon_n]$ в итерационном цикле.

Рассмотрим второй случай. Сумма заново вычисляется для каждого значения x, поэтому цикл вычисления суммы должен быть внутренним, а цикл изменения переменной x внешним. Управляющей переменной внутреннего цикла является номер слагаемого $n \in [1, N]$, ее шаг равен 1. Число повторений N должно быть известно, его можно ввести. Внешним циклом управляет переменная x, которая изменяется по закону арифметического цикла, пусть для определенности $x \in [0; 1]$, шаг 0,1.

```
void main(void)
{
  float x;          // параметр внешнего цикла
  float n;          // параметр внутреннего цикла
  float S;
  int N; // число слагаемых, включенных в сумму
  printf("Вычисление суммы чисел натурального ряда.\n");
  printf("Введите число слагаемых>1\n");
  scanf("%d", &N);
  // управление внешним циклом
  for(x=0; x<=1; x+=0.1)
  {
    // тело внешнего цикла
    // управление внутренним циклом
```

```
    S=0;
    n=1;
    while(n<=N)
    {
      // Тело внутреннего цикла.
      S+=x/n;
      n+=1;
    }
    printf("При x=%6.2f сумма = %6.2f\n", x, S);
  }
} // End of main
```

Пример 4.4. Использование рекуррентных соотношений при вычислении последовательностей. Кстати, слагаемые сумм и сомножители произведений фактически тоже образуют вычисляемые последовательности. Часто имеет смысл использовать соотношения, существующие между соседними элементами последовательности, чтобы каждое новое значение вычислять рекуррентно, т. е. исходя из значения, вычисленного на предыдущем шаге, например, $a_n = f(a_{n-1})$.

Используем этот прием для вычисления значения функции $S(x)$ в произвольной точке x по формуле разложения в ряд для 5, 10, 15 и 20 слагаемых, если формула задана:

$$S(x) = x - \frac{x^2}{2!} + \frac{x^4}{3!} - \ldots + (-1)^n \cdot \frac{x^n}{(n)!}.$$

Переменная x произвольна и не задана в условии, значит, ее следует ввести.

Помимо собственно вычисления суммы требуется сравнить вычисленные значения для разного числа слагаемых, для чего нужен сложный цикл, где внешним циклом будет арифметический цикл с управлением по количеству слагаемых, включенных в сумму: $N = 5, 10, 15, 20$, т. е. $N \in [1; 20]$, шаг 5.

Управляющей переменной внешнего цикла является N, в то время как k — управляющая переменная внутреннего цикла.

Внутренний цикл имеет некоторые особенности.

1. При $k = 1$ начальное значение суммы S равно 0, а очередное слагаемое a не определено.

2. Во внутреннем цикле очередное слагаемое вычисляется по формуле $a = \frac{x^n}{n!}$.

Для возведения числа в степень в C++ есть функция pow, а для вычисления факториала функции нет. Факториал, во-первых, растет очень быстро, и для $n = 10$ и более имеет очень большие значения. Во-вторых, для его вычисления нужен циклический алгоритм. Если для вычисления слагаемого использовать рекуррентное соотношение, то мы избавимся от этих двух недостатков.

Вычисление степени x^n — это произведение n сомножителей $x \cdot x \cdot x \cdot \ldots \cdot x$. Вычисление факториала, это произведение чисел натурального ряда, $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$. Если обозначить именем a значение очередного слагаемого, то последующее значение может быть вычислено по отношению к предыдущему по формуле

$$a = a \cdot \frac{x}{k}.$$

Действительно,

$$\text{при } k = 1 \; a = x;$$

$$\text{при } k = 2 \; a = a \cdot \frac{x}{2} = \frac{x^2}{2};$$

$$\text{при } k = 3 \; a = a \cdot \frac{x}{3} = \frac{x^2 \cdot x}{2 \cdot 3} = \frac{x^3}{6}$$

и т. д. Значит, в теле внутреннего цикла для вычисления очередного слагаемого может быть использована именно эта формула.

3. Ряд является знакопеременным, т. е. знак слагаемого отрицательный для четных слагаемых, и положительный для нечетных. Однако нет необходимости в формуле вычислений возводить (-1) в степень. Обычно для решения подобных задач вводится переменная, хранящая знак, например, переменная z принимает значения

+1 или −1. Смена знака в цикле достигается присваиванием $z = -z$ при переходе к очередной итерации.

```
#include        <stdio.h>
#include        <conio.h>
void main(void)
{
   int N;           // управление внешним циклом,
                    // целая переменная
   float x;         // аргумент функции
   float k;         // управление внутренним циклом,
                    // вещественная переменная, участвует
                    // в вычислении
   float a;         // очередное слагаемое
   float S;         // S – сумма
   int z;           // знак слагаемого
   printf("Введите x = \n");
   scanf("%f", &x);
   // управление внешним циклом по числу слагаемых
   for(N=5; N<20; N+=5)
   {
      k=1;          // начинаем с номера 1
      S=0;          // сумма = 0
      a=1;          // для рекуррентного соотношения
      z=1;          // знак слагаемого "плюс" для k = 1
      // управление внутренним циклом
      while(k<=N)
      {
         a=a*x/k;   // очередное слагаемое рекуррентно
         S=S+z*a;   // сложение с учетом знака.
         // переход к следующему слагаемому.
         k+=1;
         z=-z;      // смена знака
      };
      printf("При %4.0f слагаемых сумма = %8.4f\n",k-1, S);
   }
} // End of main
```

Небольшой итог. При вычислении сумм в подготовку цикла входит обнуление суммы. В теле цикла сумма вычисляется многократным прибавлением к значению сум-

мы очередного слагаемого. Итоговое значение одно, оно получено при завершении цикла.

При вычислении произведений в подготовку цикла входит присваивание произведению значения 1. В теле цикла произведение вычисляется многократным умножением значения произведения на очередной сомножитель. Итоговое значение также одно, получено при завершении цикла.

Пример 4.5. Вычисление последовательностей. Классическая задача — это нахождение арифметической или геометрической последовательностей. Для нахождения значения очередного элемента используется один или несколько предыдущих элементов $a_n = f(a_{n-1})$, почти как при использовании рекуррентных соотношений.

Пусть требуется вычислить элементы арифметической прогрессии, если заданы первый элемент, знаменатель и число элементов. Пусть требуется вычислить элементы геометрической прогрессии, если заданы первый элемент и знаменатель. Для возрастающей прогрессии вычислять значения, пока очередной элемент не превысит 1000, для убывающей прогрессии вычислять значения, пока очередной элемент не станет меньше, чем 0,001.

В этом примере еще раз обратим внимание на использование операторов цикла. Для организации арифметического цикла логичнее использовать for, для организации итерационного цикла while или do... while.

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
  float Ar, d_Ar;          // первый элемент и знаменатель
  float Geom, d_Geom;      // первый элемент и знаменатель
  int Cou_Ar;              // число элементов прогрессии

  //----- Задача 1. ---------------------------------
  printf("Введите первый элемент и знаменатель
        прогрессии\n");
  scanf("%f%f", &Ar, &d_Ar);
  printf("Введите число элементов\n");
```

```
   scanf("%d", &Cou_Ar);
   // управление первым циклом по числу слагаемых
   for(int k=2; k<=Cou_Ar; k++)    // номер начинается с 2
   {
      Ar=Ar+d_Ar;
      printf("%d-й элемент равен: %6.2f\n", k, Ar);
   }

//------ Задача 2.---------------------------------
printf("Введите первый элемент и знаменатель
        прогрессии\n");
scanf("%f%f", &Geom, &d_Geom);
if(d_Geom>1)
{
   printf("Возрастающая последовательность\n");
   k=2;
   while(Geom<1000)     // while, так как Geom известно
   {
      Geom=Geom*d_Geom;
      printf("%d-й элемент равен: %6.2f\n", k, Geom);
      k++;
   }
}
   else
      if(d_Geom<1)
      {
      printf("Убывающая последовательность\n");
      k=2;
      while(Geom > 0.001)
      {
         Geom=Geom*d_Geom;
         printf("%d-й элемент равен: %8.4f\n", k, Geom);
         k++;
      }
   }
   else              // здесь знаменатель равен 1
      printf("Все элементы одинаковы\n");
} // End of main
```

Пример 4.6. Использование операторов break и continue для циклов do, while, for. Оператор прерывания break внутри любого цикла прекращает выполнение цикла с передачей управления следующему за циклом оператору. Осуществляя выход из цикла при наступлении каких-либо обстоятельств, позволяет корректно завершить цикл. В случае вложения прерывает только непосредственно охватывающий цикл.

Найдем сумму чисел натурального ряда, не превышающую некоторого заданного значения. Узнаем также, сколько элементов будет включено в сумму.

```
#include <stdio.h>
void main(void)
{
  int Max, Sum=0, k; // Max – наибольшее значение суммы
  printf("\nВведите наибольшее значение\n");
  scanf("%d", &Max);
  for(k=1; ;k++)        // бесконечный цикл
  {
    Sum+=k;
    if(Sum>Max)         // условие завершения цикла
      break;            // прерывание. При выходе известно k
  }
  printf("количество элементов %d", k);
} // End of main
```

Оператор continue внутри любого цикла выполняет переход к следующей итерации тела цикла. Противоположен break. В любой точке цикла прервет текущую итерацию и перейдет к проверке условия завершения цикла.

Найдем сумму слагаемых вида $S = 1/x$ для $x \in [-1; +1]$ с шагом 0,1. Особой точкой в этом ряду является точка $x = 0$, функция в этой точке не определена. При $x = 0$ вычисление выполнять нельзя, значит, это действие следует пропустить.

```
#include <stdio.h>
#include <math.h>
void main (void)
{
  float Sum=0;
```

```
  float x;
  for(x=–1; x<=1.1; x+=0.1)
  {
    if(fabs(x)<0.0001)
      continue; // если в знаменателе 0
    Sum+=1/x;
  }
  printf("Сумма %6.2f", Sum);
} // End of main
```

Пример 4.7. Бесконечный цикл. В тех случаях, когда управление циклом осуществляется не по событиям или условиям извне, а возникает непосредственно в теле цикла, можно использовать бесконечные циклы. Бесконечным он может быть назван только с точки зрения синтаксиса, потому что условие выхода из цикла не прописано непосредственно в управлении циклом. Содержимое тела цикла может быть произвольным, но какие-то действия внутри должны привести к тому, что процесс когда-то закончится. Прерывание и выход из цикла выполняет обычно оператор break.

```
while(1)
{
    ...
  // проверка условия и выход
}
for(; ; ;)
{
    ...
  // проверка условия и выход.
}
```

Варианты заданий

Задание 1. Найти число π, используя произведение

$$\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \times \frac{4}{3} \cdot \frac{4}{5} \times \frac{6}{5} \cdot \frac{6}{7} \times \ldots$$

Точность для печати не менее 5 знаков. Во внешнем цикле выполнить вычисления для 50, 100, 200, 400 сомножителей.

Задание 2. Найти число π, используя формулу суммы ряда

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots + (-1)^{k+1} \cdot \frac{1}{2 \cdot k + 1}.$$

Во внешнем цикле выполнить вычисления для 50, 100, 200 слагаемых. Точность для печати не менее 5 знаков.

Задание 3. Вычислить значение полинома $P(x)$ в произвольной точке x, если

$$P(x) = 1 + x + x^2 + x^3 + \ldots + x^n.$$

Выполнить вычисления для 100 слагаемых. Во внешнем цикле составить таблицу, выводящую на экран значение полинома для $x \in [-0.5; 1{,}1]$.

Задание 4. Вычислить значение функции $Y(x)$ в произвольной точке x по формуле разложения в ряд

$$Y(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}.$$

Величину x вводить. Во внешнем цикле выполнить вычисления для 5, 10, 15, 20, 25 слагаемых. Точность для печати не менее 5 знаков.

Задание 5. Вычислить значение суммы ряда $Y(x)$ в произвольной точке x, если

$$Y(x) = \sin x + \sin^2 x + \sin^3 x + \ldots + \sin^n x.$$

Величину x вводить в градусной мере. Во внешнем цикле выполнить вычисления для 10, 20, 30 слагаемых. Точность для печати не менее 5 знаков.

Задание 6. Вычислить значение функции $Y(x)$ в произвольной точке x по формуле разложения в ряд

$$Y(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + (-1)^n \cdot \frac{x^{2n}}{2n!}.$$

Величину x вводить. Во внешнем цикле выполнить вычисления для 5, 10, 15, 20 слагаемых. Точность для печати не менее 5 знаков.

Задание 7. Вычислить значение функции $Y(x)$ в произвольной точке x по формуле разложения в ряд

$$Y(x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots + (-1)^{n-1} \cdot \frac{x^n}{n}.$$

Величину x вводить. Во внешнем цикле выполнить вычисления для 5, 10, 15, 20 слагаемых. Точность для печати не менее 5 знаков.

Задание 8. Составить программу для вычисления таблицы значений функции

$$Y(x) = x(x - 0{,}5) \cdot (x - 1) \cdot (x - 1{,}5) \ldots (x - 5),$$

для $x \in [0; 0{,}9]$, шаг 0,1. Параметром внутреннего цикла является величина a в сомножителе вида $(x - a)$.

Задание 9. Вычислить значение суммы ряда

$$S = \underbrace{\sqrt{\left(2 + \sqrt{\left(2 + \sqrt{\left(2 + \ldots + \sqrt{2}\right)}\right)}\right)}}_{n \text{ корней}}.$$

Во внешнем цикле выполнить вычисления для n = 5, 10, 15, 20. Один из тестовых примеров проверить вручную. Точность для печати не менее 5 знаков.

Задание 10. Для некоторого натурального числа N найти все целые x, y, такие, что выполняется равенство: $x^2 + y^2 = N$.

Во внешнем цикле значение N вводить в диалоге.

Задание 11. Вычислить значение суммы ряда $Y(x)$ в произвольной точке x, если

$$Y(x) = x \cdot \sin\left(\frac{\pi}{4}\right) + x^2 \cdot \sin\left(2 \cdot \frac{\pi}{4}\right) + \ldots + x^n \cdot \sin\left(n \cdot \frac{\pi}{4}\right).$$

Величину x вводить в градусной мере. Во внешнем цикле выполнить вычисления для 5, 10, 15, 20 слагаемых. Точность для печати не менее 5 знаков.

Задание 12. Найти произведение n сомножителей

$$S = \frac{1}{2} \cdot \frac{3}{4} \cdot \frac{5}{6} \cdot \frac{7}{8} \cdot \ldots$$

Во внешнем цикле выполнить вычисления для 50, 100, 200 сомножителей. Точность для печати не менее 5 знаков.

Задание 13. Вычислить значение функции $Y(x)$ в произвольной точке x по формуле разложения в ряд

$$Y(x) = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \ldots + (-1)^n \cdot \frac{x^{2n}}{(2n)!}.$$

Во внешнем цикле выполнить вычисления для 5, 10, 15, 20, 25 слагаемых.

Задание 14. Составить программу для вычисления таблицы значений функции

$$Y(x) = 1 + 2x + 3x^2 + 4x^3 + \ldots + 20x^{19}$$

для $x \in [0; 1{,}2]$, шаг 0,05.

Задание 15. Составить программу для вычисления таблицы значений функции

$$Y(x) = (x-1) + \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} + \frac{(x-1)^4}{4} + \ldots$$

Во внешнем цикле вычислить функции для 5, 10, 15, 40 слагаемых. Во внешнем цикле задать значение $x \in [0; 1{,}2]$, шаг 0,2.

Задание 16. Вычислить значение функции $Y(x)$ в произвольной точке x по формуле разложения в ряд

$$Y(x) = \cos x + \frac{\cos 2x}{2} + \frac{\cos 3x}{3} + \ldots + \frac{\cos nx}{n}.$$

Значение x ввести в градусной мере. Во внешнем цикле выполнить вычисления для 5, 10, 15, 20 слагаемых. Точность для печати не менее 5 знаков.

Задание 17. Вычислить значение функции $Y(x)$ в произвольной точке x:

$$Y(x) = \frac{(x-2)\cdot(x-4)\cdot(x-8)\cdot\ldots\cdot(x-64)}{(x-1)\cdot(x-3)\cdot(x-7)\cdot\ldots\cdot(x-63)}.$$

Выполнить вычисления для $x \in [-1; 1]$, шаг 0,1.

Задание 18. Вычислить значение суммы

$$S = 1\cdot 2 + 2\cdot 3\cdot 4 + 3\cdot 4\cdot 5\cdot 6 + \ldots + n(n+1)\ldots 2n.$$

Во внешнем цикле вычислить сумму для значений $n = 10, 20, 30, 40, 50$.

Задание 19. Составить программу для вычисления суммы

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + \frac{1}{9999} - \frac{1}{10000}$$

а) последовательно слева направо;

б) вычислить отдельно суммы положительных и отрицательных слагаемых, а затем вторую сумму вычесть из первой.

Если результаты различны, объяснить.

Задание 20. Вычислить значение суммы

$$S = \frac{1}{a} + \frac{1}{a^2} + \frac{1}{a^3} + \ldots + \frac{1}{a^n}.$$

Величину a вводить в диалоге во внешнем цикле. Для каждого a выполнить вычисления при различном числе слагаемых $n = 5, 10, 15, 20$.

Задание 21. Последовательность вычисляется по закону

$$1,\ 1+\frac{1}{2},\ 1+\frac{1}{2}+\frac{1}{3},\ 1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4},\ldots$$

Известно, что существует предел этой последовательности. Найти его с точностью ε знаков после запятой. Величину точности во внешнем цикле последовательно задать равной 0,1; 0,01; 0,001; 0,0001; 0,00001. Выводить в виде таблицы значение предела последовательности и значение числа слагаемых, включенных в сумму.

Задание 22. Для некоторого наперед заданного натурального числа N найти такую наименьшую степень двойки 2^k, которая не превосходит N. Вывести значение k и все степени двойки. Во внешнем цикле значение N вводить в диалоге.

Задание 23. Составить программу для вычисления суммы

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + \frac{1}{9999} - \frac{1}{10000}$$

а) последовательно справа налево;

б) вычислить отдельно суммы положительных и отрицательных слагаемых, а затем вторую сумму вычесть из первой.

Если результаты различны, объяснить.

Задание 24. Составить программу для вычисления произведения n сомножителей

$$P = \frac{1}{2} \cdot \frac{3}{4} \cdot \frac{5}{6} \cdot \ldots \cdot \frac{n-1}{n}$$

а) последовательно справа налево;

б) последовательно слева направо.

Если результаты различны, объяснить. Значение n вводить в диалоге.

Задание 25. Вычислить значение полинома в точке x:

$$Y(x) = x^{10} + 2x^9 + 3x^8 + \ldots + 10x + 11.$$

Составить таблицу значений полинома, где во внешнем цикле переменная x принимает значения $x \in [-2; 2]$ при шаге 0,2.

Задание 26. Вычислить значение полинома в точке x:

$$Y(x) = 11x^{10} + 10x^9 + 9x^8 + \ldots + 2x + 1.$$

Составить таблицу значений полинома, где во внешнем цикле переменная x принимает значения $x \in [-2; 2]$ при шаге 0,2.

Задание 27. Найти все натуральные трехзначные числа, сумма цифр которых равна некоторому натуральному числу N. Подсчитать их количество. Значение N вводить в диалоге.

Задание 28. Вычислить значение суммы ряда $Y(x)$ в произвольной точке:

$$Y(x) = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}.$$

Величину x вводить в диалоге. Во внешнем цикле выполнить вычисления для 5, 10, 15, 20 слагаемых. Точность для печати не менее 5 знаков.

Задание 29. В диапазоне от 1 до некоторого натурального N найти и вывести на печать все числа, которые делятся на 3, на 2, но не делятся на 5. Найти их количество. Величину N вводить в диалоге.

Задание 30. Числа Фибоначчи определяются формулами: $f_0 = f_1 = 1$, $f_2 = 2$, и каждое последующее число равно сумме двух предыдущих.

Составить программу, которая найдет n чисел Фибоначчи, а также сумму всех чисел Фибоначчи, не превосходящую некоторого N, введенного в диалоге.

ТЕМА 5. ИСПОЛЬЗОВАНИЕ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ В РЕШЕНИИ СОДЕРЖАТЕЛЬНЫХ ЗАДАЧ

Наиболее важным в программировании является подготовительный этап, который называют постановкой задачи, тесно связанный с этапом формализации задачи. От правильного их выполнения во многом зависит время и качество программирования.

Постановка задачи обычно заключается в ее словесном описании. Как известно, на вербальном уровне точное определение модели невозможно, поэтому следующий этап — это формализация, т. е. выбор математической или иной модели, адекватно отражающей суть задачи. Точных рекомендаций при выборе модели дать невозможно.

Важным также является выбор структур данных и определение их типов.

До сих пор мы рассматривали уже готовые, формализованные задачи, где математическое описание было дано или лежало на поверхности. Теперь приведем пример неформализованной задачи.

Пример 5.1. Червячок ползет по дереву вверх, стартуя со скоростью V м/ч. Каждый последующий час движения его скорость падает на 10% от предыдущего значения. Если высота дерева H, то за какое время червячок достигнет вершины, если только это возможно?

Для решения задачи нужно выяснить, прежде всего, что выполняется многократно: бежит время, медленно растет пройденный путь, падает скорость. В задаче все как в жизни. Очевидно, что управлять этим циклом должна переменная, обозначающая время, так как все три переменные связаны в формуле вычисления пути при заданной скорости $S = V \cdot t$. Очевидно, что точное условие завершения процесса сформулировать невозможно,

поэтому нужно использовать итерационный цикл с условием завершения $S < H$.

В подготовку цикла входит присваивание начальных значений пути $S = 0$, скорости V и времени $t = 1$. Причем если S и V имеют значения до момента первого вычисления, то $t = 1$ уже готовится к первой итерации. Для убедительности будем выводить на экран значения пути, пройденного к исходу каждого часа, и значение скорости в каждый час времени.

Момент, завершающий цикл, интересен многими особенностями и сильно зависит от входных данных.

1. Первый вариант завершения цикла — червячок сможет добраться до вершины. Условием завершения является $S < H$, при этом в теле цикла оператор break отсекает лишнюю итерацию.

2. Второй вариант — червячок не сможет добраться до конца дерева, так как скорость падает очень быстро, а высота большая. Должна быть выполнена проверка условия $V > 0$ с некоторой степенью точности, так как возможно, что скорость упадет практически до 0, а до конца дерева останется еще очень долгий путь. Кстати, проверка $S < H$ также выполняется приближенно.

```
#include <stdio.h>
void main (void)
{
  float V;    // скорость предыдущего и скорость нового часа
  float S;    // путь
  float H;    // высота дерева
  float t;    // время пути
  printf("Высота дерева и начальная скорость движения\n");
  scanf("%f%f", &H,&V);
  printf(" Таблица зависимости пути от времени.\n");
  printf("---------------------------------\n");
  printf(" Время \tПуть \tСкорость \n");
  printf("---------------------------------\n");
  // в подготовку цикла входит присваивание значений пути,
  // скорости и времени
  S=0.;
  t=1.;
```

```
  while(1)          // бесконечный цикл
  {
    S=S+V*1.;   // пройденный путь S=S+V*t, где t=1.0
    if(S>H)
    {
      printf("Я уже почти добрался до вершины.\n");
      break;      // чтобы исключить лишнюю итерацию
    }
    if(V<=0.01)
    {
      printf("Я никогда не доберусь до вершины.\n");
      break;      // цикл закончен
    }
    printf("%6.0f %8.3f %8.3f \n", t, S, V);
    V*=0.9;       // аналог записи V = V * 0.9;
    t+=1.;
  }
} // End of main
```

Варианты заданий

Задание 1. Валяльная фабрика ежегодно увеличивает объем продаж на 2%, снижая себестоимость продукции на 1%. В текущем году объем продаж составил 500 тыс. руб., а себестоимость пары валенок была равна 55 руб.

Вычислить и вывести на экран таблицу прогнозируемого увеличения объема продаж и снижения себестоимости на ближайшие K лет.

Задание 2. Ученица швеи начинает работу, сострачивая в день 2 пары рукавиц. Совершенствуя свое мастерство, она каждый день выполняет в 2 раза больше работы, чем в предыдущий день. Больше, чем 100 пар в день, сострочить нельзя.

Найти, на который день ученица достигнет вершин мастерства. Сколько всего пар рукавиц ей при этом придется сшить? Вывести на экран таблицу роста мастерства швеи по дням.

Задание 3. На день рождения ребенка бабушка открыла счет в банке и положила на него 5 долл. Каждый год

она добавляет 5 долл. Годовой процент по банковскому счету равен 12%.

Какая сумма накопится к совершеннолетию ребенка (к 18 годам), включая последний взнос. Вывести на экран таблицу ежегодного состояния счета.

Задание 4. Процент по банковскому вкладу равен 6%. Если положить в банк сумму N руб., то эта сумма будет ежегодно увеличиваться.

Как будет изменяться сумма в течение ближайших 10 лет? Если годовая инфляция составляет 3,5%, то сколько же на самом деле будут стоить эти деньги? Вывести на экран таблицу ежегодного состояния счета.

Задание 5. Пара кроликов дает приплод раз в четыре месяца, в среднем по 5 крольчат.

Вычислить и вывести в виде таблицы, каков будет ежегодный приплод от пары, двух, трех и т. д. до 20 пар.

Задание 6. Оплата труда приходящей няни осуществляется по часам. За срок до 6 ч она получает по 25 руб. в час. Начиная с 6 ч работы, каждый последующий час стоит в 2 раза дороже, т. е. 50, потом 100 и т. д. Родители, отправляясь на вечеринку, хотят знать сумму, которую они заплатят няне, но не знают, насколько задержатся.

Вычислить и вывести на экран таблицу оплат услуг няни, начиная с одного часа до 24 ч.

Задание 7. Поженившись, молодые супруги решили откладывать деньги на покупку автомобиля. Муж может вложить ежемесячно M руб., жена V руб. Если положить деньги в банк, то по срочному вкладу годовой процент равен 12%. Автомобиль мечты стоит N тыс. руб.

Через какой срок молодые поедут на юг в собственном авто? Для убедительности выведите таблицу ежемесячных накоплений с учетом процента по банковскому вкладу.

Задача 8. Маркетинговое исследование, проведенное фирмой «Рога и копыта», выявило, что каждый третий год работы предприятия является очень прибыльным, а каждый пятый убыточным. Известно, что этот год (текущий) является убыточным, а предыдущий был очень прибыльным.

Выяснить, какие из ближайших 15 лет будут прибыльными, а какие убыточными. Вывести на экран только года и комментарии.

Задание 9. По окончании сессии всегда есть некоторое количество «хвостистов». Деканат решил провести курсы для отстающих в объеме 40 ч и установил стоимость оплаты часа равной 100 руб. Из суммы, оплаченной студентами, преподавателю причитается 40%.

Найти, сколько денег получит преподаватель, если будет заниматься с одним, двумя, тремя и т. д. до M студентов. Может ли он озолотиться, если золотой горой считать сумму 20 тыс. руб. Скольких бездельников ему придется выучить?

Задание 10. Незнайка учит английский язык. В первый день он выучил 2 слова, а каждый последующий день собрался выучивать на два слова больше, чем в предыдущий.

Найдите и выведите в виде таблицы, на который день Незнайка выучит 100 слов, 200, 300, 400 и т. д. до 1000. В английском языке около 50 тыс. слов, а срок жизни Незнаек примерно 30 лет. Успеет ли до своей кончины Незнайка выучить английский язык? Если нет, то сколько незнаечих жизней понадобится, чтобы выучить английский язык?

Задание 11. Спортсмен начал тренировки, пробежав N км. Ежедневно он увеличивает длину пробегаемого пути на 20%.

Найти, к исходу какого дня спортсмен пробежит марафонскую дистанцию. Вывести таблицу длин ежедневно пройденного пути.

Задание 12. Старушка решила купить телевизор, когда внук подарил ей 1000 руб. Она положила их в Сбербанк под 8% годовых. Ежемесячно на этот же счет старушка вносит 200 руб. Самый дешевый телевизор стоит K руб.

Через сколько месяцев старушка посмотрит кино на канале СТС? Вычислить и вывести на экран состояние счета помесячно.

Задание 13. Известно, что заяц бежит в K раз быстрее, чем ползет черепаха. Они стартуют из пункта А в пункт Б, расстояние между которыми N км.

Вычислить, сколько раз заяц сбегает туда и обратно, пока черепаха доберется до пункта назначения.

Задание 14. Мама и дочка идут навстречу друг другу со скоростью V_1 км/ч. Расстояние между ними S км. В момент начала движения их собачка, которая была у мамы, видит дочку и начинает бегать от одной к другой со скоростью V_2 км/ч, большей, чем V_1.

Найти, какое расстояние пробежала собака, прежде чем мама с дочкой встретились. Вычислить число пробегов собаки от мамы к дочке.

Задание 15. В стаде K коров и k коз. Корова дает примерно 25 л молока в сутки, коза примерно 2,5 л.

Найти, сколько молока приносит ежедневно стадо, в котором число коров и коз может быть от двух до десяти.

Задание 16. Заяц бежит в K раз быстрее, чем ползет черепаха, скорость которой равна V. Они стартуют из пункта А в пункт Б, расстояние между которыми N км.

Составить таблицу перемещения объектов, если стартовый момент времени равен 0, а интервал времени 0,5 ч. В таблице учесть, что, когда заяц закончит движение, черепаха еще ползет.

Задание 17. Одноклеточная амеба каждые три часа делится на 2 клетки.

Вычислить и вывести в виде таблицы, сколько клеток будет через каждые 3 ч в течение двух суток.

Задание 18. Составить таблицу стоимости порций товара весом от 100 г до 1 кг с шагом 100 г и от 1 кг до 10 кг с шагом 500 г. Цена вводится с клавиатуры.

Задание 19. Фабрика по производству тапочек ежегодно увеличивает объем продаж на 5%. Себестоимость продукции при этом уменьшается на 1%. В текущем году объем продаж составил N тыс. руб., а себестоимость пары тапочек C руб.

Вычислить и вывести на экран таблицу увеличения объема продаж и снижения себестоимости на ближайшие K лет.

Задание 20. Ученик мастера начинает с изготовления в день одной табуретки. Совершенствуясь, он каждый

день изготавливает на одну табуретку больше, чем в предыдущий день. Больше, чем 20 табуреток в день изготовить нельзя.

Найти, на который день ученик достигнет вершин мастерства и сколько табуреток он смастерит за все время обучения. Для убедительности вывести на экран таблицу ежедневной производительности.

Задание 21. Для продавщицы Несчитайкиной разработать программу, которая по стоимости 1 кг некоторого продукта выдает таблицу стоимости 50, 100, 150, ..., 1000, 2000, 3000, ..., 10 000 г этого продукта.

Задание 22. Спортсмен бежит по кругу длиной 400 м, а тренер измеряет среднюю скорость движения на каждом круге. Скорость на первом круге была V км/ч, но на каждом круге она падает на 10%.

Если спортсмен пробежит N кругов, то какая скорость будет на последнем круге? Вычислить и вывести на экран таблицу скоростей на каждом круге пути.

Задание 23. Спортсмен бежит по кругу длиной 400 м, а тренер измеряет среднюю скорость движения на каждом круге. Начальная скорость была V_1 км/ч, но на каждом круге она падает на 10%.

Узнать, на каком круге нужно закончить движение, если скорость не должна упасть ниже V_2 км/ч? Вычислить и вывести на экран таблицу скоростей на каждом пройденном круге пути.

Задание 24. Богатый дядя подарил племяннику на пятилетний юбилей 1 долл. Мальчик отнес их в банк и положил на счет под 8% годовых. Каждый год на день рождения дядя дарит мальчику столько долларов, сколько лет мальчику.

Вычислить, сколько набежит к совершеннолетию мальчика (18 лет). Вывести состояние счета ежегодно с учетом последнего вклада.

Задание 25. Богатый дядя подарил племяннику на рождение 1 долл. Каждый день рождения сумма подарка удваивается.

Вычислить, сколько набежит к совершеннолетию мальчика (18 лет). Вывести состояние счета ежегодно.

Задание 26. Карамель стоит K руб., мармелад M руб., шоколад S руб. за кг.

Вычислить и вывести на экран таблицу стоимости каждого вида сладостей весом 100, 200 г и т. д. до 1 кг включительно.

Задание 27. Напечатать таблицу перевода температуры из градусов по шкале Цельсия (°C) в градусы по шкале Фаренгейта (F) для значений температуры от C_0 до C_n с шагом 1 градус. (Перевод осуществляется по формуле $F = 1{,}8 \cdot °C + 32$.)

Задание 28. Два пловца тренируются в бассейне, длина дорожки которого 50 м. Первый пловец начинает тренировку со скоростью V_1 км/ч, второй со скоростью V_2 км/ч. Оба начинают движение одновременно с одного края бассейна. Скорость первого падает на 5% за час, скорость второго — на 3%.

Встретятся ли пловцы в точке начала движения, если время тренировки T? Если нет, то какое время им придется плавать, чтобы встретиться в первый раз?

Задание 29. На прилавке в один ряд лежат N арбузов. Известно, что каждый арбуз на 100 г легче, чем среднее арифметическое весов его соседей.

Найти вес второго арбуза с точностью до грамма, если первый арбуз весит A кг, а последний — B кг.

Задание 30. Татьяна Ларина, читая очередной французский роман, подсчитала сумму номеров прочитанных страниц.

Определить номер последней прочитанной страницы. Учесть, что юная девица может быть не в ладах с арифметикой.

ТЕМА 6. ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ МЕХАНИЗМА ФУНКЦИЙ

Принципы модульного программирования требуют использования функций для решения любой задачи, алгоритм которой может быть описан абстрактно.

Функция — это абстрактный алгоритм решения некоторой самостоятельной задачи. Фактически, функция

является одним из конструируемых пользователем типов данных и имеет имя, тип, может иметь параметры (аргументы функции), которые обеспечивают связь функции с внешним окружением, а также имеет тело, в котором разработан алгоритм решаемой задачи.

Описание функции

В языке C++ все функции описываются на одном уровне. Вложений не допускается. Структура простой функции ничем не отличается от структуры функции main и в общем виде такова:

```
Заголовок функции
{
  описания локальных переменных;
  описание алгоритма;
return возвращаемое_значение;
// Отсутствует, если функция типа void
}
```

Заголовок функции используется для объявления функции в теле программы и содержит полное описание интерфейса функции, которое позволяет правильно обратиться к ней.

1. *Тип функции*. Это тип возвращаемого значения или void для функций, которые не возвращают значений. Если тип функции опустить, то по умолчанию тип будет int. Оператор return в теле функции должен содержать выражение, тип которого совпадает с типом функции. Для функций void этот оператор отсутствует или пустой (return;).

2. *Имя функции*. Это имя main для главной функции программы или любое, не совпадающее с ключевыми словами и именами других объектов программы.

3. *Формальные параметры функции*. Это перечисленные через запятую имена аргументов функции вместе с их типами или void, если параметров нет.

Формальные параметры функции — это ее внешние данные. Это название подчеркивает, что данные описания формальны, т. е. не участвуют в реальных действиях, а только описывают взаимосвязь данных в теле функции.

Количество формальных параметров функции и их типы могут быть любыми.

Тело функции содержит: 1) описание алгоритма; 2) описания локальных переменных, если они необходимы (их областью действия является тело функции); 3) возврат в точку вызова (оператор return, который может отсутствовать для функций типа void).

Обращение к функции

Обращение к функции — это реальное выполнение алгоритма, каждый раз с различными входными данными. Выполняется из любой другой функции программы с использованием механизма вызова функции через операцию обращения. Обращение к функции зависит от типа функции и может иметь две формы.

1. *Оператор-выражение*. Если функция возвращает значение, то им является одно значение базового типа или указатель. Оно может быть использовано в выражениях или печати в виде обращения к функции, которое имеет специальное название «оператор-выражение»:

```
переменная = имя(фактические параметры);
printf("форматная_строка ",
        имя_функции (фактические_параметры));
```

Например, при обращении к библиотечной функции sin:

```
y=sin(x)
// значение функции вычислено и присвоено y
printf("%6.2f", sin(x));     // значение функции напечатано

sin(x);
// значение функции вычислено, но что
// происходит с вычисленным значением?
```

2. *Оператор-функция*. Если функция не возвращает значения (функция типа void), то обращение к ней выглядит как обычный оператор программы и имеет специальное название «оператор-функция»:

```
имя_функции(фактические параметры);
```

Например, при обращении к библиотечной функции printf:

```
printf("%d,%d",a,b);
```

При обращении к функции ей передаются фактические параметры. Это выражения, имеющие реальные значения на момент обращения к функции, с которыми функция отрабатывает очередной вызов.

Формальные и фактические параметры функции

Тип параметров, их количество и порядок следования называются совместно «сигнатура параметров». В описании функции и в обращении сигнатуры параметров должны строго совпадать. Это означает, что формальные и фактические параметры должны соответствовать друг другу по количеству, типу и порядку следования. При несовпадении типов формальных параметров в описании и фактических при обращении, в C++ выполняется автоматическое приведение типов к типу формальных параметров.

Формальные параметры функции — это имена переменных. Фактические параметры подставляются на место формальных при каждом обращении к функции. Они могут быть в общем случае выражениями, т. е. константами, переменными или выражениями.

Параметры

В C++ существуют два способа передачи параметров в функцию.

1. *По значению*. При этом создается локальная копия фактического параметра в теле функции, следовательно, фактические значения параметров при обращении не могут быть изменены. Так можно защитить входные данные от их нежелательного изменения функцией.

Фактическим параметром, соответствующим формальному параметру-значению, может быть имя переменной, константа, выражение.

2. *По ссылке*. Изменение механизма передачи данного в функцию формально выглядит добавлением знака & к имени формального параметра:

имя_функции (тип_параметра &имя_параметра);

При этом функция и вызывающая программа работают с адресом объекта в памяти, следовательно, функция может изменить значение параметра.

Фактическим параметром, соответствующим формальному параметру-ссылке, может быть только имя переменной.

Пример 6.1. Функция с одним параметром, возвращающая значение. Функция имеет тип, имя, один параметр. Приведем пример описания функции, которая вычисляет значение некоторой функции в указанной точке, например, $y=x^2$.

```
float sqr(float x)              // имя функции sqr
{
  return x*x;                   // одно возвращаемое значение
}
```

При обращении к функции используется оператор-выражение. Фактическим параметром может быть константа, переменная или выражение.

```
#include <stdio.h>
void main(void)
{
  float a, b;
  a=2.;
  b=sqr(a);       // фактический параметр – переменная
  printf("\n%f", b);
  b=sqr(3.5);     // фактический параметр – константа
  printf("\n%f", b);
  b=(a+1.1);      // фактический параметр – выражение
  printf("\n%f", b);
  // запись обращения сокращается, если использовать
  // обращение в функции вывода.
  printf("\n%f", sqr(b));
  printf("\n%f", sqr(3.5));
  printf("\n%f", sqr(a+b*5.));
}
```

Пример 6.2. Функция со многими параметрами, возвращающая значение. Функция имеет тип, имя, несколько параметров. Приведем пример описания функции, которая находит среднее арифметическое трех чисел. Имя функции Avg. Тип функции и ее параметров float.

```
float Avg(float a, float b, float c)
{
```

```
   float S;          // локальная переменная
   S=(a+b+c)/3.;
   return S;         // тип совпадает с типом функции
}
```

Можно привести пример простой записи той же функции.

```
float Avg(float a, float b, float c)
{
   return (a+b+c)/3.;       // тип совпадает с типом функции
}
```

При обращении к функции используется оператор-выражение. Фактическим параметром может быть константа, переменная или выражение.

```
#include <stdio.h>
void main(void)
{   // фактические параметры – переменные.
    // результат присвоен y
   float x1=2.5, x2=7., x3=3.5;
   float y=Avg(x1,x2,x3);
   printf("x1=%f, x2=%f, x3=%f, y= %f\n",x1,x2,x3,y);
   // фактические параметры – константы. Результат
                                                  присвоен y
   y=Avg(2., 4., 7.);
   printf("x1=%f, x2=%f, x3=%f y= %f\n" ,2.,4.,7.,y);
   // фактические параметры – выражения
   // результат выводится на печать
   printf("x1=%f, x2=%f, x3=%f, y= %f\n",x1,x2,x3,y);
}
```

Пример 6.3. Функция, возвращающая значение. Алгоритм функции может быть достаточно сложный. Приведем пример описания функции, вычисляющей сумму n чисел натурального ряда.

```
int Sum(int n)
{
int S=0;              // переменная для накопления
                      // значения суммы
int i;                // параметр цикла для суммирования
for(i=1; i<=n; i++) // цикл завершается при i>n
   S+=i;
```

```
return S;
}
```

В этом примере локальная переменная *S* необходима, этого требует реализация алгоритма, а переменная *i* может осутствовать. Зная, что параметр, передаваемый по значению, не изменяется функцией, его можно использовать как рабочую переменную для управления циклом. По завершении работы функции фактический параметр имеет то же значение, что и до обращения.

```
int Sum(int n)
{
  for(int S=0; n>0; n– –) // при n=0 цикл будет завершен
    S+=n;
  returnS;
}                                  // функция не изменит значения n
```

Имя любого локального данного не должно совпадать с именем функции. Вот пример распространенной ошибки, когда имя функции совпадает с именем ее локального данного. Результат в этом случае непредсказуем.

```
int Sum(int n)
{
  int Sum=0;
  int i;
  for(i=1; i<=n; i++)
    Sum+=i;
  returnSum;
}
```

Пример 6.4. Функция, не возвращающая значения и не имеющая параметров.

Приведем пример описания функции, которая выводит на экран 50 символов «звездочка». Имя функции print. Список формальных параметров пуст. Функция не возвращает значения, поэтому не имеет в своем теле оператора return.

```
#include <stdio.h>
void print(void)   // функция типа void, не возвращает значения
{
  int i;              // внутренняя рабочая переменная
  for(i=1;i<=50; i++)
    printf("%c", '*');       // вывод символа "звездочка"
```

```
   printf("\n");
}
```

При обращении к функции используется оператор-функция. Фактические параметры не передаются, однако скобки после имени функции опускать нельзя. Обращение к функции выполняется дважды.

```
void main(void)
{
   print();                              // обращение к print()
   printf("\tПример вызова функции.\n");
   print();
}
```

Вывод на экран будет иметь вид:

```
**************************************************
   Пример вызова функции.
**************************************************
```

Приведем пример использования функции с параметрами, которая не возвращает значения.

Функция выводит на экран произвольное количество символов «звездочка» и не возвращает значения. Количество произвольно, значит, его значение может быть определено только на момент обращения к функции, поэтому оно является параметром функции.

```
#include <stdio.h>
#include <conio.h>
void print(int count)        // печать строки "*" длиной count
{
   int i;
   for(i=1; i<=count; i++)
   // count – количество символов в строке
      printf("%c", '*');
   printf("\n");
}
```

Если задуматься о том, почему же именно звездочки нужно печатать, можно вынести в список параметров еще и вид выводимого символа, тогда функция будет иметь два параметра, один из которых будет обозначать количество выводимых символов, а второй — их вид. Второй параметр функции имеет тип char.

```
void print(int count, char symbol)
// печать строки символов symbol длины count
{
   int i;
   for(i=1; i<=count; i++)
   // count – количество символов в строке
      printf("%c", symbol);
   printf("\n");
}
```

При обращении к функции оператор-функция может использоваться в цикле. Фактический параметр один. На экран будет выведено 12 строк, в первой строке один символ, в каждой последующей на один больше.

```
void main(void)
{
   int cou;
   clrscr();
   for(cou=1; cou<=12; cou++)
      print(cou);
   // здесь cou – количество символов при обращении
   getch();
}
```

При обращении к функции с двумя параметрами первый фактический параметр означает число выводимых символов, второй — вид символа. Оператор-функция выводит при первом обращении 80 символов тире («–»), при втором 25 символов плюс («+»). При первом обращении фактические параметры — константы, при втором — переменные.

```
void main(void)
{
   char ch;
   ch='+';
   int    cou=25;
   print(80, '–');          // 80 символов тире ("–")
   print(cou, ch);          // 25 символов плюс ("+")
   getch();
}
```

Пример 6.5. Логические функции. Логические функции используются всегда, когда нужно выполнить проверку условия, как правило, сложного, или в алгоритмах поиска. Функция возвращает одно значение логического (целочисленного) типа. Интерпретация соответствует принятой идеологии: ложно то, что равно 0, истинно то, что отлично от 0 (не обязательно 1). Это правило используется при вычислении значений логических выражений, а также при проверке условий в операторе if и в операторах цикла.

Замечание. Напоминаем, что компилятор Visual Studio 2013 в качестве логического типа позволяет использовать тип bool.

Примером логической функции с простым алгоритмом может служить функция определения четности числа. Функция должна вернуть значение логического выражения «остаток от деления числа на два равен 0».

```
int Chet(int num)          // функция возвращает логическое
                           // значение
{
  return num%2==0;         // остаток от деления на два
}
```

Примером логической функции с довольно сложным алгоритмом может служить алгоритм определения, является число простым или нет. Как известно, натуральное число является простым, если оно не имеет делителей, кроме 1 и самого себя. Для того чтобы найти, есть делители или нет, нужно последовательно делить число на все числа натурального ряда, которые могут быть его делителями, начиная с двух до значения, равного половине самого числа. Как только делитель найден, дальнейшие проверки не имеют смысла, так как уже известно, что число не является простым. Если же выполнены все возможные проверки и ни один делитель не найден, то число простое.

```
int Easy(int num)  // функция возвращает логическое значение
{
  int mod;         // делитель числа, управляющая переменная
  // управление по возможным значениям делителей числа
```

```
  for(mod=2; mod<=num/2; mod++)
    if(num%mod==0)
    return 0;      // прерывание поиска, делитель найден
  return 1;        // поиск завершен по управлению
}
```

При обращении к функции имеет смысл выполнить проверку, четно число или нет, так как любое четное число не является простым.

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
  int Number;
  do
  {
    printf("Введите число\n");
    scanf("%d", &Number);
    if(!Chet(Number)&&Easy(Number))
      printf("Число простое\n");
    else
      printf("Число не простое\n");
    printf("Продолжение – любая клавиша, выход – Esc\n");
  } while(getch()!=27)
}
```

Логика оператора условия звучит как перевод с русского на C++. Однако, если эта запись вызывает трудности в понимании, используйте явные проверки соответствия возвращаемого значения логической константе: 1— это «истина», 0 — это «ложь».

```
if(Chet(Number)!=0)      // число нечетно
{
  if(Easy(Number)==1)  // число простое
    printf("Число простое\n");
}
else
  printf("Число не простое\n");
```

Пример 6.6. Несколько слов о роли оператора return и о сложности алгоритма функции. Оператор return относится к операторам управления программой. Его назначение

не только определить и вернуть вычисленное функцией значение, но и прервать работу алгоритма функции, выполнив выход из нее правильным образом. Синтаксис оператора return имеет две формы:

return Выражение;
return;

Первая форма используется для функций, возвращающих значение, вторая для функций типа void. В любом случае оператор return прервет выполнение алгоритма функции и передаст управление в точку вызова.

Рассмотрим пример функции, которая возводит вещественное число в целочисленную степень. Для данной задачи фактически возможны три алгоритма, три ветви, выбор одной из которых зависит от входных данных. Для положительной степени вычисляется простое произведение n сомножителей, для отрицательной степени значение равно дроби $1/$(произведение n сомножителей), а для нулевой степени значение всегда равно 1. Решение начинается с анализа входных данных и, в зависимости от степени сразу расходится на три независимые ветки, каждая из которых заканчивается своим выходом со своими данными.

```
// в алгоритме этой функции три варианта выхода
float pow_1(float x, int n) // основание float, степень int
{
  float S;                  // локальная переменная
  if(n>0)                   // положительная степень
  {
    for(S=1.0; n>0; n--)
      S*=x;
    return S;               // решение найдено при n>0
  }
    else
    if(n<0)                 // отрицательная степень
      {
        for(S=.0; n<0; n++)
          S*=x;
        return 1./S;        // решение найдено при n<0
      }
```

```
    else
      return 1.0;          // решение найдено при n=0
  printf("Кому нужны эти функции?");
  // оператор вне всех ветвей
  // он никогда не будет выполнен
}
```

Этот алгоритм интересен еще и тем, что механизм передачи параметров по значению защищает внешние данные от изменения их функцией. Это позволяет не вводить рабочую переменную для управления циклом в теле функции, а использовать входное данное n в качестве рабочей переменной. При входе в функцию ей будет передано входное значение n. В процессе выполнения алгоритма функция изменяет n, но не внешнее данное, а его локальную копию. По завершении работы функции внешняя переменная имеет то же самое значение, что и при входе.

```
#include <stdio.h>
void main(void)
{
  int n;
  float x;
  printf("Введите вещественное основание и целую
        степень\n");
  scanf("%f%d", &x, &n);
  printf("\n Данные и решение: %6.2f %4d %6.2f\n", x, n,
        pow_1(x,n));
}
```

Пример 6.7. Если функция возвращает через параметры более одного значения, используется механизм передачи данных через параметры по ссылке. Синтаксически в заголовке функции к имени параметра добавляется символ "&", например int &Cou. Технически радикально изменяется механизм передачи данных: в этом случае в тело функции передается адрес объекта Cou, который выделен для него в вызывающей программе. Иллюстрацией отличия двух механизмов передачи данных будет пример функции, которая должна переменить значения двух переменных.

```
// функция Swap1 с параметрами по значению
void Swap1(int x, int y)
{
   int tmp;
   tmp=x;
   x=y;
   y=tmp;
// переменные переменились своими значениями
}

// функция Swap2 с параметрами по ссылке
void Swap2(int &x, int &y)
{
   int tmp;
   tmp=x;
   x=y;
   y=tmp;
// переменные переменились своими значениями
}
```

Как видим, функции одинаковы во всем, кроме механизма передачи параметров. Обратимся к этим функциям одинаковым образом, получим разный результат.

```
void main(void)
{
   int a=5, b=10;
   printf("Исходные значения: a=%d b=%d\n", a, b);
   Swap1(a, b);
   printf("Передача по значению: a=%d b=%d\n", a, b);
   Swap2(a, b);
   printf("Передача по ссылке: a=%d b=%d\n", a, b);
}
```

Вывод на экран покажет, что функция Swap1 работает с локальными копиями данных, а Swap2 — с адресами данных.

```
Исходные значения: a=2 b=10
Передача по значению: a=2 b=10
Передача по ссылке: a=10 b=2
```

Пример 6.8. Еще один пример функции, возвращающей значение через параметры, покажет, что такая функ-

ция не всегда типа void, а возвращаемое ею значение расширяет логику задачи, добавляя функции новый смысл.

Пусть функция находит площадь и периметр треугольника, заданного длинами сторон. Возвращаемых значений два: площадь и периметр, следовательно, их нужно возвращать через параметры. Функция будет возвращать значение типа int, смысл которого заключается в проверке условия существования треугольника. Если треугольник существует, функция вернет 1, если не существует, вернет 0.

```
#include <stdio.h>
#include <math.h>
int Triangle(float a,float b,float c,float &p,float &s)
{
// переменные p и s – внешние данные, могут быть
// входными и выходными
  float pp;
  // полупериметр, локальная переменная
  if(a+b<=c || a+c<=b || b+c<=a)
    return 0;              // треугольник не существует
  else
  {
    p=a+b+c;
    pp=p/2.;
    s=sqrt(pp*(pp–a)*(pp–b)*(pp–c));
    return 1;              // треугольник существует
  }
}
```

При обращении к функции учтем две особенности. Во-первых, функция приобрела статус логической функции. Это означает, что после обращения к ней нужно проанализировать возвращенный результат. Во-вторых, фактическими параметрами, подставляемыми на место формальных параметров-ссылок, при обращении могут быть только имена переменных.

```
void main(void)
{
float A,B,C;               // длины сторон фактические
float Perim, Square;       // периметр и площадь фактические
```

```
printf("Введите длины сторон треугольника\n");
scanf("%f%f%f", &A,&B,&C);
if(Triangle(A,B,C,Perim,Square)==1)        // проверка условия
  printf("Периметр=%6.2f, площадь=%6.2f\n", Perim, Square);
else
  printf("Треугольник не существует\n");
}
```

Пример 6.9. Функция, возвращающая указатель. Такие функции используются, когда тип возвращаемого данного не является простым, например возвращается массив, или когда в теле программы получено новое динамическое значение.

Пусть параметры функции две целочисленные переменные. Если первый параметр больше второго, функция создает новый объект со значением 1, иначе функция не создает объект и возвращает пустой адрес (NULL). Возвращаемое значение — это адрес вновь созданного объекта или пустой указатель NULL. Тип функции — указатель на целое. Возвращаемое значение должно быть присвоено указателю, объявленному в вызывающей программе.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int *New_obj(int a, int b)  // тип функции указатель
{
  int *Obj;
  // новый объект будет создан или нет в функции
  if(a>b)
  {
    Obj=new int;          // создается новый объект
    *Obj=1;               // ему присваивается значение
    return Obj;           // возвращается его адрес
  }
  else
    return NULL;
}
```

В вызывающей программе объявлен указатель int *ip, которому будет присвоено возвращаемое значение.

```
void main(void)
```

```
{
   int p1, p2;       // переменные
   int *ip;          // указатель на новый объект
   printf("Введите две переменные\n");
   scanf("%d%d", &p1, &p2);
   // обращение к функции обычное
   ip=New_obj(p1, p2);
   if(ip!=NULL)
      printf("Адрес нового объекта=%p, значение=%d\n", ip, *ip);
   else
      printf("Новый объект не создан, использовать ip нельзя,
            его значение
                  %s\n", ip);
}
```

В первом операторе вывода на экран для адреса ip использован спецификатор формата %p, во втором для вывода константы NULL использован спецификатор формата %s, чтобы увидеть текстовое представление константы. Если для вывода NULL использовать спецификатор %p, адрес выводится как 0000.

Варианты заданий

Задание 1. Описать функцию $F(x)$, вычисляющую значение периодической функции в произвольной точке. Период функции $T = 2$. На интервале $[-1; 0]$ она совпадает с функцией $y = x + 1$, на интервале $(0; 1)$ совпадает с функцией $y = -x + 1$. Обратиться к функции на интервале $x \in [-2; +4]$ с не менее чем десятью точками.

Задание 2. Описать логическую функцию $Yes(x, y)$, которая определит, принадлежит ли точка с координатами (x, y) единичной окружности, центр которой совпадает с началом координат. Обратиться к функции с координатами точек, лежащими на параболе $y = x^2$ для $x \in [-2; +2]$, шаг 0.5.

Задание 3. Описать логическую функцию $Is\_letter(c)$, которая определит, является ли некий произвольный символ c (параметр функции) одной из строчных или заглавных букв русского алфавита. Обратиться к функции в диалоге, передавая ей посимвольно текст, вводимый с клавиатуры.

Задание 4. Описать функцию $S(x, \varepsilon)$, вычисляющую значение суммы ряда в точке x с указанной точностью ε, если формула суммы:

$$S = x - \frac{x}{2} + \frac{x}{3} - \frac{x}{4} + \ldots$$

Обратиться к функции в диалоге с координатами точек $x \in [-0.5; +0.5]$, шаг 0.1.

Задание 5. Описать функцию *Is_Tri*(a, b, c), которая по значениям длин трех отрезков a, b, c определит, можно ли построить треугольник с такими сторонами. Обратиться к функции в диалоге.

Задание 6. Описать функцию *Pi*(ε), вычисляющую значение числа π по формуле

$$\frac{\pi}{2} = \frac{2 \times 4}{3 \times 3} \cdot \frac{4 \times 6}{5 \times 5} \cdot \frac{6 \times 8}{7 \times 7} \cdot \ldots$$

с произвольной точностью ε. Значение точности передать в функцию как аргумент. Обратиться к функции в цикле, вычисляя значение с точностью 0.01, 0.001, 0.0001.

Задание 7. Описать функцию $P(x)$, вычисляющую значение полинома в произвольной точке x по формуле

$$P(x) = 1 + x + x^2 + x^3 + x^4 + \ldots x^{50}.$$

Обратиться к функции в диалоге со значениями $x \in [-0.5; +0.5]$ с шагом 0.1.

Задание 8. Описать функцию *Min*(x, y, z), которая вернет значение наименьшего из трех своих аргументов. Обратиться к функции в диалоге.

Задание 9. Описать функцию *Square*(a, b, c), которая найдет площадь треугольника по значениям длин сторон a, b, c. Если треугольник не существует, функция вернет значение 0. Обратиться к функции в цикле со значениями длин сторон (1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 6) и т. д. до (10, 11, 12).

Задание 10. Описать функцию *Transform*(n), которая преобразует натуральное число n, «приписывая» к нему по единичке в начале и в конце. Обратиться к функции в диалоге со значениями 23, 234, 2345.

Задание 11. Описать функцию $R(x1, y1, x2, y2)$, которая вычисляет расстояние между двумя точками на координатной плоскости. Обратиться к функции в диалоге, чтобы определить расстояния между началом координат и вершинами некоторого квадрата, заданного координатой верхнего левого угла и длиной стороны.

Задание 12. Описать функцию *Count*(r), которая определит, сколько точек с целочисленными координатами попадают в круг радиуса R с центром в начале координат. Обратиться к функции в диалоге.

Задание 13. Описать логическую функцию $P(x)$, которая определит, является ли ее аргумент x простым числом. Обратиться к функции в диалоге со значениями чисел натурального ряда от 7 до 99.

Задание 14. Описать функцию $R(a, b, c)$, определяющую радиус вписанной окружности для треугольника со сторонами a, b, c. Предусмотреть условие существования треугольника. Если он не существует, функция должна вернуть значение 0. Обратиться к функции с равносторонними треугольниками со сторонами 2, 3, 4, 5, 6.

Задание 15. На плоскости даны две окружности. Описать логическую функцию *Yes*, которая определит, имеют ли окружности точки пересечения. Возможны варианты: окружности не пересекаются, окружности пересекаются, окружности касаются, окружности концентрические. Возвращать значение номера варианта или текстовую строку, содержащую значение соответствующего текста. Обратиться к функции в диалоге.

Задание 16. Описать функцию $S(a, b, h)$, которая найдет площадь равнобочной трапеции с заданными основаниями и высотой. Обратиться к функции в диалоге.

Задание 17. Описать функцию *Square*$(r1, r2)$, которая найдет площадь кольца с заданными радиусами. Если первый параметр меньше второго, возвращать значение 0. Обратиться к функции в диалоге.

Задание 18. Описать функцию *S_C*$(r, angle)$, которая найдет значение площади сектора круга с заданными значениями радиуса и угла (в градусах). Если угол отрица-

тельный или больше 360°, возвращать значение 0. Обратиться к функции в диалоге.

Задание 19. Описать функцию *Sum_AP*(*a*1, *d*, *n*), которая найдет сумму арифметической прогрессии. Предусмотреть проверку исходных данных, и в случае ошибки возвращать значение 0. Обратиться к функции в диалоге.

Задание 20. Описать функцию *Is_in*(*x*1, *y*1, *x*2, *y*2, *x*3, *y*3), которая по заданным координатам вершин треугольника определит, находится ли начало координат внутри этого треугольника. Возвращать значение −1, если треугольник не существует, 0, если снаружи, и 1, если внутри. Обратиться к функции в диалоге.

Задание 21. Описать функцию *R*(*long int N*), которая по заданному значению произвольного натурального числа определит его разрядность. Обратиться к функции со значениями 3, 13, 130, 1300, 13 000.

Задание 22. Описать логическую функцию *Yes*(*N*), которая по заданному значению пятизначного натурального числа определит, является ли оно палиндромом. Предусмотреть анализ входных данных. Если они заданы некорректно, функция вернет значение −1. Обратиться к функции в диалоге.

Задание 23. Описать функцию *Kvadr*(*x*, *y*), которая по заданным координатам точки определит номер четверти координатной плоскости, где находится точка. Если точка на осях координат, вернуть значение 0. Обратиться к функции, передавая ей поочередно значения (1, 1), (−1, 1), (−1, −1), (1, −1), (0, 0).

Задание 24. На плоскости даны окружность радиуса *R* и отрезок координатами концов. Описать логическую функцию, которая определит, находится ли отрезок полностью внутри окружности. Обратиться к функции в диалоге.

Задание 25. Описать функцию *Sum*(*x*, *n*), которая найдет сумму знакопеременного ряда вида

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + (-1)^{k+1} \cdot \frac{1}{k}.$$

Обратиться к функции в цикле со значениями k = 50, 100, 150, 200.

Задание 26. Описать функцию *Pi*(*eps*), которая находит значение числа π по формуле произведения

$$\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \times \frac{4}{3} \cdot \frac{4}{5} \times \frac{6}{5} \cdot \frac{6}{7} \times \ldots$$

с указанной точностью *eps*.

Обратиться к функции со значениями точности 0.01, 0.001, 0.0001.

Задание 27. Описать функцию, которая найдет объем цилиндра, если заданы радиус основания и высота. Обратиться к функции, чтобы найти объемы цилиндров высотой $H = 12, 13, 14, 15$ см и для каждой высоты с радиусами основания $R = 5, 10, 15, 20$ см.

Задание 28. Описать функцию, которая находит значение многочлена вида

$$P(x) = 1 + x + x^2 + x^3 + x^4 + \ldots + x^n$$

для заданных значений x и n. Обратиться к функции при $n = 10$ со значениями $x \in [-0.9; +0.9]$, шаг 0.1.

Задание 29. Описать функцию, которая возводит произвольное число с плавающей точкой в целочисленную положительную или отрицательную степень путем многократного умножения (в цикле). Обратиться к функции, передавая ей поочередно значения $x \in [-2; +2]$. Для каждого x вычислять положительную степень, отрицательную и нулевую.

Задание 30. Два интервала числовой оси заданы координатами своих концов. Описать логическую функцию, которая определит, имеют ли эти интервалы общие точки. Корректность данных проверить. Если интервалы заданы некорректно, функция вернет значение −1. Обратиться к функции в диалоге.

ТЕМА 7. РАБОТА С ОДНОМЕРНЫМИ МАССИВАМИ

Прежде чем приступить к программированию задач этого раздела, необходимо четко знать ответы на следующие вопросы.

1. Что такое массив?

2. Как описать массив?

3. Как размещаются в памяти элементы массива?

4. Как обратиться к элементу массива?

5. Как присвоить значения элементам массива?

Следуя правилам модульного подхода в программировании, мы здесь и далее будем использовать функции для решения задач обработки данных, тем более что основных алгоритмов немного. Выделите абстрактный алгоритм, опишите и отладьте функцию, и вы можете пользоваться ею, как говорится, всю оставшуюся жизнь. Функции программиста компонуются в библиотеки. Для библиотек исходных текстов используются заголовочные файлы. Тексты этих файлов инклюдируются в код программы пользователя и компилируются вместе с ней. Исходные тексты функций можно заранее откомпилировать и хранить в виде объектного кода (файлы с расширением .obj). Тогда в отдельный заголовочный файл, который также инклюдируется в код программы пользователя, выносятся только прототипы функций, содержащихся в объектной библиотеке.

Важной особенностью механизмов C++ является то, что имя массива — это указатель на нулевой элемент. Когда параметром функции является массив, то в функцию передается адрес массива. Это означает, что механизм передачи осуществляется по адресу, и функция возвращает измененный массив. Поясним синтаксис и механизм передачи в функцию массива.

Пусть в вызывающей программе объявлен массив
int Array[10];

Имя Array — это адрес нулевого элемента массива.

В записи заголовка функции, которой передается целочисленный массив (любой, не обязательно Array), формальный параметр должен быть указателем. Синтаксически это можно записать двумя способами.

```
void Function1(int Array[], int len)
{
  for(int i=0; i<len; i++)
    // обращение к Array[i]
}
```

Пустые скобки у формального имени массива в списке параметров подчеркивают, что передается адрес массива. Длина массива, если ее не передавать, будет равна длине массива в вызывающей программе (как он там объявлен, в данном примере 10).

```
void Function2(int *Array, int len)
{
  for(int i=0; i<len; i++)
    // обращение к Array[i]
}
```

Признак "*" у формального имени массива в списке параметров означает, что функция получает указатель на массив, т. е. при обращении фактически передается адрес массива в памяти.

При обращении к функции фактическим параметром должно быть имя любого целочисленного массива, который объявлен в вызывающей программе и получил в ней значения.

```
Function1(Array, 10); // передается весь массив
Function2(Array, 5);  // передается весь массив, но его длина
                      // ограничена значением, равным 5
```

Еще раз подчеркнем, что массив передается по адресу, функция может изменить элементы массива, и это станет известно вызывающей программе.

Теперь поясним, как функция возвращает массив.

Первый способ возвращения массива из функции, через список параметров, был показан выше. Передача массива в функцию через формальный параметр — это одновременно и его возвращение после вызова функции.

Второй способ используется, когда массив возвращается через имя функции. Тип функции должен быть указателем (он будет содержать адрес массива в памяти). В теле функции должен быть оператор return, возвращающий адрес массива. В вызывающей программе возвращаемое значение должно быть присвоено указателю. Указатель должен быть объявлен в главной программе и должен адресовать такую область памяти, которая способна вместить в себя весь массив.

Синтаксически это можно записать так:

```
int *Function3(int *Array, int len)
{
  for(int i=0; i<len; i++)
    // обращение к Array[i]
  return Array;
}
```

В том примере мало смысла, так как массив содержится в списке параметров.

Имеет смысл использовать второй способ для динамических массивов, создаваемых в теле функции. Например, функция создает динамический массив указанной длины. Динамическая память для массива выделена в теле функции.

```
int *New_Arr(int len)       // тип функции указатель
{
  int *Array=new int[len];
  // выделено len*sizeof(int) байт памяти,
  // ее адрес присвоен переменной Array
  for(int i=0; i<len; i++)
    *(Array+i)=i+1;// присвоены значения по возрастанию
return Array;          // адрес необходимо вернуть
}
```

Вызывающая программа должна быть готова к получению нового данного. Для этого в ней объявлен указатель. До обращения к функции он пустой и примет значение только после обращения. Возвращаемое функцией значение присваивается указателю.

```
#include <stdio.h>
void main (void)
{
  int *Array;                          // пустой указатель
  int len_A;
  printf("Введите длину массива\n");
  scanf("%d", &len_A);
  Array=New_mas(len_A);          // теперь он получил адрес
  for(int i=0; i<len_A; i++)
    printf ("%3d",Array[i]);
  printf("\n");
}
```

Приведем полный текст файла, содержащего множество функций, реализующих различные алгоритмы работы с массивами. Это отдельный файл, который хранит абстрактное описание алгоритмов. Такие файлы могут быть доступны многим программам. Следует создать заголовочный файл с расширением имени файла "имя.hpp" или "имя.h". Далее этот файл с помощью директивы #include легко можно подключить к любой программе, которой требуется обработка массивов. Например, если имя заголовочного файла "Task.h" ("Task.hpp"), то он будет доступен любой программе, в тексте которой есть директива

```
#include "Task.h"
```

Физически файл может находиться в том же директории, в котором находится использующая его программа (по умолчанию). Он может размещаться в системной директории, путь к которой прописан в настройках интегрированной среды разработчика. Третий вариант — это собственная директория include, на которую дополнительно нужно настроить интегрированную среду.

Каждый пример — это описание решения какой-нибудь задачи для массивов в общем виде, т. е. безотносительно к способу создания и инициализации массива. В каждой задаче значимыми параметрами массива являются тип элементов и количество (длина массива), которые являются параметрами функций. В этом примере используются статические массивы, в том числе массивы переменной длины. Для сканирования элементов массивов используется либо адресация с помощью индексов (прямая), либо адресация с использованием указателей (косвенная). В любом случае массивы можно считать массивами условно переменной длины, так как функция получает длину массива в виде переменной величины.

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

Пример 7.1. Вывод на экран целочисленного массива в общем виде. Формальные параметры — имя массива (указатель) и длина массива.

```
void Print_mas(int mas[], int len)  // len – длина массива
{
   int i;                           // рабочая переменная
   printf("Массив:\n");             // вывод заголовка
   for(i=0; i<len; i++)
      printf("%5d",mas[i]);         // вывод элемента массива
                                    // в строку
   printf("\n");                    // переход на новую строку
}
```

Пример 7.2. Функция ввода целочисленного массива в общем виде. Формальные параметры — имя массива (указатель) и длина массива. Длина массива определена в теле функции. Чтобы функция могла вернуть это значение, формальный параметр len передается по ссылке.

```
void Input_mas(int *mas, int &len)
// длина массива изменяется, &len – ссылка
{
   int *ip;
// для адресации используется указатель на элемент массива
   printf("Введите количество элементов массива \n");
   scanf("%d", &len);
   printf("Введите элементы массива \n");
   for(ip=mas; ip<mas+len; ip++)
      scanf("%5d", ip)              // & не нужен
}
```

Пример 7.3. Преобразовать целочисленный массив, увеличив значение каждого элемента в 2 раза. Массив передается в функцию как указатель. Измененные в функции значения элементов массива доступны и вызывающей программе.

```
// формальные параметры – имя массива (указатель)
// и длина массива
void Transform_mas(int *mas, int len)
// длина не изменяется, len – ее значение
{
   int i;
   for(i=0; i<len; i++)
      mas[i]=mas[i]*2;
}
```

Пример 7.4. Поиск в массиве. Для массива задан критерий отбора, которому должны соответствовать элементы массива. Простейшим способом решения задачи является прямой поиск, блок-схема алгоритма приведена в п. 1.6.1 (глава 1). Согласно алгоритму условие отбора примеряется по очереди ко всем элементам, и для тех элементов, которые соответствуют условию, выполняется определенное действие.

Для примера найдем все числа, кратные пяти.

1. Поиск всех вхождений значения.

```
void Find_all(int mas[], int len)
{
  for(int i=0; i<len; i++)
  if(mas[i]%5==0)
    // здесь с этим элементом можно что-нибудь сделать,
    // например, вывести
    printf("Кратен пяти элемент с номером %d", i);
}
```

Замечание. В случае, когда в массиве нет ни одного искомого значения, функцией будет выполнено len сравнений, но никакого сообщения пользователь не получит.

2. Поиск первого вхождения значения. Особенностью алгоритма является то, что при первом же удачном сравнении он должен закончить свою работу. Если нужного элемента не окажется, то выход из функции произойдет при завершении цикла, и результатом должно быть значение, отличное от любого номера элемента массива, например, –1. Блок-схема алгоритма поиска первого вхождения элемента, имеющего определенное значение приведена в п. 1.3.5 (глава 1).

```
int Find_first(int mas[], int len)
{
  // поиск первого вхождения числа, кратного пяти
  for(int i=0; i<len; i++)
    if(mas[i]%5==0)
      return i;   // прерывание при найденном значении
                  // функция вернет номер вхождения
  return –1;      // неудачный поиск, функция вернет –1
}
```

При обращении к функции надо учесть, что значение −1, которое она возвращает, означает неудачный поиск.

```
Num=Find_first(A, 5);
if(Num>=0)
  printf("Номер элемента, равного 5 %d\n", Num);
else
  printf("Элемент не найден");
```

Пример 7.5. Найти сумму элементов одномерного массива. Блок-схема алгоритма нахождения суммы элементов одномерного массива приведена в п. 1.3.2 (глава 1).

Функция возвращает одно значение, поэтому ее тип int. Приведем пример использования различных способов адресации элементов массива, прямую и косвенную. Выбор способа адресации не зависит от способа передачи в функцию параметра-массива.

1. Прямая адресация. Обращение к массиву через операцию [].

```
int Sum1(int *mas, int len)
{
  int i;
  int Sum=0;
  for(i=0; i<len; i++)
    Sum+=mas[i];
  return Sum;
}
```

2. Косвенная адресация. Обращение к элементам массива через операцию "*", примененную к указателю на очередной элемент массива.

```
int Sum2(int *mas, int len)
{
  int *ip;
  int Sum=0;
  for(ip=mas; ip<mas+len; ip++)
    Sum+=*ip;
  return Sum;
}
```

3. Косвенная адресация. Обращение к элементам массива через операцию "*" и смещение указателя относительно начала массива.

```
int Sum3(int *mas, int len)
{
  int i;
  int Sum;
  for(i=0, Sum=0; i<len; i++)
    Sum+=*(mas+i);
  return Sum;
}
```

Пример 7.6. Вывод на экран номеров и адресов элементов массива иллюстрирует отличие способов адресации друг от друга. Спецификатор формата для вывода адреса — %p. Так вы увидите реальные адреса памяти, выделенные для размещения массива.

```
void Out(int *mas, int len)
{
  int i;
  int *ip;
  for(ip=mas, i=0; ip<mas+len; ip++, i++)
    printf("Номер=%d Адрес=%p\n", i++, ip);
}
```

Пример 7.7. Функция поиска наибольшего (наименьшего) элемента массива. Блок-схема поиска наибольшего элемента одномерного массива приведена в п. 1.3.3.

1. Использование прямой адресации.

```
int Max1(int mas[], int len)
{
  int i;
  int max;          // наибольшее значение
                    // пусть первый элемент наибольший
  max=mas[0];
  for(i=0; i<len; i++)
    if(mas[i]>max) max=mas[i];
// запоминаем наибольшее значение
  return max;               // возвращаем найденное значение
}
```

2. Использование косвенной адресации.

```
int Max2(int mas[], int len)
{
  int *ip;                  // указатель на элемент массива
```

```
  int *imax;              // указатель на наибольшее значение
  imax=mas;               // запоминаем его адрес
  for(ip=mas+1; ip<mas+len; ip++)
    if(*ip>*imax) imax=ip; // запоминаем адрес наибольшего
  return imax–mas;        // возвращаем номер наибольшего
}
```

Пример 7.8. Функция удаления элемента из массива. Для определенности удалим все отрицательные элементы массива. Удаление элемента заключается в сдвиге части массива влево на одну позицию, начиная с удаляемого значения. Длина массива уменьшается на 1 при каждом сдвиге, поэтому длина массива возвращается по адресу. В этом алгоритме объединены поиск (пример 7.4) и удаление, цикл удаления внутренний, так как для каждого найденного значения выполняется сдвиг оставшейся части массива влево на одну позицию. Управление циклом выполняет оператор for, из которого изъято приращение параметра цикла. Это необходимо, так как приращение номера элемента должно происходить лишь тогда, когда удаления не было. Это позволит избежать ошибки, когда удаляемые элементы идут подряд.

Блок-схема удаления одного элемента одномерного массива приведена в п. 1.3.4 (глава 1).

Поиск отрицательных элементов выполняется в цикле с рабочей переменной *i*. Если отрицательный элемент будет найден, то его номер запомнит эта переменная.

```
void Del(int mas[], int &len)
{
  int i;              // рабочая переменная для поиска
  int j;              // рабочая переменная для удаления
  for(i=0; i<len; ) // приращение параметра изъято
  {
    if(mas[i]<0)  // элемент должен быть удален
    {
      for(j=i; j<len–1; j++)
        mas[j]=mas[j+1];
        len––;    // уменьшение длины по завершении цикла
    }
```

```
    else
      i++;          // переход к следующему поиску,
  }                 // если не было удаления
}
```

Пример 7.9. Элементы массива можно менять местами. Например, переставим их по кругу, это называется циклический сдвиг. Последний элемент не упадет за край массива, а станет первым.

Сдвиг массива вправо выполняется начиная с первого элемента до элемента с номером len-1, где len — длина массива.

```
void Cyrcle_Mas(int mas[], int len)
{
  // переменная tmp запомнит значение последнего
  // элемента
  int tmp=mas[len-1];
  // сдвиг массива вправо
  for(int i=len-1; i>0; i--)
    mas[i]=mas[i-1];
  mas[0]=tmp;
  // запомненный элемент записывается вперед
}
```

Пример 7.10. Функция формирования нового массива. Получим массив, в котором содержатся только положительные элементы исходного массива. Блок-схема алгоритма формирования приведена в п. 1.3.1.

Все данные должны быть объявлены вне тела функции и должны быть известны вызывающей программе. Длина нового массива и его значения получены функцией и возвращаются в вызывающую программу.

В коде программы переменная len_new индексирует новый массив. По завершении цикла поиска положительных элементов найденный элемент записывается в новый массив. Индексации в массивах различна и переменная len_new определяет длину нового массива. Она может оказаться равной 0, поэтому значение длины нового массива из функции New_mas возвращается. Функция имеет тип int, что дает ей статус функции, возвращающей логическое значение.

```
int New_mas(int mas[],int len, int *mas_new, int &len_new)
{
  len_new=0;
// в новом массиве нет значений
  // прямой поиск положительных элементов
  for(int i=0; i<len; i++)
    if(mas[i]>0)
// положительный элемент найден
      mas_new[len_new++]=mas[i];
  return len_new;
}
```

При обращении к этой функции можно выполнить проверку значения, возвращаемого функцей. Если возвращаемое значение равно 0, то новый массив не сформирован, в противном случае новый массив можно, например, напечатать.

```
if(New_mas(Array_old, 10, Array_new, len)!=0)
  print_mas(Array_new,len);
```

Пример 7.11. Функция может работать с данными базовых типов, а при обращении ей передаются элементы массива. Например, функция *R* находит расстояние между двумя точками на плоскости.

```
float R(float x1, float y1, float x2,float y2)
{
  return sqrt(pow(x1–x2, 2.)+pow(y1–y2, 2.));
}
```

Прежде чем привести текст файла, обращающегося к функциям, описанным в заголовочном файле, обсудим способы работы с данными, передаваемыми в функции. Функции все равно, как объявлен и проинициализирован массив, с которым она работает, важно только, чтобы любой массив был объявлен и проинициализирован до момента, когда он будет передан в функцию (до момента обращения). Способов объявления массива два: статический массив или динамический. Способов присваивания значений элементам массива много — это инициализация при объявлении, ввод, случайное заполнение, вычисление по формуле и прочие. Функции для решения многих из этих задач описаны выше.

В тексте примера будут только объявления переменных, их инициализация и обращения к функциям.

```
#include <stdio.h>
#include "Task.hpp"                    // включен текст
заголовочного файла
#define N 10                           // объявление длины
массива
void main(void)
{
  // объявление и инициализация массива a
  int a[]={1, –2, 3, –4, 5, –6};
  int na=sizeof(a)/sizeof(int);        // длина массива
  Print_mas(a, na);                    // вывод массива на печать


  // изменим значения элементов массива
  Transform_mas(a, na);

  // покажем, что они изменились
  Print_mas(a, na); // массив условно переменной длины
  int b[N];              // выделена память для N элементов
                         массива b.
  int nb=0;              // реальная длина массива b<N
  printf("Длина массива должна быть < %d", N);

  // ввод значений массива b
  Input_mas(b, nb);// длина будет определена при вводе
```

Вычислим сумму элементов массива с помощью трех функций, использующих различные способы адресации.

```
  printf("Прямая адресация:     Sum1=%d\n", Sum1(b, nb));
  printf("Косвенная адресация: Sum2=%d\n", Sum2(b, nb));
  printf("Косвенная адресация: Sum3=%d\n", Sum3(b, nb));

  // покажем сходство и различие механизмов адресации:
  Out(b, nb);

  // найдем наибольший элемент массива b:
  printf("Номер=%d, Значение=%d\n", Max1(b, nb),
                                        b[Max1 (b,nb)]);
```

```
// удалим из массива a отрицательные элементы
Del(a, na);
// покажем, что осталось.
printf("Массив после удаления:\n");
Print_mas(a, na);

// выполним циклический сдвиг в массиве b:
Cyrcle_Mas(b, nb);
printf("Массив после сдвига:\n");
Print_mas(b, nb);

//Многократное выполнение циклического сдвига
for(int C=1; C<=3; C++)  // C – обычный счетчик
   Circle_Mas(b, nb);
printf("Массив после сдвига:\n");
Print_mas(b, nb);

// объявим новый массив и проинициализируем его
int d[5]={–1, 1, –2, 2, –3};
// объявим новый массив, длина которого не более 5
int c[5];
int nc;                          // реальная длина нового массива
// получим новый массив
// из положительных элементов массива d
// функция дважды возвращает длину массива
nc=New_mas(d, 5, c, nc);
if(nc!=0)                        // длина нового массива больше 0,
                                 // массив получен
{
   printf("Новый массив:\n");
   print_mas(c, nc);
}
else
   printf ("Массива нет\n");

// обращение к функции с элементами массива
// массивы определяют значения координат точек
//на плоскости
float x[3]={1., 2., 3.};
```

```
    float y[3]={3., 2., 1.};
    printf("Вызываем функцию для нахождения расстояния
            от точек \n до начала координат\n");
    for(int i=0; i<3; i++)
    {
      //При обращении к функции ей передается только одна
      //точка
      printf("%6.2f %6.2f, R=%6.2f\n", x[i],y[i],R(0,0,x[i],y[i]));
    }
}
// End of main
```

Предлагаемые задания имеют основную тему «обработка массивов». Не менее важной является тема «использование механизма функций», рассмотренная ранее. Функции обработки массивов имеют особенности, связанные с передачей данных в функцию и возвращением их из нее. Не следует писать просто программу обработки массива, а затем «переписывать ее на функцию», это порочный путь. Следует воспитывать в себе алгоритмическое мышление, учась видеть абстрактный алгоритм в конкретной задаче, выделять главное и отсекать детали, а также совершенствуя свой навык многократным использованием одних и тех же механизмов.

В каждом задании предлагается три задачи. В любой из них предполагается, что функция обрабатывает абстрактный массив, для которого известны лишь тип элементов и их количество (длина массива).

В первой задаче нужно объявить массив и проинициализировать его в главной программе. Алгоритмом обработки массива является простой алгоритм сканирования его элементов или проход по массиву от начала до конца. Входным данным для функции обработки массива является имя массива и его длина. Результатом обработки является одно значение, которое и возвращает функция.

Во второй задаче нужно объявить массив в главной программе. Главная программа будет управлять процессом обработки массива путем определения последовательности вызова функций. Для увеличения функциональности массива предлагается написать и использовать функции ввода и вывода на экран элементов массива, а также

функцию его обработки, особенностью которой во многих задачах будет изменение длины массива, выполняемое функцией. Поскольку длина массива всегда передается в функцию как параметр, следует всего лишь не забыть передать ее по адресу, а не по значению.

В третьей задаче функциональность массива увеличивается добавлением функции случайной генерации элементов массива. Задачей функции обработки является получение нового массива. При решении этой задачи все реальные массивы объявляет главная программа, и она же управляет процессом получения, обработки и вывода данных. Функция должна получить в качестве параметров все входные данные и указатели на переменные, которые будут результатом обработки.

Варианты заданий

Задание 1.

1. Проинициализировать массив. Описать функцию, которая найдет количество отрицательных элементов массива. Описать функцию, которая найдет значение наибольшего из отрицательных элементов.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива наибольший из отрицательных элементов.

3. Описать функцию случайной генерации элементов массива. Описать функцию для нахождения среднего арифметического элементов массива. Описать функцию, которая получит в новом массиве все значения, меньшие, чем среднее арифметическое своих соседей. Использовать механизм указателей.

Задание 2.

1. Проинициализировать массив. Описать функцию, которая найдет количество элементов массива, имеющих четные порядковые номера, но являющихся нечетными числами. Описать функцию, которая найдет наибольшее четное значение.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит повторяющиеся элементы массива.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая получит в новом массиве только те элементы исходного, которые имеют четные порядковые номера, но являются нечетными числами. Использовать механизм указателей.

Задание 3.

1. Проинициализировать два массива произвольной длины, в каждом из которых нет повторяющихся значений. Описать логическую функцию, которая проверит, нет ли в массиве повторяющихся элементов. Описать функцию, которая найдет количество элементов, которые одинаковы в двух массивах.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая определит, является ли массив упорядоченным по возрастанию. Описать логическую функцию, которая для массива и некоторого заданного N определит, сохранится ли упорядоченность, если N «приписать» в конец массива. Если это возможно, функция действительно добавит число в конец массива, и увеличит его длину.

3. Описать функцию случайной генерации элементов массива Y, в котором могут быть как положительные, так и отрицательные значения. Описать функцию получения нового массива Z по следующему закону: $Z_i = Y_i$, если Y_i по модулю больше 1, и $Z_i = 1$ в противном случае. Найти и вернуть число единичек в новом массиве. Использовать механизм указателей.

Задание 4.

1. Проинициализировать массив. Описать функцию, которая найдет количество элементов массива, принадлежащих интервалам $[-2; -5]$ или $[2; 5]$. Описать функцию, которая найдет сумму элементов, не входящих в указанные интервалы.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая проверит, является ли массив упорядоченным по возрастанию. Описать функцию, которая выполнит вставку в упорядоченный массив некоторого произвольного значения таким образом, чтобы упорядоченность не была нарушена.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая сохранит в новом массиве только те элементы исходного, которые принадлежат интервалам [−2; −5] или [2; 5]. Использовать механизм указателей.

Задание 5.

1. Проинициализировать массив. Описать две функции для нахождения номеров наибольшего и наименьшего элементов массива.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива наибольший и наименьший элементы.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая выполнит разделение массива на два — массив положительных значений и массив отрицательных значений. Использовать механизм указателей.

Задание 6.

1. Проинициализировать два массива, которые задают n точек координатами (X, Y) в двумерном пространстве. Описать функцию, которая находит расстояние между двумя произвольными точками. Описать функцию, которая найдет расстояние между всеми точками и выведет их на экран в виде таблицы. Описать функцию, которая найдет наибольшее расстояние.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая выполнит сжатие массива (удаление всех чисел меньше 0).

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая в новом массиве получит только положительные значения элементов исходного массива. Использовать механизм указателей.

Задание 7.

1. Проинициализировать два массива, которые задают n точек координатами (X, Y) в декартовой системе координат. Описать функцию перевода декартовых координат точки в полярные (углы в градусной мере вычислять с точностью до градусов). Для всех точек перевести декартовы координаты в полярные, результаты сохранить

в массивах. Описать функцию поиска наибольшего значения. Найти номер точки, имеющей наибольший радиус-вектор, и номер точки, имеющей наибольший угол. Вывести значение декартовых и полярных координат этих точек.

2. Описать функции ввода и вывода элементов массива. Описать функции поиска и удаления из массива произвольных цепочек чисел ($M1$, $M2$, $M3$).

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая находит в массиве все элементы, значения которых принадлежат некоторому указанному диапазону [$M1$; $M2$], и формирует из них новый массив. Использовать механизм указателей.

Задание 8.

1. Проинициализировать массив. Описать функцию, которая найдет среднее арифметическое элементов массива. Описать функцию, которая найдет номер элемента массива, ближайшего к среднему арифметическому элементов массива.

2. Даны координаты и массы N материальных точек, расположенных на прямой. Описать функции ввода и вывода массивов. Использовать их для ввода данных, определяющих точки. Описать функцию, которая найдет координату центра тяжести системы. Описать функцию, которая найдет номер точки, наиболее близко расположенной к центру тяжести. Описать функцию, которая удалит точку, наиболее близко расположенную к центру тяжести.

3. Описать функцию случайной генерации элементов массива. Описать функцию, один из параметров которой символьная переменная «знак», принимающая значения «>» или «<». Функция перепишет в новый массив только те элементы исходного, которые больше среднего арифметического, если знак «>», и те, которые меньше, если знак «<». Использовать механизм указателей.

Задание 9.

1. Проинициализировать массив. Описать две функции, которые определят, являются ли элементы массива упорядоченными по возрастанию и являются ли элемен-

ты массива упорядоченными по убыванию. Описать функцию, которая одна выполняет полную проверку. Если порядок возрастания, функция вернет 1, если убывания, –1, если нет порядка, 0.

2. Даны координаты материальных точек, расположенных на прямой. Описать функции ввода и вывода массивов. Ввести координаты точек. Описать функцию, которая найдет координату центра тяжести системы. Описать функцию, которая найдет номер точки, наиболее удаленной от центра тяжести. Описать функцию, которая удалит эту точку.

3. Описать функцию случайной генерации элементов массива. Описать функцию с параметром «знак», формирующую новый массив. Если знак положителен, то функция формирует новый массив, в котором записаны только положительные элементы исходного массива, если отрицателен, то функция формирует новый массив, в котором записаны только отрицательные элементы исходного массива. Использовать механизм указателей.

Задание 10.

1. Проинициализировать массив. Описать функции, которые найдут число положительных и отрицательных элементов массива. Описать функцию преобразования массива, которая возводит в квадрат все отрицательные элементы массива, а из положительных элементов извлекает квадратный корень.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива все элементы, значения которых по модулю больше некоторого заданного N.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая выполнит разделение массива на два — массив четных чисел и массив нечетных чисел. Использовать механизм указателей.

Задание 11.

1. Проинициализировать массив, в котором могут быть как положительные, так и отрицательные числа. Описать функцию, которая найдет количество элементов массива до первого отрицательного, и функцию, которая

найдет сумму элементов после первого отрицательного. Описать функцию, которая решит обе эти задачи.

2. Даны координаты n точек на координатной плоскости массивами координат. Описать функции ввода и вывода массивов, определяющих точки. Описать функцию, которая оставит в массивах только те точки, которые принадлежат полосе, заданной системой неравенств

$$\begin{cases} y >= -1; \\ y <= +1, \end{cases}$$

а остальные удалит.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая в новом массиве получит все значения исходного массива, которые находятся после первого отрицательного элемента. Использовать механизм указателей.

Задание 12.

1. Проинициализировать массив, в котором могут быть как положительные, так и отрицательные числа. Описать функцию, которая проверит, являются ли элементы массива упорядоченными по возрастанию. Описать функцию сортировки массива по возрастанию методом пузырька. Алгоритм заключается в следующем: массив просматривается по перекрещивающимся парам чисел (a_i, a_{i+1}). Если $a_i > a_{i+1}$, они меняются местами. Перестановки подсчитываются. Алгоритм завершает работу, если при просмотре массива нет ни одной перестановки. Сортировка выполняется в том же массиве.

2. Даны координаты n точек на координатной плоскости массивами координат (X, Y). Описать функции ввода и вывода массивов, определяющих точки. Описать функцию, которая найдет координаты центра системы и добавит найденные значения в конец массивов координат.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая сформирует новый массив, в котором только положительные элементы исходного массива. Описать функцию, которая сформирует

новый массив, в котором только отрицательные элементы исходного массива. Использовать механизм указателей.

Задание 13.

1. Дан массив произвольной длины, определяющий коэффициенты многочлена степени N. Проинициализировать исходный массив. Описать функцию, которая определит, не имеет ли нулевое значение коэффициент при наивысшей степени. Описать функцию, которая найдет значение наибольшего элемента массива. Определить, при какой степени коэффициент наибольший. Описать функцию, которая найдет значение многочлена в произвольной точке x.

2. Описать функции ввода и вывода массива. Описать функцию выравнивания элементов массива, которое заключается в удалении из массива всех элементов, по модулю больших некоторого M.

3. Описать функцию случайной генерации элементов массива. Задать массив произвольной длины, определяющий коэффициенты многочлена степени N. Описать функцию, которая вычислит и сохранит в новом массиве коэффициенты многочлена, являющегося его производной первой степени. Использовать механизм указателей.

Задание 14.

1. Проинициализировать массив. Описать функции поиска наименьшего, наибольшего элементов массива и функцию выравнивания массива (замена нулем минимального и максимального его элементов).

2. Описать функции ввода и вывода массива. Описать функцию, которая проверит, является ли массив упорядоченным по убыванию. Описать функцию, которая в упорядоченный по убыванию массив включит некоторый элемент b, с сохранением упорядоченности.

3. Описать функцию случайной генерации элементов массива. Локальным минимумом называется любой элемент, который меньше своих соседей. Описать функцию, которая найдет все локальные минимумы, с записью их в новый массив. Использовать механизм указателей.

Задание 15.

1. Проинициализировать массив. Описать функцию, которая получит сумму нечетных чисел данного массива. Описать функцию, которая получит сумму отрицательных чисел данного массива. Описать функцию, которая получит сумму тех чисел данного массива, которые нечетны и отрицательны.

2. Описать функции ввода и вывода массива. Описать функцию, которая определит, является ли некоторое число простым. Описать функцию, которая удалит из массива все простые числа.

3. Описать функцию случайной генерации элементов массива. Получить два массива одинаковой длины. Описать функцию, которая сложит или вычтет массивы поэлементно с записью в новый массив. Операция (знак «+» или «−») передается в функцию как параметр. Использовать механизм указателей.

Задание 16.

1. Проинициализировать массив. Описать функцию, которая определит, являются ли палиндромом элементы массива. Описать функцию, которая расположит его элементы в обратном порядке, не используя вспомогательный массив.

2. Описать функции ввода и вывода массива. Описать функцию нахождения среднего арифметического элементов массива. Описать функцию, которая добавит найденное среднее значение в конец массива.

3. Описать функцию случайной генерации элементов массива. Описать функцию копирования массива. Использовать механизм указателей.

Задание 17.

1. Проинициализировать массив. Описать функцию, которая найдет наибольшее из нечетных по значению чисел. Описать функцию, которая найдет наименьшее из четных чисел. Описать функцию, которая одновременно решит обе эти задачи.

2. Описать функции ввода и вывода массива. Описать функцию, которая определит, является ли массив упорядоченным, например, по возрастанию. Описать функцию,

которая выполнит вставку в упорядоченный массив некоторого произвольного значения таким образом, чтобы упорядоченность не была нарушена.

3. Описать функцию случайной генерации элементов массива. Получить массив B. Описать функцию, которая сольет некоторый упорядоченный массив A со случайным массивом B, многократно выполняя вставки элементов из массива B в массив A. Использовать механизм указателей.

Задание 18.

1. Проинициализировать массив. Описать функцию поиска наибольшего значения. Описать функцию сортировки массива по возрастанию на основе алгоритма обмена. Путем просмотра отыскивается максимальное значение и меняется местами с последним элементом, затем этой же операции подвергается оставшаяся часть массива кроме последнего элемента, затем кроме двух последних и т. д. всего $(n - 1)$ раз, где n — число элементов массива. Исходные данные не сохранять, т. е. выполнить сортировку в том же массиве.

2. Описать функции ввода и вывода массива. Описать функцию преобразования массива, которая удаляет из него все отрицательные и равные нулю значения.

3. Описать функцию случайной генерации элементов массива. Пусть два массива произвольной длины хранят коэффициенты двух многочленов степеней N и M. Описать функцию сложения (вычитания) многочленов, которая получит значения коэффициентов нового многочлена в новом массиве. Характер операции (знак «+» или «−») задать как параметр функции. Использовать механизм указателей.

Задание 19.

1. Проинициализировать массив. Описать функцию, которая определит, образуют ли элементы массива арифметическую прогрессию. Если да, вернуть знаменатель прогрессии, иначе 0. Описать функцию, которая определит, образуют ли элементы массива геометрическую прогрессию. Если да, вернуть знаменатель прогрессии, иначе 1.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива первый и последний элементы.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая получит в новом массиве упорядоченный по возрастанию исходный массив. Алгоритм сортировки произвольный. Использовать механизм указателей.

Задание 20.

1. Проинициализировать массив. Описать функцию, которая найдет количество четных элементов массива. Описать функцию, которая найдет количество нечетных элементов массива. Описать функцию, которая одновременно решит обе задачи.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива те значения, которые не повторяются.

3. Описать функцию случайной генерации элементов массива. Описать функцию нахождения среднего арифметического элементов массива. Описать функцию, которая получит в новом массиве те значения исходного, которые отличаются от среднего арифметического элементов массива не более чем на 1. Использовать механизм указателей.

Задание 21.

1. Проинициализировать массив. Описать функцию, которая определит, является ли массив знакопеременным. Описать функцию, которая найдет число смен знака в массиве.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая определит, является ли массив упорядоченным по возрастанию. Описать функцию, которая объединит два упорядоченных массива в третий, тоже упорядоченный, используя механизм вставок очередного элемента второго массива в подходящее место первого.

3. Описать функцию случайной генерации элементов массива. Использовать ее, чтобы задать в виде двух массивов произвольной длины координаты n точек на плоскости. Описать функцию вычисления расстояния между двумя точками. Использовать ее, чтобы получить в новом массиве номера всех точек, принадлежащих кругу радиу-

са R с центром в начале координат. Использовать механизм указателей.

Задание 22.

1. Проинициализировать в виде двух массивов произвольной длины координаты n точек на плоскости (X, Y). Описать функцию вычисления расстояния между двумя точками. Описать функцию, которая найдет суммарное расстояние в порядке обхода от первой точки до последней.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая проверит, находится ли точка с координатами (x, y) в диапазоне значений $|x| < R$ и $|y| < R$. Описать функцию, которая выполнит для n точек на плоскости удаление из массивов координат тех точек, что не принадлежат области.

3. Описать функцию случайной генерации элементов массива. Задать случайно координаты n точек на плоскости. Описать функцию, которая для n точек сформирует массив номеров тех точек, радиус-вектор которых меньше некоторого заданного R. Использовать механизм указателей.

Задание 23.

1. Проинициализировать массив. Описать функцию для нахождения первого положительного элемента массива и функцию для нахождения первого отрицательного элемента массива.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива первый положительный и первый отрицательный элементы.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая выполнит слияние двух массивов одинаковой длины в третьем (элементы первого и второго массивов в нем чередуются). Использовать механизм указателей.

Задание 24.

1. Проинициализировать массив. Описать логическую функцию, которая найдет, есть ли в массиве хотя бы одно нулевое значение. Описать функцию, которая найдет общее число элементов до первого нулевого и после по-

следнего нулевого. Если ни одного нулевого элемента нет, функция вернет –1. Если нулевыми являются первый элемент и последний, функция возвращает 0, иначе — значение, отличное от 0.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива все значения до первого нулевого и после последнего нулевого.

3. Описать функцию случайной генерации элементов массива. Описать логическую функцию, которая в новом массиве получит все значения исходного, которые находятся от первого нулевого значения до последнего нулевого значения. Использовать механизм указателей.

Задание 25.

1. Проинициализировать два массива, которые задают n точек координатами (X, Y) в декартовой системе координат. Описать логическую функцию, которая проверит, принадлежит ли точка с координатами (x, y) кругу радиуса R с центром в начале координат. Описать функцию, которая для n точек определит, какая из них принадлежит указанной области и выведет в таблицу.

2. Описать функции ввода и вывода элементов массивов как координат точек. Описать функцию, которая удалит из массивов все точки, не принадлежащие указанной области.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая находит в массиве все элементы, значения которых не принадлежат некоторому указанному диапазону $[A; B]$, и формирует из них новый массив. Использовать механизм указателей.

Задание 26.

1. Проинициализировать массив. Описать функцию, которая найдет наименьший элемент массива. Описать функцию, которая найдет наибольший элемент массива. Описать функцию, которая переменит местами наименьшее и наибольшее значения соответственно с первым и последним элементами массива.

2. Описать функции ввода и вывода массивов. Описать функцию, которая определит, является ли число четным.

Описать функцию, которая удалит из массива все четные числа.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая определит, является ли число простым. Описать функцию, которая перепишет в новый массив только те элементы исходного, которые являются простыми числами. Использовать механизм указателей.

Задание 27.

1. Проинициализировать массив. Описать функцию, которая определит, являются ли элементы массива арифметической прогрессией. Если прогрессия возрастающая, функция вернет 1, если убывающая, –1, если не прогрессия, 0.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удаляет из массива все нулевые значения. Описать функцию, которая меняет местами элементы массива, передвигая в начало все положительные элементы, а в конец все отрицательные, выполняя перестановки в том же массиве.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая по заданным значениям первого элемента и знаменателя, генерирует арифметическую прогрессию длиной n. Использовать механизм указателей.

Задание 28.

1. Проинициализировать массив. Описать функции поиска наибольшего и наименьшего элементов массива (возвращать номера). Описать функцию, которая найдет, сколько элементов массива находятся в промежутке между номером наибольшего и номером наименьшего элементов массива.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива все элементы до первого экстремума и после последнего.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая получит в новом массиве все элементы исходного, которые находятся в промежутке между двумя экстремумами. Использовать механизм указателей.

Задание 29.

1. Проинициализировать массив, в котором могут быть как положительные, так и отрицательные числа. Описать логическую функцию, которая найдет, есть ли в массиве хотя бы один отрицательный элемент. Описать функцию, которая найдет количество элементов массива до первого отрицательного. Описать функцию, которая найдет сумму элементов массива до первого отрицательного.

2. Описать функции ввода и вывода элементов массива. Описать функцию, которая удалит из массива все элементы до первого отрицательного.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая в новом массиве получит все значения исходного массива, которые находятся после первого отрицательного элемента. Использовать механизм указателей.

Задание 30.

1. Проинициализировать массив. Описать функцию, которая выполнит сглаживание элементов массива (сглаживание — это замена каждого числа значением среднего арифметического трех стоящих рядом элементов). Описать функцию, которая найдет среднее арифметическое элементов массива.

2. Описать функции ввода и вывода массива. Описать функцию, которая заменит нулями наименьшее и наибольшее значения элементов массива. Дописать в конец массива значение среднего арифметического.

3. Описать функцию случайной генерации элементов массива. Описать функцию, которая формирует массив отклонений для каждого элемента исходного массива от среднего арифметического. Использовать механизм указателей.

ТЕМА 8. ИСПОЛЬЗОВАНИЕ ОДНОМЕРНЫХ МАССИВОВ В СОДЕРЖАТЕЛЬНЫХ ЗАДАЧАХ

В задачах этого раздела основная трудность заключается в том, что задания не формализованы. Следовательно, предваряет кодирование процесс формализации, в который должны входить:

1. Выбор математической модели.
2. Выбор способа представления данных.
3. Функциональная декомпозиция.
4. Кодирование и отладка алгоритмов обработки данных.

В качестве рекомендации напомним тему раздела: использование массивов, значит, способ представления данных — это массив (массивы). Алгоритмов обработки массивов не так много, большинство из них рассмотрены в примерах предыдущего раздела, следовательно, нужно лишь выбрать подходящий алгоритм и, возможно, модифицировать его для конкретной задачи. Следует активно пользоваться подходящими функциями, написанными ранее, такими как ввод и вывод элементов массива.

Варианты заданий

Задание 1. Сорок разбойников сдали экзамен по охране окружающей среды. Использовать функции обработки массивов, чтобы найти, сколько разбойников будут охранять среду отлично, сколько хорошо, а сколько — посредственно.

Задание 2. Для каждой детали цилиндрической формы проводят три измерения: диаметр нижнего основания, диаметр центра и диаметр верхнего основания с точностью 3 знака. Данные сохранены в массивах. Бракованными считаются детали, у которых конусность или бочковатость более 3%. Пометить бракованные детали. Определить процент бракованных деталей в партии.

Задание 3. Дети встают по кругу и начинают считалку, в которой выбывает n-й ребенок, после чего круг смыкается. Считалка повторяется, пока не останется один ребенок. Использовать функции обработки массивов, чтобы узнать, кто останется.

Задание 4. Дневная и ночная температура воздуха измеряются ежедневно и записываются в таблицу. Когда среднесуточная температура в течение трех дней подряд ниже 8°C, начинается отопительный сезон. Использовать функции обработки массивов, чтобы определить, какого числа сезон был начат в этом году, если известно, что это произошло в текущем месяце.

Задание 5. В таблице хранятся данные о расходе электроэнергии в школе помесячно в течение года. Использовать функции обработки массивов, чтобы узнать средний расход электроэнергии, минимальный и максимальный расходы, а также узнать, на сколько процентов отличаются минимальный и максимальный расходы от среднемесячного.

Задание 6. На метеостанции в компьютер введены сведения о среднесуточной температуре за март. Использовать функции обработки массивов, чтобы найти: 1) среднюю температуру месяца; 2) день, когда температура ближе всего подходила к среднемесячной.

Задание 7. Банк провел мониторинг с целью улучшения обслуживания клиентов. Собранная информация хранит номер менеджера, время регистрации клиента, время начала обслуживания и время завершения. При обработке информации требуется найти наибольшее, наименьшее и среднее время ожидания, а также наибольшее, наименьшее и среднее время обслуживания. Использовать функции обработки массивов.

Задание 8. В течение суток через каждый час произведены замеры напряжения в сети. Использовать функции обработки массивов, чтобы определить максимальный скачок напряжения и наибольшее его падение, а также узнать, в какое время суток это произошло.

Задание 9. Кот Матроскин завел 10 коров. Каждый вечер он записывает, сколько молока дала каждая корова за день. Потом ему нужно узнать, сколько всего молока получено за день, а также какая из коров сегодня отличилась, а какая дала молока меньше всех. Использовать функции обработки массивов.

Задание 10. Каждому ученику 1 класса полагается стакан молока, если его вес меньше 30 кг. Количество учеников и вес каждого известны. Выяснить, сколько литров молока необходимо для класса (1 стакан равен 0,2 л). Использовать функции обработки массивов.

Задание 11. Безумное чаепитие. За круглым столом сидят толстяки. Вес каждого известен. Каждый час они пересаживаются по кругу вправо на один стул. Известно,

что один из стульев (он помечен) не выдержит максимального веса толстяка. Используя функцию обработки массивов, определить, в котором часу все повеселятся.

Задание 12. Для того чтобы выявить наиболее популярного политического деятеля из 10 участников, проведен экспертный опрос. Три эксперта каждому деятелю проставляют баллы от 1 до 10. Использовать функции обработки массивов, чтобы найти самого популярного деятеля по сумме баллов.

Задание 13. Кот Матроскин и Шарик загадывали числа в произвольном порядке и записывали их на печке, пока не кончилось место. Использовать функции обработки массивов, чтобы определить, каких чисел, четных или нечетных, загадано больше.

Задание 14. Информация о валеологическом обследовании призывников известна. Для каждого хранится вес, рост, объем груди на вдохе. Известны предельные параметры, которым должен удовлетворять призывник. Использовать функции обработки массивов, чтобы найти, какие призывники годны к несению службы.

Задание 15. Филя, Каркуша и Степашка проводят кастинг ведущих передачи «Спокойной ночи, малыши». Каждый кандидат оценивается по трем параметрам: артистичность, фотогеничность, эрудированность. Каждый эксперт выставляет оценку от 0 до 10 баллов. Критерий выбора кандидата эксперты не смогли сформулировать, но они хотят выбрать лучшего. Также они хотят узнать лучшего в каждой отдельной номинации. Использовать функции обработки массивов.

Задание 16. Овцы пасутся примерно вместе. Отбившейся от стада считается овца, которая удалилась на максимальное расстояние от условного «центра стада». Если известны координаты всех овец (как точек на плоскости), проверить, есть ли овца, отбившаяся от стада. Она достанется волку, который с ноутбуком сидит под ближайшим к стаду кустом. Использовать функции обработки массивов.

Задание 17. Коротышки собирали огурцы. Число огурцов, собранных каждым коротышкой, записано. В оплату

каждому коротышке выдается 2 огурца, а тому, кто собрал больше всех, 3 огурца. Найти, кому достанется 3 огурца. Найти, сколько всего огурцов собрали коротышки. Найти, сколько огурцов осталось для засолки. Использовать функции обработки массивов.

Задание 18. Роща ценных деревьев расположена в живописном уголке. Известны координаты каждого дерева. Требуется огородить рощу от зайцев и оленей забором прямоугольной формы, причем потратить на это как можно меньше денег. Найти периметр охватывающего прямоугольника, используя функцию обработки массивов.

Задание 19. Улитка упорно ползет по склону вверх. В солнечный день она проползает S_1 м, в пасмурный — S_2 м. Сведения о погоде за месяц известны. Использовать функции обработки массивов, чтобы узнать: 1) сколько проползала улитка за каждую неделю; 2) сколько проползла за месяц; 3) какова средняя скорость перемещения.

Задание 20. На метеостанции в компьютер введены сведения о среднесуточной температуре за март. Использовать функции обработки массивов, чтобы найти: 1) количество дней, когда температура ниже 0°C; 2) количество дней, когда температура выше 0°C.

Задание 21. Спортсмен бежит по кругу. На каждом круге тренер определяет время прохождения дистанции и записывает его. Определить, сколько метров пробежал спортсмен. Определить, какова средняя скорость бега. Определить, на каком круге скорость была наибольшая, на каком круге наименьшая. Использовать функции обработки массивов.

Задание 22. Роща ценных деревьев расположена в живописном уголке. Известны координаты каждого дерева. Требуется огородить рощу от нашествия мартышек забором круглой формы высотой 5 м. Найти наименьший радиус окружности, охватывающей все деревья, найти площадь сетки, которая будет потрачена на забор. Использовать функции обработки массивов.

Задание 23. Популяция кроликов изменяется по следующему закону: в «хороший» год она удваивает-

ся, в «обычный» увеличивается в 1,25 раза, в «плохой» уменьшается на 0,25%. Характеристики хранятся N лет, начиная с 1990 г. В этом (нулевом) году популяция составляла M особей. Использовать функции обработки массивов, чтобы узнать численность популяции к концу N-го года, вывести на экран данные о ежегодном изменении популяции.

Задание 24. Пес Шарик каждый день фотографирует дачников и обитателей ближайшего леса. Себестоимость одной фотографии K руб. Шарик записывает в журнал, кого он сфотографировал. Дачникам фотографии стоят денег, причем Шарик берет $2K$ руб. за фотографию. Обитателям леса фотографии раздаются бесплатно, по 1 шт. на морду. Каждый день Шарик хочет знать, каков размер его прибылей (а может, убытков). Использовать функции обработки массивов.

Задание 25. Царевна Несмеяна каждому из претендентов на ее руку и сердце задает M вопросов. За очень понравившийся ответ она присуждает 2 балла, за не очень понравившийся — 6 баллов, за очень не понравившийся — 8. Определить самого понравившегося претендента. Использовать функции обработки массивов

Задание 26. В течение суток через каждый час произведены замеры температуры и влажности воздуха. Использовать функции обработки массивов, чтобы определить максимальное значение температуры и влажности, а также узнать, в какое время суток это произошло.

Задание 27. Безумное чаепитие. За круглым столом сидят толстяки. Вес каждого известен. Каждый час они пересаживаются по кругу вправо на один стул. Использовать функции обработки массивов, чтобы определить раскладку весов по стульям через K ч.

Задание 28. Сведения о росте учеников одного класса хранятся в алфавитном порядке фамилий учеников. Использовать функции обработки массивов, чтобы найти: 1) самого высокого ученика; 2) самого маленького ученика.

Задание 29. Сведения о росте и весе учеников одного класса хранятся в алфавитном порядке фамилий учени-

ков. Параллельно хранится пол ребенка (мальчик, девочка). Известен средний вес ребенка в этом возрасте (для мальчиков и для девочек). Использовать функции обработки массивов, чтобы найти: 1) детей, которые выше средней упитанности; 2) детей, которые ниже средней упитанности.

Задание 30. Есть список школьников одного класса. Для каждого из них известны вес и рост. Использовать функции обработки массивов, чтобы найти самого высокого и самого толстенького школьника.

ТЕМА 9. РАБОТА С ДВУМЕРНЫМИ МАССИВАМИ. ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ

Синтаксис C++ разрешает использовать только одномерные массивы. Массивы большей размерности трактуются как массивы массивов. Использование статических многомерных массивов имеет некоторые особенности, которые можно рассмотреть на примере двумерных массивов (матриц).

Объявление двумерного массива требует, чтобы были указаны два размера: число строк и число столбцов матрицы, например:

```
int a[8][10];       // массив из 8 одномерных массивов,
                    // в каждом из которых 10 элементов.
float b[5][5][5]; // массив из 5 матриц, каждая
                    // из которых имеет 5 строк по 5 элементов.
```

При объявлении массива таким образом в памяти выделяется место под размещение его элементов. Элементы двумерного массива размещаются линейно по возрастанию индексов (построчно). Индексы нумеруются с нуля.

Инициализация двумерных массивов синтаксически выглядит как инициализация нескольких одномерных массивов, например:

```
int a[3][4]={
            {1, 2, 3, 4}, // значения элементов
                          // нулевой строки
            {2, 4, 6, 8}, // значения элементов
                          // первой строки
```

```
                {9, 8, 7, 6}  // значения элементов
                              // второй строки
};
```

Как и одномерные массивы, матрицы могут быть условно переменной длины. При описании матрицы число строк и столбцов задает константное выражение. Это целочисленные значения, которые определяют механизм выделения памяти для матрицы. Реально из выделенного пространства можно использовать меньшее число данных. При этом следует ввести переменные, обозначающие реальное число строк, столбцов (меньшее или равное указанному в описании), которые будут управлять процессом сканирования элементов матрицы.

Пусть матрица имеет размер, определенный в описании. Пусть реальный размер обозначен переменными n, m (матрица имеет n строк и m столбцов, при этом нумерация начинается с нулевого значения). Пусть i — номер строки, j — номер элемента в столбце (текущие значения). Обращение к элементу массива a_{ij} можно выполнить по индексу (прямая адресация) или указателю (косвенная адресация):

```
a[i][j]       // адресуется элемент массива, стоящий на
              // пересечении i-го столбца и j-й строки.
*(a+i*m+j)    // адресуется элемент массива, отстоящий от его
              // адреса на i*m+j значений.
```

При косвенной адресации важно, что константа M (количество элементов в строке) определяет разбиение адресного пространства, выделенного для матрицы, на строки определенной длины.

Способы адресации элементов матрицы рассмотрим на примере. При решении любой задачи, в которой требуется обращение ко всем элементам матрицы, алгоритм просмотра содержит вложенный цикл, у которого во внешнем цикле управляющей переменной является номер строки матрицы, а во внутреннем — номер столбца. Тогда просмотр элементов происходит построчно.

Пусть есть описание матрицы:

```
#define N 5
```

```
#define M 7
int a[N][M];
```

Здесь размер матрицы определяют define-константы. Для матрицы условно переменной длины следует описать размер по максимуму и ввести переменные, реально определяющие размер матрицы.

```
int a[N][M];        // максимальное число строк, столбцов
int n, m;           // реальное число строк, столбцов
printf("Введите размер матрицы (строк, столбцов <25)\n");
scanf("%d%d", &n, &m);// теперь это наибольшие значения
                      // параметров цикла
```

Для обращения к элементам матрицы по строкам цикл записывается так:

```
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    {
      // в теле цикла обращение к переменной a[i][j]
    }
```

Если переменить эти циклы местами, просмотр будет происходить по столбцам:

```
for(j=0; j<m; j++)
  for(i=0; i<n; i++)
    {
      // в теле цикла обращение к переменной a[i][j]
    }
```

Зная, что матрица хранится как одномерный массив, можно ее описать как одномерный массив, а элементы адресовать как элементы матрицы, например:

```
int a[25];          // матрица размером 5×5
int n, m;
n=5;
m=5;
```

Независимо от способа описания двумерного массива, обращение к его элементам с использованием указателя, выполнит те же действия, что и обращение по индексам, а синтаксически оно должно быть записано так:

```
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    {
```

```
    //В теле цикла обращение к переменной *(a+i*m+j)
}
```

Как видим, запись сложного цикла нисколько не изменилась, а изменился только синтаксис обращения к переменной.

Использование функций при обработке матриц предполагает два кардинально различных подхода. В первом случае матрица существует или рассматривается как самостоятельная структура данных, к которой необходимо применить какой-либо алгоритм обработки. В качестве параметра такой функции будет использована вся матрица целиком. Чтобы задача обработки решалась в общем виде, в функцию следует передать имя матрицы и ее размеры (как параметры функции), при этом функция получает матрицу как двумерный массив. C++ должен знать, каков способ разбиения этой структуры на строки, поэтому число строк данных как константное выражение можно опустить, а число данных в каждой строке опускать нельзя. Оно должно быть указано обязательно как константное выражение в квадратных скобках при передаче матрицы. Так, прототип функции, получающей матрицу в качестве входного данного, может выглядеть так:

```
int function(int a[][m], int n, int m);
// здесь m – константное выражение
```

Пример 9.1. В качестве простого примера рассмотрим функции для ввода и вывода матрицы на экран. Предполагается, что матрица условно переменного размера, поэтому число строк и столбцов матрицы по описанию определено define-константами $N = 10$ и $M = 10$. Реальный размер матрицы определяют переменные n, m, значения которых вводятся.

```
#include <stdio.h>
#define N 10
#define M 10
void input_matr(int a[][M], int &n, int &m);
// важно, что длина строки – это константа

void print_matr(int a[][M], int n, int m);
```

```
// число строк можно не передавать


// главная программа описывает входные данные
void main(void)
{
   int n, m;                    // реальные размеры матрицы
   int matr[N][M];              // описан двумерный массив
   input_matr(matr, n, m);      // передан в функцию ввода
   print_matr(matr, n, m);      // предан в функцию вывода
}

// описание функции ввода
// параметры функции – имя и размеры массива
void input_matr(int a[][M], int &n, int &m)
// n, m возвращаются по ссылке
{
   int i, j;
   printf("Введите размер матрицы не более %d на %d\n",
           N, M);
   scanf("%d%d", &n, &m);
   printf("Введите матрицу.\n");
   for(i=0; i<n; i++)
   for(j=0; j<m; j++)
      scanf("%d", &matr[i][j]);
}

// описание функции вывода
// параметры функции – имя и размеры массива
void print_matr(int a[][M], int n, int m)
{
   int i, j;
   for(i=0; i<n; i++)
   {
      for(j=0; j<m; j++)
         printf("%5d", mas[i][j]);
      printf("\n"); // разбиение вывода на строки
   }
}
```

Во втором случае, если матрица — это массив из одномерных массивов, то для обработки отдельных строк матрицы можно использовать функции обработки одномерных массивов. Зная, что элементы матрицы — это одномерные массивы, каждый из них можно по очереди передавать в функцию, которая умеет работать с одномерным массивом. При этом для решения задач этого раздела можно и нужно использовать объявления функций, отлаженных для решения предыдущих задач раздела 7.

Пример 9.2. При решении задач на обработку одномерных массивов нами были отлажены функции обработки одномерных массивов. Их смело можно применить для решения задач обработки многомерных массивов. Пусть есть функции вывода одномерного массива и функция преобразования, которая находит наименьшее значение и переставляет его на первое место. Для ввода матрицы используем функцию ввода из примера 9.1.

```
#include <stdio.h>
#define N 5
#define M 5

// функция вывода одномерного массива
void print_mas(int mas[], int len)
{
   for(int i=0; i<len; i++)
      printf("%5d", mas[i]);
   printf("\n");
}

// функция вывода матрицы как совокупности
// одномерных массивов
void print_matr(int mas[][M], int n, int m)
{
   printf("Матрица:\n");
   for(int i=0; i<n; i++)
      print_mas(mas[i],m);
}

// функция преобразования одномерного массива
```

```
// по заданному условию
void Change(int mas[], int len)
{
   // найдем наименьший элемент
   int *ip;
   int *min=mas;
   for(ip=mas; ip<mas+len; ip++)
   if(*ip<*min)
      min=ip;
   // перестановка при завершении поиска
   *ip=*min;
   *min=*mas;
   *mas=*ip;
}

// main объявляет матрицу и управляет вызовами функций
void main(void)
{
   int n, m;
   int matr[N][M]; // матрица объявлена размером 10×10
   int i;
   input_matr(matr, n, m); // ввод матрицы
   // вызов функции преобразования
   // для каждой строки матрицы
   for(i=0;i<n;i++)
      Change(matr[i], m);  // matr[i], это i-я строка матрицы
   print_matr(matr, n, m); // вывод результирующей матрицы
}
```

Пример 9.3. Просмотр матрицы по столбцам вызовет существенные изменения в основной программе, потому что столбец нельзя передать как одномерный массив простой записью matr[j] в силу того, что двумерный массив — это массив массивов (строк для матрицы). Однако функции обработки одномерных массивов можно применить для решения задач обработки столбцов, если предварительно столбец копировать в некую промежуточную структуру, а после преобразования возвращать измененное значение столбцу матрицы. Так, основной цикл главной программы примера 9.2 будет выглядеть так:

```
int Tmp[M];              // временный массив, длина равна
                         // длине столбца
for(j=0; j<m; j++)       // цикл по номерам столбцов
{
  //Копирование перед каждым обращением.
  for(i=0; i<n; i++)     // цикл по номеру элемента в строке
    Tmp[i]=matr[i][j]; // Tmp получил копию j-го столбца
  Change(Tmp,m);     // изменение в Tmp
  // копирование после каждого обращения
  for(i=0; i<n; i++)
    matr[i][j]=Tmp[i];
  //Возвращаем значение j-му столбцу
}
// вывод обычный.
print_matr(matr, n, m);
```

Пример 9.4. Чтобы убедиться в преимуществах косвенной адресации, покажем, как можно работать с одномерным массивом как с матрицей. Найдем сумму элементов матрицы.

В этой задаче функция ввода матрицы получает одномерный массив, который разбит на n строк по m элементов в каждой. Косвенная адресация позволяет рассматривать двумерный массив как линейный.

```
#include <stdio.h>

// функция ввода матрицы
void input_matr(int mas[], int &n, int &m)
{
  int i, j;
  printf("Введи размер матрицы\n");
  scanf("%d%d", &n, &m);
  printf("Введи матрицу\n");
  for(i=0; i<n; i++)
    for(j=0; j<m; j++)
      scanf("%d", mas+i*m+j);
}
// функция вывода матрицы
void print_matr(int mas[], int n, int m)
```

```
{
   int i, j;
   for(i=0; i<n; i++)
   {
      for(j=0; j<m; j++)
         printf("%5d", *(mas+i*m+j));
      printf("\n");
   }
}

// функция нахождения суммы элементов матрицы
int Sum(int mas[], int n, int m)
{
   int i, j;
   int S=0;
   for(i=0; i<n; i++)
      for(j=0; j<m; j++)
         S+=*(mas+i*m+j);
      return S;
}

// в главной программе объявлен одномерный массив
// он передается во все функции
void main(void)
{
   int matr[25];
   int n, m;
   input_matr(matr, n, m);
   printf("Сумма элементов равна %d\n", Sum(matr, n, m));
   print_matr(matr, n, m);
}
```

Варианты заданий

Задание 1. Дана матрица размером $n \times m$. Найти суммы элементов в каждой строке матрицы, для чего использовать функцию, находящую сумму элементов одномерного массива. Дополнить матрицу найденными значениями, поместив их в конце каждой строки.

Задание 2. Дана матрица размером $n \times m$. Упорядочить матрицу по возрастанию элементов первого столбца. Использовать метод пузырька. Алгоритм заключается в следующем: матрица просматривается по перекрещивающимся парам чисел ($a[i][0]$, $a[i + 1][0]$). Если нулевой элемент i-й строки больше, чем нулевой элемент $i + 1$-й строки, то строки меняются местами. Поскольку строки матрицы — это одномерные массивы, имеет смысл написать функцию перемены указателей, адресующих строки $a[i]$ и $a[i + 1]$. Перестановки подсчитываются. Алгоритм завершает работу, если при просмотре матрицы нет ни одной перестановки.

Задание 3. Дана матрица размером $n \times m$. Найти и заменить нулем максимальное и минимальное значения в каждой строке матрицы. Использовать функцию, находящую минимум, и функцию, находящую максимум из элементов одномерного массива. Передавать им по очереди строки матрицы.

Задание 4. Дана матрица размером $n \times m$. Определить, является ли она симметричной относительно главной диагонали. Использовать логическую функцию.

Задание 5. Дана матрица размером $n \times 3$. Она содержит результаты измерений прямых круговых конусов: радиус основания, высота, длина образующей. Найти и сохранить в этой же матрице значения объема и площади поверхности каждого конуса. Использовать функции. Найти конус наибольшей площади и наибольшего объема, для чего использовать функцию поиска максимума в одномерном массиве.

Задание 6. Дана матрица размером $n \times m$. Выполнить сглаживание в строках матрицы, которое заключается в замене каждого элемента значением среднего арифметического трех стоящих рядом значений. Использовать функцию, выполняющую сглаживание в одномерном массиве. Передавать ей по очереди строки матрицы.

Задание 7. Дана матрица размером $n \times m$. Преобразовать матрицу, удалив из нее i-ю строку и (или) j-й столбец (по желанию пользователя). Использовать функцию преобразования матрицы.

Задание 8. На плоскости заданы n точек своими координатами. Построить матрицу расстояний между всеми точками. Найти наибольшее расстояние. Найти, между какими точками расстояние наибольшее. Использовать функции для вычисления расстояния между двумя точками и для поиска наибольшего значения.

Задание 9. Дана матрица размером $n \times m$. Определить, являются ли упорядоченными по возрастанию данные в каждой строке матрицы. Напечатать исходную матрицу по строкам, в конце каждой строки напечатать сообщение о том, упорядочена строка или нет. Использовать функцию, выполняющую проверку в одномерном массиве. Передавать ей по очереди строки матрицы. Использовать функцию печати строки сообщения.

Задание 10. Заданы несколько матриц произвольного размера. Описать функцию сложения (вычитания) матриц. Передавать как один из параметров функции знак действия («+» или «−»). Чтобы контролировать входные данные, функция должна быть логической.

Задание 11. Дана матрица размером $n \times m$. Преобразовать матрицу следующим образом: найти в каждой строке минимальный и максимальный элементы и поменять их местами с первым и последним элементами строки. Использовать функции, выполняющие поиск минимума и максимума в одномерном массиве. Использовать функцию для выполнения перестановки. Передавать им по очереди строки матрицы.

Задание 12. Получить трехдиагональную матрицу размером $n \times n$, у которой на главной диагонали и двух, лежащих выше и ниже ее, расположены единицы, а остальные элементы равны нулю.

Задание 13. Дана матрица размером $n \times m$. Заменить нулями элементы i-й строки и (или) j-го столбца (по желанию пользователя). Использовать функцию преобразования матрицы.

Задание 14. Дана матрица размером $n \times m$. Определить, являются ли упорядоченными по возрастанию данные в каждом столбце. Напечатать исходную матрицу по строкам, внизу каждого столбца напечатать сообщение о том,

упорядочен столбец или нет. Использовать функцию, которая решает задачу для элементов одномерного массива. Передавать ей по очереди столбцы матрицы.

Задание 15. Даны два массива коэффициентов: A и B. Вычислить матрицу решений всех линейных уравнений вида $A_i \times x + B_j = 0$. Описать и использовать функцию, возвращающую матрицу.

Задание 16. Дана матрица размером $n \times m$. Выполнить выравнивание в матрице, заменяя те значения, которые по абсолютной величине больше некоторого M (введенного в диалоге), значением M с учетом знака. Использовать функцию преобразования одномерного массива или матрицы.

Задание 17. Дана матрица размером $n \times m$. Найти и заменить нулем максимальное и минимальное значения в каждом столбце матрицы. Использовать функции, выполняющие поиск минимума и максимума в одномерном массиве. Передавать им по очереди столбцы матрицы.

Задание 18. Дана матрица размером $n \times m$. Найти норму матрицы: максимальное значение из сумм элементов в каждой строке. Использовать функцию суммирования с сохранением всех найденных сумм. Использовать функцию поиска максимума в одномерном массиве.

Задание 19. Дана матрица размером $n \times m$. Найти суммы элементов в каждом столбце матрицы и дополнить матрицу найденными значениями, поместив их в конце каждого столбца. Использовать функцию, находящую сумму элементов одномерного массива. Передавать ей по очереди столбцы матрицы.

Задание 20. На плоскости заданы n точек своими координатами. Построить матрицу расстояний между всеми точками. Найти равноудаленные точки, если такие есть, и сохранить в отдельном массиве их номера. Использовать функции построения и поиска.

Задание 21. Дана матрица размером $n \times m$. Упорядочить каждую строку матрицы по возрастанию элементов. Использовать метод простого обмена, описанный в п. 1.5.3 (глава 1). Передавать в эту функцию поочередно строки матрицы.

Задание 22. Дана матрица размером $n \times m$. Найти седловую точку матрицы и напечатать ее индексы. Седловой точкой называется элемент, имеющий наименьшее значение в строке и наибольшее в столбце. Использовать функцию обработки матрицы.

Задание 23. Дана треугольная матрица размером $n \times n$. Описать функцию ввода матрицы, пустую половину при вводе заполнить нулями. Использовать функцию обработки матрицы, чтобы найти значения и индексы наибольшего и наименьшего элементов матрицы. Описать функцию вывода матрицы на экран, выделить цветом найденные значения.

Задание 24. Дана матрица размером $n \times m$. Найти сумму элементов каждой строки матрицы. Найти наибольшее и наименьшее значения суммы, а также определить, в какой строке матрицы это значение найдено. Использовать функции для суммирования элементов одномерного массива, для поиска минимума и максимума.

Задание 25. Дана матрица размером $n \times m$. Определить и напечатать, в каких строках матрицы ее элементы образуют монотонную последовательность и какого характера (возрастающую или убывающую). Использовать функцию, выполняющую проверку в одномерном массиве. Передавать ей по очереди строки матрицы.

Задание 26. Дана матрица размером $n \times m$. Получить новую матрицу, каждый столбец которой отсортирован по убыванию элементов столбца исходной матрицы. Использовать функцию сортировки одномерного массива, где алгоритм сортировки произвольный. Передавать в эту функцию поочередно столбцы исходной матрицы.

Задание 27. Дана матрица размером $n \times m$. Найти среднее арифметическое значений ее столбцов и дописать в последнюю строку матрицы. Найти строку, в которой значения элементов наиболее близки к среднему арифметическому. Использовать функцию поиска среднего в одномерном массиве, передавая ей поочередно столбцы матрицы. Использовать функцию поиска «самой средней строки».

Задание 28. Дана матрица размером $n \times m$. Описать функцию, чтобы найти сумму ее элементов. Получить на основе исходной новую целочисленную матрицу такого же размера, у которой на месте нулевых элементов записаны нули, а на месте значений, отличных от нуля, записаны единицы. Этот алгоритм реализовать функцией, возвращающей число ненулевых элементов.

Задание 29. Дана матрица размером $n \times m$. Расстояние между k-й и l-й строками матрицы вычисляется по формуле

$$R_{kl} = \sum_{j=0}^{m-1} a_{kj} \times a_{lj}.$$

Определить, между какими строками матрицы расстояние минимально. Использовать функцию для вычисления расстояний и для поиска максимума.

Задание 30. Дана матрица размером $n \times m$. В каждой строке матрицы найти первое вхождение значения 10. Это и все последующие значения 10 заменить нулем. Если числа 10 нет в строке, оставить ее без изменения. Использовать функцию поиска и замены для одномерного массива. Передавать ей по очереди строки матрицы.

ТЕМА 10. РАБОТА СО СТРОКАМИ СИМВОЛОВ

Строка представляет собой массив символов, однобайтовых данных целого типа, объявленных как char или unsigned char. Внутреннее представление символа — это его код (ASCII кодом является число в пределах от 0 до 255 согласно кодовой таблице), где первые 32 символа — управляющие. Синтаксически признаком символьной константы являются кавычки, например: '*' , '?' , 'f' , 'Я' , '#' , '1'.

Строковая константа — это последовательность символов, заключенная в двойные кавычки " ", например: «строка символов».

При представления строки в памяти она рассматривается как массив символов, каждый символ хранится в отдельном байте, включая специальные символы. В конце строки должен быть нулевой байт '\0' как признак конца

строки. Он добавляется автоматически при инициализации строки и при вводе строки с помощью специальной функции gets. При формировании строки вручную необходимо заботиться о том, чтобы этот символ был добавлен. Поэтому число символов в строке (длина строки) всегда больше на 1 (для нулевого символа).

Объявление строковых переменных можно выполнять двумя способами.

1. Как массив: char Str[80];

Это плохой стиль объявления, так как длина строки может изменяться ограниченно.

2. Как указатель: char *Str;

Это хороший стиль, но в этом случае строка является динамической, и для указателя необходимо выделение памяти Str=new char[80];

При инициализации строковых переменных происходит автоматическое выделение памяти под запись значения строки.

```
char *Str1="строка 1 ";  // длина 11 байт, включая пробелы
char Str2[9]="строка 2";// длина 9 байт
char Str1[]="строка 3";  // специальный инициализатор
char Err[4]="ошибка";
// число символов больше, чем объявлено
char Err[10]="не ошибка";
```

Пример 10.1. Символьные константы в строках.

Управляющие символы (Esc-последовательности) в тексте строки предваряются знаком слэш "\". Такой символ, встреченный в строке, приводит к выполнению управляющего воздействия, например, "\n" вызовет перевод строки. Чтобы особые символы были видны как символы, они удваиваются в записи строки.

```
#include <stdio.h>
void main(void)
{
  // выведем на экран несколько строк
  printf("%s\n","Строка символов");      // без особенностей
  printf("%s\n","Слэш \ запишем как \\ \n");
  // символ слэш удваивается
```

```
   printf("%s\n","Символ ""апостроф"" удваивается");
   // апостроф удваивается

   // здесь слэш не только не влияет на управление строкой,
   // но и не виден при выводе
   printf("%s\n","длинные строки могут быть /
   разбиты на части произвольно, /
   на самом деле это одна строка");

   // здесь сочетание \n управляет выводом строки,
   // разбивая ее на части
   printf("%s\n","длинные строки могут\n быть разбиты на
части\n
   произвольно, на самом\n деле это несколько строк.\n");
}
```

Для ввода и вывода текстовых строк используются специальные функции gets() и puts(), аргументом которых является строка.

Пример 10.2. Инициализация строк. Ввод и вывод текстовых строк.

Для ввода и вывода строк можно использовать функции printf, scanf с управляющим признаком формата "%s". При вводе строки ее текст будет введен только до первого пробела.

Объявление и инициализации строк

```
char Str_1[10];  // выделено 10 байт, значения не присвоены
char Str_2[]={'с','и','м','в','о','л','\0'};

// выделено 7 байт, инициализация как массив символов
char Str_3[]="primer & primer";

// выделено 16 байт, инициализация как текстовой строки
char *Str_4="primer string";

// выделено 14 байт, инициализация как указателя
char *Str_5;       // указатель, память не выделена
```

Варианты вывода строк.

1. Как строка по формату %s

```
printf("%s\n", Str_2);
```

2. Как последовательность символов по формату %c

```
int i=0;
while(Str_2[i])    // пока не встречен последний символ '\0'
printf("%c", Str_2[i++]);
printf("\n");
```

3. Как строка без формата

```
puts(Str_3);
```

Ввод строк.

```
Для Str_1 можно вводить не более 9 символов.
scanf("%s",Str_1);          // текст будет введен до пробела
printf("%s",Str_1);
gets(Str_4);                // текст будет введен до нажатия
                            // клавиши [Enter]
puts(Str_4);
```

Пример 10.3. Строки и указатели.

Строку нельзя считать обычным массивом как раз благодаря наличию нулевого байта '\0' в конце строки. Символьная строка, встреченная в выражении, — это адрес массива символов. Может использоваться везде, где можно использовать указатель, но не в левой части операции присваивания. Удобно использовать косвенную адресацию, при этом текущий указатель будет адресовать один символ строки, но в отладчике будет видна оставшаяся часть строки до нулевого байта '\0'. Возможные ошибки связаны с механизмами выделения памяти для динамических строк. Если при объявлении указателя текстовая строка проинициализирована, то выделена память, равная длине этой строки. Нельзя даже пытаться записать в эту строку больше символов, чем выделено. Если же при объявлении указателя текстовая строка не проинициализирована, то память под запись строки не выделена, и этот указатель можно использовать только как рабочую переменную для косвенной адресации при работе с какой-либо строкой.

Объявлена и проинициализирована строка длиной 57 байт.

```
char *Str="Пример строки текста. Память выделена
        при инициализации.";
char *pts;                  // указатель на строку
int i=0;
```

Для ввода-вывода строк используются функции gets(), puts().

```
gets(Str);   // нельзя вводить больше, чем 57символов
puts(Str);
```

Так как память под указатель pts не выделена, нельзя записать

```
gets(pts);
```

Указателю можно присвоить значение

```
pts=Str ;    // pts показывает на начало строки
pts+=14;     // pts показывает на 14-й символ строки Str
// присваивание значения pts вносит изменения в строку
*pts=0;      // теперь здесь будет конец строки (15-й символ)
pts=Str;     // pts снова показывает на начало строки
```

Обычным механизмом перемещения по строке является смещение указателя

```
pts++;
```

Однако этим механизмом надо пользоваться осторожно, так как при таком перемещении возможно появление ошибки — выход за пределы строки.

Для полного просмотра строки используется циклический алгоритм, в котором управляющей переменной является указатель на очередной элемент (символ) строки, который изменяется от адреса начала строки до достижения нулевого байта в конце строки.

```
pts=Str;             // pts указывает на начало строки
while(*pts)          // пока его значение отлично от '\0'
{
// вывод очередного символа и смещение указателя
   printf("%c",*pts++);
}
printf("\n");
```

Такой же циклический алгоритм используется и для изменения содержимого строки. Управление циклом не изменяется, а изменение символов строки достигается изменением значения указателя на очередной элемент (символ) строки.

```
pts=Str;             // pts указывает на начало строки
while(*pts++)
{
```

```
   *pts++='@';    // Каждый второй символ заменен на '@'
}
puts(Str);
```

Пример 10.4. Внутренние коды символов.

Данные символьного типа при хранении имеют внутренние коды. ASCII кодом является число в пределах от 0 до 255. Внутри кодовой таблицы символы расположены в определенном порядке. Значения символов по их кодам можно увидеть, выполнив следующий пример. Здесь переменные i, j пробегают значения так, чтобы все значения выражения (16*i+j), которые изменяются в пределах от 0 до 255, могли быть выведены в виде матрицы. Символы с кодами 7–14 при выводе пропущены.

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
   int i, j;
   clrscr();
   for(i=0; i<6; i++)
   {
      for(j=0; j<16; j++)
      {
         if(16*i+j>7&&16*i+j<14) continue;
         // управляющие символы
         // символьное представление кода символа
         printf("%c", 16*i+j);
      }
      printf("\n");
   }
}
```

Использовать внутренний порядок следования кодов символов можно при сравнении символов и при выполнении арифметических операций над символами. Эти операции на самом деле выполняются над значениями внутренних кодов символов.

```
char c1, c2;
// пусть эти символы имеют значения
// как сравнить два символа?
```

```
int k;
k=c1==c2;                    // или c1>c2 и прочие условия
if(c1>='0' && c1<=9')
printf("Этот символ цифра\n",); // символ c1 – цифра
// для перебора в алфавитном порядке управляющая
// переменная цикла может быть символьной
char c;
for(c='a'; c<='z'; c++)
{
...
}
```

Пример 10.5. Адресация в строках.

К способам адресации в строках еще раз вернемся на примере решения задачи копирования одной строки в другую. Адресация для строк, как и для обычных массивов, может быть прямой и косвенной.

Зададим исходную строку длиной 15 байт и рабочую переменную для косвенной адресации.

```
char *Str1="Первая строка.";
char *pts;                   // рабочая переменная
char *Str2;                  // новая строка – динамическая
puts(Str1);                  // вывод исходной строки
```

Определение длины строки len выполняется в цикле. Управляет циклом переменная pts. Ее начальное значение равно адресу строки.

Пусть *pts — это значение очередного символа строки, а *(pts++) — значение символа, следующего за ним. Выход из цикла происходит, когда найден признак конца строки.

```
int len=0;
pts=Str1;  // рабочая переменная указывает на начало строки
do
  len++;
while(*(pts++)!='\0');
// поиск нулевого байта и смещение указателя
printf("Длина строки %d\n", len);
```

Выделим память для строки Str2 динамически столько же, сколько выделено для строки Str1.

```
Str2=new char[len];
```

Посимвольное копирование из строки Str1 в строку Str2 с использованием прямой адресации начинается с нулевого символа строки и заканчивается, когда встречен нулевой байт в конце исходной строки.

```
int i=0;                    // начиная с нулевого символа строки
while(Str1[i]!='\0')        // пока не встречен нулевой байт
{
  Str2[i]=Str1[i];          // посимвольное присваивание
  i++;                      // и переход к следующему символу
}
```

Цикл while заканчивается до переноса нулевого байта в новую строку, поэтому по завершении цикла нужно его дописывать программно.

```
Str2[i]='\0';               // формируется конец строки
puts(Str2);
```

В следующем примере прямой адресации тоже посимвольно копируется строка, но запись цикла короче. Использование оператора do удобнее тем, что нулевой байт переносится в новую строку, поэтому его не нужно приписывать.

```
*Str2='\0';                 // очистка строки от содержимого
i=0;
do
  Str2[i]=Str1[i];          // посимвольное присваивание
while(Str1[i++]!='\0');     // проверка достижения конца строки
// нулевой символ переносится в новую строку
puts(Str2);
```

Косвенная адресация при посимвольном копировании требует использования указателя на char в качестве рабочей переменной.

В первом случае используем только один указатель на строку, которая будет копией. Меняя адрес первой строки, покажем, как легко можно изменить значение адреса, которое приведет к потере данных.

```
// очистка строки
*Str2='\0';
// управляет циклом переменная Str1, адресующая
// исходную строку
pts=Str2;                   // pts – указатель на новую строку
```

```
while(*Str1!='\0')
   *pts++=*Str1++;          // Str1 изменяется
```

Адрес, выделенный ранее для переменной *pts, изменен операцией ++, поэтому строка потеряна. Признак конца строки нужно перенести в строку-копию.

```
*pts='\0';                  // получена вторая строка
puts(Str2);
```

Вернуться к первоначальному значению адреса первой строки можно, но мы введем новую переменную, чтобы показать в следующем цикле, что нужно сделать, чтобы косвенная адресация не изменяла адреса исходных строк. Для этого требуется две рабочие переменные, два указателя на обе строки, играющие роль синонимов строк.

```
char *Str3="Новая строка"; // ее длина не больше,
                           // чем у копии
char *pts_2;               // для работы со второй строкой
char *pts_3;               // для работы с третьей строкой
pts_2=Str2;                // знает адрес строки
pts_3=Str3;                // знает адрес строки
*Str2='\0';                // очистка строки
```

Управляет циклом переменная pts_3, адресующая исходную строку.

```
do
   *pts_2++=*pts_3;        // оба указателя смещаются
                                                одинаково
while(*pts_3++!='\0');
puts(Str2);
```

Пример короткой записи цикла управления показывает, как синтаксис C++ позволяет сократить текст программы. Тонкости алгоритма в этом случае не видны. Выполняются все те же действия, что и в предыдущем примере.

```
*Str2='\0';
pts_2=Str2;
pts_3=Str3;
while((*pts_2++=*pts_3++)!='\0');
puts(Str2);
```

Пример 10.6. Функции и строки.

При передаче строк в функции особенностей нет. Передача выполняется так же, как и для массивов. Строка должна быть параметром функции. Отдельно передавать длину строки нет необходимости, так как, во-первых, строка имеет признак конца, а во-вторых, ее длину всегда можно определить, используя функцию strlen библиотеки <string.h>, подробнее о которой далее по тексту. Рассмотрим пример функции преобразования строки, которая удалит из строки «лишние» пробелы, т. е. те, что встречаются подряд друг за другом. Эта операция может быть названа сжатием строки. Сжать строку означает удалить из нее лишние пробелы.

Приведем пример двух функций, первая из которых использует прямую адресацию, вторая — косвенную.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

// прототип функции с прямой адресацией
void compress_1(char[]);

// прототип функции с косвенной адресацией
void compress_2(char*);

void main(void)
{
   char *str=new char[80];
   // указатель на строку и выделение памяти
   printf("\nВведите строку символов с пробелами.\n");
   gets(Str);
   printf("Наша строка: \n");
   puts(Str);
   compress_1(Str);                    // обращение к функции
   printf("Наша строка: \n");
   puts(Str);
   printf("\nВведите строку символов с пробелами.\n");
   gets(Str);
   printf("Наша строка: \n");
   puts(Str);
```

```
   compress_2(Str);                  // обращение к функции
   printf("Наша строка: \n");
   puts(Str);
   getch();
}
```

Описание функции, использующей прямую адресацию строки. Длину строки Str возвращает функция strlen(Str).

В коде функции i — рабочая переменная для поиска рядом стоящих пробелов, mid — рабочая переменная для сдвига символов.

```
void compress_1(char Str[])
{
int i, mid;
i=0;                                 // внешний цикл поиска
                                     // рядом стоящих пробелов
while(i<strlen(Str)–1)
   if((Str[i]==' ')&&(Str[i+1]==' '))  // два пробела рядом
      {
      // внутренний цикл сдвига
         for(mid=i; mid<strlen(Str); mid++)
            Str[mid]=tr[mid+1];      // символ '\0'тоже сдвигается
      }
   else                              // движение по строке,
                                     // если не было сдвига
   i++;
}
```

Описание функции, использующей косвенную адресацию строки.

```
void compress_2(char *Str)
{
   Char *ip, *mid;
   ip=Str;                       // внешний цикл поиска рядом
                                 // стоящих пробелов
   while(*ip!='\0')
      if(*ip==' '&&*(ip+1)==' ')  // два пробела рядом
         {
         // внутренний цикл сдвига
            mid=p;               // запомнили адрес
               while(*mid!='\0')
```

```
            {
                *mid=*(mid+1); // копирование символа
                mid++;         // смещение указателя
            }
        }
    else                       // движение по строке,
                               // если не было сдвига
        ip++;
}
```

Пример 10.7. Функция, возвращающая строку.

Строка, передаваемая в функцию как параметр, будет изменена, потому что передается адрес строки. Многие алгоритмы требуют сохранения исходной строки, многие алгоритмы должны породить новые строки. Следовательно, функция будет порождать и формировать новую строку и должна ее вернуть. Сравним две функции, одна из которых возвращает новую строку через параметр, другая через указатель.

Функция сформирует новую строку на основе заданной, вставляя в нее указанный символ через один.

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

Основная задача — сохранить строку-образец. Новая строка будет содержать все символы исходной, но после каждого символа исходной строки будет добавлен новый символ.

Функция Transform_1 получает и возвращает строки только через параметры, поэтому ее тип void. Память для строк должна быть выделена в main.

```
void Transform_1(char *Str, char *Ind, char Symbol)
{
char *pts;          // рабочая переменная
pts=Str;            // теперь pts – это исходная строка
char *pti;          // рабочая переменная
pti=Ind;            // теперь pti – это новая срока Ind
do
{
```

```
   // циклом управляет pts
   *pti=*pts;          // символ приписывается к Ind(*pti)
                       // из Str(*pts)
   pti++;
   *pti=Symbol;        // Symbol приписывается к Ind(*pti)
   pti++;
}
while(*pts++!='\0');  // выход при достижении конца строки
// '\0' перенесен в строку назначения
}
```

Функция Transform_2 получает исходную строку через параметры, а новую возвращает через указатель, поэтому ее тип char *. Память для новой строки должна быть выделена в теле функции.

```
char *Transform_2(char *Str, char Symbol)
{
   int len;
   len=strlen(Str);          // длина строки Str
   char *Ind;
   Ind=new char[len*2];      // выделена память для новой
                             // строки
   char *pts;                // рабочая переменная
   pts=Str;                  // теперь pts — это исходная строка
   char *pti;                // рабочая переменная
   pti=Ind;                  // теперь pti — это новая срока Ind
   // алгоритм тот же самый, но запись короче
   do
   {
      *pti++=*pts;
      *pti++=Symbol;
   }
   while(*pts++!='\0');
      return Ind;            // возвращается адрес новой строки
}
```

Обращение к этим функциям выполняется различным образом. Вызов Transform_1 — это оператор-функция, вызов Transform_2 — это оператор-выражение, его значение должно быть присвоено переменной такого же типа, как и тип возвращаемого функцией значения.

```
void main(void)
{
  clrscr();
  char *Str=new char[80];
  char *Str_new_1=new char[160];
  Str="Преобразование строки без изменения образца.";
  puts(Str);
  Transform_1(Str, Str_new_1, '!');
  // для Str_new_1 память выделена
  puts(Str_new_1);
  char *Str_new_2;        // просто адрес, память выделит
                          // функция
  Str_new_2=Transform_2(Str, '*');
  puts(Str_new_2);
}
```

Пример 10.8. Использование библиотек функций работы с символами и строками.

Библиотека функций <ctype.h> используется для работы с символами, имеет много полезных функций. Например, isalpha определит, является ли символ буквой латинского алфавита, tolower и toupper позволят преобразовать символ латинского алфавита соответственно к нижнему и верхнему регистру, isascii определит, имеет ли символ код ASCII (0–127) и т. д.

Библиотека <stdlib.h> имеет функции преобразования, которые используются для преобразования строки в число (atof, atoi) или числа в строку (itoa) и т. д.

Библиотека <string.h> содержит функции преобразования строк. Например, strlen возвращает целочисленную длину строки, включая '\0', strcpy возвращает указатель на копию строки, strcat выполняет конкатенацию строк, strchr поиск первого вхождения символа, strcmp сравнение строк, strstr поиск подстроки в строке и т. д.

Рассмотрим несколько простых примеров использования библиотечных функций.

Перед использованием функций надо подключить соответствующую библиотеку.

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

```
#include <ctype.h>
#include <string.h>
```

Функция cou_dig подсчитает, сколько символов в строке являются цифрами, используя функцию isdigit, которая получает как входное данное символ (char) и возвращает значение 0 или 1.

```
int cou_dig(char *str)
{
  int count=0;
  while(*str!='\0')
  {
    if(isdigit(*str)!=0) count++;
      str++;
  }
  return count;
}
```

Пример совместного использования функции cou_dig и библиотечных функций strcpy, strcat, strstr.

```
void main(void)
{
  char *my_str="1 2 3 4 5.";
  printf("Цифр=%d",cou_dig(my_str));
  char *Old_str="Строка текста";
  char New_str[80];

  // для копирования строки Old_str по адресу New_str
  // используем функцию strcpy
  strcpy(New_str, Old_str);          // New_str="Строка текста"
  puts(New_str);

  // для слияния строк (конкатенации) используем функцию
strcat
  strcat(New_str, Old_str);
  // New_str="Строка текста Строка текста"
  puts(New_str);

  // для поиска вхождения подстроки в строку
  // используем функцию strstr.
  char *found="ока";                 // строка поиска "ока"
```

```
  char *Yes;
  Yes=strstr(Old_str, found);       // Yes="ока" текста
  if(Yes==NULL)                     //Если бы вхождения не было
    printf("Нет");
  else
    printf("Есть '%s'\n", Yes);
}
```

Пример 10.9. Работа с текстом.

Текст может быть представлен двумерными массивами строк. В качестве примера покажем реализацию функции сортировки строк текста в алфавитном порядке (по возрастанию первых символов строки) с использованием алгоритма сортировки простым обменом. Функция SortStr в качестве параметра использует указатель на массив строк. Массив фактически двумерный. Разбивку на строки определяет наибольшая длина строки LEN.

Используем функции из библиотеки string.h.

```
#include <stdio.h>
#include <string.h>
#define LEN 80              // наибольшая длина строки
#define SIZE 20             // наибольшее число строк

void SortStr(char *Srt[LEN], int num)
// массив строк, num – его длина
{
  int i;
  int flag;
  char buf[LEN];            // для перестановки строк местами
  // сортировка простым обменом
  flag=1;
  while(flag!=0)
    for(i=0, flag=0; i<num–1; i++)
    {
      if(strcmp(Str[i], Str[i+1])>0)
      //символ Str[i]>символа Str[i+1].
        {
        // перестановка строк
        strcpy(buf, Str[i]);
        strcpy(Str[i], Str[i+1]);
```

```
            strcpy(Str[i+1], buf);
            flag=1;              // запоминает факт перестановки
        }
    }
}

void main(void)
{
    char Text[SIZE][LEN];  // массив вводимых строк
    char *ps[SIZE];        // массив указателей на строки
                           // массива
    int i, num;            // num – число вводимых строк
    printf("Введи строки до пустой строки. \n");
    num=0;
    while((gets(Text[num])!=NULL)
            &&num<=SIZE&&strcmp(Text[num],"")!=0)
    // условие завершения ввода:
    // 1) пока gets возвращает не пустое значение,
    // 2) пока число введенных строк меньше, чем SIZE,
    // 3) пока строка не пуста, и strcmp позволит сравнить
    // значения символов, а не указатели
    {
        ps[num]=Text[num];// указатель на очередную строку
        num++;
    }
    printf("Вывод перед обращением к функции сортировки\n");
    for(i=0; i< um; i++)
          puts Text[i]);         // введенный массив строк
    printf("\n");
    SortStr(ps, num);
    printf("Вывод после обращения к функции сортировки\n ");
    for(i=0; i<num; i++)
          puts(Text[i]);         // отсортированный массив строк
        printf("\n");
        for(i=0; i<num; i++)
          puts(ps[i]);           // указатель на сортированные
                                 // строки
    printf("\n");
}
```

Варианты заданий

Задание 1.

1. Описать функцию работы со строкой символов, которая найдет, сколько раз входит в строку некоторый произвольный символ (задать как параметр функции).

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию преобразования строки, которая вносит изменения в строку текста, повторяя дважды каждую букву строки (знаки препинания и прочие символы не изменять). Преобразовать все строки текста и новый текст записать в файл *F*2.

Задание 2.

1. Описать функцию работы со строкой символов, которая найдет, сколько раз в строке встречаются знаки препинания ".", "?", "!" и др.

2. Дан текст (2–3 строки) в файле *F*1. Текст состоит из слов, отделенных друг от друга пробелами. Описать функцию, удаляющую из строки слово с номером *M* (номер слова *M* один из параметров функции). В вызывающей программе номер слова лучше всего вводить в диалоге. Преобразовать все строки текста и новый текст записать в файл *F*2.

Задание 3.

1. Описать функцию работы со строкой символов, которая найдет первое вхождение некоторого слова (задать как один из параметров функции).

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию преобразования строки, которая после каждого десятого символа вставит в текст сочетание символов ":!:", в начало строки — "!:", в конец — ":!". Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 4.

1. Описать функцию работы со строкой символов, которая найдет первое вхождение некоторого символа и последнее его вхождение (символ задать как один из параметров функции).

2. Дан текст (2–3 строки) в файле *F*1. Текст состоит из слов, отделенных друг от друга одним символом пробела. Описать функцию, определяющую, есть ли в тексте некоторое слово, заданное заранее (включить в список параметров

функции, а в вызывающей программе вводить в диалоге). Описать функцию, удаляющую слово из текста. Преобразовать все строки текста и новый текст записать в файл *F*2.

Задание 5.

1. Описать функцию работы со строкой символов, которая найдет, сколько всего слов в строке, если известно, что слова отделены друг от друга пробелами или сочетанием пробела со знаками препинания.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию, преобразующую строку текста следующим образом: если в строке есть подстроки, заключенные в скобки, удалить их вместе со скобками. Если есть вложения скобок, удалить всю внешнюю скобку. Преобразовать все строки текста и новый текст записать в файл *F*2.

Задание 6.

1. Описать функцию работы со строкой символов, которая найдет, есть ли в строке стоящие рядом одинаковые символы, и определит число таких вхождений.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию преобразования строки, которая заменяет все одинаковые идущие подряд символы одним вхождением этого символа. Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 7.

1. Описать функцию работы со строкой символов, которая проверит, не является ли эта строка палиндромом. Для проверки придется удалить все пробелы, отделяющие слова, и перевести символы в один регистр. Пример палиндрома «А роза упала на лапу Азора».

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию преобразования строки, которая заменяет все повторные вхождения символа пробел одним символом пробела, в начало строки добавляет слово «НАЧАЛО», в конце слово «КОНЕЦ». Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 8.

1. Описать функцию работы со строкой символов, которая найдет, сколько в строке символов, которые являются цифрами, и сформирует из них новую строку.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию обработки строки, которая удаляет из строки все знаки препинания. Преобразовать все строки исходного текста и записать новый текст в файл *F*2.

Задание 9.

1. Описать функции работы со строкой символов, которые найдут самое длинное и самое короткое слово строки.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию, которая удалит из строки символы, находящиеся рядом (справа и слева) с символом "#". Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 10.

1. Описать функцию работы со строкой символов, которая найдет вхождение в строку самой длинной последовательности пробелов и определит ее длину.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию, которая удалит из исходной строки все слова, имеющие окончание «КАЯ» или «КОЕ». Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 11.

1. Описать функцию работы со строкой символов, которая найдет первое вхождение подстроки, заключенной в скобки.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию, которая после каждого пробела вставит в текст восклицательный знак. Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 12.

1. Описать функцию работы со строкой символов, которая удалит из строки первое и последнее слова.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию, которая формирует из исходной строки новую, записывая в нее только подстроки, заключенные в круглые скобки (вместе со скобками). Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 13.

1. Описать функцию работы со строкой символов, которая найдет, сколько слов строки начинаются с некото-

рой буквы, заданной произвольно (задать как один из параметров функции).

2. Дан текст (2–3 строки) в файле *F*1. Текст состоит из слов, отделенных друг от друга одним пробелом. Описать функцию преобразования строки: из исходной строки в новую строку переписать только те слова, длина которых больше трех, отделяя их пробелами. Преобразовать все строки текста и записать новый текст в файл *F*3.

Задание 14.

1. Описать функцию работы со строкой символов, которая найдет, сколько из слов строки начинаются и заканчиваются одной буквой, например, «абракадабра», «сервис», «кулак» и пр.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию, которая удаляет из строки все сочетания букв «АЯ» и «ОЕ». Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 15.

1. Описать функцию работы со строкой символов, которая получит массив длин слов этой строки.

2. Дан текст (2–3 строки) в файле *F*1. Текст состоит из слов, отделенных друг от друга произвольным числом пробелов. Описать функцию преобразования строки, которая каждое слово отделяет от другого слова сочетанием трех символов "***". Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 16.

1. Описать функцию работы со строкой символов, которая найдет, есть ли в строке символы, отличные от букв, пробела и точки.

2. Дан текст (2–3 строки) в файле *F*1. Текст состоит из слов, отделенных друг от друга одним символом пробела. Описать функцию, которая меняет местами первое слово с последним словом. Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 17.

1. Для заданного натурального числа *n* получить его правильное символьное представление в виде стро-

ки, где триады цифр отделены друг от друга пробелами. Например, для $n = 1753967$ запись должна иметь вид: «1 753 976».

2. Дан текст (2–3 строки) в файле $F1$. Описать функцию преобразования строки, которая заменяет все последовательности цифр, стоящих подряд, одним символом решетки "#". Преобразовать все строки текста и записать новый текст в файл $F2$.

Задание 18.

1. Для заданного натурального числа n ($n < 20$) получить его правильное текстовое представление в виде строки текста, например, «три», «девятнадцать», «одиннадцать» и пр.

2. Дан текст (2–3 строки) в файле $F1$. Описать функцию, которая формирует новую строку, занося в нее символы исходной строки по 10 через каждые десять. То есть с 1-го по 10-й, с 21-го по 30-й и т. д. Преобразовать все строки текста и записать новый текст в файл $F2$.

Задание 19.

1. Описать функцию работы со строкой символов, которая проверит, начинается ли каждое слово с большой буквы.

2. Дан текст в файле $F1$ в виде:

ИМЯ ОТЧЕСТВО ФАМИЛИЯ_1

ИМЯ ОТЧЕСТВО ФАМИЛИЯ_2

...

Описать функцию, которая формирует текстовую строку в формате:

ФАМИЛИЯ И. О.

Сохранить преобразованный текст в файле $F2$.

Задание 20.

1. Дано натуральное число n ($n < 100$), означающее возраст человека в годах, например, 1, 11, 51 и т. д. Получить текстовую строку, выражающую возраст человека. Например: «Один год», «Одиннадцать лет», «Пятьдесят один год».

2. Дан текст (2–3 строки) в файле $F1$. Текст состоит из слов, отделенных друг от друга пробелами. Описать функцию, которая находит самое длинное слово в строке

и удаляет его. Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 21.

1. Описать функцию работы со строкой символов, которая найдет, есть ли в строке правильное соответствие открывающих и закрывающих скобок.

2. Дан текст (2–3 строки) в файле *F*1. Описать функцию преобразования строки, которая заменяет все подряд идущие пробелы одним вхождением пробела и удаляет все символы, отличные от букв (знаки препинания и пр.). Записать новый «чистый» текст в файл *F*2.

Задание 22.

1. Описать функцию работы со строкой символов, которая найдет количество пар открывающих и закрывающих скобок разного вида. Функции вернет 1, если есть соответствие пар, и 0, если соответствия нет.

2. Дан текст (2–3 строки) в файлах *F*1 и *F*2. Тексты состоят из слов, отделенных друг от друга одним пробелом. Описать функцию, которая определяет, насколько совпадают данные в файлах (по словам). Все несовпадающие слова вывести в файл *G* в формате: «Слово из первого файла», «Слово из второго файла».

Задание 23.

1. Описать функцию работы со строкой символов, которая найдет, сколько в строке знаков препинания.

2. Дан текст (2–3 строки) в файле *F*1. Текст состоит из слов, отделенных друг от друга пробелами. Описать функцию, которая каждый одиночный пробел заменяет пятью пробелами, идущими подряд. Преобразовать все строки текста и записать новый текст в файл *F*2.

Задание 24.

1. Описать функцию работы со строкой символов, которая найдет, стоит ли пробел после каждого знака препинания.

2. Дан текст (2–3 строки) в файле *F*1. Текст состоит из слов, отделенных друг от друга пробелами или сочетанием символов «точка пробел». Описать функцию, формирующую словарь данного текста в виде массива слов. Повторяющиеся слова не включать. Вывести словарь в файл *F*2.

Задание 25.

1. Описать функцию работы со строкой символов, которая возвращает первое слово строки.

2. Дан текст (2–3 строки) в файле $F1$. Текст состоит из слов, отделенных друг от друга пробелами. Описать функцию, которая найдет все вхождения некоторого слова в строке и заменит его некоторым другим словом (эти слова задать как параметры функции). Преобразовать все строки текста и записать новый текст в файл $F2$.

Задание 26.

1. Описать функцию работы со строкой символов, которая находит и возвращает последнее слово строки.

2. Дан текст (2–3 строки) в файле $F1$. Текст состоит из слов, отделенных друг от друга пробелами или знаками препинания. Описать функцию, формирующую частотный алфавит данного текста в виде массива символов (букв). Вывести частотный алфавит в файл $F2$.

Задание 27.

1. Описать функцию работы с текстом, представленным в виде строки символов, которая найдет среднее число слов в предложениях этого текста.

2. Дан текст (2–3 строки) в файле $F1$. Текст состоит из потока предложений, отделенных друг от друга знаками препинания. Описать функцию, которая читает текст, выделяет одно предложение и выводит в текстовый файл $F2$ по одному предложению в строке.

Задание 28.

1. Описать функцию работы с текстом, представленным в виде строки символов, которая найдет среднюю длину всех слов этого текста.

2. Дан текст (2–3 строки) в файле $F1$. Текст состоит из слов, отделенных друг от друга пробелами. Описать функцию, которая находит самое длинное слово в тексте, а затем формирует в новом файле форматированный текст, в котором каждое слово дополнено пробелами до длины самого длинного слова. Записать новый текст в файл $F2$.

Задание 29.

1. Описать функцию работы с текстом, представленным в виде строки символов, которая «нумерует» строку,

добавляя в начало строки текстовое представление номера, переданного в функцию как параметр.

2. В файле $F1$ даны натуральные числа ($n < 1000$), означающие некоторую цену в копейках, например, 317, 5005, 100 и пр. Описать функцию, которая формирует текстовую строку, выражающую цену в рублях и копейках. Например: «3 руб. 17 коп.», «50 руб. 05 коп.», «1 руб. 00 коп.». Записать числа исходного файла и новый текст в файл $F2$.

Задание 30.

1. Описать функцию работы со строкой символов, которая возвращает значение слова, номер которого задан как параметр функции.

2. Дан текст (2–3 строки) в файле $F1$. Текст состоит из слов, отделенных друг от друга пробелами. Описать функцию, которая в строке текста находит все слова, однокоренные некоторому заданному корню, и формирует из них новую строку. Преобразовать все строки текста и записать новый текст в файл $F2$.

ТЕМА 11. ИСПОЛЬЗОВАНИЕ ДВУМЕРНЫХ МАССИВОВ В СОДЕРЖАТЕЛЬНЫХ ЗАДАЧАХ

В заданиях этого раздела предполагается использование файлов данных. Работа с файлами описана в п. 2.8 (глава 2).

Задание 1. При записи данных о соревнованиях по шахматам формируется матрица турнира особого вида. Результат матча может быть 1 (выигранная партия), 0 (проигранная) или 0,5 (ничья). При вводе данных нужно получить симметричную матрицу турнира, где на главной диагонали нули, для обоих участников 0,5 в случае ничьей, а если участник выиграл, то его партнер проиграл и ему записывается 0. Написать функцию для ввода данных турнира в диалоге с формированием матрицы турнира, которую сохранить в текстовом файле. Написать функцию обработки турнира, чтобы определить победителя. Написать функцию обработки турнира, чтобы распределить участников по убыванию набранных очков.

Задание 2. Фабрика производит валенки, тапочки и боты. Данные об объемах сбыта продукции каждого вида за прошлый год помесячно хранятся в файле. При подведении итогов года дирекция требует найти суммы накопительным итогом для каждого вида продукции, и сохранить эту информацию в файле, а также выяснить, в каком месяце имеет место наибольший сбыт каждого вида продукции. Использовать функции для накопления итогов и для поиска максимума.

Задание 3. Валяльная фабрика производит валенки. Данные об объемах сбыта продукции и о ценах продаж за прошлый год помесячно хранятся в файле в виде таблицы следующего вида.

Месяц	Объем продаж, пар	Цена продажи, руб.	Себестоимость, руб.
Январь	4500	100	20
Февраль	3900	100	25
...			

При подведении итогов года необходимо выяснить динамику сбыта, т. е. найти и упорядочить по убыванию информацию о прибылях, приносимых производством. Использовать функцию для вычисления ежемесячной прибыли. Для сортировки матрицы по столбцу «прибыль» использовать функцию сортировки матрицы методом пузырька.

Задание 4. Кот Матроскин владеет стадом коров, а также организовал производство молока, сметаны, творога, масла, сыра. Ежедневно он записывает в текстовый файл дневной сбыт каждого вида продукции, причем цены товара у Матроскина могут изменяться, и цены ему приходится тоже ежедневно записывать. В конце месяца необходимо подвести итог по результатам торговли, чтобы выяснить, какая продукция пользуется наибольшим и наименьшим спросом, а также какая продукция приносит наибольшую и наименьшую прибыль.

Задание 5. В соревнованиях по фигурному катанию каждый спортсмен принимает участие в трех видах соревнований. Каждый вид судят десять судей. Для более точной оценки самый низкий и самый высокий баллы

судей отбрасывают. Составить протокол судейства для n участников в виде матрицы, сохраненной в файле. Найти победителя соревнований. Использовать функции для обработки результатов соревнований.

Задание 6. Коротышки собирают урожай огурцов, помидор, гороха, моркови и прочих плодов земли. Работают n коротышек, а Знайка подводит итоги их трудовой деятельности. Он завел текстовый файл и написал программу, которая ежедневно в диалоге позволяет ввести данные об итогах трудового дня. Эти данные суммируются с итогами предыдущих дней работы, и файл обновляется. По итогам работы в конце уборочной кампании производится оплата. Стоимость сбора одного вида овощей Знайка хранит в отдельном файле. Представьте себя на месте Знайки.

Задание 7. Разборчивая невеста занесла в файл данные о своих потенциальных женихах: внешность и богатство она оценила в баллах.

Имя	Возраст	Внешность	Богатство
Иван	25	15	10
Петр	30	15	10
Павел	40	12	50
Борис	45	8	50
Сергей	45	10	100

Составить оценочную таблицу в виде матрицы. Возраст невесты N лет. Требования к претенденту сформулированы просто: средняя красота, среднее богатство, средний возраст, но не моложе ее самой. Помогите сделать выбор. Использовать функцию выбора, возвращающую номер потенциального жениха.

Задание 8. Сорок разбойников провели турнир, в котором каждый разбойник бился с каждым. Сведения о результатах соревнований записали в файл в виде турнирной таблицы, где номера строк и столбцов — это номера разбойников. Найти самого сильного и самого слабого разбойника. Использовать функцию обработки турнира.

Задание 9. Учебное заведение проводит приемные экзамены на новый учебный год. Информация об абитури-

ентах занесена в текстовый файл в виде матрицы. Каждый должен сдать три экзамена. Те, кто получил двойку за первый экзамен, ко второму экзамену не допускаются, а кто получил двойку за второй экзамен, не допускаются к третьему. Обработать данные об абитуриентах, выполняя удаление после первого, второго и третьего экзаменов с использованием функции обработки матрицы.

Задание 10. Царевна Несмеяна, принимая претендентов на ее руку и сердце, задает каждому M вопросов. Если ответ очень понравился, она присуждает 2 балла, если не очень понравился — 6 баллов, если очень не понравился — 8. Данные опроса она сама записывает в текстовый файл. В конце дня выбирается лучший претендент. Использовать функцию обработки матрицы, чтобы определить самого понравившегося претендента. Этот кандидат приписывается в файл, хранящий многолетнюю историю испытаний, по которому, возможно, когда-то будет принято окончательное решение.

Задание 11. n коротышек собирают урожай m видов различных овощей. Знайка подводит итоги их трудовой деятельности и выполняет расчет. Он завел текстовый файл, и написал программу, которая ежедневно в диалоге позволяет ввести данные об итогах трудового дня. По количеству собранных овощей в конце дня производится оплата. Сбор одного овоща каждого вида Знайка хранит в отдельном файле. Алгоритм оплаты труда сложный. Вычисляется среднее арифметическое собранных всеми за день овощей по каждому виду. Если коротышка собрал больше, чем среднее арифметическое этого вида, то сбор каждого овоща сверху оплачивается в 2 раза дороже. Тот, кто собрал больше всех, получает премию в размере стоимости сбора трех овощей этого вида. Ежедневно Знайка подсчитывает, сколько денег получает каждый коротышка за собранный урожай. Эти данные он приписывает в итоговый текстовый файл, чтобы впоследствии подвести общий итог уборочной кампании.

Задание 12. Кот Матроскин владеет стадом n коров. Ежедневно он записывает в текстовый файл дневной удой каждой коровы. Кот желает приобрести программу, ко-

торая позволит подвести итоги производительности его коров. В конце каждого месяца Матроскин желает найти общий удой каждой коровы, а также наибольший и наименьший. Эти данные должны быть программно приписаны в новый текстовый файл, по данным которого Матроскин подведет годовой итог. Использовать функцию (функции) обработки результатов.

Задание 13. Отец Федор открыл небольшой свечной заводик под Самарой. На заводике льют свечи малые, средние, большие, очень большие и особые. Ежедневный итог по продажам каждого вида продукции в стоимостном выражении матушка дописывает в текстовый файл. По завершении месяца нужно подвести итог, какая суммарная прибыль получена от каждого вида продукции, какой вид приносит наибольшую и наименьшую прибыль.

Задание 14. Сорок разбойников провели соревнование по брейк дансингу. Судили пять судей. Запись о результатах записали в файл в виде матрицы 40×5. Чтобы результат был точнее, самый низкий и самый высокий баллы судей решили не учитывать. Составить оценочную таблицу в виде матрицы, найти победителя соревнований. Использовать функцию обработки результатов соревнований, возвращающую номер победителя.

Задание 15. Дана матрица размером 7×7. Это семеро козлят устроили турнир по копытрестлингу. Сведения о результатах соревнований записали в файл в виде турнирной таблицы, где номера строк и столбцов, это номера козлят, принявших участие в турнире. Турнирная таблица представлена в виде симметричной матрицы. Найти распределение козлят по силе. Использовать функцию обработки турнира.

Задание 16. Семеро гномов копают золотой песок шесть дней в неделю, а в седьмой отдыхают. Сколько нарыл за день каждый гном, Белоснежка вечером записывает в файл. В воскресенье подводится итоги. Самый трудолюбивый гном награждается поцелуем красавицы, самый ленивый — мытьем посуды. Найти самого старательного и самого ленивого гномов. Использовать функцию (функции) обработки результатов.

Задание 17. Коротышки провели психологическое тестирование «Узнай себя». Запись о результатах записали в файл в виде таблицы, где на перекрестье строк и столбцов записана оценка в виде итогового балла.

Имя	Умный	Смелый	Добрый
Знайка	10	5	8
Незнайка	2	10	5
Пончик	5	3	10
Шурупчик	8	8	7
...			

Составить оценочную таблицу в виде матрицы, найти самого умного, самого смелого, самого доброго из коротышек. Использовать функции.

Задание 18. Коротышки устроили турнир по футболу. Было четыре команды: Знайчики, Незнайчики, Пончики, Сиропчики. Результаты игр записали в файл в виде таблицы соревнований.

Знайчики	Незнайчики	3	2
Знайчики	Пончики	3	1
Знайчики	Сиропчики	1	2
Незнайчики	Пончики	4	1
Незнайчики	Сиропчики	2	2
Пончики	Сиропчики	2	3

Составить турнирную таблицу в виде матрицы игр, определить победителя. Использовать функцию обработки турнира.

Задание 19. Три эксперта оценивают накануне выборов популярность десяти кандидатов по 20-балльной шкале.

	Эксперт 1	Эксперт 2	Эксперт 3
Кандидат 1	15	10	10
Кандидат 2	18	12	10
...			

Составить оценочную таблицу в виде матрицы. Подвести итог по сумме баллов каждого кандидата, найти трех самых популярных кандидатов. Использовать функции обработки матрицы.

Задание 20. Для контроля состояния трубопровода ежечасно в течение суток измеряется давление на некоторых точках, где установлены датчики. Для снижения погрешностей датчиков перед обработкой исходные данные сглаживаются путем замены каждого элемента значением среднего арифметического трех стоящих рядом значений (для первого и последнего двух). Результат формируется в новом массиве. Выполнить подготовку данных для n датчиков, если снятые ими данные за сутки записаны в текстовый файл.

Задание 21. Белоснежка назначила каждому из семерых гномов день недели, когда тот должен дежурить по кухне. Каждый день она ставит оценку за дежурство по пятибалльной шкале, а в конце месяца подводит итог, чтобы наградить лучшего гнома. Сведения о результатах проверки Белоснежка записывает в файл в виде матрицы, где номера строк — это номера гномов, а столбцов — номера недель. Найти самого старательного и самого ленивого гномов. Использовать функцию (функции) обработки результатов.

Задание 22. Чипполино проводит ежедневный мониторинг погоды, записывая в текстовый файл информацию о дневной температуре, влажности, давлении, количестве выпавших осадков и скорости ветра. Пытаясь дать прогноз на завтра, он всего лишь усредняет все предыдущие данные за месяц. Текстовый файл прогнозов сохраняет все данные. Выяснить, насколько прогнозирование ошибочно, сравнивая данные файла прогнозов с данными файла мониторинга.

Задание 23. Фермер снял план своего участка прямоугольной формы и сформировал матрицу размером $n \times m$ (размер сетки 1 м), в которой на пустых местах записаны нули, а на занятых цифры, например 1 — жилой дом, 2 — зеленые насаждения, 3 — хозяйственные постройки. Матрица занесена в текстовый файл. Теперь фермер желает выяснить:

- каков процент занятости его территории;
- можно ли полностью огородить участок живой изгородью шириной 1 м, а если нет, то какие будут препятствия;

- на каком месте можно построить сарайчик размером k на l, если это возможно.

Использовать функции обработки матрицы.

Задание 24. Кафе специализируется по бизнес-ланчам для тех, кто следит за своей фигурой, поэтому в меню помимо цены указана и калорийность каждого блюда. Предлагается N комплексных ланчей, состоящих из Q блюд каждый. Стоимость и калорийность каждого блюда записаны в текстовых файлах в виде матрицы стоимостей и матрицы калорийностей. Выбрать все ланчи, калорийность которых ниже, чем указанное значение. Подсчитать стоимость. Выбрать все ланчи, стоимость которых ниже, чем указанное значение. Подсчитать калорийность.

Задание 25. В аптечном складе хранятся лекарства. Сведения о лекарствах содержатся в специальной ведомости: номер лекарства, количество (в шт.), цена. Ведомость хранится в текстовом файле. Провизор время от времени развлекается, решая следующие нестандартные задачи:

- сколько стоит самый дешевый и самый дорогой препарат;
- каково количество всех препаратов на складе;
- какова общая стоимость всех препаратов.

Использовать функции для нахождения минимального, максимального элементов массива и функцию для нахождения суммы.

Задание 26. В музее регистрируется в течение дня время прихода и ухода каждого посетителя. Таким образом, за день получены N пар значений, где первое значение в паре показывает время прихода посетителя и второе значения — время его ухода. Найти промежуток времени, в течение которого в музее одновременно находилось максимальное число посетителей, минимальное число посетителей.

Задание 27. Кукольный театр поставил n спектаклей. Ежедневно театр дает несколько различных спектаклей. Мальвина продает билеты и перед каждым сеансом знает, сколько детских и взрослых билетов продано. Чтобы знать репертуарную политику, Мальвина желает выяснить, какие спектакли имеют наибольшую популярность.

Для этого данные о посещаемости каждого спектакля она сохраняет в текстовом файле следующего вида.

Спектакль	Показов	Детских билетов	Цена	Взрослых билетов	Цена
Золушка	20	25	10	30	20
Король Лир	15	10	10	50	30
...					

Данные о спектакле вводятся в диалоге, и обновляют матрицу данных, причем данные о числе билетов суммируются, а данные о ценах усредняются. В любой момент времени можно получить информацию о том, каков интерес к спектаклям в порядке убывания отдельно для взрослых и для детей, а также узнать в порядке убывания общую прибыль от спектакля.

Задание 28. В зале кинотеатра 10 рядов по 15 мест. С 3 по 7 ряд — VIP-места. Стоимость билетов на них в 2 раза дороже, чем на другие места. 1 и 2 ряды в 1,5 раза дешевле прочих. На обычный сеанс билет стоит K_1 руб., на премьерный K_2 руб., на льготный K_3 руб. При продаже билетов данные заносятся в текстовый файл в виде матрицы занятости мест. Найти сумму, на которую продано билетов. Использовать функции формирования и обработки матрицы.

Задание 29. В зале кинотеатра n рядов, в каждом из которых m мест. При бронировании и продаже билетов формируется карта занятости мест, которая сохраняется в текстовом файле после очередной операции. Нужно помочь кассиру и зрителю выполнить процедуру бронирования, для чего организовать диалог:

- показать на экране карту занятости мест в зале;
- показать зрителю выбранное им место;
- зафиксировать выбор места по договоренности со зрителем;
- сохранить измененную карту;
- подсчитать стоимость билетов.

Задание 30. Дана матрица размером 5×3. Это жители Средиземья организовали курсы информатики. По окончании курсов всем выдали свидетельства. Похвальные листы нужно вручить трем лучшим слушателям. Най-

дите лучших, если данные записаны в текстовом файле в форме таблицы следующего вида.

	MS Word	MS Excel	MS Access
Фродо Беггинз	5	5	3
Бильбо Беггинз	3	4	5
Гэндальф	3	5	5
Двалин	5	4	3
Гвалин	3	5	5
Бифур	3	4	3
Бофур	5	4	3
Бомбур	3	4	4
Торин Оукеншильд	5	4	4

Использовать функции обработки данных.

ТЕМА 12. РАБОТА СО СТРУКТУРАМИ И ОБЪЕДИНЕНИЯМИ

Структура позволяет объединить в единое целое произвольное количество данных различных типов. Поэтому все задачи, которые требуют логического объединения каких-то данных, могут и должны использовать структурный тип. Данные, входящие в структуру, достаточно независимы. Они имеют собственные имена и возможность выполнить над ними любые операции, разрешенные их типом.

Рассмотрим пример использования структур в некоторой модельной задаче, т. е. такой, которая не имеет особенного практического смысла, но позволяет продемонстрировать достоинства структурного типа, его возможности и механизмы. Пусть структура объединяет некую информацию об одном абстрактном объекте (человеке): имя, фамилия и какие-то данные числового характера, например, возраст, рост, средняя зарплата и т. д., назовем их показателями. Пусть имеются сведения о некотором множестве объектов, т. е. о некотором сообществе людей, например, студентов какого-либо факультета, сотрудников некоторой организации, случайной статистической выборки.

Объект называется *среднестатистическим*, если на нем достигается минимум модуля разности среднего арифметического значений его показателей со средним арифметическим всей группы объектов. Аналогично определяется *уникальный* объект (на нем достигается максимум).

Объект может быть назван *среднестатистическим по k-му параметру*, (уникальным по k-му параметру), тогда рассматриваются данные о значении k-го показателя этого объекта в сравнении со средним значением k-го показателя по всей совокупности объектов.

Выясним, кто в группе объектов является:

а) среднестатистическим;

б) среднестатистическим по отдельным показателям;

в) просто поставим задачу найти объект в группе по ключевому значению.

Для описания одного объекта используется структура. Попытаемся отразить в примере возможности этого типа данных. В качестве полей структуры используем простые типы данных, массив и строки, причем одна строка будет статической, а вторая динамической. Для хранения имени используем статическую строку, для хранения фамилии динамическую, для хранения показателей используем массив (ограничимся пятью показателями). Информация о среднем арифметическом показателей каждого объекта может быть вычислена при получении объектом данных. Ее можно сохранить, для этого введем в структуру поля, имеющие смысл «Сумма показателей» и «Среднее арифметическое показателей», соответственно, данные целого и вещественного типов.

Поскольку в группе объектов несколько данных, следует объединить их в массив, где один элемент массива описывает один объект.

Выполним функциональную декомпозицию задачи, т. е. определим, какие алгоритмы обработки понадобятся для ее решения.

1. Сначала подумаем, откуда взять данные. Например, их можно ввести.

2. Для работы с одним объектом пригодятся функции ввода данных об одном объекте и вывода данных об одном

объекте. Имя формального параметра пусть будет Man, это полные данные об одном объекте.

3. Чтобы иметь возможность визуально сравнить данные, требуется вывод общей информации, идеально подходит таблица.

4. Для поиска среднего арифметического по группе объектов и для поиска среднего по какому-то показателю требуются функции, которые возвращают значение среднего по всей группе объектов и по одному показателю. Эти функции похожи внешне и по смыслу, но первая будет работать со средним данным всего объекта, а вторая, получив в качестве параметра номер показателя, будет работать только с этим элементом массива показателей.

Функция поиска может вернуть номер элемента в массиве структур или указатель на объект. Покажем это на примере функций поиска. Функция поиска среднестатистического объекта вернет его номер в массиве структур. Имея функцию вывода, мы легко выведем его данные на экран с использованием операции разыменования []. Функция поиска среднего по k-му показателю вернет указатель на структуру. Имея функцию вывода, мы легко выведем его данные на экран с использованием операции разыменования *.

Все эти функции требуют полные данные обо всей группе объектов. Формальный параметр, используемый в их описании, тоже имеет имя Man, но теперь он имеет совсем другую смысловую нагрузку, обозначая имя всей совокупности данных обо всех объектах, т. е. о массиве объектов.

Поиск по ключевому значению, например, по фамилии, не внесет ничего нового, просто добавим его, чтобы лишний раз показать механизмы работы со строками символов.

Описание абстрактного типа «Объект» с именем Person использует инструкцию typedef, потому что объявлений объектов указанного типа будет несколько.

```
#include <stdio.h>
#include <io.h>#include <conio.h>
#include <math.h>
```

```
#include <string.h>

typedef struct
{
   char Name[10];
   // поле "Имя", статический символьный массив
   char *Surname;
   // поле "Фамилия", динамический символьный массив
   // для него нужно выделить динамическую память
   // перед присваиванием значения
   int Data[5];     // поле "Показатели", статический массив
   int Sum;         // поле "Сумма" накапливает и сохраняет
                    // сумму баллов
   float Var;       // поле "Среднее" вычисляет и сохраняет
                    // средний показатель
} Person;           // имя типа данного "Person"
```

Функция ввода данных об одном объекте возвращает объект.

```
void In(Person & Man)
// одно данное типа Person возвращается по ссылке
{
   puts("Введите имя: ");
   scanf("%s", &Man.Name);
   puts("Введите фамилию: ");
   // обязательно выделить память
   // число символов фамилии не более 10
   Man.Surname=new char[10];
   scanf("%s", Man.Surname);
   //& при вводе значения указателя не пишем
   puts("Введите данные: ");
   Man.Sum=0;
   // будет накоплена сумма показателей при вводе
   for(int i=0; i<5; i++)
   {
      scanf("%4d", &Man.Data [i]);
      Man.Sum+=Man.Data[i];
   }
   Man.Var=(float)Man.Sum/5.;   // вычислен средний
                                // показатель
```

```
}
```

Функция вывода полных данных об одном объекте.

```
void Out(Person Man)     // передается данное типа Person
{
   printf("%s ", Man.Name);
   printf("%s ", Man.Surname);
   for(int i=0; i<5; i++)
      printf("%4d",Man.Data[i]);
   // показатели выводятся одной строкой
   printf("\nСумма баллов=%d ", Man.Sum);
   printf("Среднее= %6.2f\n", Man.Var);
}
```

Функция вывода данных обо всех объектах в форме таблицы.

```
void Out_All(Person Man [], int n)
// передается n данных типа Person
{
   printf("=========================================\n");
   printf("ИМЯ \t ФАМИЛИЯ \t ДАННЫЕ \t СУММА СРЕДНЕЕ\n");
   printf("=========================================\n");
   for(int k= 0;k<n; k++)
   {
      printf("% –10s ", Man[k].Name);
      printf("% –10s ", Man[k].Surname);
      for(int i=0; i<5; i++)
      printf("%4d", Man[k].Data[i]);
      printf("%4d ",Man[k].Sum);
      printf("%6.2f \n",Man[k].Var);
   }
   printf("=========================================\n");
}
```

Функция вычисляет средний показатель по всем объектам, суммируя данные полей Var по всему массиву Man.

```
float Sred(Person Man [], int n)
// передается n данных типа Person
{
   float Var=0;              // переменная Var не имеет
                             // отношения к полю структуры
                             // с именем Var
```

```
  for(int i= 0; i<n; i++)
    Var+= Man[i].Var;
  return Var/(float)n;      // среднее по всем средним
}
```

Функция вычисляет средний показатель по k-му параметру. Номер показателя (от 0 до 5) — это один из параметров функции.

```
float Sred_k(Person Man[], int n, int k)
// передается n данных типа Person и номер показателя k
{
  float Var_k=0;            // Var_k – средний по одному
                            // показателю
  for(int i=0; i<n; i++)
    Var_k+=(float)Man[i].Data[k];
// складывает один k-й показатель по всем объектам
return Var_k/(float) n;     // среднее по k-му показателю
}
```

Функция Sred_Stat определяет номер среднестатистического объекта в массиве. Ей передается весь массив структур.

```
int Sred_Stat(Person Man [], int n)
// передается n данных типа Person
{
  float Var_All=Sred(Man,n);
  // средний показатель по всем объектам.
  float Var_min=fabs(Var_All–Man[0].Var);
  int Nom=0;     // наименьший имеет номер 0
  for(int i=0; i<n; i++)
  if(fabs(Var_All–Man[i].Var)<Var_min)
    Nom=i;       // запоминаем номер самого среднего
  return Nom;
}
```

Функция Sred_Stat_k определяет объект, среднестатистический по k-му показателю. Ей передается весь массив структур, функция возвращает указатель на найденный объект.

```
Person *Sred_Stat_k(Person Man[], int n, int k)
// передается n данных типа Person и номер показателя
{
```

```
  float Var_k=Sred_k(Man, n, k);
  // находим среднее значение k-го показателя
  // по всем объектам
  float Var_min=fabs(Var_k–Man[0].Data[k]);
  Person *Nom=Man;
  // указатель Nom запоминает адрес объекта
  for(int i=0; i<n; i++)
    if(fabs(Var_k–Man[i].Data[k])<Var_min)
      Nom = Man+i;
  // запоминаем адрес самого среднего
  return Nom;
}
```

Функция поиска объекта по фамилии. Передается *n* данных типа Person, и Who — строка, содержащая фамилию искомого объекта. Выполняется прямой поиск по всей группе объектов путем последовательного сравнения искомой строки с полем Surname каждого элемента массива Man. Функция возвращает указатель на найденный объект или NULL.

```
Person *Found_Fam(char *Who, Person Man[], int n)
{
  for(int i=0; i<n; i++)
    if(strcmp (Who ,Man[i].Surname)==0)  // если найдено
      return Man+i;                      // возвращает адрес
  return NULL;
  // возвращает NULL, если поиск неудачен
}
```

Продемонстрируем текст программы, которая объявляет данные и осуществляет управление ими путем вызова функций обработки данных.

```
void main (void)
{
int n;
// объявлен массив структур
Person All_Person[20];
printf("Введите количество\n");
// реальное количество данных n
scanf("%d", &n);

// ввод данных выполняется в цикле вызова функции In
```

```
for(int i=0; i<n; i++)
{
   In(All_Person[i]);
}

// Вывод на экран полной таблицы.
Out_All(All_Person,n);

// поиск номера среднестатистического объекта
int Found=Sred_Stat(All_Person, n);
printf("Самый средний имеет номер: %d\n", Found);
printf("Его данные: \n");
Out(All_Person [Found]);

// снова выведем таблицу на экран
Out_All(All_Person,n);

// поиск значения объекта, среднестатистического
// по k-му показателю
Person *Found_k;
for(int k=0;k<5;k++)
{
   printf("Средний по %d -му показателю\n", k);
   Found_k=Sred_Stat_k(All_Person, n, k);
   Out(*Found_k);
}

// поиск объекта по фамилии
char Surname[10];
// тоже фамилия, но не та, что в Person
puts("Введите строку для поиска\n");
scanf("%s", Surname);
Person *Who;    // хранит адрес, возвращенный функцией
Who=Found_Fam(Surname, All_Person, n);
   if(Who!=NULL)          // проверяем, найден объект или нет
      Out(*Who);
   else
      puts("Такого нет.\n");
} // End of main
```

Варианты заданий

Задание 1. Определить структуру для регистрации автомашин. Она должна иметь следующие поля: фамилия владельца, дата регистрации, марка машины, год выпуска, цвет, регистрационный номер.

Написать и протестировать функции для регистрации новой машины, удаления машины из регистрационного списка, поиска машины по марке и по цвету, а также по сочетанию признаков «марка и цвет».

Задание 2. Определить структуру, описывающую багаж пассажира, с полями: количество вещей и общий вес вещей. Пусть имеются данные о багаже нескольких пассажиров, где информация о багаже каждого отдельного пассажира представляет собой соответствующую пару чисел.

Написать и протестировать функции для ввода и вывода общей информации о багаже. Найти число пассажиров, имеющих не более двух вещей. Найти число пассажиров, количество вещей которых превосходит среднее число вещей.

Задание 3. Определить структуру, описывающую багаж пассажира, с полями: количество вещей и общий вес вещей. Пусть имеются данные о багаже нескольких пассажиров, где информация о багаже каждого отдельного пассажира представляет собой соответствующую пару чисел.

Написать и протестировать функции для ввода и вывода общей информации о багаже. Определить, имеются ли два пассажира, багажи которых совпадают по числу вещей и различаются по весу не более чем на 0,5 кг. Выяснить, имеется ли пассажир, багаж которого превышает багаж каждого из остальных пассажиров и по числу вещей и по весу.

Задание 4. Определить структуру, описывающую багаж пассажира, с полями: количество вещей и общий вес вещей. Пусть имеются данные о багаже нескольких пассажиров, где информация о багаже каждого отдельного пассажира представляет собой соответствующую пару чисел.

Написать и протестировать функции для ввода и вывода общей информации о багаже. Выяснить, имеется ли пассажир, багаж которого состоит из одной вещи весом не менее 30 кг. Дать сведения о багаже, число вещей в котором не меньше, чем в любом другом багаже, а вес вещей не больше, чем в любом другом багаже с этим же числом вещей.

Задание 5. Определить структуру, описывающую сведения об ученике, с полями: имя, фамилия, название класса, в котором он учится (год обучения и буква).

Написать и протестировать функции для добавления нового ученика, для вывода информации об одном ученике и вывода общей информации. Выяснить, имеются ли в школе однофамильцы. Выяснить, имеются ли однофамильцы в каких-нибудь классах.

Задание 6. Определить структуру, описывающую сведения об ученике, с полями: имя, фамилия, название класса, в котором он учится (год обучения и буква).

Написать и протестировать функции для добавления нового ученика, для вывода информации об одном ученике и вывода общей информации. Выяснить, в каких классах больше чем n учащихся. Выяснить, сколько человек учится в каждой параллели.

Задание 7. Определить структуру, описывающую сведения об ученике, с полями: имя, фамилия, название класса, в котором он учится (год обучения и буква), а также отметки, полученные учениками в последней четверти.

Написать и протестировать функции для добавления нового ученика, для вывода информации об одном ученике и вывода общей информации. Выяснить, сколько учеников имеют отметки не ниже 4. Выяснить, сколько учеников имеют тройки в четверти.

Задание 8. Определить структуру, описывающую дату, с полями число, название месяца, год. Написать и протестировать функции для ввода и вывода даты в формате «ЧЧ. ММ.ГГ» и «ЧЧ Месяц ГГ». Пусть есть некоторое количество дат, связанных с некоторым событием, например, исторической датой. Написать и протестировать функции для сортировки этих данных по дате и по названию события.

Задание 9. Определить структуру, описывающую сведения о книге, с полями: фамилия автора, название и год издания.

Написать и протестировать функции для добавления новой книги, для вывода информации об одной книге и вывода общей информации. Написать и протестировать функции для поиска по фамилии автора, по году издания, а также по совокупности этих признаков.

Задание 10. Определить структуру, описывающую сведения о книге, с полями: фамилия автора, название, год издания.

Написать и протестировать функции для добавления новой книги, для вывода информации об одной книге и вывода общей информации. Написать и протестировать функции для поиска по названию книги, и по ключевому слову названия. Например, если название книги «Информатика и программирование», то она должна быть найдена и по ключевому слову «Информатика», и по ключевому слову «Программирование». Если таких книг несколько, то сообщить сведения обо всех книгах.

Задание 11. Определить структуру, описывающую сведения о сотрудниках учреждения, с полями: фамилия, имя, отчество, занимаемая должность, номер телефона.

Написать и протестировать функции для добавления нового сотрудника, для вывода информации об одном человеке и вывода общей информации. Написать и протестировать функции для поиска сотрудника по фамилии и по должности.

Задание 12. Определить структуру, описывающую кубики: размер (длина ребра в сантиметрах), цвет (красный, желтый, зеленый или синий) и материал (деревянный, металлический, картонный).

Написать и протестировать функции для ввода и вывода данных, а также функции поиска количества кубиков каждого из перечисленных цветов и их суммарного объема.

Задание 13. Определить структуру, описывающую кубики: размер кубика (длина ребра в сантиметрах), его цвет (красный, желтый, зеленый или синий) и материал (деревянный, металлический, картонный).

Написать и протестировать функции для ввода и вывода данных, а также функции поиска количества кубиков с указанными значениями «материал» и «длина ребра». Например, найти количество деревянных кубиков с ребром 3 см и количество металлических кубиков с ребром 5 см.

Задание 14. Определить структуру, описывающую сведения о веществах: название вещества, его удельный вес и проводимость (проводник, полупроводник, изолятор).

Написать и протестировать функции для ввода и вывода данных. Написать и протестировать функции для поиска по названию вещества и по проводимости. Найти удельные веса и названия всех проводников.

Задание 15. Определить структуру, описывающую сведения о веществах: название вещества, его удельный вес и проводимость (проводник, полупроводник, изолятор).

Написать и протестировать функции для ввода и вывода данных. Написать и протестировать функции для сортировки данных по любому из признаков.

Задание 16. Определить структуру, описывающую сведения об игрушках: название игрушки, ее стоимость и возрастные границы детей, для которых она предназначена (например, для детей от 2 до 5 лет).

Написать и протестировать функции для ввода и вывода данных. Написать и протестировать функцию поиска, чтобы получить сведения о названиях игрушек, цена которых не превышает k руб. и которые подходят детям n лет. Написать и протестировать функцию поиска, чтобы получить сведения о самом дорогом, например, конструкторе.

Задание 17. Определить структуру, описывающую сведения об игрушках: название игрушки, ее стоимость и возрастные границы детей, для которых она предназначена (например, для детей от 2 до 5 лет).

Написать и протестировать функции для ввода и вывода данных. Написать и протестировать функцию поиска по наименованию. Написать и протестировать функцию поиска, чтобы получить сведения о названиях наиболее дорогих игрушек (цена которых отличается от цены самой дорогой игрушки не более чем на 10 руб.).

Задание 18. Определить структуру, описывающую сведения об игрушках: название игрушки, ее стоимость и возрастные границы детей для которых она предназначена (например, для детей от 2 до 5 лет).

Написать и протестировать функции для ввода и вывода данных. Написать и протестировать функцию поиска по диапазону цены. Написать и протестировать функцию поиска, чтобы подобрать игрушку любую, кроме мяча, подходящую ребенку 3 лет, и дополнительно мяч так, чтобы суммарная стоимость игрушек не превосходила 100 руб.

Задание 19. Определить структуру, описывающую сведения об игрушках: название игрушки, ее стоимость и возрастные границы детей для которых она предназначена (например, для детей от 2 до 5 лет).

Написать и протестировать функции для ввода и вывода данных. Написать и протестировать функцию поиска, чтобы подобрать игрушку по названию, по цене, и по возрастному диапазону. Например, все мячи ценой *n* руб., предназначенный детям от 3 до 8 лет.

Задание 20. Имеются сведения о сотрудниках учреждения в следующем виде: табельный номер сотрудника, его фамилия, имя, отчество, оклад. Определить структуру для представления этих данных.

Написать и протестировать функции для ввода данных и вывода в виде таблицы. Написать и протестировать функции для поиска по табельному номеру, по фамилии. Вычислить среднюю заработную плату всех сотрудников.

Задание 21. Имеются сведения об изделиях, выпускаемых малым предприятием: наименование изделия, годовой план выпуска изделий в штуках, фактический поквартальный выпуск каждого изделия.

№ п/п	Наименование изделия	Годовой план выпуска	Фактический выпуск по кварталам			
			I	II	III	IV
1	Стул	100	10	20	35	35

Определить структуру для представления этих данных.

Написать и протестировать функции для ввода данных и вывода в виде таблицы. Написать и протестировать функции для определения фактического выпуска изделий и процента выполнения плана по каждому виду изделий. Написать функцию поиска по наименованию с выводом общей информации.

Задание 22. Имеются данные о выпускаемых изделиях: наименование изделия, артикул, себестоимость изделия и его цена. Определить структуру для представления этих данных. Написать и протестировать функции для ввода данных и вывода в форме полной ведомости выпускаемых изделий следующего вида.

№ п/п	Наименование изделия	Артикул	Себестоимость	Цена

Написать и протестировать функции для поиска изделий по наименованию и по диапазону цен. (Например, цена не превышает K_1 руб., но более K_2 руб.)

Задание 23. Известны сведения о себестоимости некоторых видов продукции: наименование, Jм — индекс изменения норм расхода на данный вид материальных затрат; Jц — индекс изменения оптовых цен на данный вид; Jп — индекс выпуска продукции; Do — отношение стоимости основных и вспомогательных материалов к выпуску товарной продукции по отчету; Tпп — объем товарной продукции по плану. На основании этих данных может быть вычислена экономия от снижения себестоимости, которая вычисляется по формуле:

$$Э = (1 - Jм \cdot Jц) \cdot Do \cdot Jп \cdot Tпп.$$

Определить структуру для представления этих данных. Разработать функции для ввода данных и вывода в форме выходной ведомости следующего вида.

Наименование	Jм	Jц	Jп	Do	Tпп	Э

Здесь Э — экономия, которая должна быть вычислена для каждого наименования. Написать и протестировать

функции для добавления новых данных, а также для поиска наименований, для которых достигнута наибольшая и наименьшая экономия.

Задание 24. В магазине ведется учет продажи товаров. По каждому товару известны наименование товара, цена, торговая надбавка, количество проданного товара. Определить структуру для представления этих данных. Разработать функции для ввода данных и вывода в форме отчетной ведомости следующего вида.

Наименование товара	Цена, руб.	Торговая надбавка, %	Количество	Общая стоимость

Написать и протестировать функции для добавления данных о новом товаре, для поиска по наименованию, для вычисления общей итоговой стоимости продаж.

Задание 25. Известны данные о выпускаемой продукции по цехам предприятия: название цеха, отношение стоимости основных и вспомогательных материалов к выпуску продукции по отчету Do и по плану Dп, объем товарной продукции по плану Tп.

Определить структуру для представления этих данных. Разработать функции для ввода данных и вывода в форме выходной ведомости следующего вида.

Цех	Отношение стоимости основных и вспомогательных материалов к объему выпускаемой продукции		Объем товарной продукции по плану	Экономия от снижения себестоимости
	Do	Dп	Tп	Э

Здесь экономия от снижения себестоимости по каждому цеху вычисляется по формуле:

$$Э = (Do - Dп) \cdot Tп.$$

Написать и протестировать функции, чтобы найти цеха, для которых достигнута наибольшая и наименьшая экономия, а также для поиска информации по названию цеха.

Задание 26. В выпуске продукции принимают участие несколько бригад, для которых ведется помесячный учет. Бригады имеют номера. Определить структуру для представ-

ления этих данных. Разработать функции для ввода данных и вывода в форме выходной ведомости следующего вида.

Месяц	Номер бригады				Итоговая выработка по месяцам
	1	2	3	4	

Написать и протестировать функции, чтобы найти среднюю выработку по бригадам за весь отчетный период, а также для поиска сведений о выработке за указанный месяц по названию месяца.

Задание 27. Предприятие нанимает работников на сдельную работу. Для каждого из них ведется ежедневный учет: фамилия, имя, отчество, разряд, расценка за единицу продукции, количество произведенных изделий. Определить структуру для представления этих данных. Разработать функции для ввода данных и вывода в форме расчетной ведомости следующего вида.

ФИО	Разряд	Расценки	Количество	Сумма

Написать и протестировать функции, чтобы найти общую выработку за день и общую сумму к оплате. Сохранить эти данные в виде таблицы.

Дата	Произведено (количество)	Оплачено

Задание 28. Для каждого участка цеха завода имеются сведения о количестве отпущенных для производства материалов: наименование материала, единица измерения, объем отпущенного участку материала. Определить структуру для представления этих данных. Разработать функции для ввода данных и вывода с подведением итогов в форме выходной ведомости следующего вида.

Наименование	Единицы измерения	Участок 1	Участок 2	Участок 3	Итого

Написать и протестировать функции, чтобы найти общий расход материалов каждым участком, и всего по цеху, а также для поиска по наименованию.

Задание 29. На складе ведется учет наличия товаров. Для подведения итогов месяца есть сведения: наименование товара; количество поступившего и реализованного товара за месяц. Определить структуру для представления этих данных. Разработать функции для ввода данных и вывода в форме оборотной ведомости следующего вида.

Наименование товара	Поступило за месяц	Реализовано за месяц	Разница

Написать и протестировать функции, чтобы найти общий приход товара и общий расход, а также для вывода отчета об остатках товара на складе по форме.

Наименование товара	Остаток

Задание 30. На предприятие поступает сырье разного вида. Учет поступления сырья ведется по следующим позициям: вид сырья (наименование), плановое поступление, фактическое поступление. Определить структуру для представления этих данных. Разработать функции для ввода данных и вывода в форме отчетной ведомости следующего вида.

Вид сырья	Плановое поступление	Фактическое поступление	Процент выполнения плана

Написать и протестировать функции, чтобы найти общий объем плановых поступлений и общий объем фактических поступлений, а также для поиска по наименованию.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Вирт, Н.* Алгоритмы и структуры данных / пер. с англ. Д. Б. Подшивалова. — М. : Мир, 1989. — 360 с.
2. *Гагарина, Л. Г.* Алгоритмы и структуры данных : учеб. пособие / Л. Г. Гагарина, В. Д. Колдаев. — М. : Финансы и статистика : ИНФРА-М, 2009. — 302 с.
3. *Голицына, О. Л.* Основы алгоритмизации и программирования : учеб. пособие / О. Л. Голицына, И. И. Попов. — М. : Форум, 2010. — 429 с.
4. *Канцедал, С. А.* Алгоритмизация и программирование : учеб. пособие. — М. : Форум : ИНФРА-М, 2011. — 351 с.
5. *Культин, Н. Б.* С/С++ в задачах и примерах. — СПб. : БХВ-Петербург, 2004. — 288 с.
6. *Мишенин, А. И.* Сборник задач по программированию : учеб.-метод. пособие. — М. : Финансы и статистика : ИНФРА-М, 2009. — 221 с.
7. *Незнанов, А. А.* Программирование и алгоритмизация : учебник. — М. : Академия, 2010. — 303 с.
8. *Павловская, Т. А.* С/С++. Программирование на языке высокого уровня : учебник. — СПб. : Питер, 2013. — 238 с.
9. *Павловская, Т. А.* Программирование на языке высокого уровня : учебник. — СПб. : Питер, 2013. — 432 с.
10. *Подбельский, В. В.* Программирование на языке Си : учеб. пособие / В. В. Подбельский, С. С. Фомин. — М. : Финансы и статистика, 2009. — 600 с.
11. С и С++: Алгоритмы и приемы программирования : пер с англ. / А. Фридман, Л. Кландер, М. Михаэлис, Г. Шилдт ; под ред. В. Тимофеева. — М. : Бином: Бином-пресс, 2003. — 560 с.
12. *Синицын, С. В.* Программирование на языке высокого уровня : учебник / С. В. Синицин, А. С. Михайлов, О. И. Хлытчиев. — М. : Академия, 2010. — 392 с.
13. *Тимофеева, Т. С.* Задачи по программированию : учеб. пособие / Т. С. Тимофеева, О. А. Тихонова, И. Г. Ларионова ; под ред. В. И. Васильева. — М. : Вузовская книга, 2011. — 94 с.
14. *Франка, П.* С++. — СПб. : Питер, 2012. — 491 с.
15. *Чиртик, А. А.* Программирование на С++. — СПб. : Питер, 2010. — 346 с.
16. *Шилдт, Г.* Самоучитель С++ / пер. с англ. А. Жданова. — СПб. : BHV-Петербург, 2005. — 683 с.

ОГЛАВЛЕНИЕ

Елена Александровна КОНОВА,
Галина Андреевна ПОЛЛАК

АЛГОРИТМЫ И ПРОГРАММЫ. ЯЗЫК С++

Учебное пособие

Издание второе, стереотипное

Зав. редакцией
естественнонаучной литературы *М. В. Рудкевич*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com
196105, Санкт-Петербург, пр. Ю. Гагарина, д. 1, лит. А.
Тел./факс: (812) 336-25-09, 412-92-72.
Бесплатный звонок по России: 8-800-700-40-71

ГДЕ КУПИТЬ

ДЛЯ ОРГАНИЗАЦИЙ:

Для того, чтобы заказать необходимые Вам книги, достаточно обратиться в любую из торговых компаний Издательского Дома «ЛАНЬ»:

по России и зарубежью
«ЛАНЬ-ТРЕЙД». 196105, Санкт-Петербург, пр. Ю. Гагарина, д. 1, лит. А.
тел.: (812) 412-85-78, 412-14-45, 412-85-82; тел./факс: (812) 412-54-93
e-mail: trade@lanbook.ru; ICQ: 446-869-967

www.lanbook.com
пункт меню **«Где купить»**
раздел **«Прайс-листы, каталоги»**

в Москве и в Московской области
«ЛАНЬ-ПРЕСС». 109263, Москва, 7-я ул. Текстильщиков, д. 6/19
тел.: (499) 178-65-85; e-mail: lanpress@lanbook.ru

в Краснодаре и в Краснодарском крае
«ЛАНЬ-ЮГ». 350901, Краснодар, ул. Жлобы, д. 1/1
тел.: (861) 274-10-35; e-mail: lankrd98@mail.ru

ДЛЯ РОЗНИЧНЫХ ПОКУПАТЕЛЕЙ:

интернет-магазин
Издательство «Лань»: http://www.lanbook.com

магазин электронных книг
Global F5: http://globalf5.com/

Подписано в печать 01.11.16.
Бумага офсетная. Гарнитура Школьная. Формат 84×108 $^{1}/_{32}$.
Печать офсетная. Усл. п. л. 20,16. Тираж 100 экз.

Заказ № 329-16.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в ПАО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.