Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

“Севастопольский государственный университет”

Методические указания к лабораторным работам по дисциплине “Алгоритмизация и программирование” для студентов дневной и заочной форм обучения направления 09.03.02 – “Информационные системы и технологии”

Часть 2

Севастополь

2016

УДК 004.42 (075.8)

Методические указания к лабораторным работам по дисциплине “Алгоритмизация и программирование” для студентов дневной и заочной форм обучения направления 09.03.02 — “Информационные системы и технологии”, часть 2/ Сост. В.Н. Бондарев, Т.И. Сметанина. — Севастополь: Изд-во СевГУ 2016. — 64 с.

Методические указания предназначены для проведения лабораторных работ и обеспечения самостоятельной подготовки студентов по дисциплине “Алгоритмизация и программирование”. Целью методических указаний является обучение студентов основным принципам структурного программирования, навыкам разработки программ на языках Паскаль и C/C++.

Методические указания составлены в соответствии с требованиями программы дисциплины “Алгоритмизация и программирование” для студентов направления 09.03.02 — “Информационные системы и технологии” и утверждены на заседании кафедры информационных систем протокол № 1 от 12 сентября 2016 года.

Допущено научно-методической комиссией института информационных технологий и систем управления в технических системах в качестве методических указаний.

Рецензент: Кожаев Е.А., к.т.н., доцент кафедры информационных технологий и компьютерных систем

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цели и задачи лабораторных работ

Основная цель выполнения настоящих лабораторных работ — получение практических навыков создания и отладки программ на языках Паскаль и C/C++ с использованием сложных типов данных. В результате выполнения лабораторных работ студенты должны углубить знания основных теоретических положений дисциплины ”Алгоритмизация и программирование”, решая практические задачи на ЭВМ.

Студенты должны получить практические навыки работы в интегрированной среде программирования, навыки разработки программ, использующих регулярные и комбинированные типы данных, текстовые и типизированные файлы.

Выбор вариантов и график выполнения лабораторных работ

В лабораторных работах студент решает заданную индивидуальным вариантом задачу. Варианты заданий приведены к каждой лабораторной работе и уточняются преподавателем.

Лабораторная работа выполняется в два этапа. На первом этапе — этапе самостоятельной подготовки — студент должен выполнить следующее:

- проработать по конспекту и рекомендованной литературе, приведенной в конце настоящих методических указаний, основные теоретические положения лабораторной работы и подготовить ответы на контрольные вопросы;
- разработать алгоритм решения задачи и составить схему алгоритма;
- описать схему алгоритма;
- написать программу на языке Паскаль (C/C++) и описать ее;
- оформить результаты первого этапа в виде заготовки отчета по лабораторной работе (**студенты-заочники** первый этап выполнения лабораторных работ оформляют в виде контрольной работы, которую сдают на кафедру до начала зачетно-экзаменационной сессии).

На втором этапе, выполняемом в лабораториях кафедры, студент должен:

- отладить программу;
- разработать и выполнить тестовый пример.

Выполнение лабораторной работы завершается защитой отчета. Студенты должны выполнять и защищать работы **строго по графику**. График защиты лабораторных работ студентами дневного отделения:

- лабораторная работа №1 — 1-ая неделя весеннего семестра;
- лабораторная работа №2 — 3-ая неделя весеннего семестра.
- лабораторная работа №3 — 6-ая неделя весеннего семестра;
- лабораторная работа №4 — 8-ая неделя весеннего семестра.

График уточняется преподавателем, ведущим лабораторные занятия.

Требования к оформлению отчета

Отчёты по лабораторной работе оформляются каждым студентом индивидуально. Отчёт должен включать: название и номер лабораторной работы; цель работы; вариант задания и постановку задачи; разработку и описание алгоритма решения задачи; текст и описание программы; результаты выполнения программы; выводы по работе; приложения. Содержание отчета указано в методических указаниях к каждой лабораторной работе.

Постановка задачи представляет собой изложение содержания задачи и цели её решения. На этом этапе должны быть полностью определены исходные данные, перечислены задачи по преобразованию и обработке входных данных, дана характеристика результатов, описаны входные и выходные данные.

Разработка алгоритмов решения задачи предполагает математическую формулировку задачи и описание решения задачи на уровне схем алгоритмов. Схемы алгоритмов должны быть выполнены в соответствии с требованиями ГОСТ и сопровождаться описанием.

Описание программы включает описание вычислительного процесса на языке Паскаль. Кроме текста программы, в отчёте представляется её пооператорное описание.

В приложении приводится текст программы, распечатки, полученные во всех отладочных прогонах программы с анализом ошибок, результаты решений тестового примера.

1 ЛАБОРАТОРНЯ РАБОТА №1
“ПРОГРАММИРОВАНИЕ ЛИНЕЙНЫХ СПИСКОВ”

1.1 Цель работы

Исследование списковых структур данных и приобретение навыков разработки и отладки программ, использующих динамическую память. Исследование особенностей использования переменных ссылочного типа.

1.2 Краткие теоретические сведения

1.2.1 Ссылочный тип

В предыдущих лабораторных работах рассматривались *статические переменные*. Память под такие переменные отводится заранее на этапе компиляции программы. Часть возникает необходимость в использовании переменных, которые создаются и уничтожаются в процессе выполнения программы. Такие переменные называют *динамическими*, а память, которая для них выделяется — *динамической памятью*. Для работы с динамической памятью используют *переменные ссылочного типа (указатели)*. Задание ссылочного типа выполняется следующим образом:

<задание ссылочного типа>::=^<имя типа>

Например:

```
Type
      IntPtr=^Integer;
      Massive=Array[1..10] of Real;
Var
      P:IntPtr;
      Rabmas:^Massiv;
```

Здесь ссылочная переменная P — это указатель, значением которого является адрес области памяти, в которой хранится целое значение. Ссылочная переменная RABMAS — указатель на область памяти, где хранится массив.

В языке Turbo Pascal различают *типизированные и нетипизированные* указатели. Приведенные выше указатели могут ссылаться лишь на данные соответствующего типа и называются типизированными. Нетипизированный указатель не связывается с каким-либо типом данных и описывается как переменная типа Pointer:

```
Var Ukaz:Pointer;
```

Такой указатель используют для ссылки на данные, тип и структура которых меняются во время выполнения программы.

После описания ссылочной переменной выделяется область динамической памяти, на которую она будет ссылаться. Для этого используют процедуру New(<имя ссылочной переменной>), которая выделяет необходимый объем памяти и присваивает переменной ссылочного типа значение, соответствующее начальному адресу выделенной памяти.

Для того чтобы записать значение в выделенную память, используют переменную с указателем

<переменная с указателем>::=<ссылочная переменная>^

Например:

New(P); P^:=5;

В этом случае в область динамической памяти, на которую ссылается указатель P, записывается значение 5 (рисунок 1.1). На рисунке символом "*" обозначено (в общем случае неизвестное программисту) значение ссылочной переменной P.

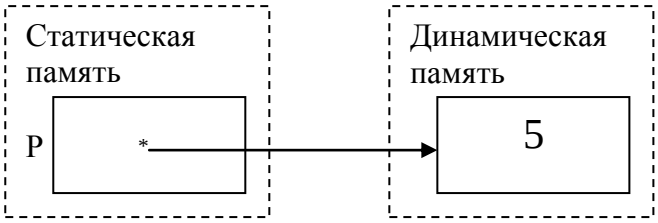

Рисунок 1.1 — Указатель и динамическая память

Для уничтожения объектов в динамической памяти применяется процедура Dispose(<имя ссылочной переменной>). Данная процедура освобождает область динамической памяти, на которую ссылается параметр процедуры, после чего эта область становится доступной для дальнейшего распределения между другими динамическими переменными.

Для работы с нетипизированными указателями в языке Turbo Pascal используются процедуры GetMem и FreeMem. Процедура GetMem выделяет область динамической памяти заданного объема, а процедура FreeMem освобождает эту область памяти. Вызов этих процедур выполняется следующим образом:

GetMem(<имя ссылочной переменной>, <объем памяти>);

FreeMem (<имя ссылочной переменной>, <объем памяти>);

Здесь ссылочная переменная должна иметь тип Pointer, а объем памяти не должен превышать 65521 байт.

1.2.2 Списковые структуры данных

Из динамических структур в программах наиболее часто используют различные линейные списки, стеки, очереди, деревья. Они различаются способами связи отдельных элементов и допустимыми операциями над ними.

Линейные структуры данных предназначены для отображения линейных связей. Линейные структуры данных могут быть последовательными и списковыми. *Последовательной* называется структура, в которой логический порядок следования элементов задается их физическим порядком. Примером последовательной структуры данных является массив. *Списковой* называется такая структура данных, при которой логический порядок следования элементов задается путем отсылок, то есть каждый элемент списка, кроме последнего, содержит указатель на следующий элемент (или предыдущий). Доступ к первому

элементу списка выполняется с помощью специального указателя — указателя на вершину (голову) списка.

Односвязным линейным списком называют список, в котором предыдущий элемент ссылается на следующий. *Двусвязный линейный список* — это список, в котором предыдущий элемент ссылается на следующий, а следующий — на предыдущий. *Односвязный циклический список* — это односвязный линейный список, в котором последний элемент ссылается на первый. *Стек* — это односвязный список, в котором компоненты добавляются и удаляются только со стороны вершины списка. *Очередь* — это односвязный список, в котором компоненты добавляются в конец списка, а удаляются со стороны вершины списка.

Для задания списковых структур необходимо определить элемент списка в виде записи, в состав которой входит поле-указатель на элемент этого же типа. Для этого в языке Pascal разрешено определять ссылочные типы на еще не описанные типы значений:

```
Type
  Data=<тип данных элемента списка>;
  Ukaz=^Element;
  Element=Record
            D:Data;
            Next:Ukaz    { поле-указатель  на элемент}
          End;
```

Определив ссылочный тип Ukaz, можно с его помощью построить следующий связный однонаправленный список (рисунок 1.2).

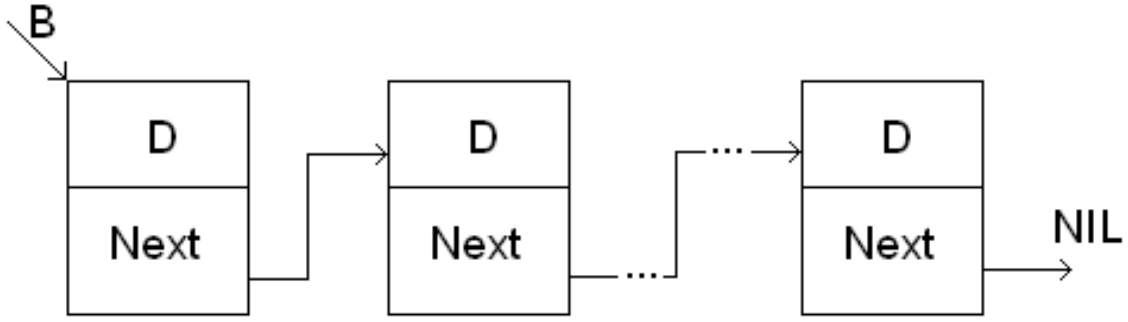

Рисунок 1.2 — Связный однонаправленный список

Здесь B является указателем на вершину списка. Каждый элемент списка содержит поле данных D и поле Next — указатель следующего элемента. Поле Next последнего элемента содержит пустую ссылку NIL.

Если Temp — переменная ссылочного типа, указывающая на элемент списка, предшествующий удаляемому, то чтобы исключить элемент из списка достаточно выполнить следующий оператор:

```
Temp^.Next:=Temp^.Next^.Next;
```

Чтобы вставить элемент в произвольное место списка, кроме начала, необходимо изменить значения двух указателей. Пусть ссылочная переменная Place указывает на элемент, после которого необходимо вставить новый элемент.

Если новый элемент адресуется указателем NewE, то операция вставки реализуется операторами:

```
NewE^.Next:= Place^.Next; Place^.Next:=NewE;
```

При добавлении элемента в начало списка или исключении начального элемента необходимо использовать указатель вершины списка B. Операция удаления первого элемента списка реализуется одним оператором

```
B:=B^.Next;
```

Операция добавления в начало списка — двумя операторами:

```
NewE^.Next:=B; B:=NewE;
```

1.2.3 Пример программы работы с очередью

Рассмотрим программу, выполняющую организацию очереди, добавление элемента в очередь, удаление элемента из очереди. Очередь — это линейный список, добавление элементов в который выполняется с одной стороны, а исключение — с другой. При этом используется специальные указатели начала и конца очереди. В примере элементами очереди являются записи, содержащие сведения о грузополучателе. Каждая запись содержит следующие поля: фамилия, имя, отчество грузополучателя, адрес.

```
Program DemoQuery;
Uses Crt;
Type
   Data=  Record  {Описание  записи о грузополучателе}
              FIO:string[15];
             Adr:string[30]
          End;
   Ukaz=^Query; {Описание указателя на элемент очереди}
   Query=Record
              Inf:Data;
              Next:Ukaz
           End;
Var  NewE,Left,Right,Temp:Ukaz; {Указатели}
                          Z:Data;   {Запись, добавляемая в очередь}
                       Key:Char;   {номер пункта меню программы}
Procedure Org; {Процедура организации очереди}
Begin
   Writeln('Выполняется процедура организации очереди');
   Writeln('Для выхода из процедуры вводите символ * ');
   Writeln('========================================= ');
   Writeln('Введите фамилию и инициалы грузополучателя');
   Readln(Z.Fio);
   Writeln('Введите адрес грузополучателя ');
   Readln(Z.Adr);
   If Z.Fio='*' Then Exit;  {Выход из процедуры при вводе '*'}
   New(NewE);               {Создание нового элемента     }
```

```
    NewE^.Inf.Fio:=Z.Fio;    {Заполнение его полей          }
    NewE^.Inf.Adr:=Z.Adr; ;
    NewE^.Next:=nil;
    Right:=NewE; { Right - указатель хвоста очереди }
    Left:=NewE;{ Left - указатель головы очереди }
    While True Do          {Повторение этих же действий     }
     Begin
      Writeln('Введите фамилию и инициалы грузополучателя');
      Readln(Z.Fio);
      Writeln('Введите адрес грузополучателя');
      Readln(Z.Adr);
      If Z.Fio='*' Then Exit;
      New(NewE);
      NewE^.Inf.Fio:=Z.Fio;
      NewE^.Inf.Adr:=Z.Adr;
      NewE^.Next:=Nil;
      Right^.Next:=NewE; {Связь с предыдущим элементом}
      Right:=NewE          {Перемещение указателя хвоста очереди}
     End
End;
Procedure Dob;  {Добавление элемента в конец очереди}
Begin
    Writeln('Введите фамилию и инициалы грузополучателя');
    Readln(Z.Fio);
    Writeln('Введите адрес грузополучателя');
    Readln(Z.Adr);
    If Z.Fio='*' Then Exit;
    New(NewE);
    NewE^.Inf.Fio:=Z.Fio;
    NewE^.Inf.Adr:=Z.Adr;
    NewE^.Next:=nil;
    If Right=Nil Then          {Если добавляется первый элемент, то}
      Left:=NewE               {инициализировать указатель головы}
    Else
      Right^.Next:=NewE; {иначе добавить в конец очереди}
    Right:=NewE;
End;
Procedure Udal; {Процедура исключения элемента}
Begin
 Writeln('Исключается головной элемент очереди');
 Writeln('Нажмите клавишу Enter');
 Readln;
 If Left<>Nil Then    {Если очередь не пустая, то}
 Begin
```

```
    Temp:=Left;        {запоминаем указатель на  голову очереди}
    Left:=Left^.Next;  {указатель головы смещаем на 2-ой элемент}
    Dispose(Temp);  {освобождаем память от головного элемента}
    If Left=Nil Then   {Если удалили последний элемент, то}
      Right:=Nil;        {указатель на конец очереди равен Nil}
  End
End;
Procedure Prosmotr; {Процедура просмотра очереди}
Var i:integer;      {Просмотр выполняется от головы к хвосту}
Begin
    Writeln('Очередь содержит следующие элементы:');
    Temp:=Left;
    i:=1;
    While Temp<>nil Do
     Begin
      Writeln(i,' ',Temp^.Inf.Fio,' ',Temp^.Inf.Adr);
      Temp:=Temp^.Next;
      i:=i+1;
    End;
    Writeln('Нажмите клавишу Enter');
    Readln;
End;
{==============основная программа===============}
begin
     Right:=Nil;      Left:=Nil;
     Repeat
     ClrScr; { очистка экрана }
   {вывод на экран пунктов меню}
     Writeln('1-Организация очереди');
     Writeln('2-Добавление элемента в очередь');
     Writeln('3-Удаление элемента из очереди');
     Writeln('4-Просмотр очереди');
     Writeln('5-Выход');
     Writeln('------------------------------');
     Writeln('Нажмите клавишу от 1 до 5');
     Key:=ReadKey; {считывание кода нажатой клавиши}
     Case Key Of { вызов необходимой процедуры по номеру}
      ‘1’:Org;
      ‘2’:Dob;
      ‘3’:Udal;
      ‘4’:Prosmotr;
      End
     Until Key=’5’ {выход из программы}
End.
```

1.3 Варианты заданий

Представить одну из приведенных ниже таблиц в виде линейного списка L, элементами которого являются строки таблицы. Написать процедуры организации, добавления элемента в список, исключения элемента из списка, просмотра списка, а также одну из процедур в соответствии с вариантом, приведенным ниже.

Значения и количество записей в таблице студент выбирает самостоятельно. Исходные данные после организации списка должны сохраняться в файле и при повторном запуске программы считываться из файла. Количество строк таблицы не задается.

Таблица 3.1 — Ведомость

N	Фамилия	Имя	Отчество	Оценки		
				Математика	История	Физика

Таблица 3.2 — Расписание

N Поезда	Станция отправления	Станция назначения	Время отправления	Время прибытия	Стоимость билета

Таблица 3.3 — Анкета

N	ФИО	Год рождения	Пол	Семейное состояние	Количество детей	Оклад

Вариант 1 Таблица 3.1. Процедуру, которая вставляет в начало списка L новый элемент E.

Вариант 2 Таблица 3.2. Процедуру, которая вставляет в начало списка L новый элемент E.

Вариант 3 Таблица 3.3. Процедуру, которая вставляет в начало списка L новый элемент E.

Вариант 4 Таблица 3.1. Процедуру, которая вставляет в конец списка L новый элемент E.

Вариант 5 Таблица 3.2. Процедуру, которая вставляет в конец списка L новый элемент E.

Вариант 6 Таблица 3.3. Процедуру, которая вставляет в конец списка L новый элемент E.

Вариант 7 Таблица 3.1. Процедуру, которая вставляет новый элемент E после первого элемента непустого списка L.

Вариант 8 Таблица 3.2. Процедуру, которая вставляет новый элемент E после первого элемента непустого списка L.

Вариант 9 Таблица 3.3. Процедуру, которая вставляет новый элемент E после первого элемента непустого списка L.

Вариант 10 Таблица 3.1. Процедуру, которая вставляет в список L новый элемент E1 за каждым вхождением элемента E.

Вариант 11 Таблица 3.2. Процедуру, которая вставляет в список L новый элемент E1 за каждым вхождением элемента E.

Вариант 12 Таблица 3.3. Процедуру, которая вставляет в список L новый элемент E1 за каждым вхождением элемента E.

Вариант 13 Таблица 3.1. Процедуру, которая вставляет в непустой список L пару новых элементов E1 и E2 перед его последним элементом.

Вариант 14 Таблица 3.2. Процедуру, которая вставляет в непустой список L пару новых элементов E1 и E2 перед его последним элементом.

Вариант 15 Таблица 3.3. Процедуру, которая вставляет в непустой список L пару новых элементов E1 и E2 перед его последним элементом.

Вариант 16 Таблица 3.1. Процедуру, которая вставляет в непустой список L, элементы которого упорядочены по возрастанию значений одного из полей таблицы, новый элемент E так, чтобы сохранилась упорядоченность.

Вариант 17 Таблица 3.2. Процедуру, которая вставляет в непустой список L, элементы которого упорядочены по возрастанию значений одного из полей таблицы, новый элемент E так, чтобы сохранилась упорядоченность.

Вариант 18 Таблица 3.3. Процедуру, которая вставляет в непустой список L, элементы которого упорядочены по возрастанию значений одного из полей таблицы, новый элемент E так, чтобы сохранилась упорядоченность

Вариант 19 Таблица 3.1. Процедуру, которая удаляет из непустого списка L первый элемент.

Вариант 20 Таблица 3.2. Процедуру, которая удаляет из непустого списка L первый элемент.

Вариант 21 Таблица 3.3. Процедуру, которая удаляет из непустого списка L первый элемент.

Вариант 22 Таблица 3.1. Процедуру, которая удаляет из непустого списка L второй элемент, если такой есть.

Вариант 23 Таблица 3.2. Процедуру, которая удаляет из непустого списка L второй элемент, если такой есть.

Вариант 24 Таблица 3.3. Процедуру, которая удаляет из непустого списка L второй элемент, если такой есть.

Вариант 25 Таблица 3.1. Процедуру, которая удаляет из непустого списка L за каждым вхождением элемента E один элемент, если такой есть и он отличен от E.

Вариант 26 Таблица 3.2. Процедуру, которая удаляет из непустого списка L за каждым вхождением элемента E один элемент, если такой есть и он отличен от E.

Вариант 27 Таблица 3.3. Процедуру, которая удаляет из непустого списка L за каждым вхождением элемента E один элемент, если такой есть и он отличен от E.

Вариант 28 Таблица 3.1.Процедуру или функцию, которая проверяет, есть ли в списке L хотя бы два элемента с равными значениями второго поля.

Вариант 29 Таблица 3.2. Процедуру или функцию, которая проверяет, есть ли в списке L хотя бы два элемента с равными значениями второго поля.

Вариант 30 Таблица 3.3. Процедуру или функцию, которая проверяет, есть ли в списке L хотя бы два элемента с равными значениями второго поля.

1.4 Порядок выполнения работы

1.4.1 Выбрать вид линейного списка, подходящий для решения задачи.

1.4.2 Разработать алгоритм решения задачи, разбив его на отдельные процедуры и функции, как указано в варианте задания.

1.4.3 Разработать программу на языке Pascal.

1.4.4 Разработать тестовые примеры, которые предусматривают проверку корректности работы программы для пустых списков.

1.4.5 Выполнить отладку программы.

1.4.6 Исследовать результаты работы программы для различных режимов ее использования.

1.5 Содержание отчета

Цель работы, вариант задания, структурные схемы процедур с описанием, текст программы с комментариями, тестовые примеры, результаты вычислений, выводы.

1.6 Контрольные вопросы

1.6.1 Чем различаются статические и динамические переменных?

1.6.2 Как осуществляется задание ссылочных переменных?

1.6.3 Для чего используется переменная с указателем?

1.6.4 Какое назначение процедур New и Dispose, GetMem и FreeMem?

1.6.5 Какие операции определены над ссылочными переменными?

1.6.6 Какие операции определены над переменными с указателем?

1.6.7 Что понимают под линейной структурой данных?

1.6.8 Что называется последовательной структурой и списковой структурой данных?

1.6.9 Определите различные виды линейных односвязных списков.

1.6.10 Как описать элемент списковой структуры на языке Паскаль?

1.6.11 Напишите на языке Паскаль процедуры для работы с однонаправленными списками:

- создания списка;
- добавления элемента в список;
- удаления элемента из списка.

2 ЛАБОРАТОРНАЯ РАБОТА №2
“ПРОГРАММИРОВАНИЕ НЕЛИНЕЙНЫХ СТРУКТУР ДАННЫХ”

2.1 Цель работы

Исследование нелинейных структур данных и приобретение навыков разработки и отладки программ, использующих древовидные структуры. Исследование особенностей работы с поисковыми бинарными деревьями.

2.2 Краткие теоретические сведения

2.2.1 Бинарные деревья и алгоритмы их обработки

Нелинейные структуры предназначены для отображения связей подчиненности элементов данных. Примером нелинейных структур являются иерархические древовидные структуры. В представлении данных наибольшее распространение получили *бинарные деревья*. Любая вершина бинарного дерева связана только с вершиной ближайшего высшего уровня и с двумя вершинами низшего уровня.

Вершина (узел) дерева, которая не имеет вышестоящей вершины (предка), называется *корнем*. Считается, что корень дерева расположен на первом уровне. Каждая вершина уровня k имеет одного предка на уровне k-1. Максимальный уровень дерева называется его *глубиной*. Среди любых пар непосредственно связанных вершин можно выделить предка и дочернюю вершину. Вершина дерева, которая не имеет дочерних вершин, называется *листом*.

В связном списке каждый элемент содержит указатель на другой элемент. Бинарное дерево похоже на связный список с тем отличием, что каждый узел содержит два указателя: на левое бинарное поддерево и правое бинарное поддерево. Ниже приведен пример описания узла бинарного дерева:

```
TYPE
   TreePtr=^Tree;
   Tree=Record
     Data: String;     {тип значения, хранимый в вершине}
     Left,Right: TreePtr  {указатели на левое и правое поддерево}
      End;
```

Наиболее распространенным условием организации бинарных деревьев является упорядоченность. Элементы дерева в этом случае снабжаются ключевыми признаками. Каждый элемент в упорядоченном дереве имеет на своей левой ветви элементы с меньшими, чем у него, значениями ключей, а на правой ветви элементы с большими значениями ключей.

Упорядоченные бинарные деревья обеспечивают быстрый поиск записи по ее ключу. Такие деревья называют *деревьями бинарного поиска*. Ниже приведен пример описания функции поиска записи по ключу в бинарном дереве.

```
Function Search(Root:TreePtr; Node:string):TreePtr;
{ Node — ключевое (поисковое) значение, например типа string;
```

```
   Root — указатель на корень дерева (поддерева)}
Var Found:Boolean; {признак успешности поиска}
{ Если дерево содержит вершину Node то функция
  возвращает указатель на эту вершину}
Begin
  Found:=False;
  While (Root<>Nil) and (not Found) Do
  {повторять цикл пока не дошли до листа или не нашли значение}
    If Root^.Data=Node Then  {Проверить значение вершины}
      Found:=True            {значение найдено}
    Else                     {значение не найдено}
      If Root^.Data>Node Then
        Root:=Root^.Left;    {поиск в левом поддереве}
      Else
        Root:=Root^.Right;   {поиск в правом поддереве}
  Search:=Root;
End;
```

Рассмотрим алгоритм построения упорядоченных бинарных древовидных структур. Пусть имеется неупорядоченный массив A[1],A[2],...,A[n]. Первый элемент A[1] назовем корнем. Сравним второй элемент A[2] с первым элементом A[1]. Если A[2]>A[1], запишем в правый указатель элемента A[1] адрес элемента A[2], иначе адрес A[2] запишем в левый указатель. На следующем шаге сравниваем A[3] с A[1]. При A[3]>A[1] сравниваем A[3] с элементом адрес, которого хранится в правом указателе, а если этого адреса нет, то записываем в правый указатель элемента A[1] адрес элемента A[3]. Если же в правом указателе элемента A[1] указан адрес элемента A[2], то сравниваем A[3] с A[2] и заносим адрес элемента A[3] в правый или левый указатель элемента A[2] в зависимости от результатов сравнения. Процесс повторяется для каждого нового элемента. В этом случае новые элементы всегда присоединяются к листьям дерева. Алгоритм добавления элемента в дерево может быть сформулирован рекурсивно:

```
Function AddTree(Top:TreePtr; NewNode: String):TreePtr;
                {Top — указатель на корень дерева (поддерева);
                 NewNode — добавляемое значение элемента}
Begin
   If Top=Nil Then {указатель нулевой, если дошли до листа}
     Begin
       New(Top);          {выделить память для нового узла}
       Top^.Data:=Node;{записать в узел значение}
       Top^.Left:=Nil;    {указатели на поддеревья пустые}
       Top^.Right:=Nil;
     End
    Else
```

```
        IF Top^.Data>NewNode Then {поиск места вставки}
          Top^.Left:=AddTree(Top^.Left,NewNode)
        Else
          Top^.Right:=AddTree(Top^.Right,NewNode);
      AddTree:=Top
  End;
```

Эффективность поиска в бинарном дереве в значительной степени зависит от его симметричности. Под *симметричным* понимается дерево, которое состоит из *n* уровней, причем *n-1* уровень занят полностью. Если листья дерева располагаются только на двух соседних уровнях *n-1* и *n,* а *n-1=trunc(log2 M)*, то такое дерево называется *выровненным* (M — количество вершин дерева).

Формирование упорядоченного выровненного дерева можно выполнить с помощью следующего алгоритма. Исходные записи должны иметь вид упорядоченного массива. За корень дерева принимается средняя запись массива, которая разделяет массив примерно на две равные части. Средние записи в обеих частях массива образуют вершины второго уровня, а массив оказывается разделенным на 4 части. Средние записи каждой из четырех частей помещаются на третьем уровне бинарного дерева. Процесс продолжается до тех пор, пока все записи массива не будут внесены в дерево.

Чтобы распечатать дерево бинарного поиска в заданном порядке используют различные *стратегии обхода дерева*:

- обход сверху;
- обход слева направо;
- обход снизу.

Для обхода дерева применяют процедуры:

1) обработки корня дерева или поддерева;
2) обход левого поддерева;
3) обход правого поддерева.

Если перечисленные процедуры выполняются в порядке 1-2-3, то выполняется обход сверху; если в порядке 2-1-3, то выполняется обход слева направо; если в порядке 2-3-1, то выполняется обход снизу. Для печати дерева с сохранением относительного порядка используют обход слева направо. Пример процедуры обхода двоичного дерева слева направо:

```
Procedure LR(Top:TreePtr);
  Begin
    If Top<>Nil Then
      Begin
        LR(Top^.Left); {обойти левое поддерево}
        Writeln(Top^.Data);
        LR(Top^.Right) {обойти правое поддерево}
      End
  End;
```

2.2.2 Пример программы обработки бинарного дерева

Пусть требуется написать процедуры создания, добавления листа в бинарное дерево, просмотра бинарного дерева, отображения структуры дерева, а также рекурсивную процедуру, которая подсчитывает число вершин на n-ом уровне непустого дерева T (корень считать вершиной 1-го уровня).

Ниже приведен текст соответствующей программы с комментариями. С каждым узлом дерева связана запись, состоящая из фамилии и адреса грузополучателя. Дерево упорядочено по фамилиям.

```
Program TreeProcess;
Uses Crt;
Type
    Zap=Record  {Описание  записи о грузополучателе}
        FIO:string[15];
        Adr:string[30]
       End;
    TreePtr=^Tree; {Описание узла дерева}
    Tree=Record
        Data:Zap;
        Left,Right:TreePtr
        End;
Var    Top:Treeptr;
          Z:Zap;
  Level,N,i:Integer;
   Number:Integer;
{Функция добавляющая лист к дереву}
Function AddTree (Top:TreePtr;Newnode:Zap):TreePtr;
Begin
     If Top=Nil THEN
       Begin
         New(Top);
         Top^.Data:=Newnode;
         Top^.Left:=Nil;
         Top^.Right:=Nil;
       End
      Else
       If Top^.Data.Fio>Newnode.Fio Then
          Top^.Left:=AddTree(Top^.Left,Newnode)
       Else
          Top^.Right:=AddTree(Top^.Right,Newnode);
      Addtree:=Top
End;
Procedure OrgTree;
Begin
```

```
 Writeln('Выполняется процедура организации дерева');
 Writeln('Для выхода из процедуры вводите символ * ');
 Writeln('======================================== ');
 Top:=nil;
 While True Do
  Begin
   Writeln('Введите фамилию и инициалы грузополучателя');
   Readln(Z.Fio);
   Writeln('Введите адрес грузополучателя');
   Readln(Z.Adr);
   If Z.Fio='*' Then Exit;
   Top:=Addtree(Top,Z);
  End
End;
Procedure DobL;
 Begin
  Writeln('Выполняется процедура добавления листа');
  Writeln('Для выхода из процедуры вводите символ * ');
  Writeln('======================================== ');
  Writeln('Введите фамилию и инициалы грузополучателя');
  Readln(Z.Fio);
  Writeln('Введите адрес грузополучателя');
  Readln(Z.Adr);
  If Z.Fio='*' Then Exit;
  Top:=Addtree(Top,Z);
 End;
 Procedure Prosmotr(Top:TreePtr);
 {Процедура просмотра значений узлов дерева слева направо}
 Begin
   If Top<>Nil Then
       Begin
         Prosmotr(Top^.Left);
         Writeln(i,' ',Top^.Data.Fio,' ',Top^.Data.Adr);
         i:=i+1;
         Prosmotr(Top^.Right)
       End;
 End;
 Procedure Otobr(Top:TreePtr;Otstup:Integer);
 {Процедура отображения структуры дерева.
  Дерево отображается повернутым на 90 градусов против
  часовой стрелки. Узлы дерева, находящиеся на одном
  уровне, отображаются с  одинаковым отступом от края
  экрана.}
 Begin
```

```
      If Top<>Nil Then
         Begin
           Otstup:=Otstup+3;
           Otobr(Top^.Right,Otstup);
           Writeln(' ':Otstup,Top^.Data.Fio);
           Otobr(Top^.Left,Otstup);
         End
    End;
    Procedure NodeCount(Top:TreePtr; Level:Integer; Var N:Integer);
    {Процедура подсчета количества вершин уровня Level}
    Begin
      If (Level>=1) and (Top<>Nil) Then
       Begin
         If Level=1 Then N:=N+1;
         NodeCount(Top^.Left,Level-1,N);
         NodeCount(Top^.Right,Level-1,N);
       End
    End;
{==============основная программа===============}
Begin
   { цикл, обеспечивающий вывод на экран  пунктов меню}
   Repeat
     ClrScr; { очистка экрана }
     Writeln('1-Организация двоичного дерева');
     Writeln('2-Добавление листа к дереву');
     Writeln('3-Просмотр дерева');
     Writeln('4-Подсчет количества вершин на n-ом уровне');
     Writeln('5-Выход');
     Writeln('------------------------------');
     Writeln('Введите номер пункта меню');
     Readln(Number);
     Case Number Of { вызов необходимой процедуры по номеру}
      1:OrgTree;
      2:Dobl;
      3:Begin
         Writeln('Выполняется процедура просмотра дерева');
         Writeln('======================================== ');
         i:=0;
         Prosmotr(Top);
         Otobr(Top,1);
         Writeln('Нажмите клавишу Enter');
         Readln
        End;
      4:Begin
```

```
        Writeln('Выполняется процедура подсчета количества');
        Writeln('вершин на n-ом уровне');
        Writeln('=================================== ');
        Write('Введите значение уровня-->');
        Read(Level);
        N:=0;
        NodeCount(Top,Level,N);
        Writeln;
        Writeln('На уровне ',Level,' находится ',N,' вершин');
        Writeln('Нажмите клавишу Enter');
        ReadKey
       End;
    End;
  Until Number=5; {выход из цикла, если введено 5}
End.
```

2.3 Варианты заданий

Представить приведенную в предыдущей работе таблицу в виде бинарного дерева. Написать процедуры создания и обхода дерева, а также одну из процедур или функций, приведенных ниже. Значения полей и количество записей в таблице студент выбирает самостоятельно. Программа должна сохранять дерево в файле и создавать его заново при её повторном запуске.

Вариант 1 Таблица 3.1. Процедуру, которая присваивает параметру E элемент из самого левого листа непустого дерева T.

Вариант 2 Таблица 3.2. Процедуру, которая присваивает параметру E элемент из самого левого листа непустого дерева T.

Вариант 3 Таблица 3.3. Процедуру, которая присваивает параметру E элемент из самого левого листа непустого дерева T.

Вариант 4 Таблица 3.1. Процедуру, которая определяет уровень, на котором находится элемент E в дереве T.

Вариант 5 Таблица 3.2. Процедуру, которая определяет уровень, на котором находится элемент E в дереве T.

Вариант 6 Таблица 3.3. Процедуру, которая определяет уровень, на котором находится элемент E в дереве T.

Вариант 7 Таблица 3.1. Процедуру, которая вычисляет среднее арифметическое всех элементов непустого дерева T (по одному из полей таблицы, которое имеет числовое значение)

Вариант 8 Таблица 3.2. Процедуру, которая вычисляет среднее арифметическое всех элементов непустого дерева T (по одному из полей таблицы, которое имеет числовое значение)

Вариант 9 Таблица 3.3. Процедуру, которая вычисляет среднее арифметическое всех элементов непустого дерева T (по одному из полей таблицы, которое имеет числовое значение)

Вариант 10 Таблица 3.1. Процедуру, которая заменяет в дереве T все элементы меньшие, чем некоторое положительное число A, на это число (по одному из полей таблицы, которое имеет числовое значение).

Вариант 11 Таблица 3.2. Процедуру, которая заменяет в дереве T все элементы меньшие, чем некоторое положительное число A, на это число (по одному из полей таблицы, которое имеет числовое значение).

Вариант 12 Таблица 3.3. Процедуру, которая заменяет в дереве T все элементы меньшие, чем некоторое положительное число A, на это число (по одному из полей таблицы, которое имеет числовое значение).

Вариант 13 Таблица 3.1. Процедуру, которая печатает элементы всех листьев дерева T.

Вариант 14 Таблица 3.2. Процедуру, которая печатает элементы всех листьев дерева T.

Вариант 15 Таблица 3.3. Процедуру, которая печатает элементы всех листьев дерева T.

Вариант 16 Таблица 3.1. Процедуру, которая находит в непустом дереве T длину (число ветвей) пути от корня до вершины с элементом E.

Вариант 17 Таблица 3.2. Процедуру, которая находит в непустом дереве T длину (число ветвей) пути от корня до вершины с элементом E.

Вариант 18 Таблица 3.3. Процедуру, которая находит в непустом дереве T длину (число ветвей) пути от корня до вершины с элементом E.

Вариант 19 Таблица 3.1. Процедуру, которая подсчитывает число вершин на n-ом уровне непустого дерева T.

Вариант 20 Таблица 3.2. Процедуру, которая подсчитывает число вершин на n-ом уровне непустого дерева T.

Вариант 21 Таблица 3.3. Процедуру, которая подсчитывает число вершин на n-ом уровне непустого дерева T.

Вариант 22 Таблица 3.1. Написать рекурсивную процедуру или функцию, которая определяет, входит ли элемент в дерево T.

Вариант 23 Таблица 3.2. Написать рекурсивную процедуру или функцию, которая определяет, входит ли элемент в дерево T.

Вариант 24 Таблица 3.3. Написать рекурсивную процедуру или функцию, которая определяет, входит ли элемент в дерево T.

Вариант 25 Таблица 3.1. Написать рекурсивную процедуру или функцию, которая определяет число вхождений элемента E в дерево T.

Вариант 26 Таблица 3.2. Написать рекурсивную процедуру или функцию, которая определяет число вхождений элемента E в дерево T.

Вариант 27 Таблица 3.3. Написать рекурсивную процедуру или функцию, которая определяет число вхождений элемента E в дерево T.

Вариант 28 Таблица 3.1. Написать рекурсивную процедуру или функцию, которая вычисляет сумму элементов (по одному из полей таблицы) непустого дерева T.

Вариант 29 Таблица 3.2. Написать рекурсивную процедуру или функцию, которая вычисляет сумму элементов (по одному из полей таблицы) непустого дерева T.

Вариант 30 Таблица 3.3. Написать рекурсивную процедуру или функцию, которая вычисляет сумму элементов (по одному из полей таблицы) непустого дерева T.

2.4 Порядок выполнения работы

2.4.1 Описать элемент бинарного дерева в соответствии с вариантом задачи.

2.4.2 Разработать алгоритм решения задачи, разбив его на отдельные процедуры и функции, как указано в варианте задания.

2.4.3 Разработать программу на языке Pascal.

2.4.4 Разработать тестовые примеры, которые предусматривают проверку корректности работы программы в разных режимах.

2.4.5 Выполнить отладку программы.

2.4.6 Исследовать результаты работы программы для различных режимов ее использования.

2.5 Содержание отчета

Цель работы, вариант задания, структурные схемы процедур с описанием, текст программы с комментариями, тестовые примеры, результаты вычислений, выводы.

2.6 Контрольные вопросы

2.6.1 Что понимают под нелинейной структурой данных?

2.6.2 Что называется бинарным деревом?

2.6.3 Как построить упорядоченное бинарное дерево?

2.6.4 Отчего зависит эффективность поиска в бинарном дереве?

2.6.5 Что понимают под симметричным и выровненным деревом?

2.6.6 Как формируется выровненное дерево?

2.6.7 Напишите на языке Паскаль процедуры для работы с бинарными деревьями:

- создания упорядоченного бинарного дерева;
- добавления элементов в дерево;
- удаления листа дерева;
- процедуры обхода дерева.

3 ЛАБОРАТОРНАЯ РАБОТА №3 “ПРОГРАММИРОВАНИЕ АЛГОРИТМОВ ЦИКЛИЧЕСКОЙ СТРУКТУРЫ”

3.1 Цель работы

Получение навыков программирования алгоритмов циклической структуры на языке C. Исследование эффективности применения различных видов циклов в задаче табулирования функции.

3.2 Краткие теоретические сведения

3.2.1 Циклы в языках C/C++

Цикл представляет собой участок программы, повторяемый многократно. В языках C/C++ имеется три разновидности циклов: **`while`**, **`do-while`** и **`for`**.

3.2.1.1 Цикл `while`

Цикл `while` имеет следующий синтаксис:

```
while (<выражение>) <инструкция>
```

В цикле `while` вначале вычисляется `<выражение>`. Если его значение отлично от нуля (т. е. имеет значение истина), то выполняется `<инструкция>` и вычисление выражения повторяется. Этот цикл продолжается до тех пор, пока `выражение` не станет равным нулю (примет значение ложь), после чего вычисления возобновятся с точки, расположенной сразу за инструкцией.

Перед входом в цикл `while` обычно инициализируют одну или несколько переменных для того, чтобы `<выражение>` имело какое-либо конкретное значение. Инструкция или последовательность инструкций (*составная инструкция*), составляющих *тело цикла*, должны, как правило, изменять значения одной или нескольких переменных, входящих в `выражение`, чтобы за конечное число *итераций,* `выражение` приняло нулевое значение и цикл завершился. Цикл `while` — это *цикл с предусловием*; это значит, что решение о выполнении еще одной итерации цикла принимается *перед* началом цикла.

Цикл `while` завершается в следующих случаях:

- обращение в ноль выражения в *заголовке цикла*;
- выполнение в теле цикла инструкции `break`;
- выполнение в теле цикла инструкции `return`.

В первых двух случаях управление передается в точку, расположенную сразу за циклом. В третьем случае происходит выход из функции.

3.2.1.2 Цикл `do-while`

Цикл `do-while` имеет следующий синтаксис:

```
do    <инструкция>
while (<выражение>);
```

В цикле `do-while` сначала выполняется `<инструкция>`, затем вычисляется `<выражение>`. Если оно истинно (отлично от нуля), то `инструкция` выполняется снова и т. д. Когда `<выражение>` становится ложным (обращается в

ноль), цикл заканчивает работу. Цикл do-while завершается в тех же случаях, что и цикл while. Цикл do-while — это *цикл с постусловием*, т.е. проверка условия продолжения цикла (<выражение>) выполняется *после* каждого прохождения тела цикла (<инструкция>).

3.2.1.3 Цикл for

Цикл for имеет следующий синтаксис:

for (<выражение1>; <выражение2>; <выражение3>)<инструкция>

Каждое из трех выражений <выражение1>, <выражение2>, <выражение3> можно опускать, но точку с запятой опускать нельзя. При отсутствии *<выражения1>* или *<выражения3>* считается, что их нет в конструкции цикла; при отсутствии *<выражения2>* полагается, что его значение всегда истинно.

Обычно <выражение1> служит для инициализации *счетчика цикла*, <выражение2> — для выполнения проверки на окончание цикла, <выражение3> — для модификации счетчика цикла.

Инструкция for эквивалентна конструкции

```
<выражение1>;
while (<выражение2>)
     { <инструкция>   <выражение3>;
     }
```

В языке C++ переменную можно объявить в *части инициализации* (выражение1) инструкции for. Например:

```
for (int counter=1;counter<=10;counter++)
{    // тело цикла
    ............
}
```

3.2.1.4 Инструкции break и continue

Инструкции break и continue изменяют поток управления. Когда инструкция break выполняется в инструкциях switch, while, do-while или for, происходит немедленный выход из соответствующей инструкции, и управление передается в точку, расположенную сразу за инструкцией. Обычное назначение инструкции break — досрочно прервать цикл или пропустить оставшуюся часть инструкции switch. Необходимо заметить, что инструкция break вызывает немедленный выход из *самого внутреннего* из объемлющих ее циклов.

Инструкция continue в циклах while, do-while или for вызывает пропуск оставшейся части тела цикла, после чего начинается выполнение следующей итерации цикла. В циклах while и do-while после выполнения инструкции continue осуществляется проверка условия продолжения цикла. В цикле for после выполнения инструкции continue выполняется *выражение приращения* (выражение3), а затем осуществляется проверка условия продолжения цикла (выражение2). Таким образом, инструкция continue вынуждает ближайший объемлющий ее цикл начать следующий шаг итерации.

3.2.1.5 Цикл `for` и оператор запятая

Иногда в цикле `for <выражение1>` и `<выражение3>` представляются как *списки выражений*, разделенных запятыми. В этом случае запятая используется как *оператор запятая* или *операция следования*, гарантирующая, что список выражений будет вычисляться *слева направо*. Оператор запятая имеет самый низкий приоритет (см. таблицу 2.1 в лабораторной работе №2). Значение и тип списка выражений, разделенных запятыми, равны значению и типу самого правого выражения в списке.

Оператор запятая наиболее часто применяется в цикле `for`. Этот оператор дает возможность использовать несколько выражений задания начальных значений и (или) несколько выражений приращения переменных, что позволяет вести несколько индексов (управляющих переменных) параллельно, например:

```
for (int left=0,right=n-1;left<right;left++,right--)
{    // тело цикла
    ............
}
```

Кроме цикла `for`, оператор запятая можно использовать в тех случаях, когда последовательность инструкций мыслится как одна операция, например:

```
temp=a[i],a[i]=a[i+1],a[i+1]=temp; // обмен
```

3.2.2 Пример программы табулирования функции

3.2.2.1 Постановка задачи

Написать программу табулирования (печати таблицы значений) кусочно-заданной функции $z = f(x)$ на интервале от $x_{нач}$ до $x_{кон}$ с шагом Δx. Таблицу снабдить заголовком и шапкой. Вид функции определяется формулой

$$z = f(x) = \begin{cases} a^2, \text{ если } x < 0, \\ ax, \text{ если } 0 \le x < 10, \\ 5a, \text{ если } x \ge 10. \end{cases}$$

Если $a > 10$, то значения функции должны выводиться в виде целых чисел. Значения параметра a, а также $x_{нач}$, $x_{кон}$ и Δx вводятся с клавиатуры. Результаты вычислений выводятся в формате с фиксированной точкой.

3.2.2.2 Описание алгоритма решения задачи в словесной форме

В *словесной форме* алгоритм решения задачи можно сформулировать следующим образом:

1) Ввести с клавиатуры исходные данные: a, $x_{нач}$, $x_{кон}$ и Δx.
2) Вывести заголовок и шапку таблицы.
3) Положить $x = x_{нач}$.
4) *Вычислить значение функции* z *по вышеприведенной формуле.*
5) *Если* $a > 10$*, то привести значение функции* z *к целому типу.*
6) *Вывести на экран строку таблицы значений функции.*
7) *Увеличить значение* x *на величину* Δx.

8) Если значение x не превышает $x_{кон}$, то перейти к пункту 4, иначе закончить выполнение программы.

Так как пункты 4 — 7 алгоритма выполняются многократно, то для их выполнения необходимо организовать цикл. Напишем два варианта программы с использованием циклов `while` и `for`.

3.2.2.3 Использование цикла `while`

Ниже представлена программа табулирования функции с использованием цикла `while`:

```
#include <conio.h>
#include <stdio.h>
main()
// программа табулирования функции z=f(x)
// на интервале от xn до xk с шагом dx
{   float a, // параметр
          x, // аргумент функции z
          xn, // начальное значение аргумента x
          xk, // конечное значение аргумента x
          dx, // шаг
          z; // значение функции z
   clrscr();
   // ввод a, xn, xk, dx
 printf("Введите параметр a: "), scanf("%f",&a);
 printf("Введите xn: "), scanf("%f",&xn);
 printf("Введите xk: "), scanf("%f",&xk);
 printf("Введите шаг dx: "), scanf("%f",&dx);
 // вывод заголовка и шапки таблицы
 printf("Таблица значений функции z=f(x)\n");
 printf("  ┌──────────────┬──────────────┐\n");
 printf("  │      x       │   z = f(x)   │\n");
 printf("  ├──────────────┼──────────────┤\n");
 // табулирование функции z
 x=xn; // начальное значение аргумента x
 while (x<=xk)
 { // вывод строки таблицы
  printf("  │  %-9.3f│",x); // вывод аргумента x
     // вычисление значения функции z
   (x<0) ? (z=a*a) : ((x<10) ? (z=a*x) : (z=5*a));
     // вывод значения функции z
   if (a>10)
     printf("  %-10d│\n",(int)z); // целочисленное z
     else
     printf("  %-10.3f│\n",z); // вещественное z
   x+=dx; // приращение аргумента x
  }
 printf("  └──────────────┴──────────────┘\n");
 printf("Нажмите любую клавишу..."); getch();
```

```
  return 0;
}
```

Поскольку формулы, определяющие функцию *z*, являются достаточно короткими, то вычисление значения функции *z* целесообразно осуществлять не с помощью инструкции `if-else`, а с помощью условного выражения.

Вывод таблицы значений функции осуществляется средствами функции `printf` стандартной библиотеки ввода-вывода `<stdio.h>`.

Воспроизвести символ большинства кодов на экране можно, нажав соответствующую ему клавишу. Но этого нельзя сделать, например, для *кодов псевдографики*, с помощью которых возможно построение рамки таблицы. Любой из символов, имеющих коды от *1* до *255*, можно воспроизвести на экране, дополнительно используя клавишу `Alt`. Для этого в среде Turbo (Borland) C++ надо установить режим работы с *цифровой клавиатурой* (правая часть клавиатуры), нажав клавишу `Num Lock`, что фиксируется индикатором `Num Lock`. Затем надо нажать клавишу `Alt`, и, не отпуская ее, на цифровой клавиатуре набрать код символа, после чего отпустить клавишу `Alt`. В результате на экране воспроизведется символ, код которого был набран. Коды символов псевдографики представлены в таблице 3.1.

Таблица 3.1 — Коды символов псевдографики (от *176* до *223*)

код	с	код	с	код	с	код	с	код	с	код	с	Код	с	код	с	код	с	код	с	код	с	код	с
176	░	180	┤	184	╕	188	╝	192	└	196	─	200	╚	204	╠	208	╨	212	╘	216	╪	220	▄
177	▒	181	╡	185	╣	189	╜	193	┴	197	┼	201	╔	205	═	209	╤	213	╒	217	┘	221	▌
178	▓	182	╢	186	║	190	╛	194	┬	198	╞	202	╩	206	╬	210	╥	214	╓	218	┌	222	▐
179	│	183	╖	187	╗	191	┐	195	├	199	╟	203	╦	207	╧	211	╙	215	╫	219	█	223	▀

3.2.2.4 Использование цикла `for`

Ниже представлена программа табулирования функции с использованием цикла **`for`**:

```
#include <conio.h>
#include <iomanip.h>
#include <iostream.h>
main()
// программа табулирования функции z=f(x)
// на интервале от xn до xk с шагом dx
{    // объявление переменных и очистка экрана
     ..................................
  // ввод a, xn, xk, dx
     cout<<"Введите параметр a:  ", cin>>a;
     cout<<"Введите xn: ", cin>>xn;
     cout<<"Введите xk: ", cin>>xk;
     cout<<"Введите шаг dx: ", cin>>dx;
 // вывод заголовка и шапки таблицы
     cout<<"Таблица значений функции z=f(x)"<<endl
```

```
        <<"  ┌──────────────────────┐"<<endl
        <<"  │      x       │  z = f(x)   │"<<endl
        <<"  ├──────────────┼──────────────┤"<<endl;
 // табулирование функции z
    cout.precision(3), cout.setf(ios::showpoint);
    cout.setf(ios::left,ios::adjustfield);
    cout.setf(ios::fixed,ios::floatfield);
 for(x=xn;x<=xk;x+=dx)
{ // вывод строки таблицы
 cout<<"  │  "<<setw(9)<<x<<'│'; // вывод аргумента x
 // вычисление значения функции z
 (x<0) ? (z=a*a) : ((x<10) ? (z=a*x) : (z=5*a));
       // вывод значения функции z
       cout<<"  "<<setw(10);
       // целочисленное или вещественное z
      (a>10) ? cout<<(int)z : cout<<z;
       cout<<'│'<<endl;
}
    cout<<"  └──────────────┴──────────────┘"<<endl;
    cout<<"Нажмите любую клавишу...";
    getch();
    return 0;
}
```

В вышеприведенной программе ввод-вывод данных осуществляется с помощью объектов `cin` и `cout`. Последовательность инструкций

```
cout.precision(3), cout.setf(ios::showpoint);
cout.setf(ios::left,ios::adjustfield);
cout.setf(ios::fixed,ios::floatfield);
```

задает формат для вывода значения аргумента *x* и значения функции *y* в строке таблицы. Для установки ширины поля используется манипулятор потока `setw`. *Функция-элемент* объекта `cout` `precision` позволяет задать точность печатаемого вещественного числа (т. е. число разрядов справа от десятичной точки). Функция-элемент `setf` используется для управления *флагами состояния формата*.

Флаг `ios::showpoint` устанавливается для вывода чисел с обязательной печатью десятичной точки и нулевых младших разрядов (нулей в конце числа). Так, например, значение `79.0` без установки `ios::showpoint` будет напечатано как `79`, а с установкой `ios::showpoint` — как `79.00000` (количество нулей определяется заданной точностью).

Флаги `ios::left` и `ios::right` содержатся в *статическом элементе данных* `ios::adjustfield` и позволяют выравнивать печать соответственно по левой или правой границам поля.

Флаги `ios::fixed` и `ios::scientific`, управляющие форматом вывода вещественных чисел, описаны в лабораторной работе №1 настоящих методических указаний.

3.3 Варианты заданий

Вычислить и вывести на экран в виде таблицы значения функции $z = f(x)$ на интервале от $x_{нач}$ до $x_{кон}$ с шагом Δx. Таблицу снабдить заголовком и шапкой. Вид функции z выбирать в соответствии с вариантами задания. Значения параметров a, b, а также $x_{нач}$, $x_{кон}$ и Δx вводятся с клавиатуры. Результаты вычислений выводятся в формате с фиксированной точкой.

1) $$z=\begin{cases}\left(1+x^2\right)/\left(2-11\sin x\right)+e^{\sinh x}, & \text{если } x\le a\\ \sqrt{1{,}57-x^3\sin^2 x}+4{,}1e^{2x}, & \text{если } a<x<b\\ \left(\arcsin x+\arccos x+\ln x\right)^{\tan x}, & \text{если } x\ge b\end{cases}$$

2) $$z=\begin{cases}e^x/\left(\sqrt{\sinh x}+8{,}9x+9\right), & \text{если } x\le a\\ \left(\cos^2 x+1{,}1\tan x\right)^{2{,}1}, & \text{если } a<x<b\\ \left|\ln x+\cos x^{2{,}4}-1{,}2x\right|x^{4{,}8}, & \text{если } x\ge b\end{cases}$$

3) $$z=\begin{cases}\sqrt[3]{e^{x-1/\arctan x}}+\tan^2 x+\ln x+6, & \text{если } x\le a\\ \left(\arcsin x+\sinh x\right)^{\cos x+x^2+e}, & \text{если } a<x<b\\ \left|x^{1{,}3/x}-\lg\left(1+x\right)\right|^{1{,}3+x^2}+x^5+x, & \text{если } x\ge b\end{cases}$$

4) $$z=\begin{cases}\ln\left(1{,}2x+2{,}4\right)+e^{3x}\sinh x, & \text{если } x\le a\\ 5^{x^2+1}\sqrt{1{,}2x^{3{,}5}+\sqrt{\left|1-x\right|}}, & \text{если } a<x<b\\ \left(1+\tan^2 x\right)^{\arctan x+\ln x+1}+\lg x, & \text{если } x\ge b\end{cases}$$

5) $$z=\begin{cases}\lg\left(\sqrt[3]{x^2}+\sqrt{x}+\sin x+\cos x\right), & \text{если } x\le a\\ \left(\tan^2 x+1{,}3e^{\cosh x+\tanh x}\right)^{x^2}, & \text{если } a<x<b\\ \left|\sin x-0{,}11\right|\arccos x+x^{\arcsin x}, & \text{если } x\ge b\end{cases}$$

6) $$z=\begin{cases}\left|\cos x+\sin x\right|^{1+3\tan^2 x}+x^{1{,}31}, & \text{если } x\le a\\ 8^{2{,}1x}\sqrt{\tanh x+\sqrt[7]{\left|1-x\right|}}, & \text{если } a<x<b\\ \lg\left(\sqrt[3]{x^{1{,}1}}+\sqrt{x}+6{,}1x+7\right), & \text{если } x\ge b\end{cases}$$

7) $$z=\begin{cases}4{,}5\sin\left(x+\pi/9\right)+1{,}2e^{-0{,}2x-1}, & \text{если } x\le a\\ \ln\left(x+e+x^2\right)+e^{2x}\cos x, & \text{если } a<x<b\\ x+8x^2+9x^3+8x^4+\cosh x, & \text{если } x\ge b\end{cases}$$

8) $$z=\begin{cases}\ln\left(x+5{,}7\right)+3{,}8e^{3x}\sin x, & \text{если } x\le a\\ \sqrt[7]{5{,}9^{x-1/\arctan x}}+\tan^2 x, & \text{если } a<x<b\\ \left|\cos x-0{,}5\right|^{1{,}1}+5\arccos x, & \text{если } x\ge b\end{cases}$$

9) $$z=\begin{cases}6{,}3e^{3{,}2x+7{,}9}\sin\left(6x+\pi/7\right)+5, & \text{если } x\le a\\ \left|\sin x-3{,}2\right|\cosh^2 x+x^{5{,}4}, & \text{если } a<x<b\\ \lg\left(11\pi+\arcsin x+\arccos x\right), & \text{если } x\ge b\end{cases}$$

10) $$z=\begin{cases}\left[1+\sinh^2\left(x^2-x-3\right)\right]x^{-4}, & \text{если } x\le a\\ 4.5\ln x+e^x/32+3.5x, & \text{если } a<x<b\\ \left(1+2\tan^4 x+\tan^8 x\right)^{\sqrt{|x|+17}}, & \text{если } x\ge b\end{cases}$$

11) $$z=\begin{cases}\left(1+\cosh x+\tanh x^2\right)^{\sin x+\cos x}, & \text{если } x\le a\\ 7{,}3x^{3{,}17x+5}+\arctan x^{-2\ln x}, & \text{если } a<x<b\\ \left|\arcsin x+e^x\right|^{1+5\sin^2 x}+\sqrt{\left|1-x\right|}, & \text{если } x\ge b\end{cases}$$

12) $$z=\begin{cases}16e^{-2.5\cos x}+2.5\ln\left(1+x^{2{,}8}\right), & \text{если } x\le a\\ \left(1+x^5\right)/\left(1{,}7-\cosh x\right), & \text{если } a<x<b\\ \sqrt[7]{x^3+\sqrt[5]{1.8+x^{1.3}}+\tan^{2{,}3} x}, & \text{если } x\ge b\end{cases}$$

13) $$z=\begin{cases}\left(\cosh x+\sinh x+x^2\right)^{11+x^2}, & \text{если } x\le a\\ \left|x^{1{,}3/x}-\sqrt[3]{x/1{,}3}+x^{x^{2{,}11}}\right|, & \text{если } a<x<b\\ \lg\left(\sqrt{e^{x+13}}+\ln x+\cosh x\right), & \text{если } x\ge b\end{cases}$$

14) $$z=\begin{cases}\sqrt{5\left(\sqrt[3]{x+1.2}+\sqrt[5]{x^2-4.6}\right)}, & \text{если } x\le a\\ 1.2e^{\cos 5x-1}+\left(x-1\right)^{4.5x+1}, & \text{если } a<x<b\\ tg^2 x/\left(1/2+\sinh^{\pi} x+\ln x\right), & \text{если } x\ge b\end{cases}$$

15) $z=\begin{cases}\left(\sqrt{x-\tanh x+x^4}\right)\Big/\cos x^2, & \text{если } x\le a\\ e^{-2\sin 4x}+\lg x+\arctan x, & \text{если } a<x<b\\ 3.7x^{7.3}\sin\left|x^2\right|+4.5\cosh x^3, & \text{если } x\ge b\end{cases}$

16) $z=\begin{cases}\left|x^{5,7/x}-\sqrt[5]{\arctan x}+\sin x^2\right|, & \text{если } x\le a\\ \ln\left(e^{x^2}+x\lg x+\cos x\right), & \text{если } a<x<b\\ \left(1+2x^2+3x^3\right)\sinh x^{x^{1,5}+7,3}, & \text{если } x\ge b\end{cases}$

17) $z=\begin{cases}5.7\cos\left(3.4x-\pi/6\right)+10.2, & \text{если } x\le a\\ \left(\cos x+2.8\lg x\right)^{\arccos 15x}, & \text{если } a<x<b\\ 2.7+1.1x+4.2x^2+5.8x^{3,1}, & \text{если } x\ge b\end{cases}$

18) $z=\begin{cases}\sqrt{1,414-x^{3,5}\sin x^{2,2}}+\ln x, & \text{если } x\le a\\ 2e^{3,14x}\sin\left(2,1x-\pi/5\right), & \text{если } a<x<b\\ \left|x^{5,1}\cosh x-1,4\right|\arctan x^{2,6}, & \text{если } x\ge b\end{cases}$

19) $z=\begin{cases}\ln\sin 5.9x+3.6x^{1.2}+\cos x, & \text{если } x\le a\\ 3^{5x^{4.4}}+\cosh x^{-3.31}+\lg x, & \text{если } a<x<b\\ \sqrt{\sinh^2\arctan x^{3,94}+9\tan x}, & \text{если } x\ge b\end{cases}$

20) $z=\begin{cases}\ln\left(x-\sinh x\right)+\arccos 5,1x, & \text{если } x\le a\\ \sin^2 2,45x+3,81e^{x^2+x+1}, & \text{если } a<x<b\\ \left(1+x^2\right)\Big/\left(\cosh^2 x^3+x^{2x+9}\right), & \text{если } x\ge b\end{cases}$

21) $z=\begin{cases}2,56e^{\sinh x-11}+\cosh\sqrt{x}+\pi, & \text{если } x\le a\\ \ln\left(2,44x+3,7\right)+\sin x, & \text{если } a<x<b\\ \left|x^{1,2}-\sqrt[5]{1+x^3}\right|+\arctan x^{2,11}, & \text{если } x\ge b\end{cases}$

22) $z=\begin{cases}\sqrt[4]{3x^2+\sqrt[3]{1+\sin^2 x^5}+e^{2,91x}}, & \text{если } x\le a\\ \left(1+\sinh x+\lg^2 x\right)^{x^2+x+9}, & \text{если } a<x<b\\ \left|x-\arctan x+\tan^{2e^x+15x+3}x\right|, & \text{если } x\ge b\end{cases}$

23) $z=\begin{cases}2,2+2,4x+4,8x^2+9,6x^3, & \text{если } x\le a\\ \sqrt{\cosh^2\arctan x^5+1,32}, & \text{если } a<x<b\\ \lg\left(\sin x+\cos x+\tan^{2e}x^{2\pi}\right), & \text{если } x\ge b\end{cases}$

24) $z=\begin{cases}4,1+7x^2+\sin\left(8,2x+\pi/6\right), & \text{если } x\le a\\ \sqrt{\cos^2\arctan^2 x^2+e^{3x+10}}, & \text{если } a<x<b\\ \ln\left(\arcsin x+\arccos x+x^{3,2}\right), & \text{если } x\ge b\end{cases}$

25) $z=\begin{cases}\arccos^2 x+\arctan x+x^4+1, & \text{если } x\le a\\ e^{\pi x}+\pi^{ex}+5e^{x+1}\cos x^{x+1}, & \text{если } a<x<b\\ \lg\left(\sqrt[5]{x^2}+\sqrt{\left|3,2-x\right|}+1,73\right), & \text{если } x\ge b\end{cases}$

26) $z=\begin{cases}\sqrt{1,414-x^2\sin^3 x^2}+\lg^{1,21}x, & \text{если } x\le a\\ \tan^2 x+\tan x+4^{x^{1,212}}+2, & \text{если } a<x<b\\ \pi-2x+3x^2-4x^3+x^4-x^6, & \text{если } x\ge b\end{cases}$

27) $z=\begin{cases}\left(1+\arcsin x\right)\Big/\left(1-\arccos x\right), & \text{если } x\le a\\ e^{x-2,11}+\left(x-1\right)^{x+1}\ln\sin x, & \text{если } a<x<b\\ \sqrt{\sinh^2 x+\tanh x^2+\cos x^{4,1}}, & \text{если } x\ge b\end{cases}$

28) $z=\begin{cases}e^{2x^2+x+5}\Big/\left(1,5+3\sin x+\lg x\right), & \text{если } x\le a\\ \left|\cosh^{2,1}x-1,2\right|\arcsin^2 x, & \text{если } a<x<b\\ \left(\arctan x+\ln x\right)^{\cos x+2\tan x}+\pi, & \text{если } x\ge b\end{cases}$

29) $z=\begin{cases}\left|\cosh x+\sinh x\right|^{9+2\tan^2 x}+\lg x, & \text{если } x\le a\\ 7^{-x-7}\sqrt{\sin x+\left|1-x-x^2\right|}, & \text{если } a<x<b\\ e^x\left(1+\arctan^2 x\right)^{\sqrt{|x|+3}}+\ln^e x, & \text{если } x\ge b\end{cases}$

30) $z=\begin{cases}x^{1,3}\ln\left(2\sinh x^2+7\cosh x^2\right), & \text{если } x\le a\\ \left(\arctan x+1\right)\Big/\left(1+x^{x+5}\right), & \text{если } a<x<b\\ \sqrt{\left|0.5-\sin x\right|}+e^{2,6x}\cos 4,1x, & \text{если } x\ge b\end{cases}$

3.4 Порядок выполнения работы

3.4.1 Проработать необходимый теоретический материал, опираясь на сведения и указания, представленные в предыдущем пункте, а также в литературе [5, 6, 7, 8].

3.4.2 Ответить на контрольные вопросы.

3.4.3 Внимательно изучить постановку задачи.

3.4.4 Выполнить анализ *области определения* и *области значений* вычисляемой функции z. Желательным является построение графика функции. Для выполнения этого пункта можно воспользоваться результатами предыдущей лабораторной работы.

3.4.5 Разработать структурную схему алгоритма решения задачи.

3.4.6 Разработать *тестовые примеры* (таблицы значений функции). При разработке тестовых примеров целесообразно использовать *отлаженную* программу из предыдущей лабораторной работы для вычисления значений функции z.

3.4.7 Написать *два* варианта программы табулирования функции z, используя циклы **`while`** и **`for`**. В одном из вариантов ввод-вывод должен базироваться на функциях **`scanf`** и **`printf`**, а в другом — на объектах **`cin`** и **`cout`**. Программы *обязательно* должны содержать комментарии.

3.4.8 Выполнить *тестирование и отладку* написанных программ, используя разработанные тестовые примеры.

3.4.9 Подготовить отчет по работе.

3.5 Содержание отчета

Цель работы; постановка задачи и вариант задания; краткие теоретические сведения; математическое обоснование задачи (включает анализ области определения и области значений функции, подлежащей вычислению, а также, факультативно, график этой функции); структурная схема алгоритма решения задачи; тестовые примеры; тексты программ; результаты тестирования и отладки программ; выводы.

3.6 Контрольные вопросы

3.6.1Что называют циклом?

3.6.2 Какие три элемента должны входить в цикл?

3.6.3 Перечислите типы циклов в языках C/C++.

3.6.4 Охарактеризуйте цикл с предусловием.

3.6.5 Охарактеризуйте цикл с постусловием.

3.6.6 Охарактеризуйте цикл **`while`**.

3.6.7 Охарактеризуйте цикл **`do-while`**.

3.6.8 Охарактеризуйте цикл **`for`**.

3.6.9 Опишите инструкции **`break`** и **`continue`**.

3.6.10 В каких случаях целесообразно применять оператор запятая?

3.6.11 Какой тип цикла лучше использовать в задаче табулирования функции? Почему?

3.6.12 Дайте характеристику флагам состояния формата, используемым в функции **`setf`** объекта **`cout`**.

4 ЛАБОРАТОРНАЯ РАБОТА №4
“ОБРАБОТКА ДВУМЕРНЫХ МАССИВОВ С ПОМОЩЬЮ ФУНКЦИЙ”

4.1 Цель работы

Изучить основные принципы работы с массивами в языках C/C++.
Исследовать способы передачи параметров в функции.

4.2 Краткие теоретические сведения

4.2.1 Описание и обработка двумерных массивов

Двумерный массив в C/C++ представляется как массив, состоящий из массивов. Для этого при описании массива в квадратных скобках указывают вторую размерность:

```
int a[3][5]; // матрица 3x5
```

Такой массив хранится в непрерывной области памяти по строкам (рисунок 4.1):

a_{00} a_{01} a_{02} a_{03} a_{04}	a_{10} a_{11} a_{12} a_{13} a_{14}	a_{20} a_{21} a_{22} a_{23} a_{24}
← 0-ая строка →	← 1-ая строка →	← 2-ая строка →

Рисунок 4.1 — Хранение двумерного массива в памяти ЭВМ

В памяти сначала располагается одномерный массив `a[0]`, т.е. нулевая строка, затем массив `a[1]` — первая строка и т.д.

Для доступа к отдельному элементу массива применяется конструкция вида `a[i][j]`, где `i` — номер строки, `j` — номер столбца. Каждый индекс изменяет свои значения, начиная с нуля. Можно обратиться к элементу массива и с помощью указателей, например, `*(*(a+i)+j)` или `*(a[i]+j)`. Следует обратить внимание на то, что `*(a+i)` должно быть указателем.

При описании массивов можно задавать начальные значения его элементов. Элементы массива инициализируются в порядке их расположения в памяти:

```
int a[3][5]={1,2,3,4,5,1,2,3,4,5,1,2,3,4,5};
```

Если количество значений в фигурных скобках превышает количество элементов в массиве, то выдается сообщение об ошибке. Если значений меньше, то оставшиеся элементы массива инициализируются значениями по умолчанию (для основных типов это ноль). Можно задать начальные значения не для всех элементов массива. Для этого список значений для каждой строки массива заключается в дополнительные фигурные скобки:

```
int a[3][5]={{1,2}, {1,2}, {1,2}};
```

Здесь нулевой и первый столбцы матрицы заполняются единицами и двойками. Остальные элементы массива обнуляются.

При явном задании хотя бы одного инициализатора для каждой строки количество строк в описании массива можно не указывать:

```
int a[][5]={{1, 1, 1, 1, 1}, {2, 2, 2}, {3, 3}};
```

Напишем программу, которая для целочисленной матрицы 10x20 определяет среднее арифметическое ее элементов и количество положительных элементов в каждой строке. Алгоритм решения этой задачи очевиден. Для вычисления среднего арифметического элементов массива требуется найти их общую сумму, после чего разделить ее на количество элементов. Порядок просмотра массива роли не играет. Определение положительных элементов каждой строки требует просмотра матрицы по строкам. Обе величины вычисляются при одном просмотре матрицы.

```
#include <iostream.h>
void main()
{
    const int nrow=10, ncol=20;
    int a[nrow][ncol];
    int i,j;
    cout<<"Введите элементы массива: "<<endl;
    for(i=0;i<nrow;i++)
        for(j=0;j<ncol;j++) cin>>a[i][j];
    int n; // счетчик положительных элементов
    float s=0; // сумма элементов
    for(i=0;i<nrow;i++)
      { n=0;
        for(j=0;j<ncol;j++)
         {
            s+=a[i][j];
            if(a[i][j]>0) n++;
         }
       cout<<"В строке "<<i
       << " кол-во положительных элементов: "<<n<<endl;
      }
    s/=nrow*ncol;
    cout<<"Среднее арифметическое равно: "<<s<<endl;
 }
```

Здесь размерности массива заданы именованными константами nrow и ncol, что позволяет легко их изменять. Для упрощения отладки рекомендуется задать небольшие значения этих констант. При вычислении количества положительных элементов для каждой строки выполняются однотипные действия: обнуление счетчика n, просмотр каждого элемента строки и сравнение его с нулем, при необходимости увеличение счетчика на единицу, а после окончания вычислений — вывод результирующего значения счетчика.

Рекомендуется после ввода матрицы выполнять ее контрольный вывод на экран. Для того чтобы элементы матрицы располагались один за другим, следует использовать манипулятор setw(), устанавливающий ширину поля для выводимого значения. Для использования манипулятора следует подключить заголовочный файл <iomanip.h> и дополнить программу следующим кодом:

```
#include <iomanip.h>
.................................
for(i=0;i<nrow;i++)
{  for(j=0;j<ncol;j++) cout<<setw(6)<<a[i][j];
   cout<<endl;
}
```

Здесь после вывода каждой строки матрицы выводится символ перевода строки `endl`. Необходимый форматированный вывод матрицы можно также выполнить с помощью функции `printf`.

4.2.2 Динамические массивы

В динамической области памяти можно создавать двумерные массивы с помощью операции `new` или функции `malloc`. Рассмотрим первый способ. При выделении памяти сразу под весь массив количество строк можно задавать с помощью переменной, а количество столбцов должно быть определено с помощью константы. После слова new записывается тип элементов массива, а затем в квадратных скобках его размерности, например [10]:

```
int n;
const int m=5;
cin>>n;
int (*a)[m]=new int[n][m]; // 1
int **b=(int **) new int[n][m]; // 2
```

Здесь в строке 1 адрес начала выделенного с помощью `new` участка памяти присваивается переменной `a`, определенной как указатель на массив из `m` элементов типа `int`. Именно такой тип значения возвращает в данном случае операция `new`. Скобки для `(*a)` необходимы, поскольку без них конструкция интерпретировалась бы как массив из указателей.

В строке 2 адрес начала выделенного участка памяти присваивается переменной `b`, которая описана как указатель на указатель типа `int`. Поэтому требуется выполнить преобразование типа `(int **)`.

Обращение к элементам динамических массивов производится точно так же, как к элементам статических массивов, например, `a[i][j]`.

Для того чтобы понять, отчего динамические массивы описываются так, напомним, что доступ к элементу двумерного массива можно выполнить с помощью конструкции `*(*(a+i)+j)`. Поскольку здесь применяются две операции раскрытия ссылки, то переменная, в которой хранится адрес начала массива, должна быть указателем на указатель.

Когда обе размерности массива задаются на этапе выполнения программы, то память под двумерный массив можно выделить следующим образом:

```
int nrow,ncol;
cout<<"Введите количество строк и столбцов: "
cin>>nrow>>ncol;
int **a=new int *[nrow]; // 1
for(int i=0;i<nrow;i++)  // 2
```

```
    a[i]=new int[ncol];  // 3
```

В строке 1 объявляется переменная `a` типа указатель на указатель типа `int` и выделяется память под массив, каждый элемент которого есть указатель на строку матрицы. Всего элементов `nrow`. В строке 2 организуется цикл для выделения памяти под строки. В строке 3 каждому указателю на строку присваивается адрес начала области памяти, достаточной для хранения `ncol` элементов типа `int`.

4.2.3 Функции

Функция — это группа операторов, вызываемая по имени и возвращающая в точку вызова предписанное значение. Формат простейшего заголовка функции: `<тип> <имя>(<список формальных параметров>)`

Например, заголовок функции `main` обычно имеет вид

```
int main();
```

Это означает, что никаких параметров функции не передается, а возвращает она одно значение типа `int`. Функция может и не возвращать значения, в этом случае должен быть указан тип `void`.

Имена формальных параметров при задании *прототипа функции* можно не указывать. Достаточно указать их тип: `double sin(double);`

Здесь записано, что функция имеет имя `sin`, вычисляет значение синуса типа `double` и для этого ей надо передать аргумент типа `double`.

Хороший стиль программирования требует, чтобы все, что передается в функцию и обратно, отражалось в ее заголовке.

Заголовок задает *объявление функции*. *Определение функции*, кроме заголовка, включает ее тело, т.е. операторы, которые выполняются при вызове функции, например:

```
int sum(int a,int b)
{    return a+b; // тело функции
}
```

Тело функции представляет собой блок, заключенный в фигурные скобки. Для возврата результата, вычисленного в функции, служит оператор `return`. После него указывается выражение, значение которого и передается в точку вызова функции. Функция может иметь несколько операторов возврата.

Для вызова функции указывают ее имя, а также передают ей значения *фактических параметров*:

```
<имя функции>(<список фактических параметров>);
```

Порядок следования аргументов в списке фактических параметров и их типы должны строго соответствовать *списку формальных параметров*. Пример обращения к определенной выше функции `sum`:

```
int summa,a=5;
summa=sum(a,5);
```

Рассмотрим *механизм передачи параметров* в функцию. Он весьма прост. Параметры передаются в функцию как *параметры-значения*. Иными словами, функция работает с копией значений, которые ей передаются. Кстати, именно по-

этому на месте таких параметров можно при вызове задавать выражения, например: `summa=sum(a*10+5,a+3);`

Для передачи в функцию параметров способом *«параметр-переменная»* в языке С используют указатели. Определим функцию `swap`, осуществляющую обмен значениями двух переменных.

```
void swap(int *x,int *y)
{   int temp;
    temp=*x;
    *x=*y;
    *y=temp;
}
```

Здесь параметры `x` и `y` являются указателями и позволяют функции осуществлять доступ к ячейкам памяти вызвавшей ее программы и менять их значения местами. Чтобы функция `swap` выполняла желаемые действия, необходимо при вызове на место параметров `x` и `y` подставить адреса соответствующих ячеек памяти. Для этого используется операция взятия адреса `&`:

```
swap(&a,&b);
```

В этом случае функция `swap` поменяет местами значения переменных `a` и `b`.

В языке С++ для передачи параметров способом «параметр-переменная» можно использовать *ссылочные переменные*. *Ссылка* — это переменная, которая ассоциирована с другой переменной и содержит ее адрес:

```
int ivar=1234;
int &iref=ivar; // ссылка iref
```

Здесь `iref` — это ссылка, которой присваивается адрес переменной `ivar`. Ссылки должны инициализироваться при их объявлении.

Не существует операторов, непосредственно производящих действия над ссылками. Все операции выполняются над ассоциированными значениями. Так, оператор `iref++` выполняет инкремент значения `ivar`.

Сами по себе ссылки применяются редко. Их обычно используют в качестве параметров функций. Так, функцию `swap` можно переписать в виде

```
void swap(int &x,int &y)
{   int temp;
    temp=x;
    x=y;
    y=temp;
}
```

При этом упрощается вызов функции: `swap(a,b);`

В этом случае `x` будет ссылаться на `a`, а `y` — на `b`, и функция `swap` выполнит обмен значениями `a` и `b`.

При определении функции входные данные обычно передаются по значению, а результаты ее работы — по ссылке или указателю.

Следует иметь ввиду, что возвращение ссылки или указателя из функции может привести к проблемам, если переменная, на которую делается ссылка, вышла из области видимости. Например,

```
int & func()
{
 int x;
    x=5;
    return x;
}
```

В этом случае попытка возвратить ссылку на локальную переменную приведет к ошибке.

Эффективность передачи в функцию адреса вместо самой переменной ощутима и в скорости работы, особенно если используются массивы.

Если в функцию требуется передать достаточно большой объект, однако его модификация не предусматривается, то используется *константный указатель* или ссылка:

```
const int * func(const int * number);
const int & func(const int & number);
```

Здесь функция `func` принимает и возвращает указатель или ссылку на константный объект типа `int`. Любая попытка модифицировать такой объект внутри функции вызовет сообщение компилятора об ошибке.

В заключение рассмотрим передачу массивов функциям. Пусть требуется написать функцию, суммирующую элементы массива. Предположим, что имеются следующие описания:

```
#define MAX 100;
int a[MAX];
```

Тогда можно объявить функцию со следующим прототипом:

```
int SumOfA(int a[MAX]);
```

Однако будет лучше оставить квадратные скобки пустыми и добавить второй аргумент, определяющий размер массива:

```
int SumOfA(int a[],int n);
```

Необходимо помнить, что в этом случае реально в функцию передается указатель на тип элемента массива. Таким образом, изменение любого элемента массива внутри функции повлияет и на оригинал. Поэтому лучше сразу передать в функцию указатель на нулевой элемент константного массива, например:

```
int SumOfA(const int *a,int n);
```

Аналогичным образом поступают и при передаче двумерных массивов в функцию. При передаче двумерного массива в функцию в списке формальных параметров обычно указывают только количество столбцов массива. Количество строк передают в виде дополнительного параметра:

```
int SumOfMatrix(int matrix[][10],int nrow);
```

Поскольку доступ к двумерному массиву можно получить с помощью указателя на указатель, то прототип функции можно объявить и так:

```
int SumOfMatrix(int **matrix,int nrow,int ncol);
```

Здесь `nrow` — количество строк матрицы, а `ncol` — количество столбцов.

4.2.4 Обработка матриц с помощью функций

Пусть требуется написать программу, которая упорядочивает строки прямоугольной целочисленной матрицы по возрастанию сумм их элементов. Начинать решение задачи необходимо с четкого описания того, что является ее исходным данными и результатами, и каким образом они будут представлены в программе.

Исходные данные. Размерность матрицы будем задавать с помощью символических констант. В качестве типа значений элементов матрицы будем использовать тип `int`.

Результаты. Результатом является та же матрица, но упорядоченная. Это значит, что не следует отводить для результата новую область памяти, а необходимо упорядочить матрицу на том же месте.

Промежуточные величины. Кроме конечных результатов, в любой программе есть промежуточные, а также служебные переменные. Следует выбрать их тип и способ хранения. Очевидно, что если требуется упорядочить матрицу по возрастанию сумм элементов ее строк, эти суммы необходимо вычислить и где-то хранить. Поскольку все они потребуются при упорядочивании, их запишем в массив, количество элементов которого соответствует количеству строк матрицы, а `i`-ый элемент содержит сумму элементов `i`-ой строки. Сумма элементов строки может превысить диапазон значений, допустимых для отдельного элемента строки, поэтому для элементов этого массива необходимо выбрать тип `long`.

После того, как выбраны структуры данных, можно приступить разработке алгоритма, поскольку алгоритм зависит от того, каким образом представлены данные.

Алгоритм работы программы. Для сортировки строк воспользуемся алгоритмом прямого выбора. При этом будем одновременно с перестановкой элементов массива выполнять обмен двух строк матрицы. Вычисления в данном случае состоят из 2-х шагов: формирование сумм элементов каждой строки и упорядочивание матрицы. Упорядочивание состоит в выборе наименьшего элемента и обмена с первым из рассматриваемых. Разработанные алгоритмы полезно представить в виде блок-схемы. Функционально завершение части алгоритма отделяется пустой строкой и/или комментарием. Не следует стремиться написать всю программу сразу. Сначала рекомендуется написать и отладить фрагмент, содержащий ввод исходных данных. Затем целесообразно перейти к следующему фрагменту алгоритма.

Ниже приведен текст программы сортировки строк матрицы по возрастанию сумм элементов.

```
#include <iostream.h>
#include <iomanip.h>
// прототипы функций
```

```
// функция, суммирующая элементы строк
void SummaStrok(int **a,int m,int n,long *v);
// функция, выполняющая вывод матрицы и
// суммы элементов в строках
void Vyvod(int **a,int m,int n,long *v);
// функция, выполняющая сортировку строк
void Sort(int **a,int m,int n,long *v);
int main()
{ int m,n,i,j;
cout<<"Введите количество строк и столбцов матрицы:";
    cin>>m>>n;
    int **a=new int *[m]; // выделение памяти
    for(i=0;i<m;i++)      // под матрицу
        a[i]=new int[n];
    for(i=0;i<m,i++) // ввод матрицы
        for(j=0;j<n;j++) cin>>a[i][j];
    int *v=new long[m]; // вспомогательный массив
 SummaStrok(a,m,n,v); // суммирование элементов строк
 Vyvod(a,m,n,v); // контрольная печать
 Sort(a,m,n,v); // сортировка
 Vyvod(a,m,n,v); // вывод результатов
return 0;
}
  //---------------------------------------
void SummaStrok(int **a,int m,int n,long *v)
{ int i,j;
  for(i=0;i<m;i++)
     {
       v[i]=0;
       for(j=0;j<n;j++)  v[i]+=a[i][j];
     }
}
  //---------------------------------------
void Vyvod(int **a,int m,int n,long *v)
{  int i,j;
   for(i=0;i<m;i++)
   { for(j=0;i<n;j++)
        cout<<setw(4)<<a[i][j]<<" ";
        cout<<endl;
   }
  for(i=0;i<m;i++)
      cout<<setw(4)<<v[i]<<" ";
  cout<<endl;
}
 //---------------------------------------
// сортировка строк матрицы методом прямого выбора
void Sort(int **a,int m,int n,long *v)
{    long buf_sum;
```

```
     int min,buf_a;
     int i,j;
for(i=0;i<m-1;i++)
   {
    min=i;
    for(j=i+1;j<m;j++) // поиск минимального элемента
      if(v[j]<v[min]) min=j;
       buf_sum=v[i]; // обмен значений v
       v[i]=v[min];
       v[min]=buf_sum;
       for(j=0;j<n;j++)//перестановка строк матрицы a
         {  buf_a=a[i][j];
            a[i][j]=a[min][j];
            a[min][j]=buf_a;
         }
   }
}
```

В функции `Sort` используются две буферные переменные: `buf_sum`, через которую осуществляется обмен двух значений сумм (имеет такой же тип, что и сумма), `buf_a` для обмена значений элементов массива (того же типа, что и элементы массива).

4.3 Варианты заданий

Каждый пункт нижеприведенного задания оформить в виде функции. Все необходимые данные для функций должны передаваться им в качестве параметров. Ввод-вывод данных и результатов также организовать с помощью соответствующих функций.

Вариант 1

Дана целочисленная прямоугольная матрица. Определить:

1) количество строк, не содержащих ни одного нулевого элемента;

2) максимальное из чисел, встречающихся в заданной матрице более одного раза.

Вариант 2

Дана целочисленная прямоугольная матрица. Определить количество столбцов, не содержащих ни одного нулевого элемента.

Характеристикой строки целочисленной матрицы назовем сумму ее положительных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с ростом характеристик.

Вариант 3

Дана целочисленная прямоугольная матрица. Определить:

1) количество столбцов, содержащих хотя бы один нулевой элемент;

2) номер строки, в которой находится самая длинная серия одинаковых элементов.

Вариант 4

Дана целочисленная квадратная матрица. Определить:

1) произведение элементов в тех строках, которые не содержат отрицательных элементов;

2) максимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

Вариант 5

Дана целочисленная квадратная матрица. Определить:

1) сумму элементов в тех столбцах, которые не содержат отрицательных элементов;

2) минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали матрицы.

Вариант 6

Дана целочисленная прямоугольная матрица. Определить:

1) сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент;

2) номера строк и столбцов всех седловых точек матрицы.

Примечание. Матрица A имеет седловую точку a_{ij}, если a_{ij} является минимальным элементом в i-й строке и максимальным в j-м столбце.

Вариант 7

Для заданной матрицы найти такие k, что k-я строка матрицы совпадает с k-м столбцом.

Найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

Вариант 8

Характеристикой столбца целочисленной матрицы назовем сумму модулей его отрицательных нечетных элементов. Переставляя столбцы заданной матрицы, расположить их в соответствии с ростом характеристик.

Найти сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.

Вариант 9

Соседями элемента a_{ij} в матрице A назовем элементы a_{kl}, для которых $i-1 \le k \le i+1,\ j-1 \le l \le j+1$. Операция сглаживания матрицы дает новую матрицу того же размера, каждый элемент которой получается как среднее арифметическое имеющихся соседей соответствующего элемента исходной матрицы. Построить результат сглаживания заданной вещественной матрицы размером 10 на 10. В сглаженной матрице найти сумму модулей элементов, расположенных ниже главной диагонали.

Вариант 10

Соседями элемента a_{ij} в матрице A назовем элементы a_{kl}, для которых $i-1 \le k \le i+1,\ j-1 \le l \le j+1$. Элемент матрицы называется локальным минимумом, если он строго меньше всех имеющихся у него соседей. Подсчитать количество локальных минимумов заданной матрицы размером 10×10. Найти сумму модулей элементов, расположенных выше главной диагонали.

Вариант 11

Соседями элемента a_{ij} в матрице A назовем элементы a_{kl}, для которых $i-1 \le k \le i+1,\ j-1 \le l \le j+1$. Элемент матрицы называется локальным максимумом,

если он строго больше всех имеющихся у него соседей. Подсчитать количество локальных максимумов заданной матрицы размером 10×10. Найти сумму модулей элементов, расположенных ниже главной диагонали.

Вариант 12

Уплотнить заданную матрицу, удаляя из нее строки и столбцы, заполненные нулями. Найти номер первой из строк, содержащих хотя бы один положительный элемент.

Вариант 13

Осуществить циклический сдвиг элементов прямоугольной матрицы на n элементов вправо или вниз (в зависимости от введенного режима), n может быть больше количества элементов в строке или столбце.

Вариант 14

Осуществить циклический сдвиг элементов квадратной матрицы вправо на k элементов таким образом: элементы 1-й строки сдвигаются в последний столбец сверху вниз, из него — в последнюю строку справа налево, из нее — в первый столбец снизу вверх, из него — в первую строку; для остальных элементов сдвиг выполняется аналогичным образом.

Вариант 15

Дана целочисленная прямоугольная матрица. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.

Характеристикой строки целочисленной матрицы назовем сумму ее отрицательных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с убыванием характеристик.

Вариант 16

Упорядочить строки целочисленной прямоугольной матрицы по возрастанию количества одинаковых элементов в каждой строке. Найти номер первого из столбцов, не содержащих ни одного отрицательного элемента.

Вариант 17

Путем перестановки элементов квадратной вещественной матрицы добиться того, чтобы ее максимальный элемент находился в левом верхнем углу, следующий по величине — в позиции (2,2), следующий по величине — в позиции (3,3) и т. д., заполнив таким образом всю главную диагональ.

Найти номер первой из строк, не содержащих ни одного положительного элемента.

Вариант 18

Дана целочисленная прямоугольная матрица. Определить:

1) количество строк, содержащих хотя бы один нулевой элемент;

2) номер столбца, в котором находится самая длинная серия одинаковых элементов.

Вариант 19

Дана целочисленная квадратная матрица. Определить:

1) сумму элементов в тех строках, которые не содержат отрицательных элементов;

2) минимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

Вариант 20

Дана целочисленная прямоугольная матрица. Определить:

1) количество отрицательных элементов в тех строках, которые содержат хотя бы один нулевой элемент;

2) номера строк и столбцов всех седловых точек матрицы.

Примечание. Матрица A имеет седловую точку a_{ij}, если a_{ij} является минимальным элементом в i-й строке и максимальным в j-м столбце.

Вариант 21

Дана прямоугольная матрица. Выполнить:

1) зеркальную перестановку столбцов матрицы относительно вертикальной оси;

2) зеркальную перестановку строк матрицы, полученной в результате выполнения предыдущего пункта, относительно горизонтальной оси.

Вариант 22

Дана прямоугольная матрица. Выполнить:

1) зеркальную перестановку строк матрицы относительно горизонтальной оси;

2) зеркальную перестановку столбцов матрицы, полученной в результате выполнения предыдущего пункта, относительно вертикальной оси.

Вариант 23

Дана прямоугольная матрица. Выполнить:

1) поиск позиций всех локальных минимумов матрицы;

2) поиск позиций всех локальных максимумов матрицы.

Вариант 24

Дана прямоугольная матрица. Осуществить поиск позиций максимального и минимального элементов матрицы. Найти среднее арифметическое всех элементов матрицы.

Примечание. Позицией элемента матрицы называется упорядоченная пара (i, j), где i — номер строки элемента, j — номер столбца.

Вариант 25

Определить, является ли заданная квадратная матрица симметрической. Найти сумму положительных и сумму отрицательных элементов матрицы.

Примечание. Матрица A является симметрической, если $A^T = A$.

Вариант 26

Определить, является ли заданная квадратная матрица антисимметрической. Найти сумму нулевых элементов матрицы.

Примечание. Матрица A является антисимметрической, если $A^T = -A$.

Вариант 27

Упорядочить строки прямоугольной целочисленной матрицы по возрастанию сумм их элементов методом прямого обмена.

Вариант 28

Упорядочить столбцы прямоугольной целочисленной матрицы по возрастанию сумм их элементов методом прямого обмена.

Вариант 29

Упорядочить строки прямоугольной целочисленной матрицы по возрастанию сумм их элементов методом вставки.

Вариант 30

Упорядочить столбцы прямоугольной целочисленной матрицы по возрастанию сумм их элементов методом вставки.

4.4 Порядок выполнения работы

4.4.1 Выбрать типы данных для хранения исходных данных, промежуточных величин и результатов.

4.4.2 Разработать алгоритм решения задачи в соответствии с вариантом.

4.4.3 Составить программу, оформив ввод данных, вывод результатов, необходимые вычисления в виде функций.

4.4.4 Разработать тестовые примеры, которые проверяют все ветви алгоритма и возможные диапазоны данных. В качестве тестового примера рекомендуется ввести значения элементов массива, близкие к максимально возможным. Дополнительно рекомендуется проверить работу программы, когда массив состоит из одной или двух строк (столбцов), поскольку многие ошибки при написании циклов связаны с неверным указанием их граничных значений.

4.4.5 Выполнить отладку программы.

4.4.6 Получить результаты для всех вариантов тестовых примеров.

4.5 Содержание отчета

Цель работы, вариант задания, структурные схемы алгоритмов всех функций, описание тестовых примеров, текст программы, анализ результатов вычислений, выводы.

4.6 Контрольные вопросы

4.6.1 Как в языках C/C++ представляются двумерные массивы?

4.6.2 Как осуществляется доступ к элементам двумерного массива?

4.6.3 Каким образом осуществляется инициализация двумерных массивов?

4.6.4 Как выделить память под динамический двумерный массив?

4.6.5 Приведите формат заголовка функции.

4.6.6 Что является телом функции?

4.6.7 В чем состоит механизм передачи параметров в функцию?

4.6.8 Что такое ссылочные переменные? В каких случаях их целесообразно использовать?

4.6.9 Что такое аргументы по умолчанию?

4.6.10 Как осуществляется передача массивов в функции?

4.6.11 Напишите функцию, выполняющую поиск номера строки матрицы, в которой находиться минимальный элемент.

4.6.12 Напишите функцию, выполняющую перестановку 2-х заданных строк (столбцов) матрицы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Павловская Т. А. Паскаль. Программирование на языке высокого уровня : практикум / Т. А. Павловская. — СПб. : Питер, 2006. — 317 с.

2. Павловская Т. А. Паскаль. Программирование на языке высокого уровня : учеб. для вузов / Т. А. Павловская. — СПб. : Питер, 2008. — 393 с.

3. Методические указания к лабораторным работам по дисциплине “Алгоритмизация и программирование” для студентов дневной и заочной форм обучения направления 09.03.02 – “Информационные системы и технологии”, часть 1/ Сост. В.Н. Бондарев, Т.И. Сметанина, А.К. Забаштанский.— Севастополь: Изд-во СевГУ, 2016. — 76с.

4. Введение в интегрированную среду разработки Eclipse CDT: методические указания к лабораторным работам по дисциплине “Основы программирования и алгоритмические языки” для студ. дневной и заочной форм обучения направления 09.03.02 — “Информационные системы и технологии” сост. В. Н. Бондарев, Т.И. Сметанина — Севастополь: Изд-во СевГУ, 2014. — 52 с.

5. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. / Н. Вирт. — М.: Мир, 1989. — 360с.

6. Керниган Б., Ритчи Д. Язык программирования Си: пер. с англ. / Под ред. и с предисл. В.С. Штаркмана.—2-е изд., перераб. и доп.— М. ; СПб. ; К. : Вильямс, 2006.—272с.

7. Павловская Т.А. С/ С++. Программирование на языке высокого уровня : учеб. для студ. вузов, обуч. по напр. «Информатика и вычислительная техника» / Т. А. Павловская.— СПб.: Питер, 2009. – 461 с.

8. Павловская Т.А. С/С++. Структурное программирование: практикум / Т.А. Павловская, Ю.А. Щупак. — СПб.: Питер, 2004.—239 с.

ПРИЛОЖЕНИЕ А
ОСНОВЫ ЯЗЫКА СИ

А.1 Структура С-программы

Любая *С-программа* состоит из *функций* и *переменных*. Функции содержат *инструкции*, описывающие вычисления, которые необходимо выполнить, а переменные хранят значения, используемые в процессе этих вычислений. Любая программа начинает свои вычисления с первой инструкции функции `main`. Таким образом, каждая С-программа должна содержать функцию `main`. Эта функция может не иметь параметров, и тогда ее заголовок — `main()`. Если функция `main` имеет параметры, то эти параметры выбираются из *строки вызова* (*командной строки*).

Левая фигурная скобка (`{`) должна начинать *тело* каждой функции. Соответствующая *правая фигурная скобка* (`}`) должна заканчивать каждую функцию. Так что тело функции представляет собой *блок*, ограниченный фигурными скобками `{}`. Блок содержит *выражения*, заканчивающиеся *точкой с запятой* (`;`), которые в языке С называют *инструкциями*. Общая структура функции `main` такова:

```
main()
{   инструкция_1
    инструкция_2
    ............
    инструкция_n
}
```

Помимо функции `main` в тексте программы могут располагаться определения *вспомогательных функций*.

В системах программирования С перед началом этапа компиляции выполняется программа предварительной *(препроцессорной)* обработки. Эта программа подчиняется специальным командам *(директивам препроцессора)*, которые указывают, что в программе перед ее компиляцией нужно выполнить определенные преобразования. Обычно эти преобразования состоят во включении других текстовых файлов в файл, подлежащий компиляции, и выполнении различных текстовых замен. Так, например, появление директивы

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

приводит к тому, что *препроцессор* подставляет на место этой директивы текст файла `<stdio.h>`, т. е. происходит включение информации о *стандартной библиотеке ввода-вывода*.

А.2 Переменные и основные типы данных языка С

В С любая *переменная* должна быть описана раньше, чем она будет использована; обычно все переменные описываются в начале функции перед первой исполняемой инструкцией. В *декларации (описании)* объявляются свойства переменных. Декларация состоит из названия типа и списка переменных, например:

```
int width, height;
float distance;
int exam_score;
```

```
char c;
```

В первом описании имеется *список переменных*, содержащий два имени (`width` и `height`). Обе переменные описываются как целые (`int`). Целочисленная переменная `exam_score` описана отдельно, хотя ее можно добавить к первому списку целых переменных.

Переменные можно инициализировать в месте их описания, например:

```
float distance=15.9;
```

Здесь переменной `distance` присваивается начальное значение `15.9`.

Имя переменной — это любая последовательность символов, содержащая *буквы*, *цифры* и *символы подчеркивания* (_), которая не начинается с цифры. Язык C *чувствителен к регистру* — прописные и строчные буквы различаются.

В C существует всего несколько *базовых типов*:

`char` — единичный байт, который может содержать одну литеру;

`int` — целое;

`float` — число с плавающей точкой одинарной точности;

`double` — число с плавающей точкой повышенной точности.

Имеется также несколько *квалификаторов*, которые можно использовать вместе с указанными базовыми типами. Например, квалификаторы `short` (короткий) и `long` (длинный) применяются к целым:

```
short int counter; long int amount;
```

В таких описаниях слово `int` можно опускать, что обычно и делается.

Квалификаторы `signed` (со знаком) и `unsigned` (без знака) можно применять к типу `char` и любому целому типу. Тип `long double` предназначен для арифметики с плавающей точкой повышенной точности.

Если операнды оператора принадлежат разным типам, то они *приводятся* к общему типу. Автоматически производятся лишь те преобразования, которые без какой-либо потери информации превращают операнды с меньшим диапазоном значений в операнды с большим диапазоном значений.

A.3 Основные операторы языка C

В таблице A.1 показаны *приоритеты* и *порядок вычисления* всех операторов языка C. Операторы, перечисленные на одной строке, имеют одинаковый приоритет; строки упорядочены по убыванию приоритетов. *Унарные операторы* +, –, и * имеют более высокий приоритет, чем те же операторы в *бинарном* варианте.

A.3.1 Арифметические операторы

К *арифметическим операторам* относятся: *сложение* (+), *вычитание* (–), *деление* (/), *умножение* (*) и *остаток* (%). Все операции (за исключением остатка) определены для переменных типа `int`, `char` и `float`. Остаток не определен для переменных типа `float`. Деление целых сопровождается отбрасыванием дробной части.

Все арифметические операции с плавающей точкой производятся над операндами *двойной точности* (`double`). После того, как получен результат с двой-

ной точностью, он приводится к типу левой части выражения, если левая часть присутствует.

Таблица А.1 — Приоритеты и порядок вычисления операторов

Операторы	Порядок выполнения
`() [] -> .`	слева направо
`! ~ ++ -- + - * & (тип) sizeof`	справа налево
`* / %`	слева направо
`+ -`	слева направо
`<< >>`	слева направо
`< <= > >=`	слева направо
`== !=`	слева направо
`&`	слева направо
`^`	слева направо
`|`	слева направо
`&&`	слева направо
`||`	слева направо
`?:`	справа налево
`= += -= *= /= %= &= ^= |= <<= >>=`	справа налево
`,`	слева направо

А.3.2 Операторы отношения

В языке С определены следующие *операторы отношения*: *проверка на равенство* (==), *проверка на неравенство* (!=), *меньше* (<), *меньше или равно* (<=), *больше* (>), *больше или равно* (>=).

Все перечисленные операторы вырабатывают результат целого типа (`int`). Если данное отношение между операндами ложно, то значение этого целого — ноль. Значения, отличные от нуля, интерпретируются как истинные.

А.3.3 Логические операторы

К *логическим операторам* относятся:

`&&` — *логическое И* (and);

`||` — *логическое ИЛИ* (or);

`!` — *логическое НЕ* (not).

Операндами логических операторов могут быть любые числа. Результат логического выражения — единица, если истина, и ноль, если ложь. Вычисление выражений, содержащих логические операторы, производится слева направо и прекращается, как только удается определить результат.

А.3.4 Операторы присваивания

К *операторам присваивания* относятся =, +=, –=, *= и /=, а также *префиксные* и *постфиксные* операторы ++ и – –. Все операторы присваивания присваивают переменной результат вычисления выражения.

В одном выражении оператор присваивания может встречаться несколько раз. Вычисления производятся *справа налево*. Например: `a=(b=c)*d;` вначале

переменной b присваивается значение переменной c, затем выполняется операция умножения на d, и результат присваивается переменной a.

Типичный пример использования *многократного присваивания*:

```
a=b=c=d=e=f=0;
```

Операторы +=, –=, *=, /= являются *укороченной* формой записи оператора присваивания. Например:

```
a+=b; // a=a+b;
a–=b; // a=a–b;
a*=b; // a=a*b;
a/=b; // a=a/b;
```

Многие компиляторы генерируют код, который выполняется быстрее при использовании таких операторов присваивания.

Префиксные и постфиксные операторы присваивания ++ и – – используют для увеличения *(инкремент)* и уменьшения *(декремент)* на единицу значения переменной. Эти операторы можно использовать как в префиксной форме (помещая перед переменной, например, ++n), так и в постфиксной форме (помещая после переменной: n++).

A.4 Основные средства ввода-вывода языков C/C++

A.4.1 Основные средства ввода-вывода языка C: функции printf и scanf

В языке C ввод-вывод данных реализуется с помощью *внешних функций*. Одними из наиболее универсальных и полезных функций ввода и вывода являются соответственно функции scanf и printf, описанные в головном файле <stdio.h>.

Функцию printf можно использовать для вывода любой комбинации символов, целых и вещественных чисел, строк, беззнаковых целых, длинных целых и беззнаковых длинных целых.

Типичный пример использования функции printf:

```
#include <stdio.h>
main()
{int age=22; // возраст Эрика
 float income=534.72; // доход Эрика
 printf("\nВозраст Эрика: %d. Его доход: $%.2f.\n",
           age,income);
 return 0;
}
```

Функция printf выводит на экран значения своих аргументов age и income в формате, который определяется *управляющей строкой*, являющейся первым параметром этой функции. Все символы этой строки, кроме *спецификаций формата* (%d и %.2f) и *управляющей последовательности* (\n), выводятся на экран без изменений.

Таким образом, при выполнении функции printf произойдет следующее. Последовательность символов \n переведет курсор на новую строку. В результате последовательность символов «Возраст Эрика: » будет выведена с

начала новой строки. Смволы %d — это спецификация формата для целой переменной. Вместо %d будет подставлено значение переменной age. Далее будет выведена литеральная строка «. Его доход: $». %.2f — это спецификация формата для вещественной переменной, а также указание формата для вывода только двух цифр после десятичной точки. Вместо %.2f будет выведено значение переменной income в указанном формате. В заключение управляющая последовательность \n вызовет переход на новую строку.

Как видно из рассмотренного примера, спецификации формата помещаются внутри печатаемой строки. Вслед за этой строкой должен стоять ноль или более переменных, разделенных запятыми. Каждой спецификации в управляющей строке функции printf должна соответствовать переменная адекватного типа. Если используется несколько спецификаций, то всем им должны соответствовать переменные того типа, который задается спецификацией.

Формально спецификацию формата можно определить следующим образом: %[флаг][ширина][.точность][h|l|L]символ_формата,
где ширина — минимальное количество позиций, отводимых под выводимое значение, точность — количество позиций, отводимых под дробную часть числа.

Модификатор h указывает, что соответствующий аргумент должен печататься как short или unsigned short; l сообщает, что аргумент имеет тип long или unsigned long; L информирует, что аргумент принадлежит типу long double.

Возможные значения полей флаг и символ_формата приведены в таблицах А.2 и А.3.

Таблица А.2 — Описание значений поля флаг

Флаг	Назначение
-	выравнивание по левому краю поля
+	печать числа всегда со знаком
Пробел	если первая литера — не знак, то числу должен предшествовать пробел

Таблица А.3 — Описание значений поля символ_формата

Символ формата	Тип выводимого объекта
C	char; единичная литера
S	char *; строка
d,i	int; знаковая десятичная запись
o	int; беззнаковая восьмеричная запись
u	int; беззнаковое десятичное целое
x,X	int; беззнаковая шестнадцатеричная запись
f	double; вещественное число с фиксированной точкой

Продолжение таблицы А.3

Символ формата	Тип выводимого объекта
e,E	double; вещественное число с плавающей точкой
g,G	double; вещественное число в виде %f или %e в зависимости от значения
p	void *; указатель
%	знак процента %

Функция scanf служит для ввода данных. Подобно функции printf, scanf использует спецификацию формата, сопровождаемую списком вводимых переменных. Каждой вводимой переменной в функции scanf должна соответствовать спецификация формата.

Перед именами переменных следует ставить символ &. Этот символ означает *«взять адрес переменной»*. Ниже приведен пример программы, использующей функцию scanf:

```
#include <stdio.h>
main()
{ int weight, // вес
      height; // рост
  printf("Введите Ваш вес:\n"); scanf("%d",&weight);
  printf("Введите Ваш рост:\n"); scanf("%d",&height);
  printf("\nВаш вес = %d;\nВаш рост = %d.\n",
              weight,height);
  return 0;
}
```

Здесь &weight и &height — это адреса переменных weight и height соответственно.

А.4.2 Средства ввода-вывода языка C++: объекты cin и cout

В C++ вывод на экран выполняется с помощью *объекта стандартного потока вывода* cout, а ввод с клавиатуры осуществляется с помощью *объекта стандартного потока ввода* cin. Для использования этих объектов необходимо подключить головной файл <iostream.h>. Для обработки форматного ввода-вывода при помощи так называемых *манипуляторов потока* необходимо подключить головной файл <iomanip.h>.

Рассмотрим пример использования объекта cout:

```
#include <iomanip.h>
#include <iostream.h>
main()
{   int age=22; // возраст Эрика
    float income=534.72; // доход Эрика
    cout<<'\n'<<"Возраст Эрика: "<<age<<'.'
        <<" Его доход: $"<<setprecision(2)
```

```
            <<income<<'.'<<endl;
    return 0;
}
```

Вывод в поток выполняется с помощью *операции поместить в поток* <<. В рассматриваемом примере объекту `cout` с помощью операции << передаются значения, которые необходимо вывести на экран. Операция << является «более интеллектуальной» по сравнению с функцией `printf`, так как определяет тип выводимых данных. Для вывода нескольких объектов операция << используется в *сцепленной форме*.

Манипулятор потока `endl` вызывает переход на новую строку и очищает *буфер вывода*, т. е. заставляет буфер немедленно вывести данные, даже если он полностью не заполнен. *Очистка (сброс)* буфера вывода может быть также выполнена манипулятором `flush`.

При выводе значения переменной `income` используется манипулятор потока `setprecision`. Использование `setprecision(2)` — это указание формата для вывода только двух цифр после десятичной точки. Поскольку манипулятор `setprecision` принимает параметр, он называется *параметризированным манипулятором потока*.

Для установки ширины поля вывода может быть использован манипулятор потока `setw`, принимающий в качестве параметра число символьных позиций, в которые будет выведено значение.

Ввод потока осуществляется с помощью *операции взять из потока* >>. Эта операция применяется к объекту стандартного потока ввода `cin`. Рассмотрим пример использования объекта `cin`:

```
#include <iostream.h>
main()
{   int weight, // вес
        height; // рост
    cout<<"Введите Ваш вес:"<<endl;     cin>>weight;
    cout<<"Введите Ваш рост:"<<endl;    cin>>height;
    cout<<"\nВаш вес = "<<weight<<';'<<endl
        <<"Ваш рост = "<<height<<'.'<<endl;
    return 0;
}
```

Здесь считываемые значения размещаются в переменных `weight` и `height`. В процессе ввода последовательность символов, набранная на клавиатуре, преобразуется в необходимое *внутреннее представление* (в рассматриваемом примере целочисленное).

A.5 Управляющие инструкции языка C: `if-else`, условное выражение, `switch`

A.5.1 Инструкция `if-else`

Инструкция `if-else` используется для принятия решения. Ниже представлен ее синтаксис:

```
if (выражение)    инструкция_1
else     инструкция_2
```

причем else-часть может быть опущена. Сначала вычисляется выражение, и, если оно истинно, то выполняется инструкция_1. Если выражение ложно и существует else-часть, то выполняется инструкция_2. Необходимо отметить, что else-*часть всегда относится к ближайшей* if-*части.*

A.5.2 Условное выражение

Условное выражение, написанное с помощью *тернарного оператора* «?:», представляет собой другой способ записи инструкции if-else. В выражении

```
выражение1 ? выражение2 : выражение3
```

первым вычисляется выражение1. Если его значение отлично от нуля, то вычисляется выражение2 и значение этого выражения становится значением всего условного выражения. В противном случае вычисляется выражение3, и его значение становится значением условного выражения. Таким образом, чтобы установить в z наибольшее из a и b, можно записать:

```
z = (a > b) ? a : b; // z=max(a,b)
```

A.5.3 Инструкция switch

Инструкция switch используется для выбора одного из многих путей. Она проверяет, совпадает ли значение выражения с одним из значений, входящих в некоторое множество целых констант, и выполняет соответствующую этому значению ветвь программы:

```
switch (выражение){
  case конст_выражение_1: последовательность_инструкций_1
  case конст_выражение_2: последовательность_инструкций_2
  ..................................................
  case конст_выражение_n: последовательность_инструкций_n
default: последовательность_инструкций_n+1
```

Константные выражения всех case-ветвей должны быть целочисленными и должны отличаться друг от друга.

Инструкция switch выполняется следующим образом. Вначале вычисляется выражение, и результат сравнивается с каждой case-константой. Если одна из case-констант равна значению выражения, управление переходит на последовательность инструкций с соответствующей case-меткой. Если ни с одной из case-констант нет совпадения, управление передается на последовательность инструкций с default-меткой, если такая имеется, в противном случае ни одна из последовательностей инструкций switch не выполняется. Ветви case и default можно располагать в любом порядке.

Для иллюстрации инструкции switch рассмотрим программу, которая определяет вид литеры, введенной пользователем с клавиатуры. Вид литеры — это цифра (digit), *пробельная литера* (white space) или другая литера (other). К пробельным литерам относят пробел (' '), *литеру табуляции* ('\t') и *литеру новая-строка* ('\n'). Ниже представлен текст программы.

```
#include <iostream.h>
main()
// определение вида литеры, введенной с клавиатуры
// (цифра, пробельная литера или другая литера)
{   char c; // литера, вводимая с клавиатуры
    c=cin.get(); // ввод литеры с клавиатуры
    switch(c){
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        // цифра
        cout<<"digit"<<endl; break;
    case ' ': case '\t': case '\n':
        // пробельная литера
        cout<<"white space"<<endl; break;
    default:
        // другая литера
        cout<<"other"<<endl; break;
    }
   return 0;
}
```

Литера, заключенная в одиночные кавычки, представляет целое значение, равное коду этой литеры (в кодировке, принятой на данной машине). Это так называемая *литерная константа*. Например, `'A'` есть литерная константа; в наборе ASCII ее значение равняется `65`. `'\t'` и `'\n'` обозначают коды литеры табуляции и литеры новая-строка соответственно.

Функция-элемент `get` объекта `cin` вводит одиночный символ из потока и возвращает этот символ в качестве *значения вызова функции*. Для вывода одиночного символа в поток используется функция-элемент `put`, которая в качестве аргумента принимает значение кода символа. Следует отметить, что стандартная библиотека ввода-вывода `<stdio.h>` включает функции для чтения и записи одной литеры `getchar` и `putchar`, аналогичные функциям-элементам `get` и `put` соответственно.

Инструкция `break` вызывает немедленный выход из инструкции `switch`. Поскольку выбор ветви `case` реализуется как переход на метку, то после выполнения одной ветви `case`, если ничего не предпринять, программа «провалится вниз» на следующую ветвь. Инструкция `break` — наиболее распространенное средство выхода из инструкции `switch`.

A.6 Математические функции файла <math.h>

Ниже приведен перечень основных математических функций, описанных в головном файле `<math.h>`. Аргументы `x` и `y` функций имеют тип `double`; все функции возвращают значения типа `double`. Углы в тригонометрических функциях задаются *в радианах*.

`sin(x)` — *синус x*;

`cos(x)` — *косинус x*;

`tan(x)` — *тангенс* x;
`asin(x)` — *арксинус* x в диапазоне $[-\pi/2, \pi/2]$, $x \in [-1, 1]$;
`acos(x)` — *арккосинус* x в диапазоне $[0, \pi]$, $x \in [-1, 1]$;
`atan(x)` — *арктангенс* x в диапазоне $[-\pi/2, \pi/2]$;
`atan2(y,x)` — *арктангенс* y/x в диапазоне $[-\pi, \pi]$;
`sinh(x)` — *гиперболический синус* x;
`cosh(x)` — *гиперболический косинус* x;
`tanh(x)` — *гиперболический тангенс* x;
`exp(x)` — *экспоненциальная функция* e^x;
`log(x)` — *натуральный логарифм* $\ln(x)$, $x > 0$;
`log10(x)` — *десятичный логарифм* $\lg(x)$, $x > 0$;
`pow(x,y)` — x^y;
`sqrt(x)` — $\sqrt{x}$, $x \geq 0$;
`ceil(x)` — *наименьшее целое* в виде `double`, которое $\geq x$;
`floor(x)` — *наибольшее целое* в виде `double`, которое $\leq x$;
`fabs(x)` — *абсолютное значение* $|x|$;
`fmod(x,y)` — *остаток от деления* x *на* y в виде числа с плавающей точкой.

A.7 Пример программы разветвляющейся структуры

Ниже представлен текст С-программы, вычисляющей значение функции

$$z = f(x) = \begin{cases} \sin(2x + \pi/7), \text{ если } x \leq a, \\ x^{3,21} + \tan(x), \text{ если } a < x < b, \\ \sqrt{x}\lg(x), \text{ если } x \geq b. \end{cases}$$

Значения параметров a, b и аргумента x вводятся с клавиатуры. Результаты вычислений выводятся на дисплей в *формате с плавающей точкой*.

```
#include <conio.h>
#include <math.h>
#include <stdio.h>
#define PI 3.14159265
main()
// вычисление значения функции z=f(x)
{
   float a, // параметр a
         b, // параметр b
         x, // аргумент x
         z; // значение функции z
    clrscr();
   // ввод значений параметров a, b и аргумента x
   printf("Введите значение параметра a: ");
   scanf("%f",&a);
```

```
    printf("Введите значение параметра b: ");
    scanf("%f",&b);
    printf("Введите значение аргумента x: ");
    scanf("%f",&x);
    // вычисление значения функции z
    if (x<=a) z=sin(2*x+PI/7);
    else if (x<b) // a<x<b
         z=pow(x,3.21)+tan(x);
    else // x>=b
         z=sqrt(x)*log10(x);
         // вывод значения функции z в формате
         //с плавающей точкой
    printf("Значение функции z = %e\n",z);
    printf("Нажмите любую клавишу...");
    getch();
    return 0;
}
```

С помощью директивы

```
#define PI 3.14159265
```

оказывается возможным указать препроцессору, чтобы он при любом появлении идентификатора `PI` в тексте программы заменял его на значение `3.14159265` ($\approx \pi$).

Функция `clrscr` библиотеки `<conio.h>` выполняет очистку экрана и перемещение курсора в левый верхний угол.

Функция `getch` библиотеки `<conio.h>` используется для ввода символов с клавиатуры без их высвечивания на экране. С помощью `getch` можно считать коды основных клавиш (ASCII-коды) и *расширенные коды*. Расширенные коды — это коды верхнего ряда клавиш, коды правой части клавиатуры и коды комбинаций клавиш `Alt`, `Ctrl`, `Shift` с другими клавишами. В случае считывания расширенных кодов при первом обращении функция `getch` возвращает нулевое значение, а ее повторный вызов позволяет получить расширенный код.

Используя возможности языка C++, вышеприведенную программу можно переписать следующим образом:

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
const double pi=3.14159265;
main()
{
    // вычисление значения функции z=f(x)
    // объявление переменных и очистка экрана
    ..................................
    // ввод значений параметров a, b и аргумента x
    cout<<"Введите параметр a: ";
    cin>>a;
```

```
        cout<<"Введите параметр b: ";
        cin>>b;
        cout<<"Введите аргумент x: ";
        cin>>x;
        // вычисление значения функции z
        ..................................
             // вывод значения функции z в формате
              //с плавающей точкой
        cout.setf(ios::scientific, ios::floatfield);
        cout<<"Значение функции z = f(x) = "<<z<<endl;
        cout<<"Нажмите любую клавишу...";
        getch();
        return 0;
}
```

Как видно, в C++ вместо *символических констант*, которые создаются директивой препроцессора `#define`, имеется возможность использования *константных переменных*. Строка `const double pi=3.14159265;` использует *спецификацию* `const` для объявления *константы* `pi`, имеющей значение `3.14159265`. После определения константной переменной изменить ее значение нельзя. Следует отметить, что в C++ отдается предпочтение использованию константных переменных, а не символических констант. Константные переменные, в отличие от символических констант, являются данными определенного типа, и их имена видны отладчику.

Строка `cout.setf(ios::scientific,ios::floatfield);` устанавливает *экспоненциальный формат* вывода вещественных чисел (*формат с плавающей точкой*). Для отображения чисел в *формате с фиксированной точкой* следует записать `cout.setf(ios::fixed,ios::floatfield);` здесь функция-элемент `setf` объекта `cout` используется для управления установкой *флагов формата* вывода вещественных чисел `ios::scientific` или `ios::fixed`, которые содержатся в *статическом элементе данных* `ios::floatfield`.

A.8 Одномерные массивы

Одномерный массив — это набор объектов одинакового типа, расположенных в *последовательной* группе ячеек памяти, доступ к которым осуществляется *по индексу*. В языке C одномерные массивы описываются следующим образом:

`тип имя_массива[размер_массива];`

В результате такого объявления компилятор отводит под массив память размером **`sizeof(тип)*размер_массива`** байтов. Операция (функция) **`sizeof`**, операндами (аргументами) которой могут являться константы, типы и переменные, в качестве результата возвращает количество байт, занимаемых ее операндом (аргументом).

Используя имя массива и индекс, можно обращаться к элементам массива: **`имя_массива[индекс]`**. Все элементы массива индексируются, *начиная с нуля*

и заканчивая величиной, на единицу меньшей, чем размер массива, указанный при его описании. Например, если массив **data** объявлен как

```
double data[245];
```

то его **245** элементами типа **double** будут: **data[0]**, **data[1]**, …, **data[244]**. Индекс массива может быть как целым числом, так и целочисленным выражением. Квадратные скобки, внутри которых записывается индекс массива, рассматриваются как *оператор индексации*. Оператор индексации имеет самый высокий приоритет (см. таблицу 1 в лабораторной работе №1).

Элементам массива можно присваивать начальные значения (*инициализировать* их) при объявлении массива с помощью списка начальных значений, разделенных запятыми, заключенного в фигурные скобки:

```
int n[10]={32,27,64,18,95,14,90,70,60,37};
```

Если начальных значений меньше, чем элементов в массиве, то оставшиеся элементы автоматически получают нулевые начальные значения. Например, элементам массива **n** можно присвоить нулевые начальные значения с помощью объявления `int n[10]={0};`

Если размер массива не указан в объявлении со списком инициализации, то количество элементов массива будет равно количеству элементов в списке начальных значений. Например, объявление `int n[]={1,2,3,4,5};`создает массив из пяти элементов.

A.9 Указатели

Указатели — это переменные, которые содержат в качестве своих значений *адреса памяти*. В общем случае переменная типа *указатель* объявляется следующим образом:

```
тип *переменная_указатель;
```

Такая запись означает, что **переменная_указатель** является указателем на объект типа **тип**. Так, объявление

```
int *countPtr,count=5;
```

объявляет переменную **countPtr** типа **int *** (т. е. указатель на целое число) и читается как «**countPtr** является указателем на целое число» или «**countPtr** указывает на объект типа **int**». Однако переменная **count** объявлена как целое число, но не как указатель на целое число. Смвол ***** в объявлении относится только к **countPtr**. *Каждая переменная, объявляемая как указатель, должна иметь перед собой знак звездочки* (*****).

Указатели должны инициализироваться либо при своем объявлении, либо с помощью оператора присваивания. Указатель может получить в качестве начального значения **0**, **NULL** или адрес. Указатель с начальными значениями **0** или **NULL** ни на что не указывает и часто используется как *ограничитель в динамических структурах*. **NULL** — это *символическая константа*, определенная в головном файле **<iostream.h>**, а также в нескольких головных файлах стандартной библиотеки языка С.

Унарный оператор адресации **&** возвращает адрес своего операнда. Например, в результате выполнения инструкции

```
countPtr=&count;
```

адрес переменной **count** будет запомнен в указателе **countPtr**.

Оператор *****, называемый *оператором раскрытия ссылки* или *оператором разыменования (разадресации)*, возвращает значение объекта, на который указывает указатель. Например, инструкция

```
cout<<*countPtr<<endl;
```

выводит на экран значение переменной **count**, а именно **5**.

Два оператора языка C++ **new** и **delete** позволяют управлять временем жизни какой-либо переменной. Переменная может быть создана в любой точке программы с помощью оператора **new** и затем при необходимости уничтожена с помощью оператора **delete**.

В результате выполнения инструкции

```
countPtr=new int;
```

переменной-указателю **countPtr** присваивается адрес начала участка памяти размером **sizeof(int)** байт. Память выделяется *динамически* из специальной области, которая называется *куча (heap)*. Оператор **delete** освобождает ранее занятую память, возвращая ее в кучу.

Наряду с операторами **new** и **delete** в языках C/C++ существуют две функции для захвата/освобождения памяти в куче: **malloc** и **free** соответственно. Эти функции описаны в головном файле **<stdlib.h>**, а также в **<alloc.h>**. Функция **malloc(size)** запрашивает память размером **size** байт из кучи и возвращает указатель на первый байт этого участка. Функция **malloc** всегда возвращает указатель на тип **void**, поэтому для получения адреса объекта определенного типа необходимо выполнить соответствующее *преобразование типа*, например:

```
int *x;
x=(int*)malloc(sizeof(int));
```

Здесь преобразование типа **(int*)** приводит значение, возвращаемое функцией **malloc** к адресу указателя на целочисленную переменную.

Память, выделенную в куче с помощью функции **malloc**, следует освобождать с помощью функции **free**. Единственным аргументом функции **free** является указатель, ссылающийся на освобождаемую память.

При работе с указателями возможны ошибки. Рассмотрим фрагмент программы

```
int *p=new int, *q=new int;
*p=255, *q=127;
p=q, *p=63;
```

Присвоение **p=q** означает, что **p**, начиная с этого момента, указывает на ту же область памяти, что и **q**. Когда по адресу, содержащемуся в **p**, заносится значение **63** (для этого используется инструкция ***p=63;**), значение, на которое

ссылается **q**, также будет равно **63** (***q==63**). Кроме того, после присвоения **p=q** значение ***p==255** оказывается *потерянным*. Не существует доступа к этой области памяти. Она остается захваченной до конца выполнения программы. Такие явления называют *утечкой памяти*.

Если указатель является *локальной переменной*, т. е. описан в некотором блоке программы, как это имеет место во фрагменте

```
{    char *q1=new char;
    *q1='c';
}
```

то при выходе из блока он *перестанет существовать* и память, на которую он ссылается, окажется недоступной. Эта память не помечается как свободная и поэтому не может быть использована в дальнейшем.

Еще один источник ошибок — *неосвобождение памяти*, запрошенной ранее с помощью оператора **new** или функции **malloc**. Сстема не способна автоматически освобождать память в куче. Неосвобождение памяти может остаться незамеченным в небольших программах, однако часто приводит к отказам в серьезных разработках и является плохим стилем программирования.

A.10 Массивы и указатели

Имя массива является *константой-указателем* и содержит адрес области памяти, которую занимает набор последовательных ячеек, соответствующих массиву.

Декларация

```
double a[10];
```

определяет массив **a**, состоящий из десяти последовательных объектов с именами **a[0]**, **a[1]**, …, **a[9]**. Если **pa** есть указатель на **double**, т. е. определен как **double *pa,** то в результате присваивания

```
pa=a;
```

или

```
pa=&a[0];
```

pa будет указывать на нулевой элемент массива **a**; иначе говоря, **pa** будет содержать адрес элемента **a[0]**.

Если **pa** указывает на некоторый элемент массива, то **pa+1** *по определению* указывает на следующий элемент, **pa-1** указывает на предыдущий элемент, **pa+i** — на *i*-й элемент после **pa**, а **pa-i** — на *i*-й элемент перед **pa**. Таким образом, если **pa** указывает на **a[0]** (**pa==&a[0]**), то ***pa** есть содержимое **a[0]**, ***(pa+1)** есть содержимое **a[1]**, а ***(pa+i)** — содержимое **a[i]**. Сделанные замечания верны безотносительно к типу и размеру элементов массива **a**. Однако нельзя выходить за границы массива и тем самым ссылаться на несуществующие объекты.

Если **p** и **q** указывают на элементы *одного и того же* массива, то к ним можно применять операторы отношения **==**, **!=**, **<**, **<=**, **>**, **>=**. Например, отношение вида **p<q** истинно, если **p** указывает на «более ранний» элемент массива,

чем **q**. Любой указатель всегда можно сравнить на равенство и неравенство с нулем.

Допускается также вычитание указателей. Например, если **p** и **q** ссылаются на элементы *одного и того же* массива и **p<q**, то **q-p+1** есть число элементов от **p** до **q** включительно.

Если до начала работы программы количество элементов в массиве неизвестно, то следует использовать *динамические массивы*. Память под них выделяется во *время выполнения* программы с помощью оператора **new** или функции **malloc**, а освобождается с помощью оператора **delete** или функции **free** соответственно, например:

```
#include <stdlib.h>
main
{   int n;
    cout<<"Введите количество элементов: ", cin>>n;
    int *a=new int[n];
    double *b=(double*)malloc(n*sizeof(double));
    // обработка массивов a и b
    ..............................
    delete []a;
    free(b);
    return 0;
}
```

A.11 Пример программы обработки динамических массивов

Рассмотрим пример работы с динамическими массивами. Ниже представлен текст программы, которая в целочисленном массиве, не все элементы которого одинаковы, заменяет набор элементов, расположенных между максимальным и минимальным элементами массива (максимальный и минимальный элементы не включаются в набор), на единственный элемент, равный сумме положительных элементов в заменяемом наборе. После этого выполняется сортировка полученного массива методом прямого обмена.

```
#include <conio.h>
#include <iostream.h>
main()
// пример программы обработки динамических массивов
{int n, // кол-во эл-тов в исходном массиве
      m; // кол-во эл-тов в результирующем массиве
int *a, // указатель на массив (исходный и результирующий)
 *temp_a; // указатель на временный (промежуточный) массив
 int i,j; // счетчики циклов
 int imax, // индекс макс. эл-та массива
     imin; // индекс мин. эл-та массива
 int ibeg, // индекс начала заменяемого набора эл-тов
     iend; // индекс конца заменяемого набора эл-тов
int s_pos; // сумма полож. эл-тов в заменяемом наборе
int temp; // буфер для сортировки методом прямого обмена
    clrscr();
```

```
        // формирование исходного массива
  cout<<"Введите количество элементов массива: ", cin>>n;
     a=new int[n];
     cout<<"Введите "<<n<<" элемента(ов) массива: ";
     for(i=0;i<n;i++) cin>>a[i];
      cout<<"Исходный массив:"<<endl;
     for(i=0;i<n;i++) cout<<"a["<<i<<"]="<<a[i]<<" ";
     cout<<endl;
    // поиск индексов макс. и мин. эл-тов массива
     for(imax=imin=0,i=1;i<n;i++)
         {  if(a[i]>a[imax]) imax=i;
            if(a[i]<a[imin]) imin=i;
         }
    cout<<"Максимальный элемент:a["  <<  imax  <<  "]="  <<
    a[imax] << endl <<"Минимальный элемент: a[" << imin << "]="
    << a[imin] << endl;
// поиск индексов начала и конца заменяемого набора эл-тов
    if(imax<imin) ibeg=imax+1, iend=imin-1;
    else ibeg=imin+1, iend=imax-1;
    cout<<"Заменяемый набор элементов:"<<endl;
     for(i=ibeg;i<=iend;i++) cout<<"a["<<i<<"]="<<a[i]<<" ";
        cout<<endl;
    // расчет суммы полож. эл-тов заменяемого набора
        for(i=ibeg,s_pos=0;i<=iend;i++)
        if(a[i]>0) s_pos+=a[i];
          cout<<"Сумма  положительных  элементов  заменяемого
    набора: "<<s_pos<<endl;
        temp_a=a; // адрес исходного массива
        m=n+ibeg-iend; // кол-во эл-тов в рез. массиве
        a=new int[m]; // результирующий массив
    /*---- формирование результирующего массива -------*/
    // запись эл-тов, расположенных до начала
    // заменяемого набора
        for(i=0,j=0;i<ibeg;i++,j++) a[j]=temp_a[i];
        // запись суммы полож. эл-тов заменяемого набора
        a[j++]=s_pos;
        // запись эл-тов, расположенных после
         //конца заменяемого набора
        for(i=iend+1;i<n;i++,j++) a[j]=temp_a[i];
        // освобождение памяти из-под исходного массива
        delete []temp_a;

        // вывод на экран результирующего массива
        cout<<"Результирующий массив:"<<endl;
        for(i=0;i<m;i++) cout<<"a["<<i<<"]="<<a[i]<<" ";
        cout<<endl;
        // сортировка массива методом прямого обмена
        for(i=m-1;i;i--)
```

```
            for(j=0;j<i;j++)
                if(a[j]>a[j+1])
                    temp=a[j],a[j]=a[j+1],a[j+1]=temp;
        // вывод на экран отсортированного массива
        cout<<"Отсортированный массив:"<<endl;
        for(i=0;i<m;i++) cout<<"a["<<i<<"]="<<a[i]<<" ";
        cout<<endl;
        cout<<"Нажмите любую клавишу..."; getch();
        delete []a; // освобождение памяти
        return 0;
}
```

Исходный целочисленный массив формируется динамически, т. е. во время выполнения программы; при этом используется значение размера массива, введенное пользователем. После того, как введены элементы массива, происходит поиск индексов максимального и минимального элементов **imax** и **imin** соответственно.

Так как порядок расположения элементов в массиве заранее не известен, то сначала может следовать как максимальный (**imax<imin**), так и минимальный элемент (**imin<imax**). С учетом этого обстоятельства осуществляется поиск индексов начала и конца заменяемого набора **ibeg** и **iend** соответственно. Далее производится расчет суммы положительных элементов заменяемого набора, т. е. тех элементов исходного массива, индексы которых лежат в диапазоне от **ibeg** до **iend** включительно.

Затем динамически происходит создание результирующего массива. При этом адрес исходного массива запоминается, чтобы после того, как формирование результирующего массива будет окончено, освободить память, которую занимает исходный массив. Для вычисления размера результирующего массива необходимо из размера исходного массива (**n**) вычесть количество элементов в заменяемом наборе (**iend-ibeg+1**) и добавить единицу, соответствующую элементу, заменяющему набор:

n-(iend-ibeg+1)+1==n+ibeg-iend

Далее происходит формирование результирующего массива. Следует обратить внимание, что при копировании элементов из исходного массива в результирующий массив используются *разные* счетчики цикла, так как копируемым элементам соответствуют *различные* индексы в соответствующих массивах. После того как результирующий массив сформирован, память из-под исходного массива освобождается. В конце результирующий массив сортируется методом прямого обмена.

Заказ № _____ от «_____» _____2016г. Тираж _____экз.
Изд-во СевГУ