Министерство образования и науки, молодежи и спорта Украины
Севастопольский национальный технический университет

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

**к лабораторному практикуму
по дисциплине
«Вычислительная техника и программирование»**

для студентов дневной и заочной форм обучения
направления 6.050901 — «Радиотехника»

Часть 2

Севастополь
2013

УДК 004.43+621.37 (076.5)

Методические указания к лабораторному практикуму по дисциплине «Вычислительная техника и программирование» для студентов дневной и заочной форм обучения направления 6.050901 — «Радиотехника» : в 4 ч. / СевНТУ ; сост. С.Н. Бердышев. — Севастополь : Изд-во СевНТУ, 2013. — Ч.2. — 48 с.

Целью методических указаний является оказание помощи студентам дневной и заочной форм обучения направления 6.050901 — «Радиотехника» при выполнении лабораторных работ по дисциплине «Вычислительная техника и программирование».

Рассмотрены и утверждены на заседании кафедры радиотехники и телекоммуникаций СевНТУ (протокол № 3 от 16 ноября 2012 г.).

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент:
Савочкин А.А., канд. техн. наук, доцент кафедры радиотехники и телекоммуникаций.

Ответственный за выпуск:
Гимпилевич Ю.Б., доктор техн. наук, проф., зав. кафедрой радиотехники и телекоммуникаций.

СОДЕРЖАНИЕ

Введение 4
1. Лабораторная работа №4
Обработка одномерных массивов 6
1.1. Цель работы 6
1.2. Варианты заданий 6
1.3. Теоретические сведения 10
1.4. Порядок выполнения работы 17
1.5. Оформление отчета 17
1.6. Контрольные вопросы 18
2. Лабораторная работа №5
Обработка двумерных массивов 19
2.1. Цель работы 19
2.2. Варианты заданий 19
2.3. Теоретические сведения 23
2.4. Порядок выполнения работы 32
2.5. Оформление отчета 32
2.6. Контрольные вопросы 33
3. Лабораторная работа №6
Пользовательские типы данных 34
3.1. Цель работы 34
3.2. Варианты заданий 34
3.3. Теоретические сведения 35
3.4. Порядок выполнения работы 43
3.5. Оформление отчета 43
3.6. Контрольные вопросы 44
Библиографический список 45
Приложение А
Пример оформления титульного листа отчета
по лабораторной работе 46

ВВЕДЕНИЕ

Дисциплина «Вычислительная техника и программирование» изучается студентами дневной формы обучения направления 6.050901 «Радиотехника» в первом и втором семестрах, студентами заочной формы обучения — в третьем или шестом семестрах.

При изучении первой части дисциплины студенты выполняют и защищают цикл из шести лабораторных работ, которые направлены на получение практических навыков программирования на языке *C*/*C*++ и изучение таких базовых структур как алгоритмы линейной, разветвляющейся и циклической структур, алгоритмы обработки одномерных и двумерных массивов. В данных методических указаниях представлены лабораторные работы №4 — №6 первой части дисциплины. Лабораторные работы №1 — №3 представлены в методических указаниях [1]. Итоговой формой контроля в первом семестре является зачет.

Вторая часть дисциплины предусматривает выполнение и защиту лабораторных работ, направленных на углубление и закрепление знаний, полученных при изучении первой части. При выполнении лабораторного практикума второй части дисциплины рассматриваются вопросы программной реализации алгоритмов работы со структурами данных, строками и файлами, а также использование графического режима. Итоговой формой контроля во втором семестре является экзамен.

После изучения дисциплины «Вычислительная техника и программирование» студент должен знать: основные понятия вычислительной техники и программирования, архитектурные особенности различных типов ЭВМ, способы разработки алгоритмов при решении технических задач, порядок работы на персональном компьютере с одной из существующих операционных систем, методику работы с программными пакетами, необходимыми в профессиональной деятельности.

В результате изучения данной дисциплины студент должен уметь: выбирать методы необходимые для решения поставленной задачи, составлять алгоритмы, разрабатывать и тестировать программы на языке *C*/*C*++, оформлять техническую документацию на программу в соответствии с действующими стандартами, читать специальную техническую литературу, осваивать новые языки программирования и типы программных комплексов.

Выполнение лабораторных работ осуществляется фронтальным методом каждым студентом индивидуально в соответствии с вариантом задания, который выдается преподавателем.

На выполнение лабораторных работ №1 — №4 и №6 отводится по четыре часа, а на выполнение работы №5 — шесть часов. Выполняются лабораторные работы в два этапа. На первом этапе студенты самостоятельно должны:

— проработать по конспекту лекций и рекомендованной литературе, приведенной в библиографическом списке, основные теоретические положения лабораторной работы и подготовить ответы на контрольные вопросы;

— разработать алгоритм решения задания, составить его структурную схему, и описать его;

— написать программу на языке *C*/*C*++ и описать её;

— оформить результаты выполнения первого этапа в виде черновика отчета по лабораторной работе.

На втором этапе, выполняемом в лаборатории университета, студенты должны:

— представить результаты выполнения первого этапа преподавателю;

— отладить и выполнить написанную программу;

— распечатать полученные результаты;

— оформить чистовик отчета по лабораторной работе;

— защитить отчет по лабораторной работе.

Выполнять и защищать лабораторные работы студенты должны строго по графику, в соответствии с расписанием учебных занятий.

В отчет должны включаться: титульный лист, цель работы, текст задания, теоретические сведения и расчеты, схема алгоритма с описанием, текст программы с комментариями и результаты выполнения программы. Содержание отчета перечислено в методических указаниях к каждой лабораторной работе.

На титульном листе обязательно требуется указать фамилию, имя и отчество полностью, группу, номер варианта, название и номер работы. Образец оформления титульного листа приведен в приложении А.

Отчет по лабораторной работе оформляется каждым студентом индивидуально на стандартных листах формата А4 с использованием персонального компьютера (ПК) и печатается с помощью принтера. Тексты разработанных программ и результаты программных расчетов следует приводить только в распечатанном виде. По краям листа должны быть оставлены поля: левое — 25 мм; верхнее и нижнее — 20 мм; правое — 15 мм.

Отчет следует оформлять в соответствии с методическими указаниями по оформлению текстовых работ [2].

Схемы и графики следует печатать на стандартных листах белой бумаги. Рисунки и таблицы должны быть пронумерованы и сопровождаться подписями и пояснениями. Схемы алгоритмов необходимо выполнять в соответствии с требованиями единой системы программной документации (ЕСПД) и ГОСТ 19.701-90 [3]. Правила оформления схем алгоритмов изложены в методических указаниях [1].

При выполнении лабораторных работ и подготовке к защите рекомендуется использовать учебную и справочную литературу [4 — 14].

1. ЛАБОРАТОРНАЯ РАБОТА №4
ОБРАБОТКА ОДНОМЕРНЫХ МАССИВОВ

1.1. Цель работы

Изучение особенностей представления и средств обработки одномерных массивов в языках *C*/*C*++ с учетом связи указателей и массивов.

Получение практических навыков реализации алгоритмов обработки одномерных массивов средствами языков *C*/*C*++.

Исследование особенностей обработки одномерных динамических массивов.

1.2. Варианты заданий

Разработать программу, которая обеспечивает ввод с клавиатуры исходных данных, выполняет их обработку в соответствии с вариантом задания и выводит результаты обработки на экран. Варианты задания выбираются по указанию преподавателя.

Вариант 1

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Найти сумму всех членов последовательности a_0, a_1, a_2, …, a_{n-1}, расположенных после первого отрицательного члена. Найти максимальный член последовательности, кратный 3.

Вариант 2

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество четных и нечетных членов последовательности a_0, a_1, a_2, …, a_{n-1}. Найти значение наименьшего четного члена последовательности.

Вариант 3

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество нечетных отрицательных членов последовательности a_0, a_1, a_2, …, a_{n-1}. Вычислить разницу наибольшего и наименьшего членов последовательности.

Вариант 4

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество четных членов последовательности a_0, a_1, a_2, …, a_{n-1}. Определить значение наибольшего четного члена этой последовательности.

Вариант 5

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить, имеются ли в последовательности a_0, a_1, a_2, …, a_{n-1} три положительных члена, идущих подряд. Среди членов указанной последовательности найти наибольший член.

Вариант 6

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить сумму отрицательных членов последовательности a_0, a_1, a_2, …, a_{n-1}. Вычислить произведение членов последовательности, которые по значению больше -1, но меньше 3.

Вариант 7

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество нечетных членов последовательности a_0, a_1, a_2, …, a_{n-1}. Определить значение наименьшего нечетного члена этой последовательности.

Вариант 8

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество отрицательных членов последовательности a_0, a_1, a_2, …, a_{n-1}, кратных 3. Найти значение наименьшего по модулю отрицательного члена последовательности.

Вариант 9

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить произведение положительных членов последовательности a_0, a_1, a_2, …, a_{n-1}, выбрать наибольшее отрицательное значение среди членов последовательности.

Вариант 10

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество членов последовательности a_0, a_1, a_2, …, a_{n-1}, расположенных между наибольшим и наименьшим членами последовательности.

Вариант 11

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество отрицательных членов последовательности a_0, a_1, a_2, …, a_{n-1}, а также вычислить сумму положительных членов последовательности.

Вариант 12

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Выбрать члены последовательности a_0, a_1, a_2, …, a_{n-1}, значение синуса которых отрицательно, а также найти наименьшее значение среди членов последовательности.

Вариант 13

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить произведение нечетных членов последовательности a_0, a_1, a_2, …, a_{n-1}. Найти член последовательности, имеющий наименьшее нечетное значение.

Вариант 14

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить сумму модулей членов последовательности a_0, a_1, a_2, …, a_{n-1}, а также вычислить произведение элементов стоящих в нечетных позициях.

Вариант 15

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить произведение ненулевых членов последовательности a_0, a_1, a_2, …, a_{n-1}, а также вычислить сумму элементов, стоящих в четных позициях.

Вариант 16

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество положительных четных членов последовательности a_0, a_1, a_2, …, a_{n-1}, а также найти наибольший по модулю отрицательный член последовательности.

Вариант 17

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить сумму нечетных членов последовательности a_0, a_1, a_2, …, a_{n-1}, и определить наименьшее значение среди нечетных членов последовательности.

Вариант 18

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Найти наибольший по модулю член последовательности a_0, a_1, a_2, …, a_{n-1}, а также вычислить произведение отрицательных членов последовательности.

Вариант 19

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить сумму положительных четных членов последовательности a_0, a_1, a_2, …, a_{n-1}. Найти значение наименьшего четного члена этой последовательности.

Вариант 20

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество членов последовательности a_0, a_1, a_2, …, a_{n-1}, кратных 3. Вычислить произведение наибольшего и наименьшего членов последовательности.

Вариант 21

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество членов последовательности a_0, a_1, a_2, …, a_{n-1}, модуль которых меньше 3. Вычислить сумму положительных членов последовательности.

Вариант 22

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить сумму членов последовательности a_0, a_1, a_2, …, a_{n-1}, для которых значение косинуса положительно. Найти наименьший член указанной последовательности.

Вариант 23

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество положительных членов последовательности a_0, a_1, a_2, …, a_{n-1}, а также найти значение наименьшего по модулю отрицательного члена последовательности.

Вариант 24

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить, имеются ли в последовательности a_0, a_1, a_2, …, a_{n-1} три отрицательных члена, идущих подряд. Среди чисел a_0, a_1, a_2, …, a_{n-1} найти ближайшее к числу 10.

Вариант 25

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить произведение членов последовательности a_0, a_1, a_2, …, a_{n-1}, стоящих в четных позициях. Найти наименьший по модулю член последовательности.

Вариант 26

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество членов последовательности a_0, a_1, a_2, …, a_{n-1}, находящихся между наибольшим и наименьшим членами. Вычислить сумму наибольшего и наименьшего членов последовательности.

Вариант 27

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить, какой из членов последовательности a_0, a_1, a_2, …, a_{n-1} наиболее близок к нулю, а также найти наибольший член указанной последовательности.

Вариант 28

Даны натуральные числа n, a_0, a_1, a_2, …, a_{n-1}. Определить количество нечетных членов и вычислить сумму четных членов последовательности a_0, a_1, a_2, …, a_{n-1}.

Вариант 29

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Вычислить сумму отрицательных членов последовательности a_0, a_1, a_2, …, a_{n-1}, а также найти значение наименьшего из членов последовательности.

Вариант 30

Даны целые числа n, a_0, a_1, a_2, …, a_{n-1}. Определить наибольшее значение среди членов последовательности $\cos(a_0), \cos(a_1), \cos(a_2), \ldots, \cos(a_{n-1})$, а также вычислить сумму положительных членов последовательности.

1.3. Теоретические сведения

Для выполнения лабораторной работы необходимо изучить особенности представления и средства обработки одномерных массивов в языках *C*/*C*++ с учетом связи указателей и массивов.

1.3.1. 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. табл. 2.1 в лабораторной работе №2 [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 помо-

щью 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.

1.3.2. Указатели

Указатели — это переменные, которые содержат в качестве своих значений адреса памяти. Их используют чаще всего при работе с динамической памятью — областью свободной памяти, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти, называемым динамическими переменными, производится только через указатели. 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. Символ ***** 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 *C*.

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 свободной области. Оператор **delete** освобождает ранее занятую память, возвращая ее свободной области.

Наряду с операторами **new** и **delete** в языках *C*/*C++* существуют две функции для захвата/освобождения памяти под динамическую переменную: **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 динамической памяти 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 динамической памяти под динамическую переменную 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 *ql=new char;
    *ql='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**. Система не способна автоматически освобождать динамически выделенную память.

Неосвобождение памяти может остаться незамеченным в небольших программах, однако часто приводит к отказам в серьезных разработках и является плохим стилем программирования.

1.3.3. 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;
}
```

1.3.4. 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.

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

main()
// пример программы обработки динамических массивов
{
    int n,        // количество элементов в исходном массиве
        m;        // количество элементов в результирующем массиве
    int *a,       // массив (исходный и результирующий)
        *temp_a;  // временный (промежуточный) массив
    int i,j;      // счетчики циклов
    int imax,     // индекс максимального элемента массива
        imin;     // индекс минимального элемента массива
    int ibeg,     // индекс начала заменяемого набора элементов
        iend;     // индекс конца заменяемого набора элементов
    int s_pos;    // сумма полож. элементов в заменяемом наборе

    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. элементов 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 элементов
    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 положительных элементов 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 элементов в результирующем 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 положительных элементов 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;

    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.

1.4. Порядок выполнения работы

Вариант задания определяется преподавателем. Перед выполнением работы необходимо ознакомится с разделами 1.1 — 1.3 данных методических указаний. Индивидуальные задания для выполнения работы см. в разделе 1.2.

1.4.1. Внимательно изучить индивидуальное задание.

1.4.2. Разработать схему алгоритма решения задачи.

1.4.3. Написать программу, решающую поставленную задачу.

1.4.4. Выполнить тестирование и отладку написанной программы.

1.4.5. Получить результаты работы программы. Рекомендуется скопировать в виде изображения сообщения, выводимые программой, для последующего включения в отчет.

1.4.6. Подготовить отчет по работе.

1.5. Оформление отчета

1.5.1. Сформулировать цель работы.

1.5.2. Привести постановку задачи и текст индивидуального задания.

1.5.3. Привести краткие теоретические сведения об одномерных массивах и указателях.

1.5.4. Изобразить схему алгоритма решения задачи.

1.5.5. Привести текст программы. Программа обязательно должна содержать комментарии.

1.5.6. Составить таблицу, в которой указать имена всех переменных используемых в программах и их тип, а также кратко описать назначение переменных.

1.5.7. Привести результаты выполнения программы.

1.5.8. Сделать выводы по работе.

1.6. Контрольные вопросы

1.6.1. Как в языках *C*/*C*++ описываются одномерные массивы?

1.6.2. Каковы особенности инициализации массива при его объявлении?

1.6.3. Как объявить переменную типа указатель?

1.6.4. Охарактеризуйте оператор адресации **&** и оператор раскрытия ссылки *****.

1.6.5. Охарактеризуйте операторы **new** и **delete**.

1.6.6. Опишите функции **malloc** и **free**.

1.6.7. Перечислите наиболее распространенные ошибки, связанные с использованием указателей.

1.6.8. Охарактеризуйте связь между массивами и указателями.

1.6.9. Какие действия можно выполнять над указателями?

1.6.10. Что такое динамические массивы?

1.6.11. Опишите алгоритмы сортировки массивов методами прямого обмена, вставки и прямого выбора.

2. ЛАБОРАТОРНАЯ РАБОТА №5
ОБРАБОТКА ДВУМЕРНЫХ МАССИВОВ

2.1. Цель работы

Изучить основные принципы работы с двумерными массивами в языках *C*/*C*++.

Исследовать способы передачи параметров в функции.

2.2. 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 задания выбирается по указанию преподавателя.

Вариант 1

Дана целочисленная прямоугольная матрица. Определить количество строк, не содержащих ни одного нулевого элемента. Поменять местами нечетные и четные столбцы, например, первый столбец поменять местами со вторым столбцом, третий столбец — с четвертым, и т.д.

Вариант 2

Дана целочисленная прямоугольная матрица. Определить количество столбцов, не содержащих ни одного нулевого элемента. Поменять местами нечетные и четные строки, например, первую строку поменять местами со второй строкой, третью строку — с четвертой, и т.д.

Вариант 3

Дана целочисленная прямоугольная матрица. Определить количество столбцов, содержащих хотя бы один нулевой элемент. Поменять местами первую и последнюю строки.

Вариант 4

Дана целочисленная квадратная матрица. Определить произведения элементов в тех строках, которые не содержат отрицательных элементов. Поменять местами строки и столбцы матрицы.

Вариант 5

Дана целочисленная квадратная матрица. Определить суммы элементов в тех столбцах, которые не содержат отрицательных элементов. Поменять местами элементы главной и побочной диагоналей матрицы.

Вариант 6

Дана целочисленная прямоугольная матрица. Определить суммы элементов в тех строках, которые содержат хотя бы один отрицательный элемент. Поменять местами первый и последний столбцы матрицы.

Вариант 7

Для заданной целочисленной квадратной матрицы найти суммы элементов в тех строках, которые содержат хотя бы один нулевой элемент. Поменять местами четные столбцы и четные строки матрицы.

Вариант 8

Дана целочисленная квадратная матрица. Определить суммы модулей отрицательных элементов в столбцах заданной матрицы. Поменять местами нечетные строки и нечетные столбцы матрицы.

Вариант 9

В целочисленной квадратной матрице вычислить произведения элементов главной диагонали. Поменять местами четные строки и столбцы, например, вторую строку поменять со вторым столбцом, четвертую стоку — с четвертым столбцом, и т.д.

Вариант 10

В целочисленной квадратной матрице вычислить сумму элементов побочной диагонали. Поменять местами первый столбец и последнюю строку матрицы относительно общего элемента, т.е. чтобы общий элемент остался на месте.

Вариант 11

Дана целочисленная прямоугольная матрица. Определить количество положительных и количество отрицательных элементов в матрице. Поменять местами вторую и предпоследнюю строки матрицы.

Вариант 12

Дана целочисленная прямоугольная матрица. Вычислить суммы элементов в строках матрицы. Удалить из матрицы столбцы, содержащие хотя бы один нулевой элемент.

Вариант 13

Вычислить произведения элементов в столбцах целочисленной прямоугольной матрицы. Заменить все элементы, находящиеся ниже главной диагонали матрицы, на нули.

Вариант 14

Дана целочисленная квадратная матрица. Вычислить сумму элементов главной диагонали матрицы. Поменять местами нечетные строки и нечетные столбцы матрицы, например, первую строку поменять с первым столбцом, третью стоку — с третьим столбцом, и т.д.

Вариант 15

Дана целочисленная квадратная матрица. Определить количество строк, содержащих хотя бы один нулевой элемент. Поменять местами строки и столбцы матрицы.

Вариант 16

Определить количество элементов прямоугольной целочисленной матрицы, модуль которых больше 3. Поменять местами второй и предпоследний столбцы матрицы.

Вариант 17

Вычислить произведение побочной диагонали целочисленной квадратной матрицы. Заменить нулями все элементы матрицы, находящиеся выше главной диагонали матрицы.

Вариант 18

Дана целочисленная квадратная матрица. Определить количество строк, содержащих хотя бы один нулевой элемент. Поменять местами последнюю строку и первый столбец матрицы относительно общего элемента, таким образом, чтобы общий элемент остался на месте.

Вариант 19

Дана целочисленная квадратная матрица. Определить суммы элементов в тех строках, которые не содержат отрицательных элементов. Поменять местами последнюю строку и последний столбец матрицы.

Вариант 20

Дана целочисленная квадратная матрица. Вычислить сумму элементов в тех строках, которые содержат хотя бы один нулевой элемент. Поменять местами четные строки и четные столбцы матрицы, например, вторую строку поменять со вторым столбцом, четвертую стоку — с четвертым столбцом, и т.д.

Вариант 21

Дана целочисленная прямоугольная матрица. Вычислить произведения элементов столбцов матрицы. Выполнить зеркальную перестановку столбцов матрицы относительно вертикальной оси.

Вариант 22

Дана целочисленная прямоугольная матрица. Вычислить суммы элементов сток матрицы. Выполнить зеркальную перестановку строк матрицы относительно горизонтальной оси.

Вариант 23

Дана целочисленная квадратная матрица. Вычислить сумму модулей элементов главной диагонали. Поменять местами вторую строку и второй столбец матрицы.

Вариант 24

Дана целочисленная прямоугольная матрица. Вычислить суммы модулей элементов в столбцах матрицы. Поменять местами первую и последнюю строки матрицы.

Вариант 25

Дана целочисленная квадратная матрица. Вычислить произведения элементов в тех строках матрицы, которые не содержат положительных чисел. Заменить в матрице все положительные элементы матрицы значением –1.

Вариант 26

Дана целочисленная квадратная матрица. Определить количество строк, содержащих хотябы один нулевой элемент. Поменять местами элементы матрицы, расположенные симметрично относительно побочной диагонали.

Вариант 27

Дана целочисленная квадратная матрица. Вычислить суммы элементов в диагоналях, параллельных главной диагонали матрицы. Заменить все отрицательные элементы матрицы нулями.

Вариант 28

Дана целочисленная квадратная матрица. Вычислить суммы модулей элементов в столбцах матрицы. Поменять местами первую строку и первый столбец матрицы.

Вариант 29

Дана целочисленная прямоугольная матрица. Вычислить произведения модулей элементов в строках матрицы. Поменять местами первый и предпоследний столбцы матрицы.

Вариант 30

Дана целочисленная прямоугольная матрица. Вычислить произведения отрицательных элементов в столбцах матрицы. Поменять местами вторую и тре-

тью строки матрицы.

2.3. Теоретические сведения

2.3.1. Описание и обработка двумерных массивов

Двумерный массив в *C*/*C*++ представляется как массив, состоящий из одномерных 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 2.1).

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 2.1 — 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 с помощью значений, записанных в фигурных скобках через запятую, и называемых инициализаторами:

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

Напишем программу, которая для целочисленной матрицы, содержащей 10 строк и 20 столбцов, 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; // вычисление среднего значение
     cout<<"Среднее арифметическое равно: "<<s<<endl;

     return;
}
```

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 её контрольный вывод на экран. Для того чтобы элементы матрицы располагались один за другим, следу-

ет 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**.

2.3.2. 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:

```
int n;
const int m=5;
cin>>n;
int (*a)[m]=new int[n][m];        // строка 1
int **b=(int **) new int[n][m];   // строка 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]; // строка 1
for(int i=0;i<nrow;i++)  // строка 2
    a[i]=new int[ncol];  // строка 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**.

2.3.3. Функции

Функция — это группа операторов, вызываемая по имени и возвращающая в точку вызова предписанное значение. Формат простейшего заголовка функции записывается следующим образом

```
<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 её 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 поэтому 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 *C* 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**. Тип функции **void** используется, поскольку функция **swap** не возвращает никаких значений.

В языке *C++* 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); // указатель
const int & func(const int & number); // ссылка
```

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.

2.3.4. 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 её 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>   // подключение библиотеки стандартных
                        // объектов и операций с потоками
                        // ввода-вывода средствами языка C++
#include <iomanip.h>    // подключение библиотеки средств
                        // манипулирования потоками

// прототипы функций

// функция, суммирующая элементы строк
void SummaStrok(int **a,int m,int n,long int *v);
// функция, выполняющая вывод матрицы и суммы элементов в строках
void Vyvod(int **a,int m,int n);
```

```
// функция, выполняющая сортировку строк
void Sort(int **a,int m,int n,long int *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];
    long int *v=new long int[m]; // вспомогательный массив
    SummaStrok(a,m,n,v);           // суммирование элементов строк
    Vyvod(a,m,n);                  // контрольная печать
    Sort(a,m,n,v);                 // сортировка
    Vyvod(a,m,n);                  // вывод результатов
    delete []v;
    delete []a;
    return 0;
}

void SummaStrok(int **a,int m,int n,long int *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){
    int i,j;
    for(i=0;i<m;i++){
        for(j=0;j<n;j++)
            cout<<setw(4)<<a[i][j]<<" ";
        cout<<endl;
    }
}

// сортировка строк матрицы методом прямого выбора
void Sort(int **a,int m,int n,long int *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).

2.4. Порядок выполнения работы

Вариант задания выбирается по указанию преподавателя. Перед выполнением работы необходимо ознакомится с разделами 2.1 — 2.3 данных методических указаний. Индивидуальные задания для выполнения работы см. в разделе 2.2.

2.4.1. Внимательно изучить индивидуальное задание.

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

2.4.3. Разработать алгоритм решения задачи в соответствии с вариантом.

2.4.4. Составить программу, оформив ввод данных, вывод результатов и обработку матрицы в виде функций.

2.4.5. Выполнить тестирование и отладку программы. Также проверить работу программы, когда массив состоит из одной или двух строк (столбцов).

2.4.6. Получить результаты работы программы. Рекомендуется скопировать в виде изображения сообщения, выводимые программой, для последующего включения в отчет.

2.4.7. Подготовить отчет по работе.

2.5. Оформление отчета

2.5.1. Сформулировать цель работы.

2.5.2. Привести постановку задачи и текст индивидуального задания.

2.5.3. Привести краткие теоретические сведения о двумерных массивах и функциях.

2.5.4. Изобразить схему алгоритма решения задачи.

2.5.5. Привести текст программы. Программа обязательно должна содержать комментарии.

2.5.6. Составить таблицу, в которой указать имена всех переменных используемых в программах и их тип, а также кратко описать назначение переменных.

2.5.7. Привести результаты работы программы и провести анализ работы программы.

2.5.8. Сделать выводы по работе.

2.6. Контрольные вопросы

2.6.1. Как в языках *C*/*C*++ описываются двумерные массивы?

2.6.2. Как осуществляется доступ к элементам двумерного массива?

2.6.3. Каким образом осуществляется инициализация двумерных массивов?

2.6.4. Каковы особенности работы с динамическими двумерными массивами в языках *C*/*C* ++?

2.6.5. Приведите формат заголовка функции.

2.6.6. Что является телом функции?

2.6.7. В чем состоит механизм передачи параметров в функцию?

2.6.8. Что такое ссылочные переменные? В каких случаях их целесообразно использовать?

2.6.9. Что такое аргументы по умолчанию?

2.6.10. Как осуществляется передача массивов в функции?

3. ЛАБОРАТОРНАЯ РАБОТА №6
ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ

3.1. Цель работы

Изучить основные принципы работы с пользовательскими типами данных в языке *C*/*C*++.

Сформировать практические навыки работы со структурами данных.

3.2. Варианты заданий

Составить схему алгоритма и написать на языке *C* программу вычисления комплексного значения функции $f(z)$ и вывести на экран действительную и мнимую части значения функции, а также модуль и аргумент функции. Варианты функций выбирать по указанию преподавателя из таблицы 3.1. Значения комплексного аргумента z выбираются произвольно и вводятся с клавиатуры. Результаты вычислений выводятся на дисплей в формате с плавающей точкой.

Таблица 3.1 — Исходные данные к лабораторной работе № 6

Вариант	Функция $f(z)$	Вариант	Функция $f(z)$
1	$f(z)=\frac{4,4}{z^2}+\frac{\mathrm{i}+z}{(z+1,4\mathrm{i})^2}$	10	$f(z)=\frac{1,01(5+1,3z)}{\mathrm{i}\,z(z+5)^2}$
2	$f(z)=\frac{10\mathrm{i}}{\left(z^2+10\right)(z+10\mathrm{i})^2}$	11	$f(z)=\frac{5}{(z+\mathrm{i})(z+1,5)}-\frac{\mathrm{i}}{z^2}$
3	$f(z)=\frac{1+z}{z(z+5)}-\frac{\mathrm{i}\,z}{(z+5,5)^2}$	12	$f(z)=\frac{1,5\mathrm{i}(7,2-z)}{z(z+1,5)^2}$
4	$f(z)=\frac{10,3}{z(z+20,5)(z+10)}$	13	$f(z)=\frac{1+z}{z(z+1,5\mathrm{i})}+\frac{1}{3,2\mathrm{i}\,z}$
5	$f(z)=\frac{1,7}{\mathrm{i}(z+3)}+\frac{2,3}{z^2(z+\mathrm{i})}$	14	$f(z)=\frac{2,3(10+1,2\mathrm{i}\,z)}{z(0,1z-2,3)}$
6	$f(z)=\frac{(6+z)(11-z)}{z(0,5z+1)^3}$	15	$f(z)=\frac{1,05z}{\mathrm{i}\left(2\mathrm{i}+z^2\right)}+\frac{1}{1,1z}$
7	$f(z)=\frac{10\mathrm{i}}{(z+10)^2}+\frac{10}{(z+10\mathrm{i})}$	16	$f(z)=\frac{6,5\mathrm{i}}{(0,2z+1)(z+\mathrm{i})^2}$
8	$f(z)=\frac{(1,2\mathrm{i}-z)(1+0,1z)}{z^2}$	17	$f(z)=(3,3\mathrm{i}+2,5z)-\frac{5,3\mathrm{i}}{(2,8z+1,6)^3}$
9	$f(z)=\frac{3,5}{7+z\mathrm{i}}+\frac{5\mathrm{i}}{z^2}$	18	$f(z)=\frac{5,4(1+z)}{z(z+5,8)(0,1z+i)}$

Продолжение таблицы 3.1

Вариант	Функция $f(z)$	Вариант	Функция $f(z)$
19	$f(z)=\dfrac{17{,}3\mathrm{i}}{z(z+12{,}3\mathrm{i})}+\dfrac{1}{z^2}$	25	$f(z)=\dfrac{1}{\mathrm{i}\,z}-\dfrac{15}{z(z+1{,}2\mathrm{i})}$
20	$f(z)=\dfrac{6\mathrm{i}+1{,}2z}{z(1{,}5z+1)^2}$	26	$f(z)=\dfrac{10\mathrm{i}}{(0{,}1z+1)^2(z+2)}$
21	$f(z)=\dfrac{0{,}5i}{z^2}+\dfrac{1}{\mathrm{i}\,z(1{,}5z+1{,}2)}$	27	$f(z)=\dfrac{0{,}1\mathrm{i}}{z^2}+\dfrac{1}{1{,}2z(1+\mathrm{i}\,z)}$
22	$f(z)=\dfrac{1{,}9(2{,}2+z)}{z^2(0{,}5z+0{,}3)}$	28	$f(z)=\dfrac{2\mathrm{i}(1-z)}{z(z+6{,}5\mathrm{i})(0{,}1z+10)}$
23	$f(z)=\dfrac{1}{1{,}2z}+\dfrac{z}{\mathrm{i}(5{,}5+z)^2}$	29	$f(z)=\dfrac{6{,}1}{z(z+1{,}3\mathrm{i})}+\dfrac{5{,}1}{2z+3}$
24	$f(z)=\dfrac{(7\mathrm{i}-z)(1+7z)}{z^3}$	30	$f(z)=\dfrac{5(\mathrm{i}+z)}{z(z+5\mathrm{i})^2}$

3.3. Теоретические сведения

Для представления информации сложной структуры используются типы данных, построенные на основе простых типов данных, массивов и указателей.

3.3.1. Переименование типов

Переименование типа используется при необходимости задать какому-либо стандартному или пользовательскому типу новое имя. Формат переименования типов имеет следующий вид:

```
typedef стандартный_тип новое_имя_типа [размерность];
```

Здесь **`стандартный_тип`** — один из стандартных типов языка *C*/*C*++; **`новое_имя_типа`** — пользовательское имя типа; **`размерность`** — указывает размер массива, если новый тип является массивом. Введенные таким образом новые имена типов можно использовать таким же образом, как и имена стандартных типов для более ясного представления программы.

3.3.2. Перечисления

При написании программ часто возникает необходимость определить несколько именованных констант, имеющих, как правило, различные значения. Для таких целей используют перечисляемый тип данных, все возможные значения которого задаются списком целочисленных констант. Формат перечисления:

```
enum [ имя_типа ] { список_констант };
```

Часть формата **имя_типа** задается в том случае, если в программе требуется определять переменные этого типа. При этом компилятор обеспечивает, чтобы эти переменные принимали значение только из списка констант. Целочисленные константы из списка могут инициализироваться обычным образом. При отсутсвии инициализатора первая константа обнуляется, а каждой последующей присваисается значение на единицу больше предыдущего:

```
enum Err_Msg{NO_ERR, READ_ERR, WRITE_ERR, UNKNOWN_ERR};
```

Здесь константам **NO_ERR**, **READ_ERR**, **WRITE_ERR** и **UNKNOWN_ERR** автоматически присваиваются значения 0, 1, 2 и 3, соответственно.

Ниже записан пример, содержащий инициализаторы:

```
enum Err_Msg{
TWO = 2,              // присваивается значение 2
THREE,                // присваивается значение 3
FOUR,                 // присваивается значение 4
TEN = 10,             // присваивается значение 10
ELEVEN,               // присваивается значение 11
FIFTY = TEN + 40      // присваивается значение 50
};
```

При выполнении арифметических операций перечисления преобразуются в целые типы.

3.3.3. Структуры

Структура может содержать элементы разных типов. В языке *C++* структура является видом класса и обладает всеми его свойствами. Описание структуры начинается с ключевого слова **struct** и содержит список элементов в фигурных скобках, которые называются полями структуры и могут иметь любой тип, кроме типа этой же структуры:

```
struct [ имя_типа ]
{
    тип_1 поле_1;
    тип_2 поле_2;
    ...
    тип_N поле_N;
} [ список_описателей ];
```

Если в определении отсутствует **имя_типа**, то обязательно должен присутствовать **список_описателей**. Если **список_описателей** отсутствует, то описание структуры определяет новый тип, имя которого (**имя_типа**) можно использовать наряду со стандартными типами.

Например, пусть требуется фиксировать сведения о проведении лекций. При этом для каждой лекции фиксируется время, тема, фамилия преподавателя и количество студентов. Для этого можно описать следующую структуру:

```
struct lecture          // объявление структуры
{
int hour, minutes;      // поля, обозначающие время (часы, минуты)
char theme[100];        // поле, обозначающее тему лекции
char name[30];          // поле, обозначающее фамилию лектора
int number;             // поле, обозначающее количество студентов
} introduction, programming[10];
```

Здесь **lecture** — это имя типа. Переменные структурного типа описываются либо одновременно с описанием структуры (см. выше), либо отдельно. Например:

```
lecture introduction,     // структура
        programming[10]; // и массив структур
```

При совместном описании структурного типа и переменных допускается тип структуры не указывать, если он нигде не используется.

Переменные структурного типа можно размещать и в динамической области памяти, для этого необходимо описать указатель на структуру:

```
lecture *p_intro=new lecture;     // указатель на структуру
lecture *p_prog=new lecture [m]; // указатель на массив структур
```

Для доступа к полям структуры используется операция выбора, обозначаемая точкой:

```
имя_структурной_переменной.имя_поля
```

Например:

```
introduction.hour=9;
introduction.minutes=45;
strcpy(introduction.theme,"Вводное занятие");
programming[1].hour=11;
programming[1].minutes=15;
strcpy(programming[1].theme,"Алгоритмы");
```

Если структуры размещены в динамической памяти, то доступ к полям выполняют через указатели:

```
(*p_intro).hour=9;
(*p_intro).minutes=30;
```

Скобки здесь необходимы, поскольку приоритет операции выбора "." выше, чем у операции раскрытия ссылки. Существует еще одна форма доступа к полям структуры при работе с указателями на структуру:

```
p_intro->number=75;
```

Если имеется указатель на массив структур, то доступ к ним выполняется также через операцию выбора:

```
programming[2].hour=11;            // (*(programming+2)).hour=15;
```

Структуры одного типа можно присваивать друг другу:

```
*p_intro=introduction;
programming[1]=introduction;
```

```
programming[2]=programming[0];
```

Ввод-вывод структур, как и массивов, выполняется поэлементно. Например, ввод-вывод структуры **introduction** с использованием классов языка *C++* записывается следующим образом:

```
cin >> introduction.hour >> introduction.minutes;
cin.getline(introduction.theme,100);
cout << introduction.hour << introduction.minutes
     << ' ' << introduction.theme << endl;
```

Этот же ввод-вывод можно выполнить и с помощью функций ввода-вывода языка *C*:

```
scanf("%d%d", &introduction.hour, &introduction.minutes);
gets(introduction.theme);
printf("%d %d %s", introduction.hour,
                   introduction.minutes,
                   introduction.theme);
```

При необходимости структуры можно инициализировать перечислением значений их полей:

```
lecture introduction ={11,15,"Системы счисления",45};
```

Разновидностью структуры являются битовые поля, которые используются для плотной упаковки данных, например, флажков типа «да/нет». При этом минимальный размер поля такой структуры имеет размер 1 бит, а минимальная адресуемая ячейка памяти — 1 байт. При описании битового поля после имени через двоеточие указывается длина поля в битах:

```
struct dialogue_win {
    bool         centerX : 1;
    bool         centerY : 1;
    unsigned int shadow  : 2;
    unsigned int palette : 4;
};
```

Битовые поля могут быть любого целого типа. Адрес битового поля получить нельзя. Доступ к битовому полю осуществляется таким же образом, как и к обычному полю структуры.

3.3.4. Объединения

Объединения представляют собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания такой же, как у структуры:

```
union [ имя_типа ]
{
    тип_1 поле_1;
    тип_2 поле_2;
    ...
    тип_N поле_N;
} [ список_описателей ];
```

Длина (размер в байтах) объединения равна наибольшей из длин его полей. В любой момент времени в объединеии хранится только одно значение (поле), правильность обращения к которому полностью контроллируется только программистом. Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно в программе не требуется:

```
union {
    char  card[125]; // поле для хранения информации об оплате
                     // платежной картой
    float cash;      // поле для ввода суммы полученной наличными
};
```

В записанном объединении может храниться либо данные о платежной карте и проведенной транзакции, либо сумма денег, полученная наличными средствами. Программа с таким вариантом объединения не предусматривает частичную оплату покупки картой, а частично наличными средствами.

3.3.5. Пример программы обработки структур

Рассмотрим пример программы, использующей структуры, который приведен ниже. Программа предназначена для вычисления модуля и аргумента коплексной величины описываемой следующим выражением:

$$y = \frac{(a+3)(a-\mathrm{i})}{(2a+3\mathrm{i})(a+2)}. \tag{3.1}$$

Для упрощения вычислений необходимо записать все числа выражения в комплексном виде:

$$3 = 3 + 0\mathrm{i}, \tag{3.2}$$
$$\mathrm{i} = 0 + 1\mathrm{i}, \tag{3.3}$$
$$2 = 2 + 0\mathrm{i}, \tag{3.4}$$
$$3\mathrm{i} = 0 + 3\mathrm{i}. \tag{3.5}$$

Затем следует переписать исходное выражение (3.1), подставляя в него числа в комплексном виде (3.2 — 3.5):

$$y = \frac{(a+(3+0\mathrm{i}))(a-(0+1\mathrm{i}))}{((2+0\mathrm{i})a+(0+3\mathrm{i}))(a+(2+0\mathrm{i}))}. \tag{3.6}$$

Полученная формула (3.6) используется в программе для вычисления комплексного значения функции.

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

#define PI 3.141592654

struct complex {
    double real;
    double image; };
```

```
// прототипы функций
// функция суммирования двух комплексных чисел: x+y
struct complex sum(struct complex x, struct complex y);
// функция нахождения разницы двух комплексных чисел: x-y
struct complex sub(struct complex x, struct complex y);
// функция нахождения произведения двух комплексных чисел: x*y
struct complex mul(struct complex x, struct complex y);
// функция нахождения произведения двух комплексных чисел: x/y
struct complex div(struct complex x, struct complex y);
// функция определения модуля комплексного числа: |x|
double abs(struct complex x);
// функция определения аргумента комплексного числа: arg(x)
double arg(struct complex x);

void main()
// пример программы обработки структур
{
    // вычисление функции:
    //       (a+3)(a-i)
    // y = ------------
    //     (2a+3i)(a+2)

    complex a,              // параметр функции
            y,              // результат вычисления функции
            numerator,      // числитель
            denominator,    // знаменатель
            temp_1, temp_2; // вспомогательные переменные


    clrscr ();

    cout << "Введите действительную часть комплексного числа: " ;
    cin >> a.real; cout << endl;

    cout << "Введите мнимую часть комплексного числа: " ;
    cin >> a.image; cout << endl;

    // вычисление числителя
    // (a+3)(a-i) = (a+(3+0i))(a-(0+1i))
    temp_1.real = 3; temp_1.image = 0;              // 3+0i
    temp_2.real = 0; temp_2.image = 1;              // 0+1i
    numerator = mul(sum(a, temp_1),sub(a, temp_2));

    // вычисление знаменателя
    // (2a+3i)(a+2) = ((2+0i)a + (0+3i)a)(a+(2+0i))
    temp_1.real = 2; temp_1.image = 0;              // 2+0i
    temp_2.real = 0; temp_2.image = 3;              // 0+3i
    denominator = mul( sum(mul(temp_1, a),mul(temp_2, a)),
                       sum(a, temp_1)                    );

    // проверка допустимого значения знаменателя
    if (denominator.real == 0 && denominator.image == 0)
```

```
    {
        cout << "Недопустимое значение в знаменателе,
                 программа завершена" << endl;
        return;
    }

    // вычисление функции
    y = div(numerator, denominator);

    // вывод на экран значения модуля функции
    cout << "Значение модуля функции: " << abs(y) << endl;

    // вывод на экран значения аргумента функции
    cout << "Значение аргумента функции: " << arg(y) << endl;

    getch(); return;
}

// функция суммирования двух комплексных чисел: x+y
struct complex sum(struct complex x, struct complex y)
{
    complex result;

    result.real = x.real + y.real;
    result.image = x.image + y.image;

    return result;
}

// функция нахождения разницы двух комплексных чисел: x-y
struct complex sub(struct complex x, struct complex y)
{
    complex result;

    result.real = x.real - y.real;
    result.image = x.image - y.image;

    return result;
}

// функция нахождения произведения двух комплексных чисел: x*y
struct complex mul(struct complex x, struct complex y)
{
    complex result;

    result.real = x.real*y.real - x.image*y.image;
    result.image = x.real*y.image + x.image*y.real;

    return result;
}

// функция нахождения произведения двух комплексных чисел: x/y
struct complex div(struct complex x, struct complex y)
```

```
{
    complex result;

    result.real  = (x.real*y.real + x.image*y.image)
                 / (y.real*y.real + y.image*y.image);
    result.image = (x.image*y.real - x.real*y.image)
                 / (y.real*y.real + y.image*y.image);

    return result;
}

// функция определения модуля комплексного числа: |x|
double abs(struct complex x)
{
    double result;

    result = sqrt(x.real*x.real + x.image*x.image);

    return result;
}

// функция определения аргумента комплексного числа: arg(x)
double arg(struct complex x)
{
    double result;

    result = atan2(x.image, x.real);

    // перевод в градусы
    result = result / PI * 180;

    return result;
}
```

Комплексные числа в программе представлены в виде структур, содержащих два поля, предназначенные для хранения действительной и мнимой частей комплексного числа:

```
struct complex {
    double real;      // действительная часть числа
    double image;     // мнимая часть числа
};
```

В приведенном объявлении **complex** является новым типом в программе, который можно использовать для обявления переменных:

```
complex a,
        y,
        numerator,
        denominator,
        temp_1, temp_2;
```

```
complex result;
```

Функции, объявленные в программе, в качестве аргументов используют объявленные структуры и могут возвращать в точку вызова не только простые стандартные типы, но и более сложные типы и структуры:

```
struct complex sum(struct complex x, struct complex y);
```

Запись объявленного типа структуры **`struct complex`** перед именем функции означает, что функция передает в точку вызова структуру данных, а перед именами аргументов, что они являются структурами этого типа.

Доступ к полям структур в программе осуществляется через операцию выбора, например, **`x.real`**, **`y.image`**.

3.4. Порядок выполнения работы

Вариант задания выдается преподавателем. Перед выполнением работы необходимо ознакомиться с разделами 3.1 — 3.3 данных методических указаний. Индивидуальные задания для выполнения работы см. в разделе 3.2.

3.4.1. Внимательно изучить индивидуальное задание.

3.4.2. Выбрать четыре комплексных значения аргумента функции таким образом, чтобы каждый из них лежал в отдельном квадранте комплексной плоскости.

3.4.3. Вычислить комплексное значение функции, действительную и мнимую части значения функции, а также модуль и аргумент функции для каждого из выбранных комплексных аргументов. Для вычислений рекомендуется использовать инженерно-математическую программу *MathCAD* [14].

3.4.4. Разработать схему алгоритма решения задачи.

3.4.5. Написать программу вычисления комплексного значения функции, действительной и мнимой частей значения функции, а также модуля и аргумента функции. Математические действия над комплексными числами оформить в виже функций. Комплексные числа представить в виде структур, содежащих поля действительной и мнимой частей.

3.4.6. Выполнить тестирование и отладку написанной программы.

3.4.7. Получить результаты работы программы для четырех значений комплексного аргумента, выбранных ранее.

Рекомендуется скопировать в виде изображения сообщения, выводимые программой, для последующего включения в отчет.

3.4.8. Подготовить отчет по работе.

3.5. Оформление отчета

3.5.1. Сформулировать цель работы.

3.5.2. Привести постановку задачи и текст индивидульного задания.

3.5.3. Привести краткие теоретические сведения, касающиеся использова-

ния в программе структур данных **`struct`**.

3.5.4. Привести математическое обоснование задачи:

— записать выбранные четыре значения аргумента;

— записать рассчитанные комплексные значения функции для каждого из комплексных аргументов;

— записать действительные и мнимые части значения функции для каждого из комплексных аргументов;

— записать значения модуля и аргумента комплексного значения функции для каждого из аргументов.

3.5.5. Изобразить схему алгоритма решения задачи. Записать краткие пояснения к алгоритму.

3.5.6. Привести текст программы. Текст программы обязательно должен содержать комментарии.

3.5.7. Составить таблицу, в которой указать имена всех переменных и структур, используемых в программе и их тип, а также кратко описать назначение переменных.

3.5.8. Привести результаты работы программы для четырех коплексных аргументов.

3.5.9. Сделать выводы по работе.

3.6. Контрольные вопросы

3.6.1. Что называют структурой в языке *C*/*C*++?

3.6.2. Приведите пример объявления и использования структурного типа.

3.6.3. Как осуществляется доступ к полям структурных переменных?

3.6.4. Как осуществляется инициализация структур?

3.6.5. Что называют битовыми полями в в языке *C*/*C*++?

3.6.6. Приведите пример объявления и использования битовых полей.

3.6.7. Для чего используются битовые поля?

3.6.8. Что называют перечислением в языке *C*/*C*++?

3.6.9. Приведите пример объявления и использования перечислений.

3.6.10. Что называют объединением в языке *C*/*C*++?

3.6.11. Приведите пример объявления и использования объединений.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Методические указания к лабораторному практикуму по дисциплине «Вычислительная техника и программирование» для студентов дневной и заочной форм обучения направления 6.050901 — «Радиотехника» : в 4 ч. / СевНТУ ; сост. С.Н. Бердышев. — Севастополь : Изд-во СевНТУ, 2013. — Ч.1. — 60 с.

2. Методические указания по оформлению текстовых работ для студентов дневной и заочной форм обучения направления 6.050901 — «Радиотехника» / СевНТУ ; сост. В.Г. Слезкин. — Севастополь : Изд-во СевНТУ, 2010. — 20 с.

3. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. — Взамен ГОСТ 19.002-80, ГОСТ 19.003-80 ; Введ. 01.01.1992. — М. : Изд-во стандартов, 1991. — 28 с.

4. Бердышев С.Н. Учебное пособие по дисциплине «Вычислительная техника и программирование» для студентов дневной и заочной форм обучения направления 6.050901 — «Радиотехника» / СевНТУ ; сост. С.Н. Бердышев, Е.А. Редькина. — Севастополь : Изд-во СевНТУ, 2013. — 170 с.

5. Савочкин А.А. Учебное пособие по дисциплине «Информатика» для студентов заочной формы обучения специальности 7.090701 — Радиотехника «Программирование на языке СИ» / СевНТУ ; сост. А.А. Савочкин. — Севастополь : Изд-во СевНТУ, 2002. — 62 с.

6. Павловская Т.А. *C*/*C*++. Программирование на языке высокого уровня / Т.А. Павловская. — СПб.: Питер, 2010. — 461 с.

7. Павловская Т.А. *C*/*C*++. Структурное программирование / Т.А. Павловская, Ю.А. Щупак. — СПб.: Питер, 2003. — 240 с.

8. Глушаков СВ. Язык программирования *C*++ / С.В. Глушаков, А.В. Коваль, С.В. Смирнов. — Харьков: Фолио, 2002. — 500 с.

9. Дейтел Х. Как программировать на *C*++: [пер. с англ.] / Х. Дейтел, П. Дейтел. — М.: Изд-во БИНОМ, 2000. — 1024 с.

10. Ковалюк Т.В. Основи програмування / Т.В. Ковалюк. — К.: Видавнича група BHV, 2005. — 384 с.

11. Франка П. *C*++: учебный курс / П. Франка. — СПб. : Питер, 2006. — 522 с.

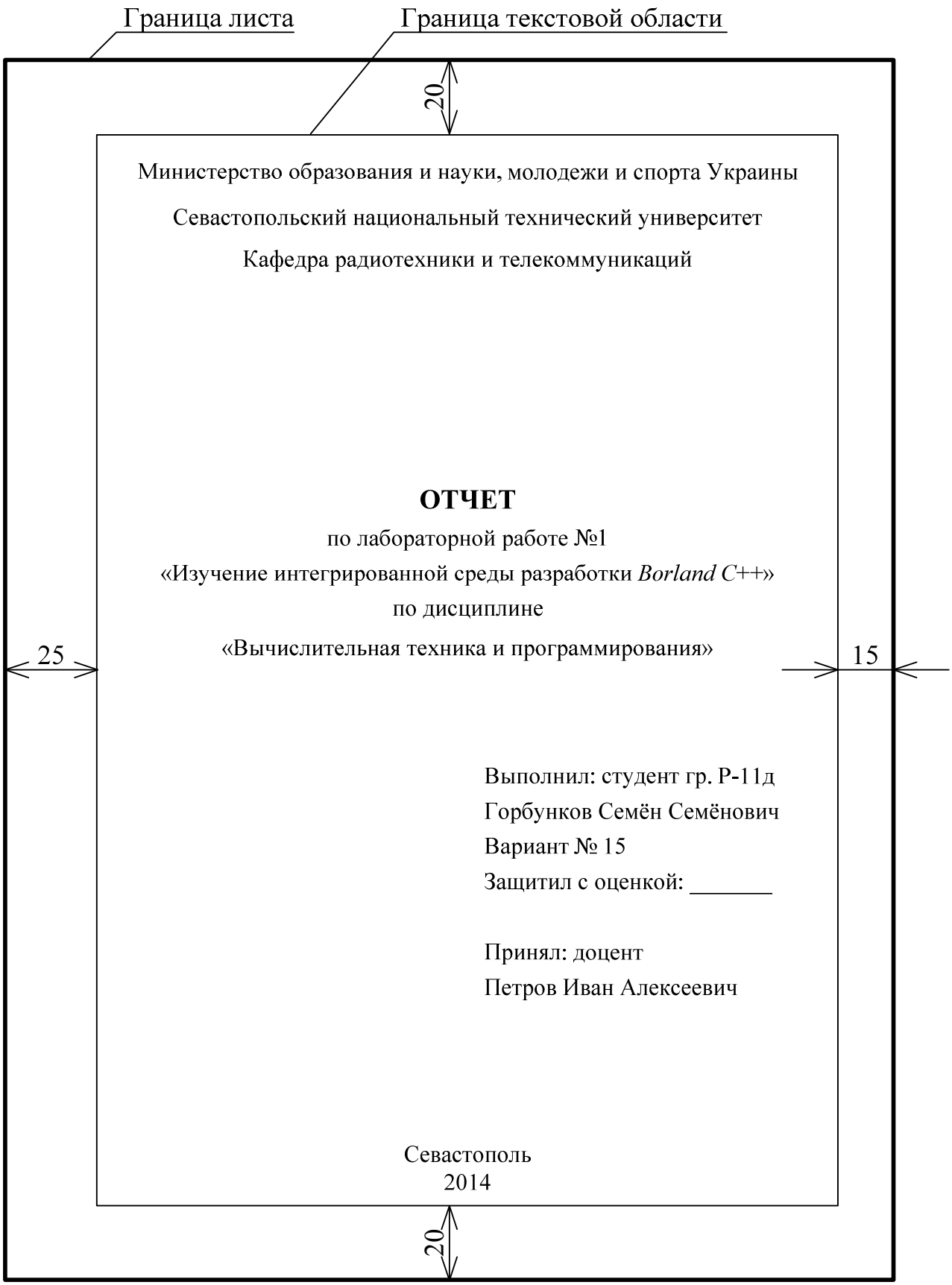
12. Громов Ю.Ю. Программирование на языке СИ: учеб. пособие / Ю.Ю. Громов, С.И. Татаренко. — Тамбов: Изд-во ТГТУ, 1995. — 169 с.

13. Вирченко Н.А. Графики функций: справочник / Н.А. Вирченко, И.И. Ляшко, К.И. Швецов. — Киев : Наук. думка, 1979. — 320 с.

14. Руководство пользователя *MathCAD* / Корпорация «*Parametric Technology Corporation*». — Нидем (США) : *PTC*, 2011. — 190 с.

ПРИЛОЖЕНИЕ А

ПРИМЕР ОФОРМЛЕНИЯ ТИТУЛЬНОГО ЛИСТА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ



Заказ № _____ от «____»_____ 20___ г. Тираж _____ экз.

Изд-во СевНТУ