МИНОБРНАУКИ РОССИИ

Санкт – Петербургский государственный электротехнический
Университет «ЛЭТИ» им. В.И. Ульянова(Ленина)

С.А. ИВАНОВСКИЙ, А.А. ЛИСС, В.П. САМОЙЛЕНКО, О.М. ШОЛОХОВА

ПРАКТИКУМ ПО ПРОЦЕДУРНОМУ ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ С++

Учебное пособие

Санкт - Петербург
Издательство СПбГЭТУ «ЛЭТИ»

УДК
00000000
ББК
00000000
И00

Ивановский С.А., Лисс А.А., Самойленко В.П., Шолохова О.М
И00 Практикум по процедурному программированию на языке С++: учеб. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2016. 64 с.
ISBN 000-0-0000-0000-0

Содержит основные сведения о процедурном программировании на языке С++. Предназначено для студентов специальностей 01.03.02 и 09.03.04.

УДК 00000000
ББК 00000000

Рецензенты: кафедра теоретических основ радиотехники Северо-западного политехнического института; д-р техн. Наук А. Т. Сидоров (НИИ «Электроприбор»).

Утверждено редакционно-издательским советом университета в качестве учебного пособия

ВВЕДЕНИЕ

Практикум по процедурному программированию на языке C++ представляет собой методические указания к лабораторным работам по курсу «Программирование» для подготовки бакалавров по направлениям «Прикладная математика и информатика» и «Программная инженерия» и соответствует рабочей программе курса.

В практикуме содержатся формулировки заданий и требования к лабораторным работам. Каждый раздел сопровождается примерами программ с пояснениями и теоретическим материалом, который содержит необходимые сведения для выполнения соответствующей лабораторной работы. Предполагается, что при выполнении лабораторной работы студент использует лекционный материал и источники из списка рекомендованной литературы.

Понятия, обязательные для изучения и понимания, выделены **жирным** шрифтом, а понятия, которые будут раскрыты в курсе (или других курсах) позднее, - ***жирным курсивом***.

Все примеры написаны на языке программирования *C++* (стандарт *C++11/14*) и проверялись в среде *Microsoft Visual Studio Community 2015*.

ОБЩИЕ ТРЕБОВАНИЯ

Для защиты лабораторной работы студент должен:

1. Представить исходный код программы, который соответствует требованиям, определенным преподавателем;
2. Ответить на вопросы по программе (используемым языковым конструкциям) и составленному алгоритму;
3. Продемонстрировать работоспособность программы на заранее составленных показательных примерах;
4. Представить отчет, выполненный в соответствии с шаблоном [1]. Отчет должен содержать следующие разделы:
   - 4.1. Цель работы;
   - 4.2. Постановка задачи;
   - 4.3. Основные теоретические положения;
   - 4.4. Спецификация программы (назначение программы, ограничения на входные данные, пример диалога с пользователем, спецификации пользовательских функций, особенности реализации и т.д.);
   - 4.5. Тестирование;
   - 4.6. Выводы;
   - 4.7. Приложение А. Исходный код;
   - 4.8. Приложение Б. Блок-схема основного алгоритма программы (по требованию преподавателя).

1. Шаблон оформления отчета по лабораторной работе. // Сайт СПбГЭТУ «ЛЭТИ», раздел Нормативные документы. URL: http://www.eltech.ru/ru/universitet/svedeniya-ob-obrazovatelnoy-organizacii/obrazovanie/normativnye-dokumenty3 (дата обращения: 17.12.2015).

ЛАБОРАТОРНАЯ РАБОТА 1. СТРУКТУРА ПРОГРАММЫ НА ЯЗЫКЕ С++. ОПЕРАТОР ВЕТВЛЕНИЯ

1 Цель работы

Ознакомление со средой программирования С++; изучение операторов выбора языка С++.

2 Теоретический материал

2.1 Структура программы на языке С++

Программа на языке программирования С++ может иметь следующую структуру:

```
// Подключение заголовочных файлов библиотек
#include <имя заголовочного файла>
// Включение в область видимости объектов из пространств имен,
// используемых в подключаемых модулях
using namespace <имя пространства имен>;
// Определение функции main
int main()
{
      операторы
      return 0;
}
```

Выполнение программы всегда начинается с выполнения ***функции*** *main*, определение которой должно присутствовать в каждой программе. Например, программа, которая выводит на экран строку *«Hello, World!»*, может выглядеть следующим образом:

Программа 1.1. hello_world.cpp

```
#include <iostream> // заголовочный файл,
                    // в котором содержится объявление объекта cout
using namespace std; // пространство имен,
                     // в котором находится объект cout
int main()
{
      cout << "Hello, World!"; // вывод на экран строки “Hello, World!”
      return 0; // завершение функции main с кодом 0
}
```

До начала **компиляции** код обрабатывается **препроцессором**, который обрабатывает **директивы препроцессора** (начинаются с символа #). Директива *#include <iostream>* означает, что в файл *hello_world.cpp* вместо строки «*#include <iostream>*» будет добавлено содержимое заголовочного файла *iostream*. Файл *iostream* (io – input/output, ввод/вывод, stream - поток) является частью ***стандартной библиотеки C++*** и содержит необходимые элементы (***классы***, ***функции***, ***переменные***) для организации ***потокового ввода-вывода*** в *C++*. В программе 1.1. используется ***объект*** *cout*, объявленный в файле *iostream*, для вывода текста на экран.

Все элементы заголовочных файлов *стандартной библиотеки C++* без расширения .h находятся в ***пространстве имен*** *std*. Это означает, что для обращения к объекту *cout* надо указать пространство имен, в котором он объявлен *std::cout*. Чтобы не указывать префикс *std* каждый раз при обращении к объекту *cout* (и другим элементам *стандартной библиотеки C++*), можно воспользоваться директивой *using* (*using namespace std*).

В коде могут присутствовать пояснения, – **комментарии**, – которые не обрабатываются компилятором и служат для улучшения понимания кода при прочтении. Комментарии языка *C++* ограничены символами «//» и концом строки. Также можно использовать комментарии языка *C*, которые ограничены символами «/*» и «*/».

Тело функции *main* (как и любой другой функции) состоит из набора операторов. Оператор представляет собой законченную инструкцию языка *C++*. После оператора должна следовать точка с запятой. В Программе 1.1. использованы следующие операторы: *using namespace std*, *cout << "Hello, World!"* и *return 0*.

2.2 Переменные и идентификаторы

Для хранения данных в программе используются **переменные** (именованная область памяти) различных **типов**. По имени переменной можно считывать и изменять значение переменной (если эти операции не ***запрещены***). Тип переменной определяет количество бит, которое отводится для хранения информации, способ интерпретации бит, а также набор операций, которые можно выполнять с переменными данного типа.

Базовые типы *C++* (символьный тип *char*, логический тип *bool*, целочисленные типы *short*, *int*, *long* и *long long*, а также вещественные типы *float*, *double* и *long double*) и их свойства будут рассмотрены позднее.

Различным сущностям программы (переменным, функциям) присваиваются **идентификаторы**, по которым можно обращаться к этим сущностям. Идентификатор может состоять из латинских букв, цифр и знака подчеркивания и не должен начинаться с цифры. Прописные и строчные буквы считаются различными.

Рассмотрим небольшой пример с использованием переменных:

Программа 1.2.

```
#include <iostream> // cout, cin
using namespace std;
int main()
{
      int age; // объявление переменной age
      cout << "Enter your age: ";
      cin >> age;
      cout << "Your age is: " << age << std::endl;
      return 0;
}
```

В программе 1.2. **объявлена** переменная *age* типа *int*. С помощью объекта *cin* переменной *age* присваивается значение, которое введено пользователем с клавиатуры (*cin >> age*). После чего на экран выводится полученное значение (*cout << age*).

2.3 Управляющие операторы. Операторы выбора

Операторы определяют порядок выполнения действий в программе и выполняются последовательно. Однако существуют управляющие операторы, позволяющие изменять порядок выполнения других операторов. В *C++* есть два оператора выбора – оператор ветвления **if** и оператор множественного выбора ***switch***.

Оператор *if* используется, когда в зависимости от некоторого условия необходимо выполнять разные действия. Рассмотрим схему оператора *if*:

```
if (<условие>) {
      <операторы 1>
}
else {
```

```
    <операторы 2>
}
```

Если *<условие>* истинно, то выполнится последовательность *операторов 1*, иначе - последовательность *операторов 2*. Часть *else* может отсутствовать. Если выполняется только один оператор, то фигурные скобки можно опустить.

Если необходимо выполнять различные действия в зависимости от целочисленного значения некоторого выражения (при трех и более вариантах) удобно использовать оператор *switch*.

3 Пример выполнения лабораторной работы

Задание: Построить программу, которая вводит координаты точки (x, y) и определяет, попадает ли точка в заштрихованную область на рисунке 1.1. Попадание на границу области считать попаданием в область.

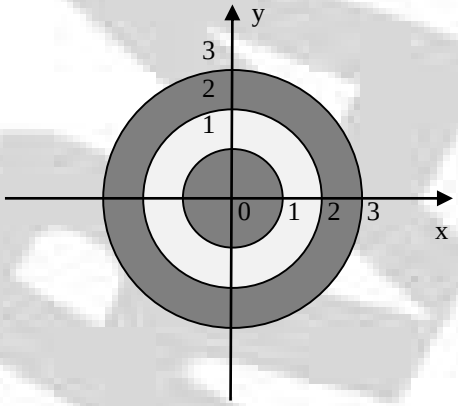


Рис 1.1. Заштрихованная область *D*.

Условие попадания точки *p* с координатами *(x, y)* в область *D*: $x^2 + y^2 \le 1^2$ *ИЛИ* ($x^2 + y^2 \ge 2^2$ *И* $x^2 + y^2 \le 3^2$). Так как координаты точек, которые принадлежат области *D*, лежат в диапазоне [0,1] и [2,3], то для представления координат имеет смысл использовать вещественные числа. В первой лабораторной работе для представления вещественных чисел рекомендуется использовать тип *double*.

Программа 1.3.

```
// Определение попадания точки в заданную область
#include <iostream>
using namespace std;
int main()
{
```

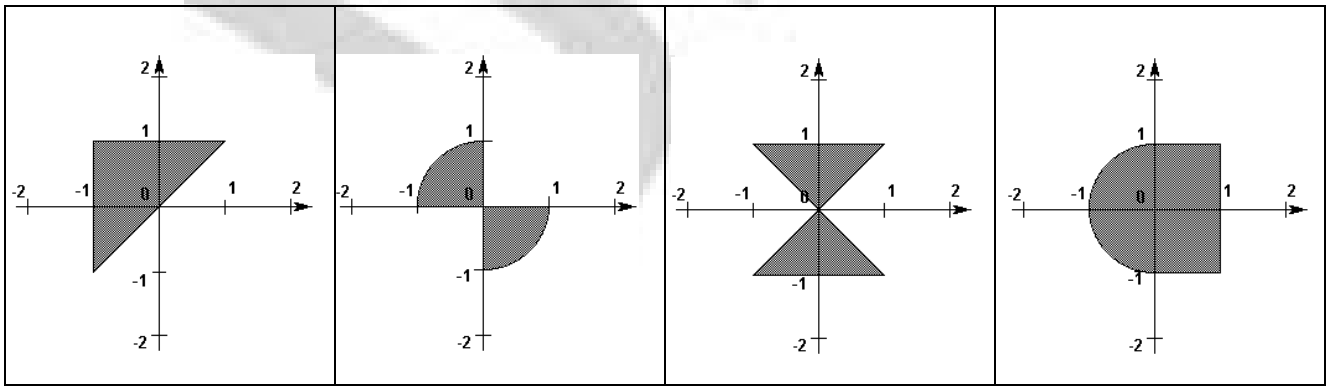
```
    setlocale(0, ""); // для использования кириллицы при вводе-выводе
    double x, y; // координаты точки p
    // считываем координаты точки с клавиатуры
    cout << "Введите координаты точки p = (x y): ";
    cin >> x >> y;
    cout << "Точка p = (" << x << ", " << y << ") " << endl;
    double r = x*x + y*y; // квадрат радиуса окружности,
                          // на которой находится точка
    bool isPointInRegion = (r <= 1) || (r >= 4) && (r <= 9);

    if (isPointInRegion)
        cout << "Точка попала в область!" << endl;
    else
        cout << "Точка не попала в область!" << endl;

    return 0;
}
```

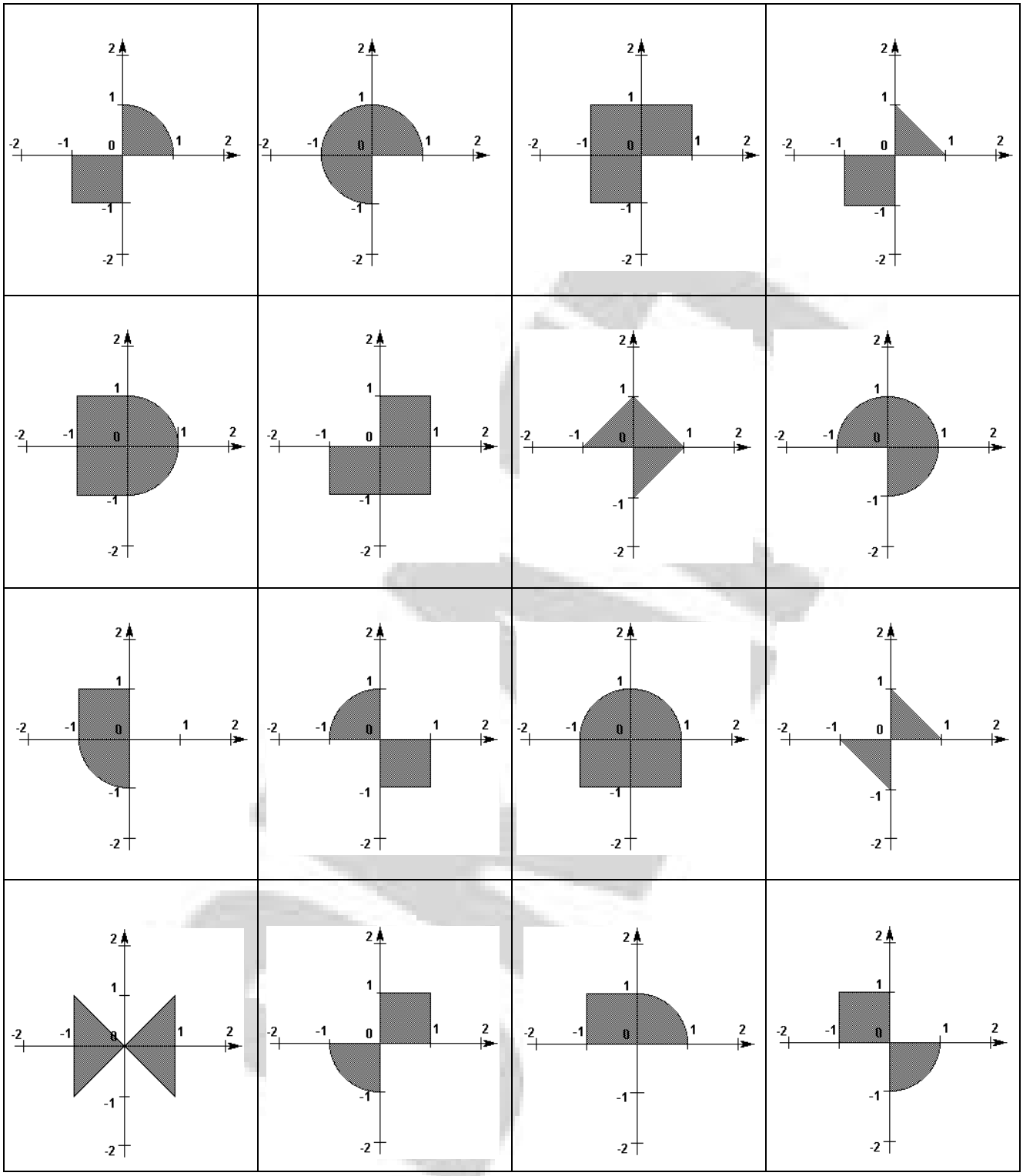
Пример работы программы 1.3:

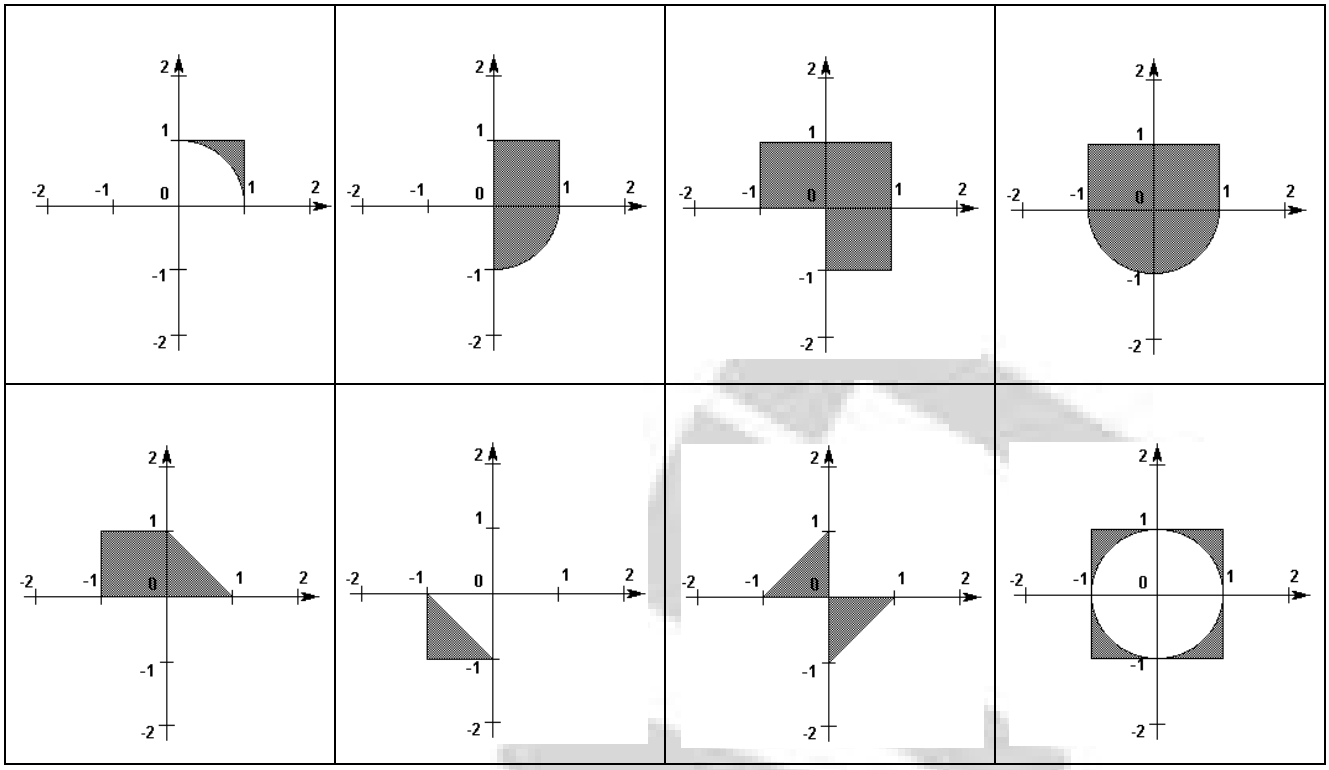
```
Введите координаты точки p = (x y): 0.5 0.5
Точка p = (0.5, 0.5)
Точка попала в область!
```

4 Задания к лабораторной работе

Построить программу, которая вводит координаты точки *(x, y)* и определяет, попадает ли точка в заштрихованную область на рисунке, который соответствует Вашему варианту. Попадание на границу области считать попаданием в область.







5 Требования к лабораторной работе

Совпадают с общими требованиями.

ЛАБОРАТОРНАЯ РАБОТА 2. РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ. ЦЕЛОЧИСЛЕННЫЕ ТИПЫ ДАННЫХ

1 Цель работы

Получение навыков использования операторов циклов с неизвестным количеством повторений. Ознакомление с особенностями вычислений с целыми числами.

2 Теоретический материал

2.1 Управляющие операторы. Циклы

Еще одна группа управляющих операторов – **циклы**. Циклы позволяют задавать многократное выполнение набора операторов. В *C++* существуют три оператора цикла: с неизвестным количеством повторений *while* (цикл с предусловием) и *do-while* (цикл с постусловием), и с известным количеством повторений *for*.

Оператор цикла *while* используется, когда необходимое количество повторений тела цикла заранее не известно и зависит от некоторого условия. При этом тело цикла может быть не выполнено ни разу или выполняться бесконечное число раз. Рассмотрим схему цикла *while*:

```
while (<условие>) {
      <операторы>
}
```

Если <*условие*> истинно, то выполнится последовательность *операторов* (**тело цикла**). После выполнения тела цикла, снова проверятся <*условие*> и т.д. Этот процесс будет продолжаться до тех пор, пока <*условие*> не станет ложно. Если в теле цикла состоит только из одного оператора, то фигурные скобки можно опустить. Рекомендуется ознакомиться с операторами ***break*** и ***continue***.

2.2 Целочисленные типы данных

Как следует из названия, целочисленные типы предназначены для хранения целых чисел. В *C++* существует несколько целочисленных типов, которые отличаются по **ширине**, то есть по количеству бит, которые выделяются для хранения чисел данного типа: *char*, *short int*, *int*, *long int* и *long long int*. Можно использовать сокращенные имена – *short*, *int*, *long* и *long long* соответственно.

Объем памяти, отводимый для хранения конкретного типа, зависит от реализации (компилятора, архитектуры процессора). Тип char имеет ширину равную количеству бит, отводимых для хранения одного символа в данной реализации, и обычно занимает один байт. Ширина остальных типов выражается через ширину *char*, и может быть найдена с помощью оператора *sizeof*. Выполняется соотношение $1 = sizeof(char) \le sizeof(short) \le sizeof(int) \le sizeof(long) \le sizeof(long\ long)$.

Каждый из целочисленных типов может хранить положительные и отрицательные значения (signed) или только положительные (unsigned). Типы *short int*, *int*, *long int* и *long long int* по умолчанию *signed*, а для *char* – зависит от реализации. То есть запись *long* означает *signed long int*, а запись *char* – *signed char* или *unsigned char* в зависимости от реализации. Выполняется соотношение $sizeof(I) == sizeof(signed\ I) == sizeof(unsigned\ I)$, где *I* – любой из целочисленных типов.

В заголовочном файле стандартной библиотеки C++ *climits* определены константы, которые содержат максимальные и минимальные значения каждого из целочисленных типов для конкретной реализации, например, константа *UINT_MAX* содержит максимальное значение для типа *unsigned int*.

Положительные числа хранятся в прямом двоичном коде, то есть число $11_{10}=1011_2$, представленное 8-битным *char*, хранится в виде последовательности бит <00001011>.

Отрицательные целые числа представляются в дополнительном коде. Дополнительный код можно получить, прибавив единицу к инверсированному представлению модуля числа, например, дополнительный код для -11_{10} будет равен <11110101>. В этом случае число 0 имеет единственное представление <00...00>, а также не требуется специальных правил для выполнения операций, так как для любого числа *a* выполняется соотношение $a_2+(-a_2) = 0_2$.

При выполнении арифметических операций может возникнуть **переполнение**. Рассмотрим пример переполнения для *signed char*. Если прибавить к числу $127_{10}=01111111_2$ число $1_{10}=00000001_2$, то по правилам сложения в двоичной системе счисления получится число 10000000_2, которое в дополнительном коде соответствует числу -128_{10}. Теперь рассмотрим пример переполнения для *unsigned char*: при вычитании из числа $0_{10}=00000000_2$ числа $1_{10}=00000001_2$, то по правилам вычитания в двоичной системе счисления получится число 11111111_2, которое соответствует числу 127_{10}. За переполнениями программист должен

следить самостоятельно, так как сообщений об ошибках при переполнении не будет выдано ни во время компиляции, ни во время выполнения программы.

2.3 Рекуррентные соотношения

Последовательность чисел $<a0, a1, a2,\ldots>$ называется **рекуррентной последовательностью m-го порядка**, если для любого $n \geq m$ элемент a_{n+m} можно выразить как функцию от m предшествующих элементов:

$$a_{n+m} = f(a_{n+m-1}, a_{n+m-2}, \ldots, a_n) \qquad (2.1)$$

Соотношение (2.1) называется **рекуррентным соотношением m-го порядка**. Рекуррентное соотношение является **линейным**, если функция f – линейная, то есть соотношение (2.1) можно представить в виде:

$$a_{n+m} = p_m(n)\, a_{n+m-1} + p_{m-1}(n)\, a_{n+m-2} + \ldots + p_1(n)\, a_n + u_n, \qquad (2.2)$$

где коэффициенты $p_i(n)$ являются некоторым функциями от n. Если коэффициенты $p_i(n)$ не зависят от n, то рекуррентное соотношение называются **рекуррентным соотношением с постоянными коэффициентами**. Если u_n равен нулю, то рекуррентное соотношение называют **однородным**. Пример линейного однородного рекуррентного соотношения второго порядка с постоянными коэффициентами – числа Фибоначчи:

$$F_{n+2} = F_{n+1} + F_n;\ F_0 = 0 \text{ и } F_1 = 1. \qquad (2.3)$$

2.4 Схема итерации

Вычисления элементов рекуррентной последовательности (и других вычислительных задач) удобно организовать в виде следующей схемы:

Схема 2.4.

```
x = x0;
while B(x) {
      x = F(x);
},
```

где x – множество переменных, x_0 – их начальные значения, $B(x)$ – функция от x, которая возвращает значение типа *bool* (**предикат**), а $F(x)$ – функция пересчета значений x. В соответствии с семантикой цикла *while* предикат $B(x)$ – условие продолжения цикла.

Схему (2.4) иногда называют **схемой итерации** (от латинского *iteratio* – повторение), а отдельные шаги цикла — **итерациями**.

Рассмотрим свойства программ, порождаемых схемой (2.4). Пусть цикл *while* в (2.4) завершился за n шагов и переменные x принимали последовательно

значения, $x_0, x_1, \ldots, x_n$, тогда последовательность x_i $(i \in [0; n])$ обладает свойствами:

1. $x_i = F(x_{i-1})$ для всех $i \in [1; n]$;
2. $x_i \neq x_j$ при $i \neq j$ и $i, j \in [0; n]$ (необходимое условие завершения цикла);
3. $B(x_i)$ истинно для всех $i \in [0; n-1])$;
4. $B(x_n)$ ложно.

Итерации заканчиваются, если существует n, для которого выполняется свойство 4.

Рассмотрим задачу вычисления числа Фибоначчи F_n с заданным номером $n \geq 1$. Построим решение в соответствии со схемой (2.4), где:

x – переменные $<i, a_i, b_i>$, где i - номер итерации, $a_i = F_i$ и $b_i = F_{i-1}$;

x_0 – начальные значения $<i, a_i, b_i> = <1,1,0>$;

$F(x) = F(<i, a_i, b_i>) = <i+1, a_i + b_i, a_i>$;

$B(x) = B(<i, a_i, b_i>) = (i < n)$.

Теперь построим программу в соответствии со схемой (2.4):

Программа 2.2.

```
unsigned int a = 1;
unsigned int b = 0;
unsigned int i = 1;
unsigned int tmp;
while (i < n)
{
      tmp = a;
      a = a + b;
      b = tmp;
      i = i + 1;
}
```

Проверить корректность программы 2.2 до этапа тестирования, можно с помощью понятия **инвариант цикла**. Инвариант цикла – это условие, которое истинно 1) до входа цикл, 2) после каждой итерации и 3) после выхода из цикла. Инвариантом цикла в программе 2.2. является условие $a=F(i)$ и $b=F(i-1)$.

3 Пример выполнения лабораторной работы

Задание: Для заданного целого $n \geq 0$ вычислить $S(n) = \sum_{i=0...n} a(i)$, где $a(i) = A(i \mid m)$, m — заданное целое ($m \geq n$) и $A(i \mid m) = m! / (m-i)!$.

Рекуррентное соотношение для суммы $S(i)$: $S(i + 1) = S(i) + a(i + 1)$. Найдем рекуррентное соотношение для $a(i + 1)$. Для этого рассмотрим отношение $p(i) = a(i + 1) / a(i) = [m! / (m – i – 1)!] [(m – i)! / m!] = m – i$. Таким образом, имеем рекуррентное соотношение первого порядка с непостоянными коэффициентами $a(i + 1) = (m – i)\, a(i)$. При $i = 0$ имеем $a(0) = m! / (m – 0)! = 1$ и $S(0) = 1$.

В программе будем отслеживать переполнение при вычислении очередного значения $S(i)$, т. е. $S(i)$ может быть вычислено только при выполнении неравенства $S + (m – i)a \le UINT\_MAX$ (при использовании типа unsigned int). Прямая проверка неравенства невозможна, так как сумма $S + (m – i)\, a$ может превысить $UINT\_MAX$ (при этом произойдет переполнение и неравенство будет верным), поэтому заменим неравенство на $a \le (UINT\_MAX – S) / (m – i)$.

Программа 2.3.

```
//Программа вычисляет сумму s = sum (i=0...n) (m!/(m-i)!), m>=n>=0
#include <climits> // UINT_MAX
#include <iostream>

using namespace std;
using uint = unsigned int;

int main()
{
	setlocale(0, "");
	uint m, n;
	cout << "Вычисление суммы s = sum(i = 0...n) (m!/ (m - i)!), "
		<< " m >= n >= 0" << endl;
	cout << "Введите n > 0: ";
	cin >> n;
	cout << "n = " << n << endl;
	cout << "Введите m > " << n << ": ";
	cin >> m;
	cout << "m = " << m << endl;

	uint a = 1;     // текущее значение a
	uint s = 1;     // текущая сумма последовательности
	uint i = 0;     // номер шага
	bool isOverflow = false; // флаг переполнения
```

```
	// Инвариант цикла: a=a(i) && s = sum (j=0..i) (a(j))
	while ((i < n) && !isOverflow)
	{
		cout << "i = " << i << ", a = " << a << ", s = " << s << endl;
		a = (m - i) * a;
		s = s + a;
		i = i + 1;
		if ((i < n) && ((UINT_MAX - s) / (m - i) < a))
			isOverflow = true;
	}

	cout << "При i = " << i << ", последнее слагаемое a = " << a
		<< ", сумма s = " << s << endl;

	if (isOverflow)
		cout << "Ошибка: При i = " << i + 1
			<< " произойдет переполнение." << endl;
}
```

Пример работы программы 2.3:

```
Вычисление суммы S = Sum(i = 0...n) (m!/ (m - i)!), m >= n >= 0
Введите n > 0: 10
n = 10
Введите m > 10: 15
m = 15
i = 0, a = 1, s = 1
i = 1, a = 15, s = 16
i = 2, a = 210, s = 226
i = 3, a = 2730, s = 2956
i = 4, a = 32760, s = 35716
i = 5, a = 360360, s = 396076
i = 6, a = 3603600, s = 3999676
i = 7, a = 32432400, s = 36432076
i = 8, a = 259459200, s = 295891276
При i = 9, последнее слагаемое a = 1816214400, сумма s = 2112105676
Ошибка: При i = 10 произойдет переполнение.
Для продолжения нажмите любую клавишу . . .
```

4 Задания к лабораторной работе

1. Для заданного целого $n \geq 0$ вычислить:

a. $(2n + 1)!! = 1 \cdot 3 \cdot 5 \cdot \ldots \cdot (2k + 1)$;

b. $(2n)!! = 1 \cdot 2 \cdot 4 \cdot 6 \cdot \ldots \cdot (2n)$;

n!!, n!!= (2k + 1)!! при нечетном n = 2k + 1, (2k)!! при четном n = 2k.

2. Для заданного целого $n \geq 0$ вычислить $S(n) = \sum_{i = 0 \ldots n} a(i)$; вид слагаемых $a(i)$ определяется вариантами:

a. $a(i) = i!$;

b. $a(i) = i! \cdot i$; для "контроля" учесть, что $S(n) = (n + 1)! - 1$;

c. $a(i) = k^i$, где $k > 0$ – заданное целое;

d. $a(i) = (n - i)\, k^i$, где $k > 1$ – заданное целое;

e. $a(i) = A(n - i \mid m)$, где m – заданное целое ($m \geq n$);

f. $a(i) = A(n \mid m + i)$, где m – заданное целое ($m \geq n$);

Примечание: для вариантов e и f $A(t \mid r) = r!/(r - t)!$.

3. Для целых $p \geq q \geq 0$ определим $(p \mid q) = p!/(q!\,(p - q)!)$. Для заданного целого $n \geq 0$ вычислить $S(n) = \sum_{i = 0 \ldots k} a(i)$, где:

a. $k = [\, n / 2]$, $a(i) = (n \mid 2i)$, для "контроля" учесть, что $S(n) = 2^{n-1}$;

b. $k = [\, n / 2]$, $a(i) = (n \mid 2i + 1)$, для "контроля" учесть, что $S(n) = 2^{n-1}$;

c. $k = n$, $a(i) = (n \mid i)$, для "контроля" учесть, что $S(n) = 2^n$;

d. $k = n$, $a(i) = i \cdot (n \mid i)$; $n > 1$; для "контроля" учесть, что $S(n) = n\, 2^{n-1}$;

e. $k = n$, $a(i) = (m + i \mid m)$, где m — заданное целое $(m > 0)$; для "контроля" учесть, что $S(n) = (m + n + 1 \mid m + 1)$;

f. $k = n$, $a(i) = (m \mid i)\,(-1)^i$, где m — заданное целое $(m > n)$; для "контроля" учесть, что $S(n) = (m - 1 \mid n) \cdot (-1)^n$.

4. Задана последовательность чисел Фибоначчи $\{F(n)\}$. Для заданного целого $n \geq 0$ вычислить:

a. $F(n)$;

b. $S(n) = \sum_{i = 0 \ldots n} a(i)$, где $a(i) = F(i)$; для «контроля» учесть, что $S(n) = F(n + 2) - 1$;

c. $\Phi(n)$ — элемент последовательности Фибоначчи второго порядка, определяемой условиями $\Phi(0) = 0$, $\Phi(1) = 1$, $\Phi(n + 2) = \Phi(n + 1) + \Phi(n) + F(n)$; для "контроля" учесть, что $\Phi(n) = ((3n + 3)/5)\, F(n) - (n/5)\, F(n + 1)$.

5. Задана последовательность чисел Фибоначчи *{F(n)}*. Для заданного целого *m>1* вычислить:

a. первое по порядку число Фибоначчи *F(n)*, большее *m*;
b. первое по порядку число Фибоначчи второго порядка *Ф(n)*, большее *m* (см. задание 4c) ;
c. наименьшее *n*, такое, что *S(n) > m*, S(n) определено в задании 4b.

6. Пусть задано целое *m > 0*. Последовательность *{A(n)}* определяется условиями *A(0) = 0, A(1) = 1, A(n + 2) = A(n + 1) + A(n) + (n | m)*, где *(n | m)* определено в задании 3 при $n \geq m \geq 0$, а при *n < m* положим *(n | m) = 0*. Требуется:

a. для заданного целого $n \geq 0$ вычислить *A(n)*;
b. для заданного целого *k > 1* вычислить первое по порядку *A(n)*, большее *k*.

7. Целое число *k* задано в десятичной системе записи. Пользуясь стандартными операциями с целыми числами, вычислить следующие характеристики двоичной записи числа *k*:

a. *N(k)* – количество двоичных цифр;
b. $N_0(k)$ – количество позиций с цифрой *0*;
c. $N_1(k)$ – количество позиций с цифрой *1*;
d. номер позиции первой слева (справа) цифры *0 (1)*.

5 Требования к лабораторной работе

1. Организовать вычисления в соответствии со схемой (2.4). Для представления данных использовать только целочисленные типы;
2. Реализовать проверку вводимых данных (на соответствие условиям задачи и на корректность ввода – например, буквы вместо числа);
3. Организовать вывод промежуточных данных, иллюстрирующий работу алгоритма и процесс получения конечного результата;
4. В отчете представить вывод рекуррентных соотношений, использованных при вычислениях, и указать инвариант основного цикла вычислений (показать, что цикл не нарушает инвариант);

5. Отслеживать переполнение и прерывать вычисления при их обнаружении;

6. Определить диапазон значений входных данных, для которых программа выдает корректные выходные данные, для различных целочисленных типов – *unsigned short, unsigned int, unsigned long, unsigned long long.*

ЛАБОРАТОРНАЯ РАБОТА 3. ВЫЧИСЛЕНИЕ СУММЫ РЯДА С ЗАДАННОЙ ТОЧНОСТЬЮ. ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ

1 Цель работы

Изучение представления и особенностей использования чисел с плавающей точкой.

2 Теоретический материал

2.1 Числа с плавающей точкой

Формат чисел с плавающей точкой является приближением вещественных чисел (бесконечное множество) на ограниченном участке памяти (конечное подмножество вещественных чисел) и определяется стандартом *IEEE 754*.

Вспомним известный из курса школьной алгебры **стандартный вид** числа в десятичной системе счисления: $x = a \times 10^m$, где $1 \le a < 10$ (например, $123 = 1{,}23 \times 10^2$). В стандарте *IEEE 754* используется **нормализованное представление** числа в двоичной системе счисления, которое определяется аналогичным образом: $x = a \times 2^m$, где $1 \le a < 2$. В общем виде число x можно представить в виде $x = (-1)^{знак} \times мантисса \times 2^{порядок}$. В таком виде невозможно представить ноль, для представления которого используется специальная последовательность бит.

В памяти число $x = (-1)^{знак} \times мантисса \times 2^{порядок}$ представляется в виде последовательности бит $\underbrace{S}_{Знак}\ \underbrace{EE...E}_{Порядок}\underbrace{MM...M}_{Мантисса}$, где количество бит для хранения порядка (n бит) и мантиссы (m бит) зависят от **формата** (будут рассмотрены в п. 2.2). Из неравенства $1 \le мантисса < 2$ следует, что старший бит мантиссы всегда равен единице, поэтому в целях экономии памяти он не хранится (битовое представление мантиссы *00…0MM…M* означает *1.MM…M*$_2$, таким образом, $M_{min} = 1$, $M_{max} = 2-2^{-m}$). Порядок представлен в виде **кода со сдвигом** (битовое представление порядка *EE…E* означает $EE\ldots E_2 - 2^{n-1}$, таким образом, $E_{min} = -2^{n-1}$, а $E_{max} = 2^{n-1}-1$).

Специальные значения:

1. **Положительный и отрицательный нули**. Представлены в виде последовательности: $\underbrace{S}_{Знак}\ \underbrace{00...0}_{Порядок}\underbrace{00...0}_{Мантисса}$. Таким образом, в зависимости от значения бита S ноль может быть, как положительным, так и отрицательным.

2. **Положительная и отрицательная бесконечности**. Представлены как $\underbrace{S}_{Знак}\ \underbrace{11...1}_{Порядок}\underbrace{00...0}_{Мантисса}$. Возникают при переполнении, например $x \div 0$.

3. **Неопределенность**. (*NaN* «*Not a Number*», «*Не число*»). Представлена в виде последовательности бит $\underbrace{S}_{Знак}\ \underbrace{11...1}_{Порядок}\underbrace{MM...M}_{Мантисса}$. Возникает, когда математическая операция имеет неопределенный результат, например, *NaN* будет результатом следующих операций: $-\infty+\infty$, $0\times\infty$, $\pm 0 \div \pm 0$.

4. **Денормализованные числа (subnormal)**. Используются для увеличения точности представления чисел, близких к нулю. Представляются последовательностью бит $\underbrace{S}_{Знак}\ \underbrace{00...0}_{Порядок}\underbrace{MM...M}_{Мантисса}$. При этом битовое представление мантиссы *00…0MM…M* означает не *1.MM…M*$_2$, а *0.MM…M*$_2$. При этом $M_{min} = 2^{-m}$, $M_{max} = 1-2^{-m}$. Порядок денормализованных чисел равен -2^{n-1}.

2.2 Представление чисел с плавающей точкой в C++

В таблице 3.1 приведено соответствие типов, определенных стандартом *IEEE 754*.

Таблица 3.1. Форматы *IEEE 754* чисел

Тип по стандарту *IEEE 754*	Всего бит	Порядок, *w* бит	Мантисса, *t* бит
binary16 (половинная точность)	16	5	10
binary32 (одинарная точность)	32	8	23
binary64 (двойная точность)	64	11	52
binary128 (расширенная точность)	128	15	112
BinaryK $\{k \geq 128\}$	*k*	*round(4×log2(k)) – 13*	*k-w-1*

В *C++* стандартом определено три типа для хранения вещественных типов: *float*, *double* и *long double*. Как и в случае целочисленных типов, количество байт, выделенных под каждый вещественный тип зависит от реализации и может быть найдено с помощью оператора *sizeof*. Выполняется соотношение

sizeof(float) ≤ sizeof(double) ≤ sizeof (long double). В заголовочном файле *cfloat* хранятся константы, которые содержат информацию о количестве бит, выделенных для каждого вещественного типа, максимальное и минимальное значение и т.д.

Примеры записи вещественных чисел:

```
double a = 1.23; // 1.23
double a = 2.;   // 2.0
double a = .005; // 0.005
double a = 1.23e-10;  // 1.23 x 10^-10
```

По умолчанию тип любого вещественного литерала – double. Чтобы явно указать тип можно использовать суффиксы, например: *1.23f* (float), *1.23L* (long double).

2.3 Особенности арифметики чисел с плавающей точкой

При выполнении арифметических операций с вещественными типами необходимо учитывать следующие особенности:

1. При сложении чисел с большой разницей между порядками, меньшее число может не повлиять на сумму, так как бит мантиссы может не хватить для включения меньшего числа.

В связи с этой особенностью, часто в качестве характеристики точности представления вещественных чисел, в ЭВМ используют так называемое **машинное эпсилон**, определяемое как eps = min {e: (e + 1) > 1}. Значение машинного эпсилон для типа double хранится в константе DBL_EPSILON в файле *cfloat* и может быть вычислено с помощью следующего алгоритма:

```
double eps = 1.0;
while (eps + 1 > 1.0) {
     eps = eps * 0.5;
}
eps = eps * 2;
```

2. Нельзя напрямую сравнивать два вещественных числа из-за погрешности в представлении. Например, условие 1.0/7+2.0/7 == 3.0/7 может быть ложным. Для сравнения вещественных чисел на равенство рекомендуется сравнивать модуль разности чисел с некоторой малой величиной, например, машинным эпсилон: abs(a-b) < eps.

3 Пример выполнения лабораторной работы

Задание: Вычислить частичную сумму сходящегося бесконечного ряда разложения функции, $f(x) = \sin(x) = \Sigma_{i=0\ldots\infty}(-1)^i x^{2i+1}/(2i+1)!$, $x \in R$, *т. е.* $D = [-\infty, +\infty]$.

Найдем рекуррентное соотношение для вычисления очередного элемента ряда: $\frac{u_i}{u_{i-1}} = \frac{(-1)^{i+1}x^{2i+3}\times(2i+1)!}{(2i+3)!\times(-1)^i x^{2i+1}} = \frac{-x^2}{(2i+2)(2i+3)}$. Получилось линейное рекуррентное соотношение первого порядка: $u_{i+1} = p(i)u_i$, где $p(i) = \frac{-x^2}{(2i+2)(2i+3)}$. Первый элемент последовательности $u_0 = x$.

Вычисления суммы ряда построим по схеме итерации:

x – *переменные* $<i, u_i, s_i>$, где i - *номер итерации,* u_i = *элемент ряда и* $s_i = \Sigma_{j=0\ldots i}u_j$;

x_0 – начальные значения $<i, u_i, s_i> = <0,x,x>$;

$F(x) = F(<i, u_i, s_i>) = <i+1, u_i \times p(i), s_i + u_i>$;

$B(x) = B(<i, u_i, s_i>) = i < N$.

По условию $N=\infty$. Однако ясно, что вычисления не могут продолжаться бесконечно. Так как ряд сходящийся, то $\lim(u_n)_{n\to\infty} = 0$ и при некотором k разность порядков текущего значения суммы и u_k будет достаточно велика, чтобы при сложении $u_{i>k}$ не влияли на значение суммы. В качестве критерия завершения вычислений возьмем неравенство $|u_{i+1}| \le eps$.

Программа 3.1.

```
//Программа вычисляет значение sin через разложение в бесконечный ряд
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;
using real = double;

int main()
{
	setlocale(0, "");
```

```
	// Вычисление машинного эпсилон
	real eps = 1.0;
	while (1 + eps > 1)
		eps /= 2;
	eps *= 2;
	cout << "Машинное эпсилон = " << eps << endl;

	// Ввод параметра x
	real x;
	cout << "Введите x : ";
	cin >> x;

	cout << scientific << setprecision(10) << endl; // манипуляторы вывода
	cout << "x = " << x << endl;
	real s = x;                  // сумма ряда
	real u = x;                  // слагаемое
	int i = 0;                // номер шага

	// Инвариант цикла: s=s(i) & u=u(i)
	while (abs(u) > eps) {
		i++;
		u *= -x*x / (2 * i) / (2 * i + 1);
		s += u;

		cout << "Слагаемое u(" << setw(2) << i << ") = " << showpos << u
			<< " сумма  s(" << noshowpos << setw(2) << i << ") = "
			<< showpos << s << noshowpos << endl;
	}

	cout << showpos << "S(x) = " << s << " sin x = " << sin(x) << noshowpos << " n = " << i << endl;
	cout << showpos << "Абсолютная погрешность = " << fabs(s - sin(x)) << endl; return 0;

	}
```

Пример вывода программы 3.1:

```
Машинное эпсилон = 2.22045e-16
Введите x: 5
```

```
x = 5.0000000000e+00
Слагаемое u( 1) = -2.0833333333e+01  сумма  s( 1) = -1.5833333333e+01
Слагаемое u( 2) = +2.6041666667e+01  сумма  s( 2) = +1.0208333333e+01
Слагаемое u( 3) = -1.5500992063e+01  сумма  s( 3) = -5.2926587302e+00
Слагаемое u( 4) = +5.3822889109e+00  сумма  s( 4) = +8.9630180776e-02
Слагаемое u( 5) = -1.2232474798e+00  сумма  s( 5) = -1.1336172990e+00
Слагаемое u( 6) = +1.9603324996e-01  сумма  s( 6) = -9.3758404902e-01
Слагаемое u( 7) = -2.3337291662e-02  сумма  s( 7) = -9.6092134068e-01
Слагаемое u( 8) = +2.1449716601e-03  сумма  s( 8) = -9.5877636902e-01
Слагаемое u( 9) = -1.5679617398e-04  сумма  s( 9) = -9.5893316520e-01
Слагаемое u(10) = +9.3331055943e-06  сумма  s(10) = -9.5892383209e-01
Слагаемое u(11) = -4.6112181790e-07  сумма  s(11) = -9.5892429321e-01
Слагаемое u(12) = +1.9213409079e-08  сумма  s(12) = -9.5892427400e-01
Слагаемое u(13) = -6.8423821507e-10  сумма  s(13) = -9.5892427468e-01
Слагаемое u(14) = +2.1066447508e-11  сумма  s(14) = -9.5892427466e-01
Слагаемое u(15) = -5.6630235238e-13  сумма  s(15) = -9.5892427466e-01
Слагаемое u(16) = +1.3406779176e-14  сумма  s(16) = -9.5892427466e-01
Слагаемое u(17) = -2.8165502470e-16  сумма  s(17) = -9.5892427466e-01
Слагаемое u(18) = +5.2863180311e-18  сумма  s(18) = -9.5892427466e-01
     S(x) = -9.5892427466e-01 sin x = -9.5892427466e-01 n = 18
     Абсолютная погрешность = +1.1102230246e-16
```

4 Задания к лабораторной работе

Рассматриваются ряды $f(x) = \Sigma i = 0...\infty u_i$. Для каждого индивидуального задания определены вид элемента ряда u_i, функция f(x), область сходимости D, если $D \neq R$.

1. $u(i) = (-1)^i x^{2i} / (2i)!;\ f(x) = \cos(x).$
2. $u(i) = x^i / i!;\ f(x) = \exp(x).$
3. $u(i) = (-1)^i x^{2i} / i!;\ f(x) = \exp(-x^2).$
4. $u(i) = x^i (i + 1) / i!\ ;\ f(x) = \exp(x)(1+x).$
5. $u(i) = x^{3i} / (3i)!;\ f(x) = (1/3)\exp(x) + (2/3)\exp(-x/2)\cos(x\ \mathrm{sqrt}(3)/2).$
6. $u(i) = x^{3i + 1} / (3i + 1)!;$
   $f(x) = (1/3)\exp(x) - (2/3)\exp(-x/2)\sin((\pi/6 - x\ \mathrm{sqrt}(3)/2\ )).$
7. $u(i) = x^{4i} / (4i)!;\ f(x) = (1/2)(\mathrm{ch}(x) + \cos(x)).$
8. $u(i) = (-1)^i x^{4i} / (4i)!;\ f(x) = \cos(x / \mathrm{sqrt}(2))\ \ \mathrm{ch}(x / \mathrm{sqrt}(2)).$
9. $u(i) = x^{4i + 1} / (4i + 1)!;\ f(x) = (1/2)(\mathrm{sh}(x) + \sin(x)).$
10. $u(i) = x^{4i + 3} / (4i + 3)!;\ f(x) = (1/2)(\mathrm{sh}(x) - \sin(x)).$
11. $u(i) = (-1)^i 2^{2i} x^{4i} / (4i)!;\qquad i \geq 1;\ f(x) = \mathrm{ch}(x)\cos(x) - 1.$
12. $u(i) = (-1)^{i + 1} 2^{2i - 1} x^{4i - 2} / (4i - 2)!;\ i \geq 1;\ f(x) = \mathrm{sh}(x)\sin(x).$

13. $u(i) = 2^{2i}x^{2i+1} / (2i + 1)!; \quad i \geq 1; \; f(x) = sh(x)*ch(x)-x.$

14. $u(i) = (-1)^{i+1} 2^{2i-1}x^{2i} / (2i)!; \; i \geq 1; \; f(x) = \sin^2(x).$

15. $u(i) = (-1)^i(2i - 1)! \, x^{2i} / 2^{2i} / (i!)^2 ; \; i \geq 1;$
$f(x) = \ln 2 - \ln(1 + \sqrt{1 + x^2}); \quad x^2 \leq 1.$

16. $u(i) = (-1)^i 2^{2i-1}(i - 1)! \, i! \, x^{2i+1}/(2i + 1)!; \; i \geq 1;$
$f(x) = x - \sqrt{1 + x^2} \ln(x + \sqrt{1 + x^2}); \; x^2 < 1.$

17. $u(i) = (-1)^i 2^{2i}(i!)^2 x^{2i+1} / (2i + 1)!;$
$f(x) = \ln(x + \sqrt{1 + x^2}) / \sqrt{1 + x^2}; \; x^2 < 1.$

18. $u(i) = (-1)^i(2i - 1)! / 2^{2i-1} / i! / (i - 1)! / (2i + 1) / x^{2i+1}; \; i \geq 1;$
$f(x) = \ln(1 + \sqrt{1 + x^2}) - \ln(x) - 1/x; \quad x^2 \geq 1.$

19. $u(i) = (2i)! \, x^{2i+1} / 2^{2i} / (i!)^2 / (2i + 1); \; f(x) = \arcsin(x); \; x^2 < 1.$

20. $u(i) = 2^{2i} (i!)^2 \, x^{2i+1} / (2i + 1)! / (i + 1); \; f(x) = \arcsin^2(x); \; x^2 \leq 1.$

21. $u(i) = (-1)^i \, x^{2i+1} / (2i + 1); \; f(x) = \operatorname{arctg}(x); \; x2 \leq 1.$

22. $u(i) = x^i \sin(i \cdot p)/i! \; ; \; i \geq 1 \; ; \; f(x) = \exp(x \cos(p)) \sin(x \sin(p)).$

Примечание.

$sh(x) = (\exp(x) - \exp(-x)) / 2;$

$ch(x) = (\exp(x) + \exp(-x)) / 2.$

5 Требования к лабораторной работе

1 Значение аргумента *x* считывать с клавиатуры, значение машинного эпсилон вычислять в программе. Вычислить абсолютную и относительную погрешность вычислений;

2 Опробовать вычисления с различными вещественными типами (*float, double, long double*). Сравнить значения погрешностей для одного значения *x* (в отчете представить результаты сравнения);

3 Использовать рекуррентное соотношение для вычисления значения очередного слагаемого.

ЛАБОРАТОРНАЯ РАБОТА 4. ФУНКЦИИ. ПЕРЕДАЧА АРГУМЕНТОВ ПО ССЫЛКЕ И ПО ЗНАЧЕНИЮ

1 Цель работы

Ознакомление с парадигмой процедурного программирования. Получение навыков реализации функций на языке программирования C++. Изучение различных способов передачи аргументов в функцию.

2 Теоретический материал

2.1 Процедурное программирование

Программирование в процедурном стиле подразумевает оформление программы в виде набора подпрограмм. Исходная задача разбивается на несколько подзадач, каждая из которых реализована в виде отдельной подпрограммы. При таком подходе снижается сложность, как написания, так и чтения кода, так как код становится более структурированным.

Язык C++ поддерживает парадигму процедурного программирования и позволяет оформлять подпрограммы в виде функций. Функция – именованный фрагмент кода, к которому можно обратиться из другой части программы. Функции могут принимать на вход аргументы, а также возвращать некоторое значение. В третьей лабораторной работе уже использовались различные математические функции библиотеки *C* из заголовочного файла *cmath*. Например, функция *sin*:

```
double sin(double x);
float sin(float x);
long double sin(long double x);
```

В заголовочном файле *cmath* объявлено три функции с именем sin, которые работают с разными вещественными типами (функция sin ***перегружена***, в зависимости от типа фактического параметра будет использована соответствующая функция).

Вызову функции должно предшествовать **объявление функции** (описание **прототипа** функции), то есть описание имени функции, списка формальных параметров (для каждого параметра необходимо указать тип и, возможно, имя параметра, параметры разделяются запятыми) и типа возвращаемого значения:

```
<Тип возвращаемого значения> <Имя функции> (список формальных параметров);
```

Например, для использования функции *sin* необходимо подключить заголовочный файл *cmath*, чтобы после обработки препроцессором в код попало объявление функции *sin* (можно было бы включить объявление функции *sin* вручную, но заголовочные файлы позволяют значительно облегчить поддержку соответствия). После чего, функцию можно вызвать по ее имени: double s = sin(2.39);

В результате вызова будут выполнены операторы тела функции *sin*, то есть операторы, содержащиеся в **определении** функции, которое может располагаться как до, так и после вызова функции (в том числе в другом файле).

Определение функции имеет следующую структуру:

```
<Тип возвращаемого значения> <Имя функции> (список формальных параметров)
{
     операторы, среди которых есть хотя бы один оператор return,
          аргументом которого является результат выполнения функции
}
```

Типом возвращаемого значения может быть любой из стандартных и пользовательских типов языка C++, за исключением массивов. Если функция не должна ничего возвращать (подобно процедурам в Паскале), то в качестве типа возвращаемого значения используется тип *void*, при этом оператор *return* используется без аргументов и может быть опущен.

В примере double s = sin(2.39) значение, которое вернет функция *sin* через оператор *return* будет присвоено переменной *s*.

На этапе компиляции компилятору необходимо знать только прототип функции, чтобы проверить корректность использования функции (соответствие типов и количество аргументов), а также выполнить необходимые приведения типов.

Рассмотрим пример, в котором определяется функция подсчета площади прямоугольника по его сторонам.

Программа 4.1.

```
#include <iostream>
using namespace std;
```

```
double square(double a, double b) {
	return a*b;
}

int main() {
	double a = 5.0;
	double d = 2.0;
	cout << square(a, b);
	return 0;

}
```

Как видно из примера 4.1, если определение функции расположено до вызова, то включать объявление не обязательно. В лабораторных работах для определяемых функций рекомендуется размещать объявление функции до функции *main*, а определение – после (поощряется вынесение определения функций в отдельный файл).

2.2 Область видимости и продолжительность хранения переменных

Область видимости идентификатора определяет область программы, в пределах которой идентификатор доступен. Переменные, объявленные внутри функции (или блока, то есть внутри фигурных скобок), называются **локальными**. То есть обращаться к этим переменным по имени можно только внутри функции или блока, в котором они объявлены. Остальные переменные являются **глобальными**. К ним можно обращаться в любой части файла, после их объявления.

При этом локальные переменные (без модификатора ***static***) имеют **автоматическую продолжительность хранения**, то есть они создаются при входе в функцию (или блок), в которой они определены и удаляются при выходе из нее. Глобальные переменные имеют **статическую продолжительность хранения**, то есть они существуют во время всего времени выполнения программы.

2.3 Передача параметров в функцию

Рассмотрим способы передачи параметров в функцию. Параметры, которые указываются при объявлении/определении функции, называются **формальными**, а параметры, которые передаются непосредственно при вызове

функции – **фактическими**. Параметры функции имеют локальную область видимости.

В C++ существует три способа передачи аргументов в функцию — по значению, по (константному) указателю, по (константной) ссылке.

1. Объявление функции с параметрами, передаваемыми по значению, например, имеет вид:

```
int func(int i, float f, char c);
```

При вызове данной функции для всех формальных параметров выделяется память и присваиваются значения фактических параметров (это могут быть переменные, выражения, константы). После выполнения функции память освобождается. Изменение формальных параметров не сказывается на значениях фактических.

2. Заголовок процедуры с параметрами, передаваемыми по **указателю**, может, например, иметь вид:

```
int func(int* i, const float* f, char* c);
```

Указатель – переменная, которая хранит адрес, по которому расположено значение заданного типа. Чтобы получить адрес, по которой расположена переменная, существует оператор взятия адреса &, а для получения значения, адрес которого содержит указатель – оператор разыменовывания *, например:

```
int a = 5; // Переменная типа int
int* pa = &a; // Указатель на a
int va = *pa; // Значение, на которое указывает pa (равно 5)
```

При передаче параметров по указателю выделяется память для хранения самого указателя, который содержит адрес, по которому расположено значение параметра. Через указатель можно менять значения, на которые он указывает. Данный способ может использоваться для избегания копирования и выделения памяти для параметров, значение которых внутри функции меняться не должно, в этом случае для передачи параметра следует использовать указатель на константу (например, const float* f). Более подробно указатели будут изучены во втором семестре в курсе «Программирование. Дополнительные главы», в текущем семестре для передачи аргументов не по значению рекомендуется использовать ссылки.

3. Заголовок процедуры с параметрами, передаваемыми по **ссылке**, может, например, иметь вид:

```
int func(int& i, const float& f, char& c);
```

При использовании указателей для передачи параметров в функцию необходимо использовать дополнительные операции – оператор взятия адреса при передаче и оператор разыменовывания при каждом обращении ко значению. Ссылки работают аналогично указателям – сами значения аргументов не копируются, все действия производятся с фактическими аргументами. При этом нет необходимости выполнять дополнительные операции – передача аргументов при вызове функции и работа с аргументами внутри функции происходит аналогично передаче аргументов по значению. Если значение внутри функции меняться не должно, то следует использовать константную ссылку (const float& f).

2.4 Спецификация функций

Начиная с данной лабораторной работы, каждая функция, определённая в программе должна сопровождаться **спецификацией**. Спецификация представляет собой комментарий, который содержит краткое описание назначения функции, **предусловие** (описание всех входных данных, включая глобальные переменные, и ограничения, которые накладываются функцией на входные данные) и **постусловие** (описывается результат работы функции, а также все “побочные эффекты”, то есть изменения глобальных переменных и т.д.). Пример спецификации функции представлен в следующем разделе.

3 Пример выполнения лабораторной работы

Задание: Реализовать задачу, решаемую в примере 2.4, с использованием функций.

В виде функций оформляются фрагменты ввода исходных данных, их обработки и вывода результатов. Заголовки сопровождаются комментариями с указанием вида параметров.

Программа 4.1.

```
//Программа вычисляет сумму s = sum (i=0...n) (m!/(m-i)!), m>=n>=0
#include <climits>
#include <iostream>
#include <windows.h>
```

```
using namespace std;
using uint = unsigned int;

// Вычисление суммы sum (i=0...n) (m!/(m-i)!);
// Предусловие: m >= n >= 0
// Постусловие:
// i - номер последнего слагаемого
// a - значение последнего слагаемого
// isOverflow - флаг переполнения (если isOverflow true то
//              вычисления прервались из-за переполнения суммы)
// Возвращает значение вычисленной суммы
uint computeSum(uint n, uint m, uint& i, uint& a, bool &isOverflow);

int main()
{
      setlocale(0, "");
      uint m, n;
      cout << "Вычисление суммы s = sum(i = 0...n) (m!/ (m - i)!), m >= n
>= 0" << endl;
      cout << "Введите n > 0: ";
      cin >> n;
      cout << "n = " << n << endl;
      cout << "Введите m > " << n << ": ";
      cin >> m;
      cout << "m = " << m << endl;

      bool isOverflow; // флаг переполнения
      uint a, i;
      uint s = computeSum(n, m, i , a, isOverflow);

      cout << "При i = " << i << ", последнее слагаемое a = " << a
            << ", сумма s = " << s << endl;

      if (isOverflow)
            cout << "Ошибка: При i = " << i + 1
                  << " произойдет переполнение." << endl;
}


uint computeSum(uint n, uint m, uint& i, uint& a, bool &isOverflow)
```

```
{
	a = 1;	// текущее значение a
	i = 0;	// номер шага
	isOverflow = false;  // флаг переполнение
	uint s = 1;			// текущая сумма последовательности

	// Инвариант цикла: a=a(i) && s = sum (j=0..i) (a(j))
	while ((i < n) && !isOverflow)
	{
		cout << "i = " << i << ", a = " << a << ", s = " << s << endl;
		a = (m - i) * a;
		s = s + a;
		i = i + 1;
		if ((i < n) && ((UINT_MAX - s) / (m - i) < a))
			isOverflow = true;
	}

	return s;
}
```

Пример вывода программы 4.1:

```
Вычисление суммы S = Sum(i = 0...n) (m!/ (m - i)!), m >= n >= 0
Введите n > 0: 10
n = 10
Введите m > 10: 15
m = 15
i = 0, a = 1, s = 1
i = 1, a = 15, s = 16
i = 2, a = 210, s = 226
i = 3, a = 2730, s = 2956
i = 4, a = 32760, s = 35716
i = 5, a = 360360, s = 396076
i = 6, a = 3603600, s = 3999676
i = 7, a = 32432400, s = 36432076
i = 8, a = 259459200, s = 295891276
При i = 9, последнее слагаемое a = 1816214400, сумма s = 2112105676
Ошибка: При i = 10 произойдет переполнение.
Для продолжения нажмите любую клавишу . . .
```

4 Задания к лабораторной работе

Разработать программу для выполнения индивидуального задания с использованием функций языка C++. Первая цифра шифра задания определяет номер одной из предыдущих лабораторных работ, а вторая – номер варианта задачи соответствующей лабораторной работы.

5 Требования к лабораторной работе

1 Оформить в виде функций фрагменты программы, обеспечивающие ввод и проверку исходной информации, ее преобразование, вывод результатов (промежуточных и окончательных);

2 При реализации функций обязательно обеспечить передачу данных с помощью параметров;

3 Для всех определяемых функций указать спецификацию.

ЛАБОРАТОРНАЯ РАБОТА 5. ИНДУКТИВНЫЕ ФУНКЦИИ. ОБРАБОТКА ФАЙЛОВ

1 Цель работы

Изучение однопроходных алгоритмов обработки последовательностей на примере индуктивных функций. Обработка файлов.

2 Теоретический материал

2.1 Индуктивные функции

Пусть заданы некоторые множества *X* и *Y*. Рассмотрим множество функций *F*, аргументами которых являются последовательности элементов множества *X*, а значениями – элементы множества *Y*. Пусть $\Omega_k(X)$ – пространство всех конечных последовательностей ω с элементами из множества *X* длины не менее, чем *k*. Обозначим $\Omega_0(X)$ через $\Omega(X)$, а пустую последовательность - через Δ.

Функция $f:\Omega(X)\to Y$ называется индуктивной, если $f(\omega\times x)$ можно вычислить, зная $f(\omega)$ и *x*, т. е. если существует функция $g:Y\times X\to Y$, такая, что для всех $\omega\in\Omega(X)$ и $x\in X$ выполняется соотношение $f(\omega\times x)=g(f(\omega),x)$.

Примером индуктивной функции является вычисление максимального элемента последовательности $\max(\omega\times x)=\max(\max(\omega),x)$ и суммы элементов последовательности $sum(\omega\times x)=sum(\omega)+x$. Чтобы доказать, что функция индуктивная достаточно описать функцию g.

Не все функции являются индуктивными, например, значение среднего арифметического элементов последовательности. После добавления нового элемента невозможно пересчитать значение среднего арифметического без дополнительной информации.

Существует критерий индуктивности. Пусть даны две различные последовательности ω_1 и ω_2, при этом $f(\omega_1)=f(\omega_2)$. Функция *f* является индуктивной тогда и только тогда, когда для любого *x* $f(\omega_1\times x)=f(\omega_2\times x)$. Таким образом для того, чтобы доказать, что функция не является индуктивной, достаточно привести пример ω_1, ω_2, и x таких, что $f(\omega_1)=f(\omega_2)$ и $f(\omega_1\times x)\neq f(\omega_2\times x)$.

Для неиндуктивных функций можно построить индуктивное расширение. То есть найти индуктивную функцию $\hat{f}:\Omega(X)\to\hat{Y}$ такую, что существует функ-

ция $\pi : \hat{Y} \to Y$ и $\pi(\hat{f}(\omega)) = f(\omega)$. Построим индуктивное расширение для функции вычисления среднего арифметического элементов последовательности. Определим функцию $\hat{f}(\omega) = (sum(\omega), length(\omega))$, где $\hat{Y} = X \times N$, N – множество натуральных чисел. При этом $\pi(sum(\omega), length(\omega)) = \frac{sum(\omega)}{length(\omega)}$.

Вычислить значение индуктивной функции можно за один проход по последовательности, сохраняя при этом только значение функции на предыдущем шаге.

2.2 Обработка текстовых файлов

Последовательность может быть представлена различными способами (в виде массива, линейного списка и т.д.). В данной лабораторной работе будем считать, что последовательность хранится в текстовом файле.

Для обработки текстовых файлов удобно использовать объекты *fstream* (чтение/запись), *ifstream* (чтение) и *ofstream* (запись), которые объявлены в заголовочном файле *fstream* стандартной библиотеки *C++*. Эти объекты служат для потокового ввода/вывода в файлы. Работа с этими объектами аналогична работе с *cin* и *cout*, за исключением того, что *cin* и *cout* автоматически **открываются** при запуске программы (и по умолчанию ассоциируются со стандартным устройством ввода (клавиатурой) и вывода (монитором)), а *ifstream* и *ofstream* необходимо ассоциировать с определенным файлом до начала и закрывать после окончания работы с файлом.

Например, если необходимо открыть на запись файл с именем *output.txt*, то сделать это можно несколькими способами:

```
1. ofstream fout("output.txt");
2. ofstream fout;
   fout.open("output.txt");
3. fstream fout;
   fout.open("output.txt", fstream::out);
```

После окончания работы с файлом его необходимо закрыть:

```
fout.close();
```

Открытие и закрытие файла на чтение происходит аналогично (в случае использования объекта *fstream* при открытии надо использовать флаг *fstream::in*).

3 Пример выполнения лабораторной работы

Задача: Вычислить сумму элементов отрезка заданной последовательности, имеющего максимальную сумму элементов.

Очевидно, что функция не является индуктивной. Докажем это по критерию индуктивности. Возьмем $\omega_1 = \langle 1,2,3,-4 \rangle$ и $\omega_2 = \langle 3,2,1 \rangle$, для которых выполняется условие $f(\omega_1) = f(\omega_2) = 6$. Пусть $x = 5$, тогда $f(\omega_1 \times 5) = 6 \neq f(\omega_2 \times 5) = 11$. Следовательно, функция не является индуктивной.

Построим индуктивное расширение $\hat{f}(\omega) = (f(\omega), cur(\omega))$, где $cur(\omega)$ – максимальная сумма среди хвостовых отрезков. При этом $\hat{f}(\omega \times x) = (\max(f(\omega), cur(\omega)), \max(cur(\omega), 0) + x))$ (сначала пересчитывается $cur(\omega)$). При этом $f(\omega)$ является частью индуктивного расширения, поэтому специальных вычислений проводить не надо.

Программа 5.1.

```
//Программа вычисляет сумму элементов отрезка заданной
// последовательности, имеющего максимальную сумму элементов.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <algorithm>

using namespace std;

int main()
{
	setlocale(0, "");

	ifstream infile("in_seq.txt");
	if (!infile.is_open()) {
		cout << "исходной файл не открыт!" << endl;
		return 0;
	}
```

```
	int x;
	int f = INT_MIN;
	int cur = 0;
	short i = 0;

	while (infile >> x)
	{
		i++;
		// обновление конкурента:
		cur = max(cur, 0) + x;
		// обновление рекорда:
		f = max(f, cur);

		cout << setw(2) << i << ". x = " << setw(4) << x
			<< "; f = " << setw(4) << f << ",  конкурент = " << cur
<< endl;
	}
	infile.close();

	cout << endl;
	cout << "Число элементов в последовательности = " << i << endl;
	cout << "Сумма элементов отрезка с максимальной суммой: " << f <<
endl;
	return 0;
}
```

Пример вывода программы 5.1:

```
 1. x =    5; f =    5,  конкурент = 5
 2. x =    4; f =    9,  конкурент = 9
 3. x =    8; f =   17,  конкурент = 17
 4. x =   -5; f =   17,  конкурент = 12
 5. x =    2; f =   17,  конкурент = 14
 6. x =  -16; f =   17,  конкурент = -2
 7. x =    4; f =   17,  конкурент = 4
 8. x =    9; f =   17,  конкурент = 13
 9. x =   18; f =   31,  конкурент = 31
10. x =   12; f =   43,  конкурент = 43
11. x =  -45; f =   43,  конкурент = -2
12. x =    7; f =   43,  конкурент = 7
```

```
13. x =   10; f =   43,  конкурент = 17
14. x =   28; f =   45,  конкурент = 45
15. x =   15; f =   60,  конкурент = 60
16. x =    4; f =   64,  конкурент = 64
17. x =  -20; f =   64,  конкурент = 44

Число элементов в последовательности = 17
Сумма элементов отрезка с максимальной суммой: 64
```

4 Задания к лабораторной работе

При формулировке задания используются следующие обозначения: Z – целые числа, N – натуральные, N_0 – натуральные с нулем, R – вещественные, R_0 – неотрицательные вещественные, $R+$ – положительные вещественные, S – символы, B – логический тип, $R_2 = R \times R$ (декартово произведение).

В формулировке каждого задания указана функция $f(\omega)$, а также типы X и Y, такие, что $f: \Omega(X) \to Y$.

1. Значение неотрицательного целого числа, записанного в позиционной системе счисления с основанием p $(1 < p \le 10)$, $f: \Omega_1(\{0, 1, ..., p - 1\}) \to N_0$.

2. Число пар соседних элементов последовательности ω, удовлетворяющих условию $P(x_1, x_2)$; $f: \Omega(X) \to N_0$. Варианты:

а) $X = N$, $P(a, b) = (a > b)$;

б) $X = N$, $P(a, b) = (НОД(a, b) = 1)$;

в) $X = N$, $P(a, b) = (e1(a) = e1(b))$, где $e_1(a)$ – число единиц в двоичной записи числа a;

г) $X = N$, $P(a, b) = (Odd(a)$ & $Odd(b))$.

3. Максимум среди сумм пар соседних элементов, т.е. $max\{x_i + x_{i+1}: 1 \le i < n\}$, где $n = length(\omega)$ – длина последовательности $\omega = x_1\ x_2\ \ldots\ x_n$. Здесь $f: \Omega_2(Z) \to Z$.

4. Максимум среди сумм троек соседних элементов, т. е. $max\{x_i + x_{i+1} + x_{i+2} : 1 \le i < n - 1\}$, где $n = length(\omega)$ – длина последовательности $\omega = x_1\ x_2\ \ldots\ x_n$. Здесь $f: \Omega_3(Z) \to Z$.

5. Размах (разность максимального и минимального) значений элементов последовательности, $f: \Omega(Z) \to N_0$.

6. Число локальных максимумов, $f: \Omega(R) \to N_0$. (Элемент последовательности называется локальным максимумом, если у него нет соседа, большего, чем сам элемент. Например, в любой одноэлементной последовательности ровно один локальный максимум.)

7. Число элементов числовой последовательности, больших всех предыдущих элементов, $f: \Omega(N) \to N_0$.

8. Число изменений знака (переходов через нуль) числовой последовательности, $f: \Omega(Z) \to N_0$.

9. Значение записанного по возрастающим степеням многочлена в точке t, $f: \Omega(R) \to R$, $f(\Delta) = 0$.

10. Значение производной записанного по возрастающим степеням многочлена в точке t, $f: \Omega(R) \to R$, $f(\Delta) = 0$.

11. Значение k-й производной записанного по убывающим степеням многочлена в точке t, $\Omega(R) \to R$, $f(\Delta) = 0$.

12. Размерность пространства, натянутого на последовательность векторов R_2, $f: \Omega(R_2) \to \{0, 1, 2\}$.

13. Последовательность обладает заданным свойством, $f: \Omega_1(Z) \to B$. Варианты свойства:

а) последовательность возрастает;

б) все элементы последовательности одного знака;

в) последовательность знакочередующаяся;

г) последовательность ограничена сверху заданным порогом d, т. е. состоит из элементов x_i, таких, что $x_i \le d$, где d – заданное число;

д) последовательность «пилообразна», т. е. каждый элемент является строгим локальным минимумом или максимумом (строгий локальный минимум (максимум) не имеет соседа, равного или меньшего (большего), чем сам этот элемент).

14. Количество отрезков с заданным свойством, $f: \Omega_1(Z) \to N_0$. Варианты свойства отрезка:

а) возрастающий; б) состоящий из равных элементов;

в) знакочередующийся; г) знакопостоянный;

д) ограниченный сверху заданным порогом d, т. е. состоящий из элементов x_i, таких, что $x_i \le d$, где d – заданное число;

е) ограниченный снизу и сверху заданными порогами d_1 и d_2, т. е. состоящий из элементов x_i, таких, что $d_1 \le x_i \le d_2$, где d_1 и d_2 — заданные числа ($d_1 < d_2$).

ж) «пилообразный» (см. задание 13.д).

15. Максимальная длина отрезков с заданным свойством, $f: \Omega_1(Z) \to N$. Варианты свойства отрезка такие же, как в задании 14.

16. Средняя длина отрезков с заданным свойством, $f: \Omega_1(Z) \to R_0$. Варианты свойства отрезка такие же, как в задании 14.

17. Номер первого элемента с заданным свойством, $f: \Omega(Z) \to N_0$. Варианты свойства элемента:

а) максимальный элемент;

б) равен заданному числу x_0;

в) последний элемент отрезка с заданным свойством; свойство отрезка – по вариантам задания 14.

18. Номер последнего элемента с заданным свойством, $f: \Omega(Z) \to N_0$. Варианты свойства элемента:

а) максимальный элемент; б) равен заданному числу x_0;

в) первый элемент отрезка с заданным свойством; свойство отрезка – по вариантам задания 14.

19. Среднее значение (заданного вида) элементов последовательности $\omega = x_1x_2\ldots x_n$ длины $n > 0$, $f: \Omega(R) \to R$. Варианты видов среднего значения:

а) среднее арифметическое xa последовательности ω: $xa = (x_1 + x_2 + \ldots + x_n)/n$;

б) среднее гармоническое xg последовательности положительных элементов ω ($f: \Omega(R+) \to R+$): $xg = n/(1/x_1 + 1/x_2 + \ldots + 1/x_n)$;

в) среднее логарифмическое xl последовательности ω: $xl = x_1 + x_2/2 + \ldots + x_n/n$;

г) среднее хронологическое xh последовательности ω: $xa = (x_1/2 + x_2 + \ldots + x_{n-1} + x_n/2)/(n - 1)$;

д) средний квадрат $x2$ последовательности ω $x2 = (x_1^2 + x_2^2 + ... + x_n^2)/n$;

е) дисперсию d последовательности ω $d = ((x_1 - xa)^2 + (x_2 - xa)^2 + ... + (x_n - xa)^2)/n$, где xa определено в п.19а;

ж) взвешенное среднее арифметическое xva последовательности ω с положительными весами $v_1v_2...v_n$ $(f: \Omega(R_2) \to R)$: $xva = (x_1v_1 + x_2v_2 + ... + x_nv_n))/(v_1 + v_2 + ... + v_n)$;

з) взвешенное среднее гармоническое xvh последовательности положительных элементов ω с положительными весами $v_1v_2...v_n$ $(f: \Omega((R+) \times (R+)) \to R+)$: $xvh = (v_1 + v_2 + ... + v_n)/(v_1/x_1 + v_2/x_2 + ... + v_n/x_n)$;

и) взвешенный средний квадрат $xv2$ последовательности ω с положительными весами $v_1v_2...v_n$ $(f: \Omega(R2) \to R)$ $xv2 = (x_1^2v_1 + ... + x_n^2v_n)/(v_1 + v_2 + ... + v_n)$;

к) взвешенную дисперсию dv последовательности ω с положительными весами $v_1v_2...v_n$ $(f: \Omega(R2) \to R)$ $dv = ((x_1 - xva)^2v_1 + ... + (x_n - xva)^2v_n))/(v_1 + v_2 + ... + v_n)$, где xva определено в п. 19ж.

20. Длина возрастающего отрезка с заданным дополнительным свойством, $f: \Omega(Z) \to N$. Варианты свойства отрезка:

а) с максимальным средним значением элементов; вид среднего – по вариантам задания 19;

б) с минимальным средним значений элементов; вид среднего – по вариантам задания 19;

в) с максимальным “размахом” значений элементов отрезка (о размахе см. п. 5);

г) с минимальным “размахом” значений элементов (о размахе см. п. 5);

д) с максимальной “крутизной” (крутизна возрастающего отрезка $x_ix_{i+1}...x_j$ при $j > i$ есть $(x_j - x_i)/(j - i)$.

21. Длина знакочередующегося отрезка числовой последовательности с дополнительным свойством, $f: \Omega(N) \to N$. Варианты свойства отрезка:

а) с максимальным “размахом” значений элементов (о размахе см. п. 5);

б) с минимальным “размахом” значений элементов (о размахе см. п. 5);

в) с максимальным средним значений элементов; вид среднего – по вариантам задания 19;

г) с минимальным средним значений элементов; вид среднего – по вариантам задания 19.

5 Требования к лабораторной работе

1Алгоритм решения задачи должен быть однопроходным и не должен использовать дополнительную память, пропорциональную размеру входного файла (использование массивов для хранения входной последовательности запрещено);

2Рекомендуется использовать схему вычисления индуктивной функции, в которой на каждом шаге основного цикла из входного файла читается и обрабатывается один элемент последовательности;

3Программа должна выводить поочередно обрабатываемые элементы входного файла, а также другие (промежуточные) данные в виде, позволяющем проанализировать и понять процесс формирования окончательного результата;

4Выходные данные должны выводиться на экран и в выходной файл (в режиме формирования отчета);

5Требуется протестировать основные особые случаи входных данных (например, входной файл пуст или содержит лишь один элемент), корректную обработку последнего элемента файла, проверить, учитывается ли наличие стационарного значения индуктивной функции и т. п.

ЛАБОРАТОРНАЯ РАБОТА 6. ОБРАБОТКА ОДНОМЕРНЫХ МАССИВОВ

1 Цель работы

Изучение массивов, индексации в массивах, способов передачи массивов в функции.

2 Теоретический материал

2.1 Массивы

Очень часто возникает потребность хранить информацию, которую неудобно (невозможно) хранить в виде отдельных переменных. Например, статистические данные, строки (последовательность значений символьного типа) и т.д. **Массив** – это **структура данных**, которая содержит множество значений одного типа (**элементы массива**), при этом все элементы массива расположены в памяти последовательно и доступ к конкретному элементу осуществляется через его **индекс** (порядковый номер). Массивы можно описать следующим образом:

```
Тип_элементов_массива имя_массива[количество элементов];
```

Типом элементов массива может быть любой тип *C++* (включая массивы).

Примеры:

```
int arr1[10];          //Массив из 10 элементов, тип элементов – int
int arr2[15][5];       //Массив из 15 элементов, тип элементов – массив из
                       // 5 элементов типа int
char* arr3[30];        //Массив из 30 элементов, тип элементов – указатель
                       // на char
```

Число индексов при определении массивов не ограничено. Важен суммарный объем данных в программе, на который накладываются ограничения. При объявлении массива выражение, определяющее размер массива, должно быть **константным выражением**, то есть выражением, значение которого известно на этапе компиляции.

Нумерация элементов массива в *C++* начинается с нуля. Обращение к конкретному элементу массива выполняется с помощью оператора *[]*, внутри скобок указывается индекс элемента. То есть элементами массива *arr1* (размер массива - *10*) являются *arr1[0]*, *arr1[1]*,..,*arr1[9]*.

Имя массива практически во всех контекстах трактуется компилятором как константный указатель на первый элемент (с индексом ноль), кроме случая использования оператора *sizeof*. С помощью оператора *sizeof* можно вычислить

размер массива: *sizeof(имя_массива) = sizeof(тип_элементов)×количество_элементов*.

Фактически индекс элемента массива означает смещение от начала массива (первый элемент размещен в начале массива, поэтому его смещение - ноль). Обращаться к элементам массива можно и с помощью указателей: *arr[i] = *(arr+i) = *(i+arr) = i[arr]*.

При обращении к элементам массива (для чтения или записи) диапазон допустимых значений не проверяется ни на этапе компиляции, ни во время выполнения программы. Таким образом присвоение значения элементу *arr1[15]* не вызовет ошибок, однако, при этом произойдет запись в область памяти, в которой хранились другие данные (или их часть), это может привести к повреждению данных и другим нежелательным последствиям. Во многих вредоносных программах использована возможность выхода за границы массива. За корректностью обращения к элементам массива должен следить программист.

В отличие от базовых типов, нельзя присваивать один массив другому, однако, можно присваивать значения поэлементно (с помощью цикла, при работе с массивами особенно удобно использовать цикл **for**). В примере 6.1. выполняется поэлементное заполнение массивов *arr1* и *arr2*, каждому элементу присваивается значения индекса. Индексация выполнена двумя способами – с помощью оператора [] (для *arr1*) и указателей (для *arr2*).

Программа 6.1. arrays.cpp

```
#include <iostream>
const int MAX_SIZE = 100;
int main() {
      int arr1[MAX_SIZE];
      int arr2[MAX_SIZE];

      for (int i = 0; i < MAX_SIZE; ++i) {
            arr1[i] = i;
      }


      for (int *p = arr2, i = 0; i < MAX_SIZE; ++i, ++p) {
            *p = i;
      }
  }
```

Инициализировать массив можно списком или поэлементно. Инициализация списком допустима только при объявлении массива:

```
int array[4]{ 1, 2, 3, 4 };
int array[14]{ 1, 2, 3, 4 };      // Остальным элементам будет присвоено
                                  // значение 0
int array[]{ 1, 2, 3, 4 };        // array – массив из 4х элементов
int array[14];                    // элементы массива содержат “мусор”
```

2.2 Передача массивов в функции

Используемые в программе массивы могут передаваться при вызове функций в качестве фактических параметров. Можно использовать несколько способов передачи массивов в функцию:

```
void f1(int arr[15], int n);
void f2(int arr[], int n);
void f3(int *arr, int n);
void f4(int(&arr)[10]);
```

В функции *f1*, *f2* и *f3* аргумент *arr* является указателем на начало массива, поэтому теряется информация о размере массива, длина передаваемого массива может быть любой (в *f1* может быть передан массив, размер которого отличен от *15*, компилятор не проверяет совпадения размеров, поэтому лучше не использовать такой способ).

```
void f2(int arr[], int n) {
     // sizeof(arr) = sizeof(int*)
}
```

Так как в функцию передается адрес, по которому расположены элементы массива, то изменения значений элементов массива в функции отразятся на исходном массиве.

В *f4* массив передается по ссылке (как и в случае с обычными переменными, изменения, вносимые в массив в функции, отразятся на переданном массиве). При передаче по ссылке размер массива можно вычислить с помощью оператора *sizeof* .

```
void f4(int(&arr)[10])) {
     // sizeof(arr)/sizeof(int) = 10
}
```

Однако в этом случае передавать только массивы, размер которых определен в заголовке функции.

```
int array1[10]; // можно передать в f4
int array2[9];  // нельзя передать в f4
```

С помощью директив *typedef* или *using* можно задать псевдоним типа ссылки на массив определенного размера, для более удобной записи передачи массива в функцию.

```
typedef int array1_t[15];
void f5(array1_t &arr);

using array2_t = int[15];
void f6(array2_t &arr);
```

Заметим, что массив нельзя передать в функцию по значению, то есть копию массива. Если внутри функции массив не меняется, то для передачи массива в функцию следует использовать константный указатель или ссылку:

```
void f1(const int arr[15]);
void f2(const int arr[]);
void f3(const int *arr);
void f4(const int(&arr)[10]);
```

3 Пример выполнения лабораторной работы

Задание: Написать функцию, которая выполняет сортировку массива вставками (Insertion sort).

Массив *arr* размера *n* является отсортированным по неубыванию, если выполняется свойство: $arr[0] \le arr[1] \le ... \le arr[n]$. Существует множество алгоритмов сортировки, которые рассматриваются в курсе «Алгоритмы и структуры данных» в третьем семестре. Алгоритм сортировки вставками относится к классу простых алгоритмов сортировки, с ***квадратичным*** временем работы, то есть количество операций пропорционально квадрату размера массива.

Опишем алгоритм сортировки вставками. Считаем, что на каждом шаге i ($i \in [1; n-1]$) часть массива $arr[0...i]$ уже отсортирована (на первом шаге часть из одного элемента $arr[0]$ является отсортированной). Затем выполняем вставку элемента $arr[i]$ в отсортированную часть. Для этого выполняем сдвиг элементов $(i-1...k: arr[k] \le arr[i] < arr[k+1])$ на один элемент вправо. На освободившее место помещаем элемент $arr[i]$.

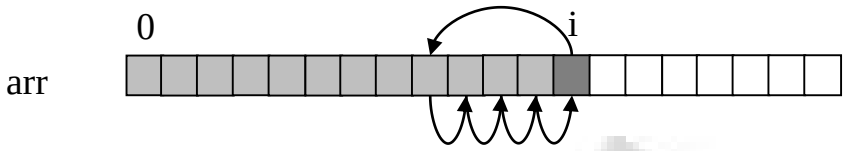


Рис. 6.1. Добавление элемента *arr[i]* в отсортированную часть *[0…i-1]*.

Программа 6.2.

```
//Программа выполняет сортировку массива, используя алгоритм
// сортировки вставками.
#include <iostream>
#include <fstream>

using namespace std;

using uint = unsigned int;
const uint MAX_SIZE = 100;

uint read_arr(istream &in, int *arr);
void write_arr(ostream &out, const int *arr, uint n);
void insertion_sort(int *arr, uint n);

int main()
{
	setlocale(0, "");

	int arr[MAX_SIZE];
	ifstream fin("in_array.txt");

	if (!fin) {
		cout << "Не удалось открыть файл!\n";
		return 1;
	}

	uint len = read_arr(fin, arr);
	cout << "Считано " << len << " элементов: ";
	write_arr(cout, arr, len);

	insertion_sort(arr, len);

	cout << "Отсортированный массив: ";
```

```
	write_arr(cout, arr, len);

	return 0;
}

uint read_arr(istream &in, int *arr) {
	uint len = 0;
	for (; (len < MAX_SIZE) && (in >> arr[len]); ++len);
	return len;
}

void write_arr(ostream &out, const int *arr, uint len) {
	for (int i = 0; i < len; ++i)
		out << arr[i] << " ";
	cout << endl;
}

void insertion_sort(int *arr, uint len) {

	for (int cur = 1; cur < len; cur++)
	{
		for (int i = cur - 1; (i >= 0) && (arr[i] > arr[i + 1]); --i)
			swap(arr[i + 1], arr[i]);

		write_arr(cout, arr, len);
	}
}
```

Пример вывода программы 6.2:

```
Считано 11 элементов: 9 10 2 4 56 78 23 10 87 67 54
9 10 2 4 56 78 23 10 87 67 54
2 9 10 4 56 78 23 10 87 67 54
2 4 9 10 56 78 23 10 87 67 54
2 4 9 10 56 78 23 10 87 67 54
2 4 9 10 56 78 23 10 87 67 54
2 4 9 10 23 56 78 10 87 67 54
2 4 9 10 10 23 56 78 87 67 54
2 4 9 10 10 23 56 78 87 67 54
2 4 9 10 10 23 56 67 78 87 54
2 4 9 10 10 23 54 56 67 78 87
Отсортированный массив: 2 4 9 10 10 23 54 56 67 78 87
```

4 Задания к лабораторной работе

1 Проверить, можно ли получить последовательность значений элементов заданного массива *a[n]* из последовательности значений элементов другого заданного массива *b[m]* “вычеркиванием” некоторых элементов второй последовательности.

2 Проверить выполнение следующего свойства массивов *a[n]* и *b[m]*: для каждого элемента *a[i] (i = 0...n-1)* найдется равный ему элемент массива *b*.

3 Проверить выполнение следующего свойства упорядоченных по возрастанию массивов *a[n]* и *b[m]*: для каждого элемента *a[i] (i = 0...n-1)* найдется равный ему элемент массива *b*.

4 Даны два упорядоченных массива: *a[n]* и *b[m]*. Известно, что среди элементов этих массивов нет совпадающих. Найти количество элементов, встречающихся как в массиве *a*, так и в *b*.

5 Рассматривая массивы *a[n]* и *b[m]* как последовательности цифр десятичной записи некоторых неотрицательных чисел, получить массив *c[k]* – аналогичное представление для суммы этих двух чисел.

Задания 6-17 взяты из книги А. Шень. Программирование. теоремы и задачи. - 4е изд. М.: МЦИНМО, 2011

6 В массиве *a[n]* встречаются по одному разу все числа от *0* до *n*, кроме одного. Найти пропущенное число за время порядка *n* и с конечной дополнительной памятью.

7 Дан массив *a[n]* и число $m \le n$. Для каждого участка из *m* стоящих рядом членов (таких участков, очевидно, *n-m+1*) вычислить его сумму. общее число действий должно быть порядка *n*.

8 Коэффициенты многочлена лежат в массиве *a[n]* (*n* - степень многочлена). Вычислить значение этого многочлена в точке *x*, то есть $a[n] \times x^n + .. + a[1] \times x + a[0]$.

9 В целочисленных массивах *a[k]* и *b[l]* хранятся коэффициенты двух многочленов степеней *k* и *l*. Поместить в массив *c[m]* коэффициенты их произведения. ($m=k+l$; элемент массива с индексом *i* содержит коэффициент при степени *i*).

10 Даны два целочисленных массива $x[0]<x[1]<...<x[p]$ и $y[0]<y[1]<...<y[q]$. Найти количество общих элементов в этих массивах, то есть количество тех целых *t*, для которых $t=x[i]=y[j]$ для некоторых *i* и *j*. Число действий должно быть порядка $p+q$.

11 Даны два целочисленных массива $x[0] \le x[1] \le .. \le x[p]$ и $y[0] \le y[1] \le .. \le y[q]$. Найти количество общих элементов в этих массивах, то есть количество тех целых t, для которых $t=x[i]=y[j]$ для некоторых i и j. Число действий должно быть порядка $p+q$.

12 Даны два целочисленных массива $x[0] \le x[1] \le .. \le x[p]$ и $y[0] \le y[1] \le .. \le y[q]$. Найти число различных элементов среди $x[0],x[1],...,x[p], y[0], y[1],..,y[q]$. Число действий должно быть порядка $p+q$.

13 Даны два массива $x[0] \le x[1] \le .. \le x[p]$ и $y[0] \le y[1] \le .. \le y[q]$. "Соединить" их в массив $z[0] \le z[1] \le .. \le z[r]$ ($r=p+q$; каждый элемент должен входить в массив z столько раз, сколько раз он входит в общей сложности в массивы x и y). Число действий должно быть порядка r.

14 Даны два массива $x[0] \le x[1] \le .. \le x[p]$ и $y[0] \le y[1] \le .. \le y[q]$. Найти "пересечение", то есть массив $z[0] \le z[1] \le .. \le z[r]$, содержащий их общие элементы, причем кратность каждого элемента в массиве z равняется минимуму из его кратности в массиве x и y. Число действий должно быть порядка $p+q$.

15 Даны два массива $x[0] \le x[1] \le .. \le x[p]$ и $y[0] \le y[1] \le .. \le y[l]$ и число q. Найти сумму вида $x[i]+y[j]$, наиболее близкую к числу q. Число действий должно быть порядка $p+q$. Дополнительная память - фиксированное количество целых переменных, сами массивы менять не разрешается.

16 Некоторое число содержится в каждом из трёх целочисленных массивов $x[0] \le x[1] \le .. \le x[p]$, $y[0] \le y[1] \le .. \le y[q]$ и $z[0] \le z[1] \le .. \le z[r]$. Найти одно из таких чисел. Число действий должно быть порядка $p+q+r$.

17 Двоичный поиск. Дана последовательность $x[0] \le x[1] \le .. \le x[n]$ целых чисел и число a. Выяснить, содержится ли число a в этой последовательности, то есть существует ли i из $0..n$, для которого $x[i]=a$.

5 Требования к лабораторной работе

1. Обязательно наличие следующих функций: функция ввода массива из файла, вывода массива в файл/на экран и функция, которая решает поставленную задачу;
2. Не использовать глобальных объявлений, кроме объявления констант;
3. В отчет включить описание спецификации реализованных функций.

ЛАБОРАТОРНАЯ РАБОТА 7. ОБРАБОТКА ДВУМЕРНЫХ МАССИВОВ

1 Цель работы

Изучение двумерных массивов и способов передачи двумерных массивов в функции.

2 Теоретический материал

2.1 Двумерные массивы

В языке *C++* нет специального типа «двумерный (многомерный) массив». Двумерный массив – это массив, элементами которого являются массивы. Задается многомерный массив следующим образом:

`<Имя типа> <Имя массива>[K1][K2]…[KN]`, где N – размерность массива.

Пример: массив `int arr[4][5]`. Двумерный массив удобно представлять себе, как матрицу (рис. 7.1а), однако в памяти элементы хранятся последовательно (рис. 7.1б).

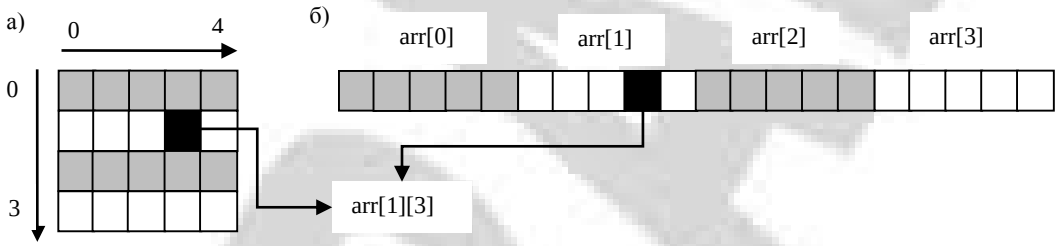


Рис. 7.1. Представление массива int arr[4][5].

Все правила индексации в двумерном массиве аналогичны правилам индексации в одномерном массиве. Рассмотрим массив *int arr[N][M]*. При обращении к элементу *arr[i][j]* мы сначала смещаемся на *i* элементов (каждый элемент – массив размера *M*) массива *arr*, а затем на *j* элементов в массиве *arr[i]*. Имя массива является константным указателем на начало массива. Обратиться к элементу *arr[i][j]* можно и с помощью указателей: `*(*(arr + i) + j)`. После выполнения сложения *arr+i* будет выполнен сдвиг на $i \times (M \times sizeof(\text{int}))$ байт, а всего – на $i \times (M \times sizeof(\text{int})) + j \times sizeof(\text{int})$.

Инициализировать двумерный массив можно также списком или поэлементно. Инициализация списком допустима только при объявлении массива: `int arr[4][2]{ { 1, 2 },{ 3, 4 },{ 5,6 },{ 7,8 } }`.

2.2 Передача двумерных массивов в функцию

Примеры заголовков функций при передаче двумерного массива:

```
void f1(int(*arr)[15], int n);
void f2(int arr[][15], int n);
void f3(int arr[10][15], int n);
```

Также, как и в случае с одномерным массивом, в функции *f1*, *f2* и *f3* могут переданы массивы любой длины, однако, элементами этих массивов должны быть массива размера *15*. Все изменения с массивом внутри функции отразятся на исходном массиве.

Обратите внимание, что для правильного обращения к элементу массива по индексу необходимо знать размер элементов массива (чтобы знать смещение от начала массива), поэтому вторая размерность обязательно должна быть указана, а в случае многомерных массивов должны быть указаны все размерности, кроме первой.

Теперь рассмотрим передачу массива по ссылке (как и в случае с обычными переменными, изменения, вносимые в массив в функции, отразятся на переданном массиве):

```
void f4(int(&arr)[10][15]);
```

В этом случае передавать только массивы, размер которых определен в заголовке функции.

```
int array1[10][15];   // можно передать в f4
int array2[9][14];    // нельзя передать в f4
```

С помощью директив typedef или using можно задать псевдоним типа ссылки на массив определенного размера, для более удобной записи передачи массива в функцию.

```
typedef int array1_t[10][15];
void f5(array1_t &arr);

using array2_t = int[10][15];
void f6(array2_t &arr);
```

3 Пример выполнения лабораторной работы

Задание: В матрице A (размерности m × n) найти минимальное значение среди максимальных элементов строк. Исходные данные и результаты расположены в файлах.

Для решения задачи найдем для каждой строки максимальный элемент, а затем выберем из полученных значений минимальное.

Программа 7.1.

```
//Программа находит минимальное значение среди максимальных
// элементов строк матрицы.
#include <iostream>
#include <climits>
#include <clocale>

#define n_elements(a) (sizeof(a)/sizeof(a[0])) // число элементов в мас-
сиве

using namespace std;

using uint = unsigned int;
const uint MAX_COL = 5;
const uint MAX_ROW = 4;

using row_t = int[MAX_COL];
using array_t = int[MAX_ROW][MAX_COL];

int max_num(const row_t &arr);
int min_max(const array_t &arr);

void write_arr(ostream &out, const row_t &arr);
void write_arr(ostream &out, const array_t &arr);

int main()
{
      setlocale(0, "");

      array_t arr = { { 90, 91, 92, 93, 94 },
                      { 14, 13, 12, 11, 10 },
                      { 20, 21, 22, 23, 24 },
                      { 34, 33, 32, 31, 30 } };
      cout << "Входной массив: " << endl;
      write_arr(cout, arr);

      int arr_min_max = min_max(arr);

      cout <<"Минимальное значение среди максимальных элементов строк
матрицы = " << arr_min_max << endl;
```

```
	return 0;
}

int max_num(const row_t &arr) {
	uint len = n_elements(arr);

	int cur_max = INT_MIN;
	for (uint i = 0; i < len; i++) {
		if (arr[i] > cur_max)
			cur_max = arr[i];
	}

	return cur_max;
}

int min_max(const array_t &arr) {
	uint row_num = n_elements(arr);
	int cur_min_max = max_num(arr[0]);

	for (uint i = 1; i < row_num; ++i)
	{
		int row_max = max_num(arr[i]);
		cout << "Максимальный элемент в строке ";
		write_arr(cout, arr[i]);
		cout << " равен " << row_max << endl;

		if (row_max < cur_min_max)
			cur_min_max = row_max;
	}

	return cur_min_max;
}

void write_arr(ostream &out, const row_t &arr) {
	uint len = n_elements(arr);

	for (int i = 0; i < len; ++i)
		out << arr[i] << " ";
}
```

```
void write_arr(ostream &out, const array_t &arr) {
      uint row_num = n_elements(arr);

      for (int i = 0; i < row_num; ++i) {
            write_arr(out, arr[i]);
            cout << endl;
      }
}
```

Пример вывода программы 7.1:

```
Входной массив:
90 91 92 93 94
14 13 12 11 10
20 21 22 23 24
34 33 32 31 30
Максимальный элемент в строке 14 13 12 11 10  равен 14
Максимальный элемент в строке 20 21 22 23 24  равен 24
Максимальный элемент в строке 34 33 32 31 30  равен 34
Минимальное значение среди максимальных элементов строк матрицы =
14
```

4 Задания к лабораторной работе

1. Матрица является ортонормированной, если скалярное произведение каждой пары различных строк равно *0*, а скалярное произведение каждой строки на себя равно *1*. Определить, является ли заданная матрица *A* размера $N \times M$ ортонормированной.

2. Задана квадратная таблица размера $N \times N$. Преобразовать ее, осуществив поворот элементов вокруг ее центра на 90° по часовой стрелке.

3. Элемент матрицы *A* размера $N \times M$ называют седловой точкой, если он является наименьшим в своей строке и одновременно наибольшим в своем столбце или является наибольшим в своей строке и одновременно наименьшим в своем столбце. Найти индексы всех седловых точек матрицы.

4. Найти сумму элементов заштрихованной области таблицы *A* размера $N \times N$ (рис. 8. 1).

а) для любого $N>1$

X	X	X	X	X	1
X				X	2
X				X	3
X				X	4

б) для любого $N>1$

	X	X	X	
	X	X	X	
	X	X	X	

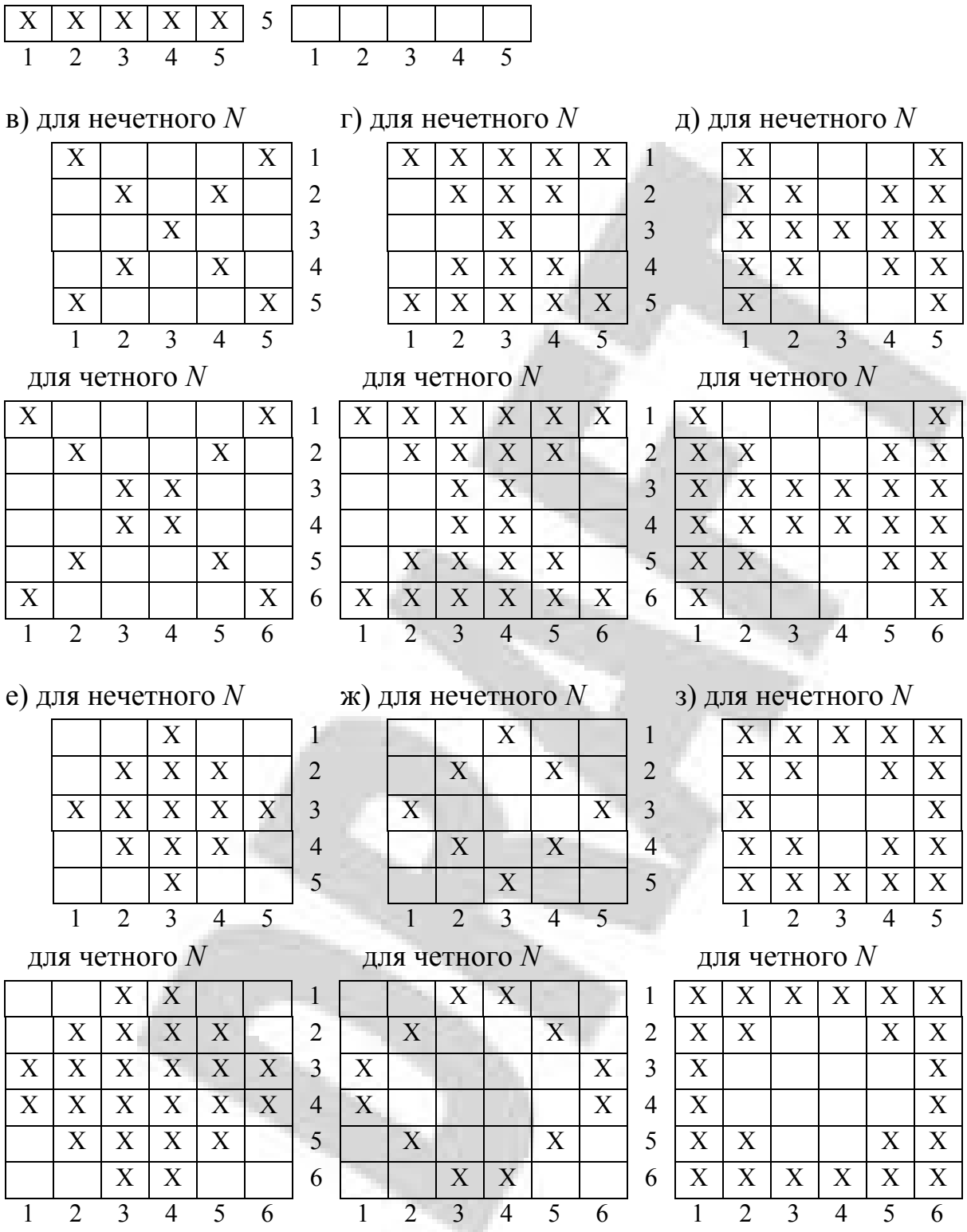


Рис. 7. 2

5. Дана матрица A размера $N \times M$. Переставляя ее строки и столбцы, добиться того, чтобы наибольший элемент (какой-либо) оказался в левом верхнем углу.

6. Найти наибольший элемент заштрихованной области таблицы *A* размера $N \times N$ (рис. 8. 2).

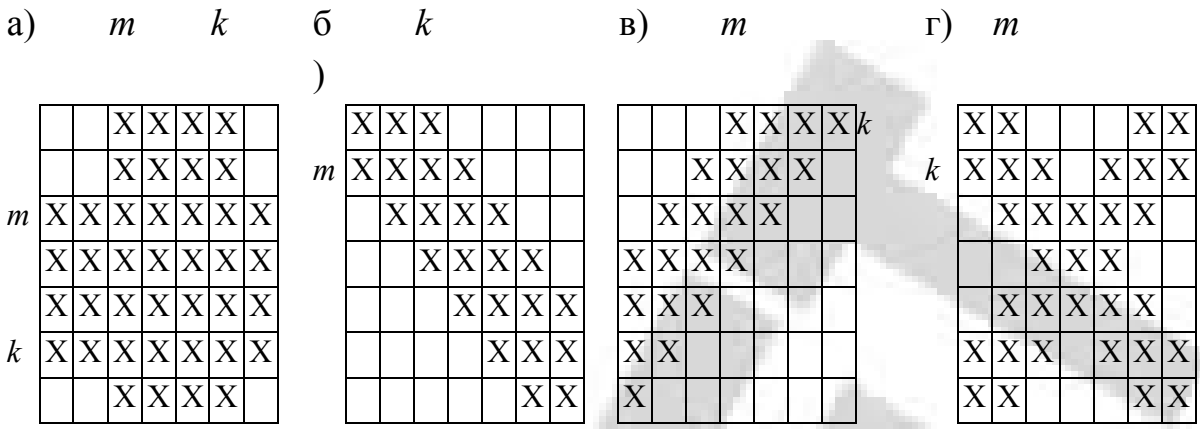


Рис. 7. 3

7. Имеется таблица размера N × N с именем Табл результатов некоторого спортивного турнира, в котором участвовало N команд (N > 2). Элемент таблицы Табл[i, j] = 2, если i-я команда выиграла у j-й. Если i-я команда проиграла j-й, то Табл[i, j] = 0. Если i-я и j-я команды сыграли вничью, то Табл[i, j] = 1. Кроме того, Табл[i, i] = 0. Перечислить команды в порядке невозрастания набранной ими суммы очков.

5 Требования к лабораторной работе

Аналогичны требованиям к лабораторной работе 6.

СПИСОК ЛИТЕРАТУРЫ

1. Ивановский С.А. Разработка корректных программ: Учеб.пособие. - СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2003.
2. Павловская Т.А. С/С++. Программирование на языке высокого уровня: Учеб. для вузов по направлению "Информатика и вычисл. Техника". - СПб. : Питер, 2003.
3. Павловская Т.А., Щупак Ю.А. С/С++. Структурное программирование: практикум. - СПб. : Питер, 2004.
4. Дейтел Х.М. Как программировать на С++. - М. : Бином, 2001, 2005.
5. Керниган Б., Ритчи Д. Язык программирования С, 2-е издание. - М.: Вильямс, 2006, 2007.
6. Подбельский В. Язык С++: Учеб. пособие для вузов. - М.: Финансы и статистика, 2001.
7. Шилдт Г. Самоучитель С++. - СПб.: БХВ-Петербург, 2000.
8. Прата С. Язык программирования С++: лекции и упражнения: учеб. - Киев: Диа Софт, 2001, 2003. М.: Вильямс, 2015.
9. Липпман С. Б. Язык программирования С++. Базовый курс, 5-е издание. - М.: Вильямс, 2014.
10. Эккель Б. Философия С++. Введение в стандартный С++: монография. - СПб.: Питер, 2004.
11. Голуб. А.И. Правила программирования на С и С++. - М.: Бином, 1996.

ОГЛАВЛЕНИЕ

Ивановский Сергей Алексеевич,
Лисс Анна Александровна,
Самойленко Владимир Петрович,
Шолохова Ольга Михайловна

Процедурное программирования на языке C++
Учебное пособие

Редактор

Подписано в печать 00.00.00. Формат 60×84 1/16.
Бумага офсетная. Печать цифровая. Печ. л. 0,0.
Гарнитура «Times New Roman». Тираж 000 экз. Заказ 000.

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5