Огнёва М.В., Кудрина Е.В.

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

ЧАСТЬ 1

УДК 681.3.026(076.1)
ББК 32.973-01я73
О38

Огнёва М.В., Кудрина Е.В.
О38 Основы программирования на языке С++: Учеб. пособие в 2 ч. Часть 1. - Саратов: Изд-во "Научная книга", 2008. - 100 с.
ISBN 978-5-9758-0895-0

Данное пособие представляет собой учебно-методическую разработку по изучению основ программирования на языке С++. Пособие состоит из двух частей. Первая часть пособия содержит 6 разделов, в которых рассматриваются такие базовые средства языка С++ как операции, операторы, функции, указатели и массивы, и 4 приложения. Каждый раздел пособия содержит: теоретический материал, примеры решения типовых задач и набор упражнений, предназначенных для закрепления материала. В приложениях содержится справочная информация.

Пособие предназначено для студентов естественно-научных факультетов, изучающих язык С++ с «нуля» в рамках дисциплин компьютерного цикла. Мы надеемся, что пособие окажется полезным и для преподавателей дисциплин компьютерного цикла при подготовке и проведении соответствующих занятий.

Рецензенты:

Федорова А.Г., кандидат физико-математических наук, доцент кафедры информатики и программирования, декан факультета компьютерных наук и информационных технологий Саратовского государственного университета им. Н.Г. Чернышевского

Кондратов Д. В., кандидат физико-математических наук, доцент кафедры математики и статистики, проректор по информатизации Поволжской академии государственной службы им. П.А. Столыпина

УДК 681.3.026(076.1)
ББК 32.973-01я73

Работа издана в авторской редакции

ISBN 978-5-9758-0895-0

Данное пособие представляет собой учебно-методическую разработку по изучению основ программирования на языке C++ и состоит из двух частей. Пособие не претендует на полноту изложения материала, для этого существуют справочники, документация и контекстная помощь.

Первая часть пособия содержит 6 разделов и 4 приложения.

В первом разделе пособия рассматриваются базовые элементы языка C++: стандартные типы данных, константы и переменные, организация потокового ввода/вывода данных, операции и выражения.

Во втором разделе пособия рассматриваются принципы проектирования, разработки и применения функций для решения практических задач.

В третьем разделе пособия рассматриваются операторы языка C++: следования, ветвления, цикла и безусловного перехода.

В четвертом разделе пособия рассматриваются рекуррентные соотношения, а в пятом – приемы вычисления конечных и бесконечных сумм. Данные разделы предназначены для закрепления навыков программирования на языке C++.

В шестом разделе пособия рассматриваются одномерные и двумерные массивы (статические и динамические).

Каждый раздел пособия содержит: теоретический материал, примеры решения типовых задач и набор упражнений, предназначенных для закрепления материала. В приложениях содержится справочная информация.

Пособие предназначено для студентов естественно-научных факультетов, изучающих язык C++ с «нуля» в рамках дисциплин компьютерного цикла. Мы надеемся, что пособие окажется полезным и для преподавателей дисциплин компьютерного цикла при подготовке и проведении соответствующих занятий.

Материалы этого пособия использовались на лекционных и практических занятий со студентами факультета компьютерных наук и информационных технологий и механико-математического факультета Саратовского государственного университета им. Н.Г. Чернышевского.

Авторы благодарят за помощь, поддержку и критические замечания Федорову Антонину Гавриловну и всех сотрудников кафедры информатики и программирования и кафедры математической кибернетики и компьютерных наук Саратовского государственного университета им. Н.Г. Чернышевского, принимавших участие в апробации материала, изложенного в данном пособии.

1. БАЗОВЫЕ ЭЛЕМЕНТЫ ЯЗЫКА С++

1.1. Состав языка

Алфавит языка

Алфавит – совокупность допустимых в языке символов. Алфавит языка С++ включает:

1) прописные и строчные латинские буквы и знак подчеркивания;
2) арабские цифры от 0 до 9, шестнадцатеричные цифры от A до F;
3) специальные знаки: " { } , | ; [ ] ( ) + - / % * . \ ' : ? < = > ! & # ~ ^
4) пробельные символы: пробел, символ табуляции, символ перехода на новую строку.

Из символов алфавита формируются лексемы языка: идентификаторы, ключевые (зарезервированные) слова, знаки операций, константы, разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами такими, как разделители или знаки операций. В свою очередь, лексемы входят в состав выражений (выражение задает правило вычисления некоторого значения) и операторов (оператор задает законченное описание некоторого действия).

Замечание. Некоторые символы, например, буквы русского алфавита, не используются в С++, но их можно включать в комментарии и символьные константы.

Идентификаторы

Идентификатор – это имя программного объекта: константы, переменной, метки, типа, экземпляра класса, функции и поля в записи. Идентификатор может включать латинские буквы, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, myname, myName и MyName — три различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы внутри имен не допускаются. Язык С++ не налагает никаких ограничений на длину имен, однако для удобства чтения и записи кода не стоит делать их слишком длинными.

Ключевые слова

Ключевые слова – это зарезервированные идентификаторы, которые имеют специальное значение для компилятора, например, include, main, int и т.д. Ключевые слова можно использовать только по прямому назначению. С ключевыми словами и их назначением можно ознакомиться в справочной системе С++.

Замечание. Другие лексемы (знаки операций и константы), а также правила формирования выражений и различные виды операторов будут рассмотрены чуть позже.

1.2. Структура программы

Каждая программа на языке С++ состоит из одной или несколько функций. Каждая функция содержит четыре основных элемента:

1) тип возвращаемого значения;
2) имя функции;
3) список параметров, заключенный в круглые скобки (он может быть пустым);

4) тело функции (последовательность операторов), которое помещается в фигурные скобки.

При запуске программы вызывается функция с именем main – это главная функция программы, которая должна присутствовать в программе *обязательно*. Приведем пример простейшей функции main():

```
int main()
{ return 0;}
```

В приведенном примере:

1) тип возвращаемого значения *int* – целый;
2) имя функции *main;*
3) список параметров пуст;
4) тело функции состоит из одного оператора *return 0,* который возвратит значение 0 вызывающему окружению (либо компилятору, либо операционной системе).

Замечание. В более старых версиях C++ для функции *main()* можно было указать тип *void.* Это означает, что данная функция ничего не возвращает. В этом случае оператор *return* не нужен. В современных версиях C++ функция *main()* всегда должна возвращать тип *int,* который сообщает вызывающему окружению о нормальном или сбойном завершении работы программы. А вот список параметров функции *main()* может быть и не пуст, хотя ограничен по количеству параметров.

Теперь рассмотрим пример простейшей программы, которая подсчитывает сумму двух целых чисел, значения которых вводит с клавиатуры пользователь, а результат выводится на экран.

Замечание. Прежде чем вводить программу в память компьютера, следует ознакомиться с принципами работы в специализированной среде разработки Microsoft Visual Studio (см. приложение 1) и с ошибками, возникающими при разработке программ (см. приложение 2).

```
#include <iostream>                          //директива препроцессора
/* пример программы, которая подсчитывает сумму двух целых чисел, значения
которых вводит с клавиатуры пользователь, а результат выводится на экран */
using namespace std;
int main()
{ int a,b;                                   //описание переменных
 cout <<"Введите 2 целых числа" <<endl;      // оператор вывода
 cin >>a  >>b;                               //оператор ввода
 cout << "Их сумма равна " << a+b;           //оператор вывода
 return 0; }
```

Рассмотрим первую строку кода: *#include <iostream>* – это директива препроцессора, которая указывает компилятору на необходимость подключить перед компиляцией содержимое заголовочного файла библиотеки подпрограмм C++. Имя заголовочного файла помещено в скобки, в данном случае это iostream. В этом файле содержатся готовые программные средства для организации форматированного ввода-вывода. В общем случае в программе может быть несколько директив, каждая из которых должна начинаться с новой строки и располагаться вне тела функции.

Замечание. Название библиотеки *iostream* происходит от сокращения следующих слов ***input-output (ввод-вывод) stream*** (поток). *Поток* – это последовательность символов, записываемая или читаемая с устройств ввода-вывода информации каким-либо способом. Поэтому библиотеку *iostream* еще называют библиотекой для организации потокового ввода-вывода данных.

Далее идет *комментарий.* Комментарии используются для кратких заметок об используемых переменных, алгоритме или для дополнительных разъяснений сложного фрагмента кода, т.е. помогают понять смысл программы. Комментарии игно-

рируются компилятором, поэтому они могут использоваться для «скрытия» части кода от компилятора.

В языке C++ существуют 2 вида комментариев:

1) Комментарий, содержание которого умещается на одной строке, начинается с символов // и заканчивается символом перехода на новую строку.
2) Если содержание комментария не умещается на одной строке, то используется парный комментарий, который заключается между символами /* */. Внутри парного комментария могут располагаться любые символы, включая символ табуляции и символ новой строки. Парный комментарий не допускает вложения.

Замечание. В нашем примере вторая и третья строка – парные комментарии.

В четвертой строке содержится директива *using namespace std*, которая означает, что все определенные ниже имена в программе будут относиться к пространству имен std. *Пространством имен* называется область программы, в которой определена некоторая совокупность имен. Эти имена неизвестны за пределами данного пространства имен. В нашем случае это означает, что глобальные (т.е. определенные вне программы имена из библиотеки *iostream*) имена cout, cin и endl определены внутри пространства имен *std*, более того, в нем определены собственные имена, a, b. Пространство имен std – это пространство имен, используемое стандартной библиотекой. Программист вправе определить собственное пространство имен.

Замечание. Каждое имя, которое встречается в программе, должно быть уникальным. В больших и сложных приложениях используются библиотеки разных производителей. В этом случае трудно избежать конфликта между используемыми в них именами. Пространства имен предоставляют простой механизм предотвращения конфликтов имен. Они создают разделы в глобальном пространстве имен, разграничивая область видимости программных объектов и уникальность их имен.

Следующая строка нашей программы представляет собой заголовок функции *main*, которая не содержит параметров и должна возвращать значение типа int. Тело функции помещено в фигурные скобки.

В шестой строке производится объявление двух переменных типа int (целый тип данных). Подробнее встроенные типы языка C++ будут рассмотрены в разделе 1.5, переменные – в разделе 1.7.

Седьмая строка начинается с идентификатора *cout*, который представляет собой объект C++, предназначенный для работы со стандартным потоком вывода, ориентированным на экран. Далее идет операция <<, которая называется операцией вставки или помещения в поток. Данная операция копирует содержимое, стоящее в правой части от символов <<, в объект, стоящий слева от данных символов. В одной строке может быть несколько операций помещения в поток. В нашем случае в результате выполнения команды *cout*<< на экране появляется строка «Введите два целых числа», и курсор перемещается на новую строчку. Перемещение курсора на новую строку происходит за счет использования специализированного слова *endl*, называемого манипулятором: при его записи в поток вывода происходит перевод сообщения на новую строку.

Следующая строка начинается с идентификатора *cin*. Идентификатор *cin* представляет собой объект C++, предназначенный для работы со стандартным потоком ввода, ориентированным на клавиатуру. Операция >> называется операцией извлечения из потока: она читает значения из объекта cin и сохраняет их в перемен-

ных, стоящих справа от символов >>. В результате выполнения данной операции с клавиатуры будут введены два значения, первое из которых сохраняется в переменной *a*, второе – в переменной *b*.

Далее идет оператор вывода, в результате выполнения которого на экране появится сообщение «Их сумма равна ...». Например, если с клавиатуры были введены числа 3 и 4, то на экране появится сообщение «Их сумма равна 7».

Замечания

1) Если у вас старый компилятор, то заголовок вида #include <iostream> придется заменить на #include <iostream.h> и убрать строчку using namespace std.
2) Из языка С язык С++ унаследовал функции ввода-вывода scanf() и printf(), их можно использовать подключив файл stdio.h.

1.3. Стандартные типы данных С++

Данные – это формализованное (понятное для компьютера) представление информации. В программах данные фигурируют в качестве значений переменных или констант. Данные, которые не изменяются в процессе выполнения программы, называются константами. Данные, объявленные в программе и изменяемые в процессе ее выполнения, называются переменными. Особенности представления данных:

1) каждое значение (переменной, константы и результата, возвращаемого функцией) имеет свой тип;
2) тип переменной или константы объявляется при их описании;
3) тип определяет:
   - внутреннее представление данных в памяти компьютера;
   - объем оперативной памяти, необходимой для размещения значения данного типа;
   - множество значений, которые могут принимать величины этого типа;
   - операции и функции, которые можно применять к величинам этого типа.

Все типы данных можно разделить на простые и составные. К простым относятся: стандартные (целые, вещественные, символьные, логический) и определенные пользователем (перечислимые типы). К составным типам относятся массивы, строки, объединения, структуры, файлы и объекты. Кроме этого существует специальный тип void, который не предназначен для хранения значений и применяется обычно для определения функций, которые не возвращают значения.

В данном разделе мы рассмотрим только стандартные типы данных.

Целые типы данных

Существует семейство целых типов данных - *short, int, long*; и два спецификатора – *signed* и *unsigned* (таблица 1.1).

Таблица 1.1

Тип	Диапазон значений	Размер (байт)
short	-32 768 ... 32 767	2
unsigned short	0 ... 65 535	2
int	-32 768 ... 32 767 или -2 147 483 648 ... 2 147 483 647	2 или 4
unsigned int	0 ... 65 535 или 0 ... 4 294 967 295	2 или 4
long	-2 147 483 648 ... 2 147 483 647	4
unsigned long	0 ... 4 294 967 295	4

Тип int является аппаратно-зависимым, это означает, что для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного — 4 байта. Размер типов *short* – 2 байта, *long* – 4 байта всегда фиксирован.

Спецификатор *unsigned* используется для установления беззнакового формата представления данных, *signed* – знакового. По умолчанию все целые типы являются знаковыми, и поэтому спецификатор *signed* можно не использовать.

Замечание. Символьные типы *char* и *wchar_t* также могут использоваться для хранения целых чисел, но мы не будет рассматривать эти типы в данном контексте.

Вещественные типы данных

Существует семейство вещественных типов данных: *float, double, long doube* (таблица 1.2).

Таблица 1.2

Тип	Диапазон значений	Размер (байт)	Количество значащих цифр после запятой
float	3.4e-38 ... 3.4e+38	4	6
double	1.7e-308 ... 1.7e+308	8	15
long double	3.4e-4932 ... 3.4e+4932	10	20

Замечание. В таблице 1.2 приведены абсолютные величины минимальных и максимальных значений.

Символьные типы

Для описания символьных данных используются типы *char* и *wchar_t*.

Под величину типа char отводится 1 байт, что достаточно для размещения любого символа из 256-символьного набора ASCII. Под величину типа *wchar_t* отводится 2 байта, что достаточно для размещения любого символа из кодировки Unicode.

Логический тип

Для описания логических данных используется тип *bool*, который может принимать всего два значения: *true* (истина) и *false* (ложь). Теоретически размер переменной типа *bool* должен составлять 1 бит, но на практике большинство компиляторов выделяет под него 1 байт, т.к. организовать доступ к байту на аппаратном уровне легче, чем к биту.

1.4. Константы

Константами называются данные, которые не изменяются в процессе выполнения программы. В C++ можно использовать именованные и неименованные константы.

Неименованные константы

Неименованные константы или литералы - это обычные фиксированные значения. Например, в операторе:

```
a = b + 2.5;          // 2.5 – неименованная константа
```

Различаются целые, вещественные, символьные и строковые литералы. Компилятор относит литерал к одному из типов данных по его внешнему виду.

Для записи *целочисленного литерала* можно использовать одну из трех форм: десятичную, восьмеричную или шестнадцатеричную. Десятичная форма: последовательность десятичных цифр, начинающаяся не с нуля, если это не число нуль. Вось-

меричная: нуль, за которыми следуют восьмеричные цифры (0,1,2,3.4,5,6,7). Шестнадцатеричная: Ох или ОХ, за которыми следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

Например, значение 20 можно записать:

1) в десятичной форме как 20;
2) в восьмеричной форме как 024;
3) в шестнадцатеричной форме как 0x14.

По умолчанию целочисленные литералы имеют тип *int* или *long*. Реальный тип зависит от значения. Добавив суффикс можно явно задать тип целочисленной константы: *long, unsigned, unsigned long*. Например, 1L (тип *long*), 8Lu (*unsigned long*), 128u (*unsigned)*.

Для записи *вещественного литерала* можно использовать десятичную или экспоненциальную форму.

Десятичная форма записи имеет вид: **[целая часть].[дробная часть]**, например, 2.02, -10.005. При этом в записи числа может быть опущена либо целая часть, либо дробная, но не обе сразу. Например: 2 или .002

Экспоненциальная форма имеет вид **[мантиса]{E|e}[+|-][порядок].** В этом случае значение константы определяется как произведение мантиссы на 10 в степени, определенной порядком числа. Например, запись 0.2E6 эквивалентна записи $0.2 \cdot 10^6$, а 1.123E-2 – эквивалентна записи $1.123 \cdot 10^{-2}$.

По умолчанию константы с плавающей запятой имеют тип *double*. Для указания одинарной точности после значения располагают суффикс *f* или *F*, для повышенной – суффикс *l* или *L*. Например:

Десятичная форма	Экспоненциальная форма
3.14159F	3.14159E0f
.001f	1E-3F
12.345L	1.2345E1L
0	0e0

Символьный литерал – это один или два символа, заключенные в апострофы. Например, 'f', '.', '3 ', '\n'. Большинство символьных литералов являются печатаемыми символами, т.е. они отображаются на экране в том виде, в котором записаны в апострофах. Некоторые символы являются непечатаемыми, т.е. никак не отображаются ни на экране, ни при печате. Для ввода непечатаемых символов используется управляющая последовательность. Управляющая последовательность начинается символом наклонной черты влево. Некоторые управляющие последовательности C++ приведены в таблице 1.3

Таблица 1.3.

Вид	*Назначение*	*Вид*	*Назначение*
\a	Звуковой сигнал, оповещение (alert)	\t	Горизонтальная табуляция (horizontal tab)
\b	Возврат на 1 символ (backspace)	\v	Вертикальная табуляция (vertical tab)
\f	Перевод страницы (formfeed)	\\	Обратная косая черта (backslash)
\n	Перевод строки, новая строка (newline)	\'	Апостроф, одинарная кавычка (single quote)
\r	Возврат каретки (carriage return)	\"	Кавычки

Строковый литерал – это произвольное количество символов, помещенное в кавычки, например, *"Ура! Сегодня информатика!!!"*.

В состав строковых литералов могут входить и управляющие последовательности из табл.1.3. Например, если внутри строки требуется записать кавычки, то их предваряют косой чертой, по которой компилятор отличает их от кавычек, ограничивающих строку: *"Дж.Р.Р. Толкин\" Властелин Колец\" "*.

В конец каждого строкового литерала компилятором добавляется *нулевой символ*, представляемый управляющей последовательностью \0. Поэтому длина строки всегда на единицу больше количества символов в ее записи. Таким образом, пустая строка имеет длину 1 байт.

Замечания

Два строковых литерала, которые расположены рядом и разделены только пробелами, символами табуляции или новой строки, объединяются в один новый строковый литерал. Это облегчает запись длинных строковых литералов, занимающих несколько отдельных строк. Существует и более примитивный способ создания длинных строк: поместив в конце строки символов символ наклонной черты влево можно указать, что следующая строка составляет с данной строкой единое целое. Символ наклонной черты влево должен располагаться в строке последним, после него не должно быть никаких комментариев и пробелов. Кроме того, все пробелы и символы табуляции в начале следующей строки окажутся частью полученной длинной строки.

Обратите внимание на разницу между строкой из одного символа, например, "A", и символьной константой 'A'. Более того, пустой строковый литерал допустим, а символьный нет.

Именованные константы

Если при решении задачи используются постоянные значения, имеющие смысл для этой задачи, то в программе можно определить их как именованные константы. Формат объявления именованной константы:

[<класс памяти>] const <тип> <имя именованной константы> = <выражение>;

где *класс памяти* – эти спецификатор, определяющий время жизни и область видимости программного объекта (см. раздел 2.4); *выражение* определяет значение именованной константы, т.е инициализирует ее.

Константа обязательно должна быть инициализирована при объявлении. В C++ разрешается использовать при объявлении констант выражения, операндами которого могут быть ранее определенные константы. А также в одном операторе можно описать несколько констант одного типа, разделяя их запятыми. Например:

```
const int I=-124;
const float k1=2.345,  k2 =1/k1;
```

1.5. Переменные

Переменная — это именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение. Во время выполнения программы значение переменной можно изменять. Перед использованием любая переменная должна быть описана. Формат описания (объявления) переменных:

[<класс памяти>] <тип> <имя> [=<выражение> | (<выражение>)];

Например, опишем переменные i, j –типа int и переменную x типа double:

```
int i, j;
double x;
```

Перед использованием значение любой переменной должно быть определено. Это можно сделать с помощью:

1) Оператора присваивания:

```
int a;      //описание переменной
...
a=10;       //определение значения переменной
```

2) Оператора ввода:

```
int a;      //описание переменной
...
cin>> a;    //определение значения переменной
```

3) Инициализации – определении значения переменной на этапе описания. Язык C++ поддерживает две формы инициализации переменных: инициализация копией и прямая инициализация. Например:

```
int i=100;      //инициализация копией
int i (100);    //прямая инициализация
```

При одном описании можно инициализировать две и более переменных, задавая каждой собственное значение. Кроме того, в одном описании могут находиться как инициализированные, так и неинициализированные переменные. Например:

```
int i, day=3, year=2007;
```

Замечание. Необходимо помнить о том, что при использовании неинициализированной переменной для любых целей, кроме присвоения ей значения или использовании в операторе ввода, будет получен неопределенный результат, т.к. будет использовано случайное значение, находящееся в памяти по адресу, на который указывает переменная. Подобные ошибки очень трудно обнаружить. Поэтому нужно пользоваться простым правилом: если первым действием с переменной будет присвоение ей значения, то в инициализации нет необходимости, в противном случае, следует обязательно присвоить ей значение при описании.

1.6. Организация консольного ввода/вывода данных

В разделе 1.1 уже рассматривался ввод-вывод данных. Напомним, что в C++ для организации консольного ввода-вывода данных (ввод-вывод в режиме «черного» окна) необходимо подключить заголовочный файл iostream. В этом файле определены:

1) объект cin, который предназначен для ввода данных со стандартного устройства ввода - клавиатуры;
2) объект cout, который предназначен для вывода данных на стандартное устройство вывода – экран;
3) операция >>, которая используется для извлечения данных из входного потока;
4) операция <<, которая используется для помещения данных в выходной поток;
5) операции форматированного ввода-вывода.

Рассмотрим некоторые операции форматирования выходного потока. По умолчанию данные, помещаемые в выходной поток, форматируются следующим образом:

1) значение типа char помещается в поток как единичный символ и занимает в потоке поле, размерность которого равна единице;
2) строка помещается в поток как последовательность символов и занимает в потоке поле, размерность которого равна длине строки;

3) значение целого типа помещается в поток как десятичное целое число и занимает в потоке поле, размерность которого достаточна для размещения всех цифр числа, а в случае отрицательного числа еще и для знака минус; положительные числа помещаются в поток без знака;
4) значение вещественного типа помещается в поток с точностью 6 цифр после «десятичной запятой»; в зависимости от величины числа оно может быть выведено в экспоненциальной форме (если число очень маленькое или очень большое) или десятичной форме.

Для форматирования потоков используются манипуляторы. Мы уже рассматривали манипулятор *endl*, который позволяет при выводе потока на экран перенести фрагмент потока на новую строку. Теперь рассмотрим манипуляторы, которые позволяют управлять форматом вещественных типов данных и их размещением на экране.

Замечание. Для использования манипуляторов с аргументами требуется подключить заголовочный файл *iomanip*

Управление форматом вещественных типов данных

Существуют три аспекта оформления значений с плавающей запятой, которыми можно управлять: *точность* (количество отображаемых цифр); *форма записи* (десятичная или экспоненциальная); *указание десятичной точки* для значений с плавающей запятой являющихся целыми числами.

Точность задает общее количество отображаемых цифр. При отображении значения с плавающей запятой округляются до текущей точности, а не усекаются. Изменить точность можно с помощью манипулятора *setprecision*. Аргументом данного манипулятора является устанавливаемая точность.

Рассмотрим следующий пример:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{ double i=12345.6789;
  cout << setprecision(3) << i <<endl;
  cout << setprecision(6) << i <<endl;
  cout << setprecision(9) << i <<endl;
  return 0;}
```

Результат работы программы:

```
1.23e+004
12345.7
12345.6789
```

По умолчанию *форма записи* значения с плавающей запятой зависит от его размера: если число очень большое или очень маленькое, оно будет отображено в экспоненциальном формате, в противном случае – в десятичном формате. Чтобы самостоятельно установить тот или иной формат вывода, можно использовать манипуляторы *scientific* (отображать числа с плавающей запятой в экспоненциальном формате) или *fixed* (отображать числа с плавающей запятой в десятичном формате). Рассмотрим следующий пример:

```
#include <iostream>
using namespace std;
int main()
{ double i=12345.6789;
  cout << scientific<< i <<endl;
  cout << fixed << i <<endl;
  return 0;}
```

Результат работы программы:

```
1.234568e+004
12345.678900
```

Десятичная точка по умолчанию не отображается, если дробная часть равна 0. Манипулятор *showpoint* позволяет отображать десятичную точку принудительно.

```
#include <iostream>
using namespace std;
int main()
{ double i=10; cout << i <<endl;
  cout << showpoint << i <<endl;
  return 0;}
```

Результат работы программы:

```
10
10.0000
```

Управление размещением данных на экране

Для размещения отображаемых данных используются манипуляторы:

1) *left* – выравнивает вывод по левому краю;
2) *right* – выравнивает вывод по правому краю;
3) *internal* - контролирует размещение отрицательного значения: выравнивает знак по левому краю, а значение по правому, заполняя пространство между ними пробелами;
4) *setprecision(int w)* – устанавливает максимальное количество цифр в дробной части для вещественных чисел в формате с фиксированной точкой (манипулятор fixed) или общее количестов значащих цифр для чисел в экспоненциальном формате (манипулятор scientific);
5) *setw(int w)* – устанавливает максимальную ширину поля вывода.

Рассмотрим примеры использования данных манипуляторов.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{ cout << "1." <<setw(10) <<"Ivanov" <<endl;
  cout << "2." <<setw(10) <<left <<"Ivanov" <<endl;
  cout << "3." <<setw(10) <<right <<"Ivanov" <<endl;
  return 0; }
```

Результат работы программы:

```
1.    Ivanov
2.Ivanov
3.    Ivanov
```

Замечание. По умолчанию выравнивание происходит по правому краю (см. первый оператор вывода).

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{ cout << "1." <<setw(10) <<-23.4567 <<endl;
  cout << "2." <<setw(10) <<setprecision(3) <<-23.4567 <<endl;
  cout << "3." <<setw(10) <<internal <<-23.4567 <<endl;
  return 0;}
```

Результат работы программы:

```
1.  -23.4567
2.      -23.5
3.-     23.5
```

Замечание. Обратите внимание на то, что установки манипулятора *setprecision* распространяются и на все последующие операторы вывода.

1.7. Операции

Полный список операций C++ в соответствии с их приоритетами (по убыванию приоритетов, операции с разными приоритетами разделены чертой) приведен в приложении 2. В данном разделе мы подробно рассмотрим только часть операций, остальные операции будут вводиться по мере необходимости.

Замечание. Операции можно классифицировать по количеству операндов на: унарные – воздействуют на один операнд, бинарные – воздействуют на два операнда, тернарные – воздействуют на три операнда. Некоторые символы используются для обозначения как унарных, так и бинарных операций. Например, символ * используется как для обозначения унарной операции разадресации, так и для обозначения бинарной операции умножения. Будет ли данный символ обозначать унарную или бинарную операцию, определяется контекстом, в котором он используется.

Унарные операции

Операции увеличения и уменьшения на 1 (++ и --)

Эти операции называются также инкрементом и декрементом соответственно. Они имеют две формы записи — *префиксную*, когда операция записывается перед операндом, и *постфиксную* – операция записывается после операнда. Префиксная операция инкремента (декремента) увеличивает (уменьшает) свой операнд и возвращает измененное значение как результат. Постфиксные версии инкремента и декремента возвращают первоначальное значение операнда, а затем изменяют его.

Рассмотрим эти операции на примере.

```
#include <iostream>
using namespace std;
int main()
{ int x=3, y=4;
  cout  <<++x <<"\t"<< --y<<endl;
  cout  <<x++ <<"\t"<< y--<<endl;
  cout  <<x <<"\t"<< y<<endl;
  return 0;}
```

Результат работы программы:

```
4 3
4 3
5 2
```

Замечание. Префиксная версия требует существенно меньше действий: она изменяет значение переменной и запоминает результат в ту же переменную. Постфиксная операция должна отдельно сохранить исходное значение, чтобы затем вернуть его как результат. Для сложных типов подобные дополнительные действия могут оказаться трудоемкими. Поэтому постфиксную форму имеет смысл использовать только при необходимости.

Операция определения размера sizeof

Эта операция предназначена для вычисления размера выражения или типа в байтах, и имеет две формы: *sizeof <выражение>*, *sizeof (<тип>)*. При применении операции sizeof к выражению возвращается размер типа, который получается в результате вычисления выражения. При применении к типу – возвращается размер заданного типа.

Рассмотрим данную операцию на примере.

```
#include <iostream>
using namespace std;
int main()
{ int x=3;
  cout  << sizeof (int) <<endl;
  cout  << sizeof (x*10)<<endl;
  cout  << sizeof (x*0.1)<<endl;
  return 0; }
```

Результат работы программы:

```
4
4
8
```

Замечание. Последний результат определен тем, что вещественные константы по умолчанию имеют тип double, к которому, как к более длинному, приводится тип всего выражения. Скобки необходимы для того, чтобы выражение, стоящее в них, вычислялось раньше операции приведения типа, имеющей больший приоритет, чем сложение.

Операции отрицания (-, !)

Арифметическое отрицание (или унарный минус -) изменяет знак операнда целого или вещественного типа на противоположный.

Замечание. В C++ на экран в качестве значения константы *false* (ложь) выводится 0, а вместо значения константы *true* (истина) выводится 1. При этом в логическом выражении нулевое значение любого типа, в том числе и пустой указатель, преобразуется к логической константе false, а ненулевое значение любого типа - к логической константе *true*.

Логическое отрицание (!) дает в результате значение 0 (ложь), если операнд отличен от нуля (истина), и значение 1 (истина), если операнд равен нулю (ложь). Тип операнда может быть логический, целочисленный, вещественный или указатель. Рассмотрим данные операции на примере.

```
#include <iostream>
using namespace std;
int main()
{ int x=3, y=0;
  bool f=false, v=true;
  cout  << -x<<"\t" <<!x <<endl;
  cout  << -y<<"\t" <<!y <<endl;
  cout  << f<<"\t" <<!f <<endl;
  cout  << v<<"\t" <<!v <<endl;
  return 0;}
```

Результат работы программы:

```
-3 0
 0 1
 0 1
 1 0
```

Бинарные операции

Арифметические операции

Арифметические операции – операции, применимые ко всем арифметическим типам (например, к целым и вещественным типам), а также к любым другим типам, которые могут быть преобразованы в арифметические (например, символьные типы). К бинарным арифметическим операциям относятся (в порядке убывания приоритета):

1) умножение (*), деление (/), остаток от деления (%);
2) сложение (+) и вычитание (-).

Операция *деления* применима к операндам арифметического типа. Однако, если оба операнда целочисленные, то тип результата преобразуется к целому, и в качестве ответа возвращается целая часть результата деления, в противном случае, тип результата определяется правилами преобразования. Операция *остаток от деления* применяется только к целочисленным операндам. Знак результата зависит от реализации.

Рассмотрим на примере операции *деления* и *остаток от деления*.

```
#include <iostream>
using namespace std;
int main()
{ cout  << 100/24<<"\t"<<100/24.0<<endl;
  cout  << 100/21<<"\t"<<100.0/24<<endl;
  cout  << 21%3<<"\t" <<21%6<<"\t" <<-21%8<<endl;
  return 0;}
```

Результат работы программы:

```
4  4.7619
4  4.1667
0  3  -5
```

Операции отношения (<, <=,>, >=, ==, !=)

Операции отношения <, <=,>, >=, = =, != (меньше, меньше или равно, больше, больше или равно, равно, не равно соответственно) - это стандартные операции, ко-

торые сравнивают первый операнд со вторым. Операнды могут быть арифметического типа или указателями. Результатом операции является значение *true* (истина) или *false (ложь)* (напомним, что любое значение, не равное нулю, интерпретируется как истина, нулевое – как ложь). Операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции сравнения.

Логические операции (&& и ||)

Логическая операция *И* (&&) возвращает значение истина тогда и только тогда, когда оба операнда принимают значение истина, в противном случае, операция возвращает значение ложь. Логическая операция *ИЛИ* (||) возвращает значение истина тогда и только тогда, когда хотя бы один операнд принимает значение истина, в противном случае, операция возвращает значение ложь. Логические операции выполняются слева направо, при этом приоритет операции (&&) выше приоритета операции (||). Если значения первого операнда достаточно, чтобы определить результат всей операции, то второй операнд не вычисляется.

Рассмотрим на примере данные операции.

```
#include <iostream>
using namespace std;
int main()
{ cout << "x \t y \t && \t ||" <<endl;
  cout  << "0 \t 0 \t "<< (0 && 0) <<" \t "<< (0 || 0) <<endl;
  cout  << "0 \t 1 \t "<< (0 && 1) <<" \t "<< (0 || 1) <<endl;
  cout  << "1 \t 0 \t "<< (1 && 0) <<" \t "<< (1 || 0) <<endl;
  cout  << "1 \t 1 \t "<< (1 && 1) <<" \t "<< (1 || 1) <<endl;
  return 0; }
```

Результат работы программы:

x	y	&&	||
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Замечания

Фактически была построена таблица истинности для логических операций И и ИЛИ.

Мы рассматриваем операции в порядке убывания их приоритета. Сейчас этот порядок будет нарушен, т.к. следующей по этому признаку должна быть условная операция, которая является тернарной, а мы еще не закончили рассматривать бинарные операции. Поэтому мы сначала закончим рассматривать бинарные операции присваивания, а затем перейдем к тернарной. Если вам потребуется уточнить приоритеты операций, то можно обратиться к приложению 2.

*Операции присваивания (=, *=, /=, %=, +=, -=)*

Операция простого присваивания обозначается знаком =. Формат операции *простого присваивания* (=):

операнд_2 = операнд_1;

В результате выполнения этой операции вычисляется значение операнда_1, и результат записывается в операнд_2. Возможно связать воедино сразу несколько операторов присваивания, записывая такие цепочки: a=b=c=100. Присваивание такого вида выполняется справа налево: результатом выполнения c=100 является число 100, которое затем присваивается переменной b, результатом чего опять является 100, которое присваивается переменной a.

Кроме простой операции присваивания существуют *сложные операции присваивания*, например, умножение с присваиванием (*=), деление с присваиванием (/=), остаток от деления с присваиванием (%=), сложение с присваиванием (+=), вычитание с присваиванием (-=) и т.д (см. приложение 3).

В сложных операциях присваивания, например, при *сложении с присваиванием*, к операнду_2 прибавляется операнд_1, и результат записывается в операнд_2. То есть, выражение c += a является более компактной записью выражения c = c + a . Кроме того, сложные операции присваивания позволяют сгенерировать более эффективный код засчет того, что в простой операции присваивания для хранения значения правого операнда создается временная переменная, а в сложных операциях присваивания значение правого операнда сразу записывается в левый операнд.

Тернарная операция

Условная операция (? :)

Формат условной операции:

операнд_1 ? операнд_2 : операнд_3

Операнд_1 – это логическое или арифметическое выражение. Он оценивается с точки зрения его эквивалентности константам *true* (истина) и *false* (ложь). Если результат вычисления операнда_1 равен истине, то результатом условной операции будет значение операнда_2, иначе — операнда_3. Вычисляется всегда либо операнд_2, либо операнд_3. Их тип может различаться. Условная операция является сокращенной формой условного оператора if, который будет рассмотрен позже.

Рассмотрим тернарную операцию на примере.

```
#include <iostream>
using namespace std;
int main()
{ int x, y, max;
  cin >>x>>y;
  (x>y)? cout <<x : cout<<y<<endl;     //1
  max=(x>y)? x : y;                    //2
  cout<<max<<endl;
  return 0;}
```

Результат работы программы для x=11 и y=9:

```
11
11
```

Обратите внимание на то, что *строка 1* и *строка 2* решают одну и ту же задачу: нахождение наибольшего значения из двух целых чисел. Но в *строке 2* в зависимости от условия (x>y) условная операция записывает в переменную max либо значение x, либо значение y. После чего значение переменной max можно неоднократно использовать. В *строке 1* наибольшее значение просто выводится на экран, и в дальнейшем это значение использовать будет нельзя, т.к. оно не было сохранено ни в какой переменной.

1.8. Выражения и преобразование типов

Выражение – это синтаксическая единица языка, определяющая способ вычисления некоторого значения. Выражения состоят из операндов, операций и скобок. Каждый операнд является в свою очередь выражением или одним из его частных случаев – константой, переменной или функций.

Замечание. Математические функции заголовочного файла *math* (*cmath*), которые могут применяться в арифметических выражениях, приведены в приложении 4.

Примеры выражений:

```
(a + 0.12)/6          x && y || !z          (t * sin(x)-l.05e4)/((2 * k + 2) * (2 * k + 3))
```

Операции выполняются в соответствии с приоритетами (см.приложение 3). Для изменения порядка выполнения операций используются круглые скобки. Если в одном выражении записано несколько операций одинакового приоритета, то унарные операции, условная операция и операции присваивания выполняются *справа налево*, остальные — *слева направо*. Например,

a = b = c означает a=(b=c),
a+b+c означает (a + b) + c.

Порядок вычисления подвыражений внутри выражений не определен: например, нельзя считать, что в выражении *(sin(x + 2) + cos(y + 1))* обращение к синусу будет выполнено раньше, чем к косинусу, и что *x + 2* будет вычислено раньше, чем *y + 1*.

Результат вычисления выражения характеризуется значением и типом. Например, если *a* и *b* — переменные целого типа и описаны так:

int a = 2, b = 5,

то выражение *a + b* имеет значение *7* и тип *int*.

В выражение могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат операции будет иметь тот же тип. Если операнды разного типа, то перед вычислениями выполняются преобразования более коротких типов в более длинные для сохранения значимости и точности. Иерархия типов данных приведена в таблице 1.4.

Таблица 1.4.

Типы данных	***Старшинство***
long double	Высший
double	
float	
long	
int	
short	
char	Низший

Преобразование типов в выражениях происходит *неявно* (без участия программистов) следующим образом: если их операнды имеют различные типы, то операнд с более «низким» типом автоматически будет преобразован к более «высокому» типу. Тип *bool* в арифметических выражениях преобразуется к типу *int*, при этом константа *true* заменяется 1, а *false* – 0.

В общем случае неявные преобразования могут происходить с потерей точности или без потери точности. Рассмотрим небольшой пример:

```
#include <iostream>
using namespace std;
int main()
{ int a=100, b;  float c=4.5, d;
 d= a/c;        //1 – без потери точности
 cout <<"d=" <<d<<endl;
 b=a/c;         //2 – с потерей точности
 cout <<"b=" <<b<<endl;return 0;}
```

Результат работы программы:

```
d=22.2222
b=22
```

В *строке 1* переменная *a* (тип *int*) делится на переменную *c* (тип *float*). В соответствии с иерархией типов результат операции деления будет преобразован к типу

float. Полученное значение записывается в переменную *d* (тип *float*). Таким образом, в *строке 1* было выполнено одно преобразование от «низшего» типа к «высшему» и мы получили точный результат.

В *строке 2* в результате операции деления также было получено значение, тип которого *float*. Но этот результат был записан в переменную *b* (тип *int*). Поэтому произошло еще одно преобразование – от «высшего» типа к «низшему», что повлекло потерю точности.

Рассмотренные преобразования происходили неявно (автоматически), т.е. без участия программиста. Однако программист может совершать подобные преобразования сам с помощью операций *явного преобразования типов* (см. приложение 3). Для демонстрации этих операций рассмотрим небольшой пример:

```
#include <iostream>
using namespace std;
int main()
{ int a=1500000000;
  a=(a*10)/10;                    //1 – неверный результат
  cout << "a=" <<a<<endl;
  int b=1500000000;
  b=(static_cast<double>(b)*10)/10; //2 – верный результат   Результат работы программы:
  cout << "b=" <<b<<endl;                                    a=211509811
  return 0; }                                                b=1500000000
```

В *строке 1* мы умножили переменную *a* на 10 и получили результат, равный 15000000000, который нельзя сохранить даже с помощью *usigned int*. В этом случае было выполнено неявное преобразование типов, которое привело к потере точности вычисления.

В *строке 2* перед умножением переменной *b* на 10 было выполнено явное преобразование значения переменной к типу *double*. Полученное значение было сохранено во временную переменную. Затем временная переменная умножается на 10, и поскольку результат 150000000000 попадает в диапазон допустимых значений для типа *double*, то переполнения не происходит. После деления на 10 тип *double* неявно преобразуется к типу *int*, и мы получаем верный результат.

Замечание. Явное преобразование типов следует использовать только в случае полной уверенности в его необходимости и понимания, для чего оно делается. Т.к. в случае явного преобразования компилятор не может проконтролировать корректность действий при изменении типов данных, то повышается возможность ошибок.

1.9. Примеры простейших программ

1. Составить программу, которая для заданного значения x вычисляет значение выражения $\frac{x^2 + \sin(x+1)}{25}$.

Указание по решению задачи. Прежде чем составлять программу, перепишем данное выражение с учетом приоритета операций по правилам языка C++: (pow(x, 2)+sin(x+1))/25.

```
#include <iostream>
#include <iomanip>
#include <math.h>
using namespace std;
int main()
```

```
{ int x; double y;
  cin >>x;
  y=(pow(x, 2)+sin(x+1))/25;  //1
  cout <<"y=" << setprecision(5) << y <<endl;
  return 0;}
```

Результат работы программы при x=10:

y=3.96

Замечание. В некоторых версиях компилятора функции *pow* и *sin* могут обрабатывать только вещественные числа. В этом случае вам потребуется изменить строку 1 следующим образом (pow(x, 2.0)+sin(x+1.0))/25.

2. Написать программу, подсчитывающую площадь квадрата, периметр которого равен p.

Указания по решению задачи. Прежде чем составить программу, проведем математические рассуждения. Пусть дан квадрат со стороной a, тогда:

периметр вычисляется по формуле $p=4a$

площадь вычисляется по формуле $s=a^2$

$$\Rightarrow \quad \begin{matrix} a=\frac{p}{4} \\ a=\sqrt{s} \end{matrix} \quad \Rightarrow \quad \frac{p}{4}=\sqrt{s} \quad \Rightarrow \quad s=\left(\frac{p}{4}\right)^2$$

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{ float p, s;
  cout <<"Введите периметр квадрата: ";  cin >>p;
  s=pow(p/4, 2);
  cout <<"Площадь данного квадрата = " << s;
  return 0;}
```

Результат работы программы для p=20:

Площадь данного квадрата=25

3. Определить, является ли целое число четным.

Указание по решению задачи. Напомним, что число является четным, если остаток от деления данного числа на 2 равен нулю.

```
#include <iostream>
using namespace std;
int main()
{ int x;
  cout <<"Введите x";   cin >>x;
  (x % 2 ==0)? cout << "четное\n": cout << "нечетное\n";   //1
  return 0; }
```

Результат работы программы:

x	ответ
45	нечетное
88	четное

Замечание. В *строке 1* операция % находит остаток от деления на 2. Если число четное, то остаток будет равен 0, а в C++ нулевое значение трактуется как ложь. Если число нечетное, то остаток будет равен 1, а в C++ ненулевое значение трактуется как истина. Поэтому в *строке 1* можно обойтись без операции сравнения. В этом случае условная операция будет выглядеть следующим образом: *(x % 2)? cout << "четное\n": cout << "нечетное\n";*

1.10. Упражнения

I. Написать программу, которая вычисляет значение выражения.

Замечания

1) Для вывода результата использовать манипуляторы для форматирования выходного потока.

2) Считать, что вводимые значения x принадлежат области определения выражения.

1) $10\sin x+|x^4-x^5|$;

2) $e^{-x}-\cos x+\sin 2xy$;

3) $\sqrt{x^4+\sqrt{|x+1|}}$;

4) $\dfrac{\sin x+\cos y}{\operatorname{tg}x}+0{,}43$;

5) $\dfrac{0{,}125x+|\sin x|}{1{,}5x^2+\cos x}$;

6) $\dfrac{x+y}{x+1}-\dfrac{xy-12}{34+x}$;

7) $\dfrac{\sin x+\cos y}{\cos x-\sin y}\operatorname{tg}xy$;

8) $\dfrac{1+e^{y-1}}{1+x^2|y-\operatorname{tg}x|}$;

9) $|x^3-x^2|-\dfrac{7x}{x^3-15x}$;

10) $1+\dfrac{x}{3}+|x|+\dfrac{x^3+4}{2}$;

11) $\dfrac{\ln|\cos x|}{\ln(1+x^2)}$;

12) $\dfrac{1+\sin\sqrt{x+1}}{\cos(12y-4)}$;

13) $\dfrac{a^2+b^2}{1-\dfrac{a^3-b}{3}}$;

14) $\dfrac{1+\sin^2(x+y)}{2+\left|x-\dfrac{2x}{1+x^2y^2}\right|}+x$;

15) $x\cdot\ln x+\dfrac{y}{\cos x-\dfrac{x}{3}}$;

16) $\sin\sqrt{x+1}-\sin\sqrt{x-1}$;

17) $\dfrac{\cos x}{\pi-2x}+16x\cdot\cos xy$;

18) $2\operatorname{ctg}3x-\dfrac{1}{12x^2+7x-5}$;

19) $\dfrac{b+\sqrt{b^2+4ac}}{2a}-a^3+b^{-2}$;

20) $\ln\left|(y-\sqrt{|x|})(x-\dfrac{y}{x+\dfrac{x^2}{4}})\right|$.

II. Написать программу, которая подсчитывает:

1) периметр квадрата, площадь которого равна a;
2) площадь равностороннего треугольника, периметр которого равен p;
3) расстояние между точками с координатами a, b и c,d;
4) среднее арифметическое кубов двух данных чисел;
5) среднее геометрическое модулей двух данных чисел;
6) гипотенузу прямоугольного треугольника по двум данным катетам a, b.
7) площадь прямоугольного треугольника по двум катетам a, b.
8) периметр прямоугольного треугольника по двум катетам a, b.
9) ребро куба, площадь полной поверхности которого равна s;
10) ребро куба, объем которого равен v;
11) периметр треугольника, заданного координатами вершин x1, y1, x2, y2, x3, y3;
12) площадь треугольника, заданного координатами вершин x1, y1, x2, y2, x3, y3;
13) радиус окружности, длина которой равна l;
14) радиус окружности, площадь круга которой равна s;
15) площадь равнобедренной трапеции с основаниями a и b и углом α при большем основании;
16) площадь кольца с внутренним радиусом r1 и внешним r2;
17) радиус окружности, вписанной в равносторонний треугольник со стороной a;
18) радиус окружности, описанной около равностороннего треугольника со стороной a;
19) сумму членов арифметической прогрессии, если известен ее первый член, разность и число членов прогрессии;

20) сумму членов геометрической прогрессии, если известен ее первый член, знаменатель и число членов прогрессии.

III. Написать программу, которая определяет:

1) максимальное значение для двух различных вещественных чисел;
2) является ли заданное целое число четным;
3) является ли заданное целое число нечетным;
4) если целое число M делится на целое число N, то на экран выводится частное от деления, в противном случае выводится сообщение «M на N нацело не делится»;
5) оканчивается ли данное целое число цифрой 7;
6) имеет ли уравнение $ax^2+bx+c=0$ решение, где a, b, c – данные вещественные числа;
7) какая из цифр двухзначного числа больше: первая или вторая;
8) одинаковы ли цифры данного двухзначного числа;
9) является ли сумма цифр двухзначного числа четной;
10) является ли сумма цифр двухзначного числа нечетной;
11) кратна ли трем сумма цифр двухзначного числа;
12) кратна ли числу A сумма цифр двухзначного числа;
13) какая из цифр трехзначного числа больше: первая или последняя;
14) какая из цифр трехзначного числа больше: первая или вторая;
15) какая из цифр трехзначного числа больше: вторая или последняя;
16) все ли цифры трехзначного числа одинаковые;
17) существует ли треугольник с длинами сторон a, b, c;
18) является ли треугольник с длинами сторон a, b, c прямоугольным;
19) является ли треугольник с длинами сторон a, b, c равнобедренным;
20) является ли треугольник с длинами сторон a, b, c равносторонним.

2. ФУНКЦИИ В C++

С увеличением объема программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является разделение ее на части – подпрограммы. Разделение программы на подпрограммы позволяет также избежать избыточности кода, поскольку подпрограммы описывают один раз, а вызывают на выполнение многократно из различных участков программы.

Как мы уже знаем, в C++ любая программа может состоять из нескольких функций, но в программе обязательно должна присутствовать функция *main* – главная функция, с которой начинается выполнение программы. Теперь мы научимся разрабатывать программы, состоящие из нескольких функций. А также рассмотрим такие важные понятия как классы памяти и модели памяти.

2.1. Основные понятия

Функция – это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая функция должна быть *объявлена* и *определена*. *Объявление функции* (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, еще и *тело* функции. Функция может быть объявлена несколько раз, но определена только один раз. Синтаксис определения функции:

```
[<класс памяти>] <тип результата> <имя функции> ([<список параметров>] )
{ <тело функции>}
```

Рассмотрим основные части определения функции:

1) *Класс памяти* (необязательный элемент описания) – эти спецификатор, определяющий время жизни и область видимости программного объекта (см. раздел 2.4).
2) *Тип результата* возвращаемого функцией может быть любым, кроме массива или функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, то указывается тип *void*. Функция *main* должна возвращать значение типа *int*.
3) *Список параметров* определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра указывается его тип и имя. Список параметров может быть пустым.
4) *Тело функции* представляет собой последовательность описаний и операторов. Если тип результата функции не *void*, то тело функции должно содержать команду *return <возвращаемое значение>*.

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых параметров. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого значения не *void*, то она может входить в состав выражений. В частности, может располагаться в правой части от оператора присваивания.

Рассмотрим пример функции, возвращающей сумму двух чисел.

```
#include <iostream>
using namespace std;
int sum(int x,int y)  // определение функции
{ return x+y; }
int main()            // главная функция
{ int a=5, b=3, c;
  c=sum(a,b);                          //1
  cout <<"sum="<<c<<endl;
  cout <<"sum="<<sum(a,b)<<endl;       //2
  return 0; }
```

Результат работы программы:

```
sum=8
sum=8
```

В данном примере функция *sum* сразу определена, поэтому ее предварительное объявление не требуется. Функция возвращает целочисленное значение, которое

формируется выражением после команды *return*. На этапе определения функции были указаны два *формальных* целочисленных параметра. На этапе вызова функции (строки 1 и 2) в функцию передаются *фактические* параметры, которые по количеству и по типу совпадают с формальными параметрами. Если количество фактических и формальных параметров будет различным, то компилятор выдаст соответствующее сообщение об ошибке. Если параметры будут отличаться типами, то компилятор выполнит неявное преобразование типов. Обратите внимание на то, что в *строке 1* вызов функции входит в состав выражения, располагаясь справа от знака присваивания. В общем случае выражения могут быть и более сложными. А в *строке 2* результат, возвращаемый функцией, сразу помещается в выходной поток.

Рассмотрим другой пример: функция находит наибольшее значение для двух вещественных чисел.

```
#include <iostream>
using namespace std;
float max(float x, float y);  // объявление функции
int main()    // главная функция
{ float a=5.5, b=3.2, c=14.1, d;
  d=max(max(a,b),c);    //1
  cout <<"max="<<d<<endl;
  return 0; }
float max(float x, float y)  //определение функции
{ return (x>y)?x:y; }
```

Результат работы программы:

```
max=14.1
```

В данном примере функция вначале объявлена, а затем определена. Обратите внимание на то, что в *строке 1* происходит два обращения к функции *max*. Вначале в функцию *max* будут переданы значения переменных *a* и *b* (5.5 и 3.3 соответственно). Функция вернет в качестве результата значение наибольшего из них – 5.5. А затем в функцию будут переданы значение 5.5 и значение переменной *c*, т.е. 14.1, из которых также будет найдено наибольшее значение. В результате, в переменную *d* будет записано значение 14.1.

2.2. Локальные и глобальные переменные

Все величины, описанные внутри функции, а также ее параметры, являются *локальными* переменными. Область их видимости (см. раздел 2.4) – тело функции. При вызове функции в стеке (см. раздел 2.5) выделяется память под локальные автоматические переменные. Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции для того, чтобы при выходе из нее можно было продолжить выполнение вызывающей функции. При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются.

Замечание. Во всех примерах, рассмотренных выше, использовались только локальные переменные.

Глобальные переменные описываются вне функций, в том числе и вне функции *main*, поэтому они видны во всех функциях программы и могут использоваться для передачи данных между всеми функциями. Если имена глобальных и локальных

переменных совпадают, то внутри функций локальные переменные «заменяют» глобальные, а после выхода из функции значение глобальной переменной восстанавливается. Однако с помощью операции доступа к области видимости (::) можно получить доступ к одноименной глобальной переменной. Рассмотрим небольшой пример:

```
#include <iostream>
using namespace std;
int a=100, b=20;  //глобальные переменные a и b
void f1(int a) //локальная переменная a
{ a+=10;                                          //1
  cout <<"f1:\t"<<a <<"\t" << ::a <<endl; }       //2
void f2 (int b)  //локальная переменная b
{ b*=2;                                           //3
  cout <<"f2\t"<<b <<"\t" <<a <<endl;}            //4
int main()
{ cout <<"main:\t"<<a <<"\t" <<b <<endl;  //5
  f1(a); f2(b);                           //6
  cout <<"main:\t"<<a <<"\t" <<b <<endl;  //7
  return 0; }
```

Результат работы программы:

```
main:   100   20
f1:     110   100
f2:     40    100
main:   100   20
```

В *строке 1* происходит изменение значения локальной переменной *a*. В *строке 2* на экран выводится значение локальной переменной *a* и через обращение к области видимости значение глобальной переменной *a*.

В *строке 3* изменяется значение локальной переменной *b*. В *строке 4* на экран выводится значение локальной переменной *b* и значение глобальной переменной *a*. Так как в функции *f2* нет локальной переменной с именем *a*, то использовать операцию :: для обращения к глобальной переменной *a* не нужно.

В *строках 5* и *6* на экран выводятся значения глобальных переменных *a* и *b*. В *строке 6* происходит вызов вначале функции *f1*, затем *f2*. Так как тип результата данных функций *void*, то для вызова каждой из них достаточно указать только имя функции и список параметров.

Замечание. Глобальные переменные не рекомендуется использовать, поскольку они затрудняют отладку программы, и препятствую помещению функций в библиотеки общего пользования.

2.3. Параметры функции

Механизм параметров является основным способом обмена информацией между вызываемой и вызывающей функциями. Существует два способа передачи параметров в функцию: по значению и по адресу.

При передаче параметров по значению в стек заносятся копии значений фактических параметров, и операторы функции работают с этими копиями. Доступа к исходным значениям у параметров функций нет, и, следовательно, нет возможности их изменять. Во всех рассмотренных ранее примерах параметры передавались по значению.

При передаче параметров по адресу в стек заносятся копии адресов фактических параметров, и функция осуществляет доступ к ячейкам памяти по этим адресам, и, следовательно, может изменять исходные значения аргументов. Передача па-

раметров по адресу делится на *передачу по указателю* и *передачу по ссылке*. *При передаче параметра по указателю* в объявлении функции перед именем параметра указывается операция разадресации *, и при обращении к параметру в теле функции используется эта же операция. При вызове функции перед именем соответсвующего фактического параметра указывается операция взятия адреса &. Например:

```
void f (int *a)
{ (*a)++;}
int main ()
{ int x=10;  f(&x); }
```

При передаче параметра по ссылке в объявлении функции перед именем параметра указывается операция взятия адреса &. В этом случае в теле функции и при вызове функции операция разадресации выполняется неявным образом. т.е. без участия программиста. Например:

```
void f (int &a)
{ a++;}
int main ()
{ int x=10;  f(x); }
```

Рассмотрим на примере разницу в передаче параметров.

```
#include <iostream>
using namespace std;
void f(int a, int *b, int &c)   //определение функции f
{ a+=10;
  (*b)+=10;
  c+=10;
  cout <<"f:\t"<<a <<"\t" << *b <<"\t" <<c <<endl; }
int main()
{ int x=10, y=20, z=30;
  cout <<"main:\t"<<x <<"\t" <<y <<"\t" <<z <<endl;
  f(x, &y,z);    //вызов функции f
  cout <<"main:\t"<<x <<"\t" <<y <<"\t" <<z <<endl;
  return 0;}
```

Результат работы программы:

```
main:   10  20  30
f:      20  30  40
main:   10  30  40
```

В данном примере:

1) параметр *a* передается по значению, поэтому фактический параметр *x* не изменил свое значение после завершения работы функции *f*;
2) параметр *b* передается по указателю, а *c* - по ссылке, поэтому фактические параметры *y* и *z* изменили свое значение после завершения работы функции *f*.

Замечания

1) Более подробно про указатели можно прочитать в разделе 6.1.
2) Если изменения, произошедшие с параметром внутри функции, не должны отразиться на значении фактических параметров, то параметры передаются по значению, иначе – по адресу.
3) Использование ссылок вместо указателей при передаче параметров в функцию улучшает читаемость программы, избавляя ее от необходимости использовать операцию разадресации.
4) Использование передачи параметров по ссылке вместо передачи параметров по значению более эффективно, поскольку не требует копирования параметров, что имеет значение при передаче структур данных большого объема.
5) Если требуется запретить изменение параметра, передающегося по адресу, внутри функции, используется модификатор const. Например: void f (const int *a).

Внесем небольшие изменения в предыдущий пример:

```
#include <iostream>
using namespace std;
void f(const int *b, int &c)
{ (*b)+=10;                                    //1
  c+=10; }
int main()
{ int y=20, z=30;
  f(&y,2);                                     //2
  cout <<"main:\t" <<y <<"\t" <<z <<endl;
  return 0;}
```

В *строке 1* возникнет ошибка, т.к. мы попытались изменить константный параметр. В *строке 2* также возникнет ошибка, т.к. константу (в нашем случае целое число 2) нельзя преобразовать в параметр, передаваемый по адресу.

2.4. Классы памяти

Рассмотрим основные правила использования спецификаторов класса памяти:

- Необязательный спецификатор *класса памяти* может принимать одно из значений auto, extern, static и register.
- Место описания переменной и спецификатор класса памяти определяют *область действия, время жизни и область видимости* переменной.

Область действия — это часть программы, в которой переменную можно использовать для доступа к связанной с ней области памяти. В зависимости от области действия переменная может быть локальной или глобальной. Если переменная описана внутри блока (блок соответствует содержимому парных фигурных скобок), она называется *локальной,* область ее действия — от точки описания до конца текущего блока, включая все вложенные блоки. Если переменная описана вне любого блока, она называется *глобальной,* и областью ее действия считается файл, в котором она определена, от точки описания до его конца.

Класс памяти определяет *время жизни* и *область видимости* программного объекта (в частности, переменной). Если класс памяти не указан явным образом, он определяется компилятором исходя из контекста объявления.

Время жизни может быть постоянным (в течение выполнения программы) и временным (в течение выполнения блока).

Областью видимости идентификатора называется часть текста программы, из которой допустим обычный доступ к связанной с идентификатором области памяти. Чаще всего область видимости совпадает с областью действия. Исключением является ситуация, когда во вложенном блоке описана переменная с таким же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя он (блок) и входит в ее область действия. Однако, если эта переменная глобальная, то к ней можно обратиться, используя операцию доступа к области видимости ::.

Для задания *класса памяти* используются следующие спецификаторы:

auto — *автоматическая* переменная. Память под нее выделяется в стеке и при необходимости инициализируется каждый раз при выполнении оператора, содержащего ее определение. Освобождение памяти происходит при выходе из блока, в

котором описана переменная. Время ее жизни — с момента описания до конца блока. Для глобальных переменных этот спецификатор не используется, а для локальных он принимается по умолчанию, поэтому задавать его явным образом особого смысла не имеет.

extern — означает, что переменная определяется в другом месте программы (в другом файле или дальше по тексту). Используется для создания переменных, доступных во всех модулях программы, в которых они объявлены. Если переменная в том же операторе инициализируется, то спецификатор extern игнорируется.

static — *статическая* переменная. Время жизни — постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. В зависимости от расположения оператора описания статические переменные могут быть глобальными и локальными. Глобальные статические переменные видны только в том модуле, в котором они описаны.

register — аналогично auto, но память выделяется по возможности в регистрах процессора. Если такой возможности у компилятора нет, переменные обрабатываются как auto.

Пример описания переменных:

```
int a;              // 1 глобальная переменная a
int main()
{ int b;            // 2 локальная переменная b
  extern int x;     //3 переменная x определена в другом месте
  static c;         //4 локальная статическая переменная c
  a =1;             // 5 присваивание глобальной переменной
  int a;            // 6 локальная переменная a
  a = 2;            // 7 присваивание локальной переменной
  ::a = 3;          //8 присваивание глобальной переменной
  return 0;}
int x = 4;          //9 определение и инициализация x
```

В этом примере глобальная переменная *a* определена вне всех блоков. Память под нее выделяется в сегменте данных в начале работы программы, областью действия является вся программа. Область видимости — вся программа, кроме строк 6-7, так как в них определяется локальная переменная с тем же именем, область действия которой начинается с точки ее описания и заканчивается при выходе из блока. В строке 8 происходит обращение к глобальной переменной *a* через операцию доступа к области видимости ::.

Переменные *b* и *c* — локальные, область их видимости — блок, но время жизни различно: память под *b* выделяется в стеке при входе в блок и освобождается при выходе из него, а переменная *c* располагается в сегменте данных и существует все время, пока работает программа.

Если при определении начальное значение переменных явным образом не задается, компилятор присваивает глобальным и статическим переменным случайное значение соответствующего типа. Автоматические переменные не инициализируются. Имя переменной должно быть уникальным в своей области действия (например, в одном блоке не может быть двух переменных с одинаковыми именами).

2.5. Модели памяти

Вся память, используемая программой, распределяется между сегментом данных, стеком и кучей. Дополнительно память выделяется под код самой программы.

В сегменте данных хранятся внешние (глобальные) и статические идентификаторы (имеющие спецификаторы extren и static). Память под данные идентификаторы в сегменте данных выделяется при их описании и освобождается перед непосредственным завершением работы программы.

Стек используется для хранения локальных (auto – автоматических) идентификаторов. Память под данные идентификаторы выделяется в стеке при их описании и освобождается при завершении работы блока, в котором они описаны.

Куча используется для хранения данных, работа с которыми реализуется через указатели и ссылки. Сами указатели хранятся либо в сегменте данных, либо в стеке (в зависимости от указанного спецификатора класса памяти), а память для размещения данных динамически выделяется или освобождается в куче программистом с помощью специальных средств языка C++ (см. раздел 6.1).

2.6. Примеры использования функций при решении задач

Пример 1. Вычислить Z=(v1+v2+v3)/3, где v1, v2, v3 - объемы шаров с радиусами r1, r2, r3 соответственно. Объем шара вычислять по формуле $V = \frac{4}{3}\pi R^3$.

Указания по решению задачи. При решении данной задачи эффективней всего использовать функцию, которая будет вычислять объем шара. Радиус шара будет передаваться в качестве параметра.

```
#include <iostream>
#include <cmath>
using namespace std;
float volume(float r) //функция вычисляет объема шара
{  const float pi=3.14;
   return 4.0/3*pi*pow(r,3); }
int main()
{ float r1, r2, r3, z;
  cout <<"Введите радиусы трех шаров =";
  cin >>r1 >>r2 >>r3;
  z=(volume(r1)+volume(r2)+volume(r3))/3;
  cout <<"z=" <<z <<endl;
  return 0;}
```

Результат работы программы:

r1	r2	r3	z
1	2	3	50.24

Пример 2. Дана функция $f(x)=x^3-x^2+x-1$. Найти значение выражения z=f(2a)+f(b+c), где a, b, c – вещественные числа.

```
#include <iostream>
#include <cmath>
using namespace std;
float f(float x) //функция вычисляет f(x)
{  return pow(x,3)-pow(x,2)+x-1;}
int main()
{ float a,b,c, z;
```

Результат работы программы:

```
 cout <<"Ведите значение a, b, c =";  cin >>a >>b >>c;
 z=f(2*a)+f(b+c); cout <<"z=" <<z <<endl;
 return 0; }
```

a	b	c	z
1	2	3	109

Пример 3. Разработать функцию, которая увеличивает положительное число в два раза, а отрицательное число заменяет на противоположное. Продемонстрировать работу данной функции на примере.

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

void func(float &x)
{ x=(x>=0)? 2*x: -x; }

int main()
{ float a;
  cout <<"Введите число =";  cin >>a;
  func(a);  cout <<"a=" <<a <<endl;
  return 0;}
```

Результат работы программы:

a	ответ
12	24
0	0
-4	4

Пример 4. Дана сторона квадрата. С помощью одной функции вычислить его периметр и площадь. Определить, что у заданного квадрата больше – периметр или площадь.

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

void func(float x, float &p, float &s)
{ p=4*x;   s=x*x; }

int main()
{ float a, p, s;
  cout <<"a=";  cin >>a;
  func(a, p, s);
  (p>s)? cout << "периметр": cout <<"площадь";
  return 0;}
```

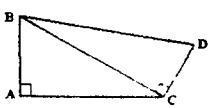
Результат работы программы:

a	ответ
1	периметр
5	площадь

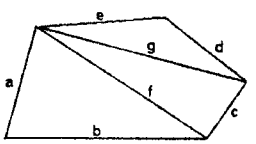
2.7. Упражнения

1. Разработать функцию min(a,b) для нахождения минимального из двух чисел. Вычислить с помощью нее значение выражения z=min(3x,2y)+min(x-y,x+y).
2. Разработать функцию min(a,b) для нахождения минимального из двух чисел. Вычислить с помощью нее минимальное значение из четырех чисел x, y, z, v.
3. Разработать функцию max(a,b) для нахождения максимального из двух чисел. Вычислить с помощью нее значение выражения z=max(x,2y-x)+max(5x+3y,y).
4. Разработать функцию f(x), которая вычисляет значение по следующей формуле: $f(x)=x^3-\sin x$. Определить, в какой из двух точек a или b, функция принимает наибольшее значение.
5. Разработать функцию f(x), которая вычисляет значение по следующей формуле: f(x)=cos(2x)+sin(x-3). Определить, в какой из двух точек a или b, функция принимает наименьшее значение.
6. Разработать функцию f(x), которая возвращает младшую цифру натурального числа x. Вычислить с помощью нее значение выражения z=f(a)+f(b).

7. Разработать функцию f(x), которая возвращает вторую справа цифру натурального числа x. Вычислить с помощью нее значение выражения z=f(a)+f(b)-f(c).
8. Разработать функцию f(n), которая для заданного натурального числа n находит значение $\sqrt{n+n}$. Вычислить с помощью нее значение выражения $\frac{\sqrt{6}+6}{2}+\frac{\sqrt{13}+13}{2}+\frac{\sqrt{21}+21}{2}$.
9. Разработать функцию f(n, x), которая для заданного натурального числа n и вещественного x находит значение выражения $\frac{x^n}{n}$. Вычислить с помощью данной функции значение выражения $\frac{x^2}{2}+\frac{x^4}{4}+\frac{x^6}{6}$.
10. Разработать функцию f(x), которая нечетное число заменяет на 0, а четное число уменьшает в два раза. Продемонстрировать работу данной функции на примере.
11. Разработать функцию f(x), которая число, кратное 5, уменьшает в 5 раз, а остальные числа увеличивает на 1. Продемонстрировать работу данной функции на примере.
12. Разработать функцию f(x), которая в двузначном числе меняет цифры местами, а остальные числа оставляет без изменения. Продемонстрировать работу данной функции на примере.
13. Разработать функцию f(x), которая в трехзначном числе меняет местами первую с последней цифрой, а остальные числа оставляет без изменения. Продемонстрировать работу данной функции на примере.
14. Разработать функцию f(a, b), которая по катетам a и b вычисляет гипотенузу. С помощью данной функции найти периметр фигуры ABCD по заданным сторонам AB, AC и DC.



15. Разработать функцию f(x, y, z), которая по длинам сторон треугольника x, y, z вычисляет его площадь. С помощью данной функции по заданным вещественным числам a, b, c, d, e, f, g найти площадь пятиугольника, изображенного на рисунке.



16. Разработать функцию f(x_1, y_1, x_2, y_2), которая вычисляет длину отрезка по координатам вершин (x_1, y_1) и (x_2, y_2), и функцию d(a, b, c), которая вычисляет периметр треугольника по длинам сторон a, b, c. С помощью данных функций найти периметр треугольника, заданного координатами своих вершин.
17. Разработать функцию f(x_1, y_1, x_2, y_2), которая вычисляет длину отрезка по координатам вершин (x_1, y_1) и (x_2, y_2), и функцию max(a, b), которая вычисляет максимальное из чисел a, b. С помощью данных функций определить, какая из трех точек на плоскости наиболее удалена от начала координат.
18. Разработать функцию f(x_1, y_1, x_2, y_2), которая вычисляет длину отрезка по координатам вершин (x_1, y_1) и (x_2, y_2), и функцию min(a, b), которая вычисляет минимальное из чисел a, b. С помощью данных функций найти две из трех заданных точек на плоскости, расстояние между которыми минимально.
19. Разработать функцию f(x_1, y_1, x_2, y_2), которая вычисляет длину отрезка по координатам вершин (x_1, y_1) и (x_2, y_2), и функцию t(a, b, c), которая проверяет, суще-

ствует ли треугольник с длинами сторон a, b, c. С помощью данных функций проверить, можно ли построить треугольник по трем заданным точкам на плоскости.

20. Разработать функцию $f(x_1, y_1, x_2, y_2)$, которая вычисляет длину отрезка по координатам вершин (x_1, y_1) и (x_2, y_2), и функцию t(a, b, c), которая проверяет, существует ли треугольник с длинами сторон a, b, c. С помощью данных функций проверить, сколько различных треугольников можно построить по четырем заданным точкам на плоскости.

3. ОПЕРАТОРЫ С++

Программа на языке С++ состоит из последовательности операторов, каждый из которых определяет законченное описание некоторого действия и заканчивается точкой с запятой. Все операторы можно разделить на 4 группы: операторы следования, операторы ветвления, операторы цикла и операторы передачи управления.

3.1. Операторы следования

Операторы следования выполняются компилятором в естественном порядке: начиная с первого до последнего. К операторам следования относятся: оператор выражение и составной оператор.

Любое *выражение*, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении значения выражения или выполнении законченного действия. Например:

```
++i;                 //оператор инкремента
x+=y;                //оператор сложение с присваиванием
f(a, b);             // вызов функции
x=max(a, b)+a*b;     //вычисление сложного выражения
```

Частным случаем оператора выражения является *пустой оператор* ; Он используется, когда по синтаксису оператор требуется, а по смыслу — нет. В этом случае лишний символ ; является пустым оператором и вполне допустим, хотя и не всегда безопасен. Например, случайный символ ; после условия оператора *while* или *if* может совершенно поменять работу этого оператора.

Составной оператор или *блок* представляет собой последовательность операторов, заключенных в фигурные скобки. Блок обладает собственной *областью видимости*: объявленные внутри блока имена доступны только внутри данного блока или блоков, вложенных в него. Составные операторы применяются в случае, когда правила языка предусматривают наличие только одного оператора, а логика программы требует нескольких операторов. Например, тело цикла *while* должно состоять только из одного оператора. Если заключить несколько операторов в фигурные скобки, то получится блок, который будет рассматриваться компилятором как единый оператор.

3.2. Операторы ветвления

Операторы ветвления позволяют изменить порядок выполнения операторов в программе. К операторам ветвления относятся условный оператор *if* и оператор выбора *switch*.

Условный оператор if

Условный оператор *if* используется для разветвления процесса обработки данных на два направления. Он может иметь одну из форм: *сокращенную* или *полную*.

Формат *сокращенного оператора if*:

```
if (B) S;
```

где *B* –логическое или арифметическое выражение, истинность которого проверяется; *S* – один оператор: простой или составной.

При выполнении сокращенной формы оператора *if* сначала вычисляется выражение *B*, затем проводится анализ его результата: если *B* истинно, то выполняется оператор *S*; если *B* ложно, то оператор *S* пропускается. Таким образом, с помощью сокращенной формы оператора *if* можно либо выполнить оператор S, либо пропустить его.

Формат *полного оператора if*:

```
if (B) S1; else S2;
```

где *B* – логическое или арифметическое выражение, истинность которого проверяется; *S1, S2*- один оператор: простой или составной.

При выполнении полной формы оператора *if* сначала вычисляется выражение *B*, затем анализируется его результат: если *B* истинно, то выполняется оператор *S1*, а оператор *S2* пропускается; если *B* ложно, то выполняется оператор *S2*, а *S1* – пропускается. Таким образом, с помощью полной формы оператора *if* можно выбрать одно из двух альтернативных действий процесса обработки данных.

Рассмотрим несколько примеров записи условного оператора *if*:

```
if (a > 0) x=y;          // Сокращенная форма с простым оператором
if (++i) {x=y; y=2*z;}   // Сокращенная форма с составным оператором
if (a > 0 || b<0) x=y; else  x=z;    // Полная форма с простым оператором
if (i+j-1) { x= 0; y= 1;} else {x=1; y:=0;}  // Полная форма с составными операторами
```

Операторы S1 и S2 могут также являться операторами *if*. Такие операторы называют вложенными. При этом ключевое слово *else* связывается с ближайшим предыдущим словом *if*, которое еще не связано ни с одним *else*. Рассмотрим несколько примеров алгоритмов с использованием вложенных условных операторов:

Пример 1

```
if (A< B)
   if (C < D)
      if (E < F) X= Q;
      else X = R;
   else X= Z;
else X = Y;
```

Уровни вложенности операторов if: 3, 2, 1

Пример 2

```
if (A < B)
   if (C < D) X =Y;
   else X = Z;
else
   if (E < F) X= R;
   else X = Q;
```

Уровни вложенности операторов if: 2, 2, 1

Оператор выбора switch

Оператор выбора *switch* предназначен для разветвления процесса вычислений на несколько направлений. Формат оператора:

```
switch ( <выражение> )
{ case <константное_выражение_1>: [<оператор 1>]
  case <константное_выражение_2>: [<оператор 2>]
  ...
  case <константное_выражение_n>: [<оператор n>]
  [default: <оператор> ] }
```

Выражение, стоящее за ключевым словом *switch*, должно иметь арифметический тип или тип указатель. Все константные выражения должны иметь разные значения, но совпадать с типом выражения, стоящим после *switch*, или приводиться к нему. Ключевое слово *case* и расположенное после него константное выражение называют также меткой *case*.

Выполнение оператора начинается с вычисления выражения, расположенного за ключевым словом *switch*. Полученный результат сравнивается с меткой *case*. Если результат выражения соответствует метке *case*, то выполняется оператор, стоящий после этой метки. Затем последовательно выполняются все операторы до конца оператора *switch* , если только их выполнение не будет прервано с помощью оператора передачи управления *break* (см. пример). При использовании оператора *break* происходит выход из *switch*, и управление переходит к первому после него оператору. Если же совпадения выражения ни с одной меткой *case* не произошло, то выполняется оператор, стоящий после слова *default*, а при его отсутствии управление передается следующему за *switch* оператору.

Пример. Известен порядковый номер дня недели. Вывести на экран его название.

```
#include <iostream>
using namespace std;
int main()
{ int x; cin >>x;
  switch  (x)
  { case 1: cout <<"понедельник" ; break;
    case 2: cout <<"вторник"; break;
    case 3:  cout <<"среда"; break;
    case 4:  cout <<"четверг"; break;
    case 5:  cout <<"пятница"; break;
    case 6:  cout <<"суббота"; break;
    case 7:  cout <<"воскресенье"; break;
    default: cout <<"вы ошиблись";}
  return 0; }
```

Результат работы программы:

x	Сообщение на экране
2	вторник
4	четверг
10	вы ошиблись

Если в нашем примере исключить операторы break, то получим следующий результат:

```
#include <iostream>
using namespace std;
int main()
{ int x;  cin >>x;
  switch  (x)
  { case 1: cout <<"понедельник" ;
```

```
  case 2: cout <<"вторник";
  case 3:  cout <<"среда";
  case 4:  cout <<"четверг";
  case 5:  cout <<"пятница";
  case 6:  cout <<"суббота";
  case 7:  cout <<"воскресенье";
  default: cout <<"вы ошиблись";}
 return 0; }
```

Результат работы программы:

x	Сообщение на экране
4	четвергпятницасубботавоскресеньевы ошиблись
6	субботавоскресеньевы ошиблись
10	вы ошиблись

Существует одна стандартная ситуация, когда оператор *break* не нужен. Речь идет о случае, когда одна и та же последовательность действий должна выполняться для нескольких меток *case*. В этом случае метки *case* располагают последовательно одну за другой через двоеточие. После последней метки указывают действие, которое нужно выполнить. Например:

```
#include <iostream>
using namespace std;
int main()
{ int x; cin >>x;
 switch  (x)
 { case 1: case 2: case 3: case 4:
   case 5: cout <<"рабочий день"; break;
   case 6:
   case 7:  cout <<"выходной"; break
   default: cout <<"вы ошиблись"; }
 return 0; }
```

Результат работы программы:

x	Сообщение на экране
4	рабочий день
6	выходной
10	вы ошиблись

В этом примере для меток со значениями из диапазона от 1 до 5 выполняется одно действие, а для меток со значениями 6-7 выполняется другое действие.

3.3. Примеры использования операторов ветвления при решении задач

1. Для произвольных значений аргументов вычислить значение функции, заданной следующим образом: $y(x)=\frac{1}{x}+\sqrt{x+1}$. Если в некоторой точке вычислить значение функции окажется невозможно, то вывести на экран сообщение «функция не определена».

Указание по решению задачи. Данную задачу можно решить двумя способами.

1 способ. Заданная функция не определена в том случае, когда:

знаменатель первого слагаемого равен нулю		x=0		x=0
Или	⇔	*или*	⇔	*или*
подкоренное выражение второго слагаемого отрицательное		x+1<0		x<-1

В остальных случаях функция определена. Программа выглядит следующим образом:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{  float x,y;
   cout <<"x:=";  cin >> x;
   if (!x || x<-1)              //проверка условия неопределенности функции
    cout <<"Функция не определена" <<endl;
    else {y=1/x+sqrt(x+1); cout <<"x=" << x <<"\t" <<"y=" <<y <<endl;}
```

```
return 0;}
```

II способ. Заданная функция определена в том случае, когда:

знаменатель первого слагаемого не равен нулю		$x\neq0$		$x\neq0$
И	$\Leftrightarrow$	*и*	$\Leftrightarrow$	*и*
подкоренное выражение второго слагаемого неотрицательно		$x+1\geq0$		$x\geq-1$

В остальных случаях функция не определена. Программа выглядит следующим образом:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ float x,y;
 cout <<"x=";  cin >> x;
 if ((x) && (x>=-1)) //проверка условия определенности функции
  {y=1/x+sqrt(x+1); cout <<"x=" << x <<"\t" <<"y=" <<y <<endl;}
  else cout <<"Функция не определена" <<endl;
return 0;}
```

Обе программы дадут нам следующий результат:

x	y(x)
0	функция не определена
2	1.50

2. Для произвольных значений аргументов вычислить значение функции, заданной следующим образом: $y(x)=\begin{cases}\left(x^3+1\right)^2, \text{ при } x<0;\\ 0, \text{при } 0\le x<1;\\ \left|x^2-5x+1\right|, \text{при } x\ge 1.\end{cases}$

Указания по решению задачи. Вся числовая прямая Ox разбивается на три непересекающихся интервала, (-∞; 0), [0; 1), [1; +∞). На каждом интервале функция задается своею ветвью. Заданная точка x может попасть только в один из указанных интервалов. Чтобы определить, в какой из интервалов попала точка, воспользуемся следующим алгоритмом. Если $x<0$, то x попадает в первый интервал, и функцию высчитываем по первой ветви, после чего проверка заканчивается. Если это условие ложно, то истинно условие $x\geq0$, и для того чтобы точка попала во второй интервал достаточно, чтобы выполнялось условие $x<1$. Если выполняется это условие, то точка x попадает во второй интервал и мы определяем функцию по второй ветви, после чего заканчиваем вычисления. В противном случае, точка может принадлежать только третьему интервалу, поэтому дополнительная проверка не проводится, а сразу вычисляем функцию по третьей ветви. Приведенный алгоритм можно реализовать с помощью вложенных операторов *if*

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ float x,y;
 cout <<"x="; cin >>x;
 if (x<0)   y=pow(pow(x,3)+1,2);        //проверяем условие первой ветви
  else if (x<1) y=0;                    //проверяем условие второй ветви
       else y=fabs(x*x-5*x+1);
cout << "f( " << x << ")= " <<y;
return 0; }
```

Результат работы программы:

координата точки	ответ
0	*0*

1	*3*
-2	*49*

3. Дана точка на плоскости с координатами (x, y). Составить программу, которая выдает одно из сообщений «Да», «Нет», «На границе» в зависимости от того, лежит ли точка внутри заштрихованной области, вне заштрихованной области или на ее границе.

Указания по решению задачи. Всю плоскость можно разбить на три непересекающихся множества точек: I_1 – множество точек, лежащих внутри области; I_2 – множество точек, лежащих вне области; I_3 – множество точек, образующих границу области. Точка с координатами (x, y) может принадлежать только одному из них. Поэтому проверку можно проводить по аналогии с алгоритмом, приведенном в примере 2. Однако множества I_1, I_2, I_3 значительно труднее описать математически, чем интервалы в примере 2. Поэтому для непосредственной проверки выбираются те два множества, которые наиболее просто описать математически. Обычно труднее всего описать точки границы области. Например, для рис. 2.1 множества задаются следующим образом:

I_1: $x^2 + y^2 < 10^2$; I_2: $x^2 + y^2 > 10^2$; I_3: $x^2 + y^2 = 10^2$.

Для рис. 2.2 множества задаются следующим образом:

I_1: $|x| < 10$ и $|y| < 5$; I_2: $|x| > 10$ или $|y| > 5$;

I_3: ($|x| <= 10$ и $y = 5$) или ($|x| <= 10$ и $y = -5$) или ($|y| < 5$ и $x = 10$) или ($|y| < 5$ и $x = -10$).

Таким образом, для рис. 2.1 описание всех множеств равносильно по сложности, а для рис. 2.2 описать множество I_3 значительно сложнее.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ float x,y;
  cout <<"x="; cin>>x;
  cout <<"y="; cin >>y;
   if (x*x+y*y<100)    //точка внутри области?
    cout <<"Да";
    else if (x*x+y*y>100) //точка вне области?
          cout<<"Нет";
          else cout <<"на границе";
  return 0;}
```

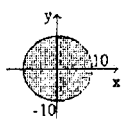


рис. 2.1

Результаты работы программы:

координаты точек		ответ
0	0	да
10	0	на границе
-12	13	нет

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ float x,y;
  cout <<"x="; cin>>x;
  cout <<"y="; cin >>y;
  if (fabs(x)<10 && fabs(y)<5) //точка внутри области?
  cout <<"Да";
  else if (fabs(x)>10 || fabs(y)>5) //точка вне области?
        cout<<"Нет";
        else cout <<"на границе";
 return 0;}
```

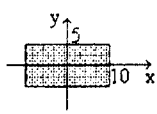


рис. 2.2

Результаты работы программы:

координаты точек		ответ
0	0	да
10	5	на границе
-12	13	нет

4. Дан номер фигуры (1- квадрат, 2 – треугольник). По номеру фигуры запросить необходимые данные для вычисления площади, произвести вычисление площади фигуры и вывести полученные данные на экран.

```
#include <iostream>
```

```
#include <cmath>
using namespace std;
int main()
{ int x;
 cout << "Программа подсчитывает площадь:\n1. квадрата;\n2. треугольника.\n3. выход из
программы";
 cout <<"Укажите номер фигуры или завершите работу с программой.\n";
 cin >>x;
 switch (x)
 {case 1 :{ cout << "введите длину стороны квадрата\n" ;
            float a; cin >>a;
            if (a>0) cout << "Площадь квадрата со стороной" <<a <<"равна\t" <<a*a;
             else cout <<"Квадрат не существует\n";
            break;}
  case 2: {cout << "введите длины сторон треугольника\n";
           float a,b,c,p, s; cin >> a >>b >>c;
           if (a+b>c && a+c>b && b+c>a)
            {p=(a+b+c)/2;   s= sqrt(p*(p-a)*(p-b)*(p-c));
             cout <<"Площадь треугольника со сторонами" <<a <<b <<c <<"равна\t" <<s;}
            else cout<<"Треугольник не существует\n";
            break;}
  case 3:break;
  default: cout <<"Номер фигуры указан не верно\n";}
return 0; }
```

3.4. Операторы цикла

Операторы цикла используются для организации многократно повторяющихся вычислений. К операторам цикла относятся: цикл с предусловием *while*, цикл с постусловием *do while* и цикл с параметром *for*.

Цикл с предусловием while

Оператор цикла *while* организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Формат цикла *while*:

while (B) S;

где *B* – выражение, истинность которого проверяется (условие завершения цикла); *S* – тело цикла: один оператор (простой или составной).

Перед каждым выполнением тела цикла анализируется значение выражения *B*: если оно истинно, то выполняется тело цикла, и управление передается на повторную проверку условия *B*; если значение *B* ложно – цикл завершается и управление передается на оператор, следующий за оператором *S*.

Если результат выражения *B* окажется ложным при первой проверке, то тело цикла не выполнится ни разу. Отметим, что если условие *B* во время работы цикла не будет изменяться, то цикл не сможет завершить работу. Возникнет ситуация зацикливания. Поэтому внутри тела должны находиться операторы, приводящие к изменению значения выражения *B* так, чтобы цикл мог корректно завершиться.

В качестве иллюстрации выполнения цикла *while* рассмотрим программу вывода на экран целых чисел из интервала от 1 до *n*.

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

```
int main()
{ int n, i=1;
  cout <<"n="; cin >>n;
  while (i<=n)          //пока i меньше или равно n
   { cout<<i<<"\t";     //выводим на экран значение i
     ++i; }             // увеличиваем i на единицу
  return 0; }
```

Результаты работы программы:

n	ответ
10	1 2 3 4 5 6 7 8 9 10

Замечание. Используя операцию постфиксного инкремента, тело цикла можно заменить одной командой *cout <<i++ <<"\t".*

Цикл с постусловием do while

Оператор цикла *do while* также организует выполнение одного оператора (простого или составного) неизвестное заранее число раз. Однако в отличие от цикла *while* условие завершения цикла проверяется после выполнения тела цикла. Формат цикла *do while*:

do S while (B);

где *B* – выражение, истинность которого проверяется (условие завершения цикла); *S* – тело цикла: один оператор (простой или блок).

Сначала выполняется оператор *S*, а затем анализируется значение выражения *B*: если оно истинно, то управление передается оператору *S*, если ложно - цикл завершается, и управление передается на оператор, следующий за условием *B*.

В операторе *do while*, так же как и в операторе *while*, возможна ситуация зацикливания в случае, если условие *B* всегда будет оставаться истинным, Но так как условие *B* проверяется после выполнения тела цикла, то в любом случае тело цикла выполнится хотя бы один раз.

В качестве иллюстрации выполнения цикла *do while* рассмотрим программу вывода на экран целых чисел из интервала от 1 до *n*.

```
#include <iostream>
using namespace std;
int main()
{ int n, i=1;
  cout <<"n="; cin >>n;
  do       //выводим на экран i, а затем увеличиваем
    cout<<i++<<"\t";     //ее значение на единицу
  while (i<=n);  //до тех пор пока i меньше или равна n
 return 0; }
```

Результаты работы программы:

n	ответ
10	1 2 3 4 5 6 7 8 9 10

Цикл с параметром for

Цикл с параметром имеет следующую структуру:

for (<инициализация>; <выражение>; <модификация>) <оператор>;

Инициализация используется для объявления и присвоения начальных значений величинам, используемым в цикле. В этой части можно записать несколько операторов, разделенных запятой. Областью действия переменных, объявленных в части инициализации цикла, является цикл и вложенные блоки. Инициализация выполняется один раз в начале исполнения цикла. *Выражение* определяет условие выполнения цикла: если его результат истинен, цикл выполняется. Истинность выражения проверяется перед каждым выполнением тела цикла, таким образом, цикл с параметром реализован как цикл с предусловием. *Модификация* выполняется после каж-

дой итерации цикла и служит обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую. *Оператор* (простой или составной) представляет собой тело цикла.

Любая из частей оператора *for* (инициализация, выражение, модификация, оператор) может отсутствовать, но точку с запятой, определяющую позицию пропускаемой части, надо оставить.

```
#include <iostream>
using namespace std;
int main()
{ int n;  cout <<"n="; cin >>n;
 for (int i=1; i<=n;i++)   //для i от 1 до n с шагом 1
  cout<<i<<"\t";     //выводить на экран значение i
 return 0;}
```

Результаты работы программы:

n	ответ
10	1 2 3 4 5 6 7 8 9 10

Замечание. Используя операцию постфиксного инкремента при выводе данных на экран, цикл *for* можно преобразовать следующим образом: *for (int i=1; i<=n;) cout<<i++<<"\t";*. В этом случае в заголовке цикла *for* отсутствует блок модификации.

Вложенные циклы

Циклы могут быть простые или вложенные (кратные, циклы в цикле). Вложенными могут быть циклы любых типов: *while, do while, for*. Структура вложенных циклов на примере типа *for* приведена ниже:

```
                                          Уровни вложенности
for ( i=1;i<ik;i++)                                  }
{ ...                                                |
   for (j=10; j>jk; j--)                             |
     { ... for( k=1; k<kk; j+=2){ ...}    }3 }2 }1
     ...}                                            |
...}                                                 }
```

Каждый внутренний цикл должен быть полностью вложен во все внешние циклы. «Пересечения» циклов не допускается.

Рассмотрим пример использования вложенных циклов, который позволит вывести на экран следующую таблицу:

```
2   2   2   2   2
2   2   2   2   2
2   2   2   2   2
2   2   2   2   2
```

```
#include <iostream>
using namespace std;
int main()
{ for (int i=1; i<=4; ++i,cout<<endl) //внешний цикл
   for (int j=1; j<=5; ++j)          //внутренний цикл
    cout<<"2\t";                     //тело внутреннего цикла
   return 0;}
```

Замечание. Внешний цикл определяет количество строк, выводимых на экран. Обратите внимание на то, что в блоке модификации данного цикла стоят два оператора. Первый *++i* будет увеличивать значение *i* на единицу после каждого выполнения внутреннего цикла, а второй – *cout <<endl* будет переводить выходной поток на новую строку. Внутренний цикл является телом внешнего цикла. Внутренний цикл определяет, сколько чисел нужно вывести в каждой строке, а в теле внутреннего цикла выводится нужное число.

Рассмотрим еще один пример использования вложенных циклов, который позволит вывести на экран следующую таблицу:

```
1
1   3
1   3   5
```

```
#include <iostream>
using namespace std;
int main()
```

```
1   3   5   7                { for (int i=1; i<=5; ++i,cout<<endl) //внешний цикл
1   3   5   7   9              for (int j=1; j<=2*i-1; j+=2)     //внутренний цикл
                                cout<<j<<"\t";          //тело внутреннего цикла
                               return 0;}
```

Замечание. В данном случае таблица состоит из пяти строчек, в каждой из которых печатаются только нечетные числа. Причем последнее нечетное число в строчке зависит от ее номера. Эта зависимость выражается через формулу $k = 2i-1$ (зависимость проверить самостоятельно), где k – последнее число в строке, i – номер текущей строки. Внешний цикл следит за номером текущей строки i, а внутренний цикл будет печатать нечетные числа из диапазона от 1 до $2i-1$.

3.5. Примеры использования операторов цикла при решении задач

1. Написать программу, которая выводит на экран квадраты всех целых чисел от A до B (A и B целые числа, при этом A≤B).

Указания по решению задачи. Необходимо перебрать все целые числа из интервала от A до B. Эти числа представляют собой упорядоченную последовательность, в которой каждое число отличается от предыдущего на 1. При решении данной задачи можно использовать любой оператор цикла.

```
#include <iostream>
using namespace std;
int main()
{ int a, b;
  cout <<"a="; cin >>a;
  cout <<"b="; cin >>b;
  int i=a;
  while (i<=b)
   cout<<i*i++<<"\t";
  return 0;}
```

```
#include <iostream>
using namespace std;
int main()
{ int a, b;
  cout <<"a="; cin >>a;
  cout <<"b="; cin >>b;
  int i=a;
  do cout<<i*i++<<"\t";
  while (i<=b);
  return 0;}
```

Замечание. Рассмотрим выражение i*i++, значение которого выводится на экран в теле каждого из циклов. С учетом приоритетов операций (см. Приложение 3) вначале выполнится операция умножения, результат которой будет помещен в выходной поток, а затем постфиксный инкремент увеличит значение i на единицу.

```
#include <iostream>
using namespace std;
int main()
{ int a, b;
  cout <<"a="; cin >>a;
  cout <<"b="; cin >>b;
  for (int i=a; i<=b; i++)
   cout<<i*i<<"\t";
  return 0; }
```

Три программы дадут нам одинаковый результат:

a	b	ответ
3	9	9 16 25 36 49 64 81

2. Написать программу, которая выводит на экран квадраты всех четных чисел из диапазона от A до B (A и B целые числа, при этом A≤B).

Указания по решению задачи. Из диапазона целых чисел от A до B необходимо выбрать только четные числа. Напомним, что четными называются числа, которые делятся на два без остатка. Кроме того, четные числа представляют собой упорядоченную последовательность, в которой каждое число отличается от предыдущего на 2. Решить эту задачу можно с помощью каждого оператора цикла.

```
#include <iostream>
using namespace std;
int main()
```

```
#include <iostream>
using namespace std;
int main()
```

```
{ int a, b, i;
  cout <<"a="; cin >>a;
  cout <<"b="; cin >>b;
  i=(a%2)? a+1: a;
  while (i<=b)
    {cout<<i*i<<"\t"; i+=2;}
  return 0; }
```

```
{int a, b, i;
 cout <<"a="; cin >>a;
 cout <<"b="; cin >>b;
 i=(a%2)? a+1: a;
 do cout<<i*i<<"\t"; i+=2;
 while (i<=b);
 return 0;}
```

Замечание. Начальное значение переменной *i* определяется с помощью тернарной операции. Если *a* нечетное число, то при делении на 2 мы получим остаток 1, который трактуется компилятором как *истина*, и, следовательно, переменной *i* будет присвоено значение *a+1*. Если же *a* четное число, то при делении на 2 мы получим остаток 0, который трактуется компилятором как *ложь*, и следовательно переменной *i* будет присвоено значение *a*. В результате, независимо от значения переменной *a*, переменной *i* будет присвоено четное значение.

```
#include <iostream>
using namespace std;
int main()
{ int a, b;
  cout <<"a="; cin >>a;
  cout <<"b="; cin >>b;
  for (int i=(a%2)? a+1: a; i<=b; i+=2)
   cout<<i*i<<"\t";
  return 0; }
```

Три программы дадут нам одинаковый результат:

a	b	ответ
3	9	16 36 64

3. Постройте таблицу значений функции $y(x)=\frac{1}{x}+\sqrt{x+1}$ для $x\in[a, b]$ с шагом h.

Если в некоторой точке *x* функция не определена, то выведите на экран сообщение об этом.

```
#include <iostream>
#include <cmath>
using namespace std;
/*вспомогательная функция: выводит на экран значение функции в точке x или сообщение о том, что функция не определена*/
void f(float x)
{ if (!x || x<-1) cout <<"функция неопределена";
  else cout <<1/x+sqrt(x+1); }
int main() //главная функция
{ float a,b,h,x;
  cout <<"a="; cin>>a;
  cout <<"b="; cin>>b;
  cout <<"h="; cin>>h;
  cout<<"x\tf(x)\n";          //выводим заголовок таблицы
  for (x=a; x<=b; x+=h)       //перебираем все числа из отрезка [a, b] с шагом h
   {cout <<x <<"\t";          //выводим на экран значение x
    f(x);                     //выводим на экран значение функции в точке x
    cout <<endl;}             //переводим выходной поток на новую строку
 return 0; }
```

Результат работы программы для a=-2, b=4, h=2:

x	f(x)
-2	функция не определена
0	функция не определена
2	2.23205
4	2.48607

4. Постройте таблицу значений функции $y(x)=\begin{cases}(x^3+1)^2, & \text{при } x<0;\\ 0, & \text{при } 0\le x<1;\\ |x^2-5x+1|, & \text{при } x\ge 1.\end{cases}$ для $x\in[a, b]$

с шагом h.

```
#include <iostream>
#include <cmath>
using namespace std;
float f(float x)  //вспомогательная функция: возвращает значение функции в точке x
{ if (x<0) return pow(pow(x,3)+1,2);      //условие первой ветви
   else if (x<1) return 0;                //условие второй ветви
        else return fabs(x*x-5*x+1); }    // по умолчанию третья ветвь
int main()  //главная функция
{ float a,b,h,x;
  cout <<"a="; cin>>a;
  cout <<"b="; cin>>b;
  cout <<"h="; cin>>h;
  cout<<"x\tf(x)\n";        //выводим заголовок таблицы
  for (x=a; x<=b; x+=h)  //перебираем все числа из отрезка [a, b]  с шагом h и выводим на
   cout <<x <<"\t"<< f(x)<<endl;   // экран значение x и значение функции в точке x
  return 0; }
```

Результат работы программы для a=-3, b=3, h=1.5:

x	f(x)
-3	676
-1.5	5.64063
0	0
1.5	4.25
3	5

3.6. Операторы безусловного перехода

В C++ есть четыре оператора, изменяющие естественный порядок выполнения операторов: оператор безусловного перехода *goto*, оператор выхода *break*, оператор перехода к следующей итерации цикла *continue*, оператор возврата из функции *return*.

Оператор безусловного перехода goto

Оператор безусловного перехода *goto* имеет формат:

```
goto <метка>;
```

В теле той же функции должна присутствовать ровно одна конструкция вида:

```
<метка>: <оператор>;
```

Оператор *goto* передает управление на помеченный меткой оператор. Рассмотрим пример использования оператора goto:

```
#include <iostream>
using namespace std;
int main()
{ float x;
  metka: cout <<"x=";          //метка
         cin>>x;
```

```
    if (x) cout<<"y="<<1/x<<endl;
     else {cout<<"функция не определена\n";
           goto metka;}          // передача управления метке
 return 0;}
```

Замечание. В данном примере при попытке ввести 0 на экран будет выведено сообщение «функция не определена», после чего управление будет передано метке, и программа повторно попросит ввести значение x.

Следует учитывать, что использование оператора goto затрудняет чтение больших по объему программ, поэтому использовать его нужно только в крайних случаях.

Оператор выхода break

Оператор *break* используется внутри операторов ветвления и цикла для обеспечения перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится *break*.

В разделе 3.2. оператор *break* применялся для выхода из оператора *switch*, аналогичным образом он может применяться для выхода из других операторов.

Оператор перехода к следующей итерации цикла continue

Оператор перехода к следующей итерации цикла *continue* пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации (повторение тела цикла). Рассмотрим оператор *continue* на примере.

```
#include <iostream>
using namespace std;
int main()
{ for (int i=1; i<100; ++i)  // перебираем все числа от 1 до 99
   {if (i % 2) continue;     // если число нечетное, то переходим к следующей итерации
    cout<<i<<"\t"; }         //выводим число на экран
 return 0;}
```

Замечание. В результате данной программы на экран будут выведены только четные числа из интервала от 1 до 100, т.к. для нечетных чисел текущая итерация цикла прерывалась и команда *cout<<i<<"\t"* не выполнялась.

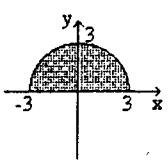
Оператор возврата из функции return

Оператор возврата из функции *return* завершает выполнение функции и передает управление в точку ее вызова. Данный оператор мы неоднократно использовали при разработке функций, возвращающих значение.

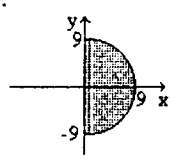
3.7. Упражнения

I. Дана точка на плоскости с координатами (x, y). Составить программу, которая выдает одно из сообщений «Да», «Нет», «На границе» в зависимости от того, лежит ли точка внутри заштрихованной области, вне заштрихованной области или на ее границе. Области задаются графически следующим образом:

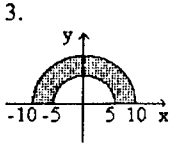
1.



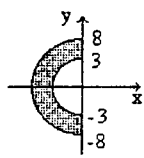
2.



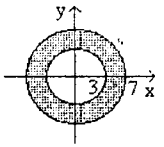
3.



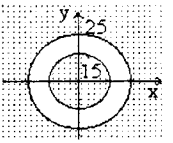
4.



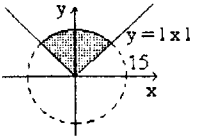
5.



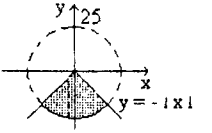
6.



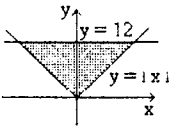
7.



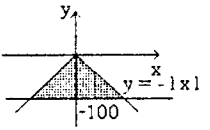
8.



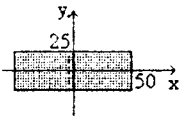
9.



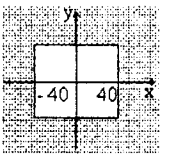
10.



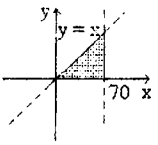
11.



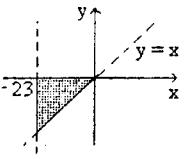
12.



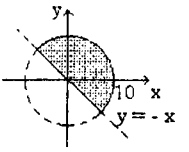
13.



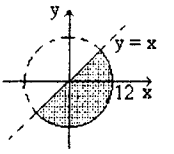
14.



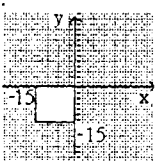
15.



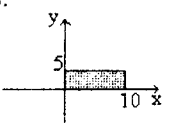
16.



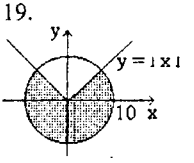
17.



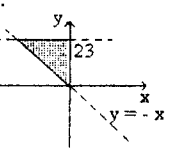
18.



19.



20.



II. Составить программу.

1) Дан порядковый номер месяца, вывести на экран его название.
2) Дан порядковый номер дня месяца, вывести на экран количество дней оставшихся до конца месяца.
3) Дан номер масти m ($1 \le m \le 4$), определить название масти. Масти нумеруются: «пики» - 1, «трефы» - 2, «бубны» - 3, «червы» - 4.
4) Дан номер карты k ($6 \le k \le 14$), определить достоинство карты. Достоинства определяются по следующему правилу: «туз» - 14, «король» - 13, «дама» - 12, «валет» - 11, «десятка» - 10, …, «шестерка» - 6.
5) Дан номер масти m ($1 \le m \le 4$) и номер достоинства карты k ($6 \le k \le 14$). Определить полное название соответствующей карты в виде «дама пик», «шестерка бубен» и т.д.
6) С 1 января 1990 года по некоторый день прошло m месяцев, определить название текущего месяца.
7) Дано расписание приемных часов врача. Вывести на экран приемные часы врача в заданный день недели (расписание придумать самостоятельно).
8) Проведен тест, оцениваемый в целочисленный баллах от нуля до ста. Вывести на экран оценку тестируемого в зависимости от набранного количества баллов: от 90 до 100 – «отлично», от 70 до 89 – «хорошо», от 50 до 69 – «удовлетворительно», менее 50 – «неудовлетворительно».

9) Даны два числа и арифметическая операция. Вывести на экран результат этой операции.
10) Дан год. Вывести на экран название животного, символизирующего этот год по восточному календарю.
11) Дан возраст человека мужского пола в годах. Вывести на экран возрастную категорию: до года – «младенец», от года до 11 лет – «ребенок», от 12 до 15 лет – «подросток», от 16 до 25 лет – «юноша», от 26 до 70 лет – «мужчина», более 70 лет – «пожилой человек».
12) Дан пол человека: м – мужчина, ж – женщина. Вывести на экран возможные мужские и женские имена в зависимости от введенного пола.
13) Дан признак транспортного средства: а – автомобиль, в – велосипед, м - мотоцикл, с – самолет, п – поезд. Вывести на экран максимальную скорость транспортного средства в зависимости от введенного признака.
14) Дан номер телевизионного канала (от 1 до 5) . Вывести на экран наиболее популярные программы заданного канала.
15) Дан признак геометрической фигуры на плоскости: к – круг, п – прямоугольник, т - треугольник. Вывести на экран периметр и площадь заданной фигуры (данные, необходимые для расчетов, запросить у пользователя).

III. Вывести на экран:

Замечание. Решите каждую задачу тремя способами: используя операторы цикла *while, do while* и *for*.

1) целые числа 1, 3, 5, …, 21 в строчку через пробел:
2) целые числа 10, 12, 14, …, 60 в обратном порядке в столбик;
3) числа следующим образом:

10 10.4
11 11.4
…
25 25.4

4) числа следующим образом:

25 25.5 24.8
26 26.5 25.8
…
35 35.5 34.8

5) таблицу соответствия между весом в фунтах и весом в килограммах для значений 1, 2, 3, …, 10 фунтов (1 фунтов = 453г);
6) таблицу перевода 5, 10, 15, …, 120 долларов США в рубли по текущему курсу (значение курса вводится с клавиатуры);
7) таблицу стоимости для 10, 20, 30,…, 100 штук товара, при условии что одна штука товара стоит *x* руб (значение *x* водится с клавиатуры);
8) таблицу перевода расстояний в дюймах в сантиметры для значений 2, 4, 6, …, 12 дюймов (1 дюйм = 25.4 мм);
9) кубы всех целых чисел из диапазона от A до B (A≤B) в обратном порядке;
10) все целые числа из диапазона от A до B (A≤B), оканчивающиеся на цифру X;
11) все целые числа из диапазона от A до B (A≤B), оканчивающиеся на цифру X или У;
12) все целые числа из диапазона от A до B (A≤B), оканчивающиеся на любую четную цифру;
13) только положительные целые числа из диапазона от A до B (A≤B);
14) все целые числа из диапазона от A до B, кратные трем (A≤B);
15) все четные числа из диапазона от A до B, кратные трем (A≤B);

16) только отрицательные четные числа из диапазона от A до B (A≤B);
17) все двухзначные числа, в записи которых все цифры разные;
18) все двухзначные числа, в которых старшая цифра отличается от младшей не больше чем на 1;
19) все трехзначные числа, которые начинаются и заканчиваются на одну и ту же цифру;
20) все трехзначные числа, в которых хотя бы две цифры повторяются.

IV. Вывести на экран числа в виде следующей таблицы:

1)
```
5 5 5 5 5 5
5 5 5 5 5 5
5 5 5 5 5 5
5 5 5 5 5 5
```

2)
```
1 2 3 ... 10
1 2 3 ... 10
1 2 3 ... 10
1 2 3 ... 10
```

3)
```
-10 -9 -8 ... 12
-10 -9 -8 ... 12
-10 -9 -8 ... 12
-10 -9 -8 ... 12
-10 -9 -8 ... 12
```

4)
```
41 42 43 ... 50
51 52 53 ... 60
61 62 63 ... 70
...
71 72 73 ... 80
```

5)
```
5
5 5
5 5 5
5 5 5 5
5 5 5 5 5
```

6)
```
1 1 1 1 1
1 1 1 1
1 1 1
1 1
1
```

7)
```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

8)
```
6 6 6 6 6
7 7 7 7
8 8 8
9 9
10
```

9)
```
7
6 6
5 5 5
4 4 4 4
3 3 3 3 3
```

10)
```
8 8 8 8 8
7 7 7 7
6 6 6
5 5
4
```

11)
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

12)
```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

13)
```
0 1 2 3 4
0 1 2 3
0 1 2
0 1
0
```

14)
```
4 3 2 1 0
3 2 1 0
2 1 0
1 0
0
```

15)
```
1
0
2 2
0 0
3 3 3
0 0 0
4 4 4 4
0 0 0 0
5 5 5 5 5
0 0 0 0 0
```

16)
```
8
7
7 7
6 6
6 6 6
5 5 5
5 5 5 5
4 4 4 4
```

17)
```
1
6
2 2
7 7
3 3 3
8 8 8
4 4 4 4
9 9 9 9
```

18)
```
9
4
8 8
3 3
7 7 7
2 2 2 2
6 6 6 6 6
1 1 1 1 1
```

19)
```
3
0
2 3
9 0
2 2 3
8 9 0
2 2 2 3
7 8 9 0
2 2 2 2 3
6 7 8 9 0
```

20)
```
2 2 2 2 2
3 4 5 6 7
2 2 2 2
2 3 4 5
2 2 2
1 2 3
2 2
0 1
2
-1
```

V. Постройте таблицу значений функции y=f(x) для $x \in [a, b]$ с шагом h. Если в некоторой точке x функция не определена, то выведите на экран сообщение об этом.

Замечание. Для решения задачи использовать вспомогательную функцию.

1. $y = \dfrac{1}{(1+x)^2}$;
2. $y = \dfrac{1}{x^2-1}$;
3. $y = \sqrt{x^2-1}$;
4. $y = \sqrt{5-x^3}$;
5. $y = \ln(x-1)$;
6. $y = \ln(4-x^2)$;
7. $y = \dfrac{x}{\sqrt{2x-1}}$;
8. $y = \dfrac{3x+4}{\sqrt{x^2+2x+1}}$;
9. $y = \dfrac{1}{x-1} + \dfrac{2}{1-4x}$;
10. $y = \ln|x-2|$;
11. $y = \ln\dfrac{x}{x-2}$;
12. $y = \ln(x^4-1)\ln(1+x)$;
13. $y = \dfrac{\ln(x-2)}{\sqrt{5x+1}}$;
14. $y = \dfrac{\sqrt{x^2-2x+1}}{\ln(4-2x)}$;
15. $y = \ln|3x|\sqrt{2x^5-1}$;
16. $y = \dfrac{3}{|x^3+8|}$;
17. $y = \dfrac{x+4}{x^2-2} + \sqrt{x^3-1}$;
18. $y = \sqrt{x^2+1} - \sqrt{x^2+5}$;
19. $y = \dfrac{\sqrt{x^3-1}}{\sqrt{x^2-1}}$;
20. $y = \dfrac{1}{x+7} + \ln(1-|x|)$.

VI. Постройте таблицу значений функции y=f(x) для $x \in [a, b]$ с шагом h.

Замечание. Для решения задачи использовать вспомогательную функцию.

1. $y = \begin{cases} \dfrac{1}{(0.1+x)^2}, & \text{если } x \ge 0.9; \\ 0.2x + 0.1, & \text{если } 0 \le x < 0.9; \\ x^2 + 0.2, & \text{если } x < 0. \end{cases}$

2. $y = \begin{cases} \sin(x), & \text{если } |x| < 3; \\ \dfrac{\sqrt{x^2+1}}{\sqrt{x^2+5}}, & \text{если } 3 \le |x| < 9; \\ \sqrt{x^2+1} - \sqrt{x^2+5}, & \text{если } |x| \ge 9. \end{cases}$

3. $y = \begin{cases} 0, & \text{если } x < a; \\ \dfrac{x-a}{x+a}, & \text{если } x > a; \\ 1, & \text{если } x = a. \end{cases}$

4. $y = \begin{cases} x^3 - 0.1, & \text{если } |x| \le 0.1; \\ 0.2x - 0.1, & \text{если } 0.1 < |x| \le 0.2; \\ x^3 + 0.1, & \text{если } |x| > 0.2. \end{cases}$

5. $y=\begin{cases} a+b, \text{ если } x^2-5x<0; \\ a-b, \text{ если } 0\le (x^2-5x)<10; \\ ab, \text{ если } x^2-5x\ge 10. \end{cases}$

6. $y=\begin{cases} x^2, \text{ если } (x^2+2x+1)<2; \\ \dfrac{1}{x^2-1}, \text{ если } 2\le (x^2+2x+1)<3; \\ 0, \text{ если } (x^2+2x+1)\ge 3. \end{cases}$

7. $y=\begin{cases} -4, \text{если } x<0; \\ x^2+3x+4, \text{если } 0\le x<1; \\ 2, \text{ если } x\ge 1. \end{cases}$

8. $y=\begin{cases} x^2-1, \text{ если } |x|\le 1; \\ 2x-1, \text{ если } 1<|x|\le 2; \\ x^5-1, \text{ если } |x|>2. \end{cases}$

9. $y=\begin{cases} (x^2-1)^2, \text{ если } x<1; \\ \dfrac{1}{(1+x)^2}, \text{ если } x>1; \\ 0, \text{ если } x=1. \end{cases}$

10. $y=\begin{cases} x^2, \text{ если } (x+2)\le 1; \\ \dfrac{1}{x+2}, \text{если } 1<(x+2)<10; \\ x+2, \text{ если } (x+2)\ge 10; \end{cases}$

11. $y=\begin{cases} x^2+5, \text{ если } x\le 5; \\ 0, \text{если } 5<x<20; \\ 1, \text{ если } x\ge 20. \end{cases}$

12. $y=\begin{cases} 0, \text{ если } x<0; \\ x^2+1, \text{если } x\ge 0 \text{ и } x\ne 1; \\ 1, \text{если } x=1. \end{cases}$

13. $y=\begin{cases} 1, \text{если } x=1 \text{ или } x=-1; \\ \dfrac{-1}{1-x}, \text{если } x\ge 0 \text{ и } x\ne 1; \\ \dfrac{1}{1+x}, \text{если } x<0 \text{ и } x\ne -1. \end{cases}$

14. $y=\begin{cases} 0.2x^2-x-0.1, \text{ если } x<0; \\ \dfrac{x^2}{x-0.1}, \text{если } x>0 \text{ и } x\ne 0.1; \\ 0, \text{ если } x=0.1. \end{cases}$

15. $y=\begin{cases} 1, \text{ если } (x\text{-}1)<1; \\ 0, \text{ если } (x\text{-}1)=1; \\ -1, \text{если } (x\text{-}1)>1. \end{cases}$

16. $y=\begin{cases} x, \text{ если } x>0; \\ 0, \text{ если } -1\le x\le 0; \\ x^2, \text{ если } x<-1. \end{cases}$

17. $y=\begin{cases} a+bx, \text{ если } x<93; \\ \text{b - ac, если } 93\le x\le 120; \\ \text{abx, если } x>120. \end{cases}$

18. $y=\begin{cases} x^2-0.3, \text{ если } y<3; \\ 0, \text{ если } 3\le x\le 5; \\ x^2+1, \text{ если } x>5. \end{cases}$

19. $y=\begin{cases} \sqrt{5x^2+5}, \text{если} |x|<2; \\ \dfrac{|x|}{\sqrt{5x^2+5}}, \text{ если } 2\le |x|<10; \\ 0, \text{ если } |x|\ge 10. \end{cases}$

20. $y=\begin{cases} \sin(x), \text{ если } |x|<\dfrac{\pi}{2}; \\ \cos(x), \text{ если } \dfrac{\pi}{2}\le |x|\le \pi; \\ 0, \text{ если } |x|>\pi. \end{cases}$

4. РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ

4.1. Вычисление членов рекуррентной последовательности

Пусть $a_1, a_2, \ldots, a_n$ - произвольная числовая последовательность. Рекуррентным соотношением называется такое соотношение между членами последовательности, в котором каждый следующий член выражается через несколько предыдущих, т.е $a_k = f(a_{k-1}, a_{k-2}, \ldots, a_{k-l}),\ k > l$ (1).

Последовательность задана рекуррентно, если для нее определено рекуррентное соотношение вида (1) и заданы первые *l* ее членов.

Самым простым примером рекуррентной последовательности является арифметическая прогрессия. Рекуррентное соотношение для нее записывается в виде: $a_k = a_{k-1} + d$, где d – разность прогрессии. Зная первый элемент и разность прогрессии, и, используя данное рекуррентное соотношение, можно последовательно вычислить все остальные члены прогрессии.

Рассмотрим пример программы, в которой вычисляются первые n членов арифметической прогрессии при условии, что $a_1 = \frac{1}{2}$ и $d = \frac{1}{4}$.

```
#include <iostream>
using namespace std;
int main()
{ float a=0.5, d=0.25;  //задали первый член последовательности и разность прогрессии
 int n; cout <<"n="; cin>>n;  //ввели количество членов последовательности
 cout<<"a1: "<<a<<endl;  //вывели первый член последовательности
 for (int i=2; i<=n; i++)  //организуем вычисление 2, 3, ... ,n члена последовательности
  {a=a+d;                  //для этого прибавляем к предыдущему члену значение d
   cout<<"a"<<i<<"="<<a<<endl; }  //и выводим новое значение a на экран
 return 0;}
```

Результат работы программы:

n	*состояние экрана*
5	a1: 0.5
	a2: 0.75
	a3: 1.
	a4: 1.25
	a5: 1.5

Другим примером рекуррентной последовательности является геометрическая прогрессия. Рекуррентное соотношение для нее записывается в виде: $b_k = b_{k-1} * q$, где q – знаменатель прогрессии. Рассмотрим пример программы, в которой вычисляются первые n членов арифметической прогрессии при условии, что $b_1=1$, $q=2$.

```
#include <iostream>
using namespace std;
int main()
{ int b=1, q=2,n;  //задали первый член последовательности и знаменатель прогрессии
 cout <<"n="; cin>>n;  //ввели количество членов последовательности
 cout<<"b1="<<b<<endl;  //вывели первый член последовательности
 for (int i=2; i<=n; i++)  //организуем вычисление 2, 3, ... ,n члена последовательности
  {b=b*q;  //для этого умножаем предыдущее значение b на q
   cout<<"b"<<i<<"="<<b<<endl; }  //и выводим новое значение b на экран
 return 0;}
```

Результат работы программы:

n	*состояние экрана*
5	b1: 1
	b2: 2
	b3: 4
	b4: 8
	b5: 16

В арифметической и в геометрической прогрессиях каждый член последовательности зависит только от одного предыдущего значения. Более сложная зависимость представлена в последовательности Фибоначчи: $a_1 = a_2 = 1$, $a_n = a_{n-1} + a_{n-2}$. В этом случае каждый член последовательности зависит от значений двух предыдущих членов. Рассмотрим пример программы, в которой вычисляются первые n членов последовательности Фибоначчи.

```
#include <iostream>
using namespace std;
int main()
{ int a1=1, a2=1,a,n;   //задали первый и второй члены последовательности Фибоначчи
 cout <<"n="; cin>>n; //ввели количество членов последовательности
 cout<<"a1="<<a1<<endl <<"a2="<<a2<<endl;    //вывели известные члены
  /*Организуем цикл для вычисления членов последовательности с номерами 3, 4,..., n.
  При этом в переменной a1 будет храниться значение члена последовательности с
  номером i-2, в переменной a2 -  члена с номером i-1, переменная a будет использо-
  ваться для вычисления члена с номером i. */
 for (int i=3; i<=n; i++)
 { a=a1+a2;   //по рекуррентному соотношению вычисляем член последовательности
  cout<<"a"<<i<<"="<<a<<endl;       //с номером i и выводим его значение на экран
  //выполняем рекуррентный пересчет для следующего шага цикла
  a1=a2;  //в элемент с номером i-2 записываем значение элемента с номером i-1
  a2=a;}   //в элемент с номером i-1 записываем значение элемента с номером i
 return 0;}
```

Результат работы программы:

n	*состояние экрана*
5	a1: 1
	a2: 1
	a3: 2
	a4: 3
	a5: 5

В рассмотренных случаях мы выводили на экран значения первых n элементов рекуррентной последовательности. Иногда нам бывает необходимо найти только значение n-ного элемента последовательности. Для этого в программе нужно исключить вывод значения на каждом шаге цикла. В качестве примера рассмотрим программу, в которой вычисляется n-ный элемент последовательности, заданной следующим образом: $b_1 = 1$, $b_2 = 2$, $b_n = \dfrac{b_{n-1} - b_{n-2}}{(n-1)^2}$.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ float b1=1, b2=2, b;       //задали первый и второй элементы последовательности
 int n;  cout <<"n="; cin>>n;    // ввели количество элементов последовательности
```

```
/*Организуем цикл для вычисления элементов с номерами 3, 4,..., n. При этом в переменной b1 будет храниться значение элемента последовательности с номером i-2, в переменной b2 - элемента с номером i-1, переменная b будет использоваться для вычисления элемента с номером i. */
for (int i=3; i<=n; i++)
 { //по рекуррентному соотношению вычисляем i-ый элемент последовательности
   b=(b1-b2)/pow(i-1.0,2);
   b1=b2; b2=b;}  // выполняем рекуррентный пересчет для следующего шага цикла
 cout<<"b"<<n<<"="<<b<<endl; //выводим значение b на экран
return 0;}
```

Результат работы программы:

n	*состояние экрана*
5	5 элемент: -0.03125

4.2. Упражнения

Написать программу, вычисляющую первые n элементов заданной последовательности:

1. $b_1 = 9, b_n = 0.1b_{n-1} + 10$;
2. $b_1 = -1, b_n = 9 - 2b_{n-1}$;
3. $b_1 = 1, b_n = 0.2b_{n-1}^4 + 1$;
4. $b_1 = 4.7, b_n = \sin(b_{n-1}) + \pi$;;
5. $b_1 = 0.1, b_n = \frac{1}{6}(0.05 + b_{n-1}^3)$;
6. $b_1 = 2, b_n = 0.5(\frac{1}{b_{n-1}} + b_{n-1})$
7. $b_1 = 5, \quad b_n = (-1)^n b_{n-1} - 8$;
8. $b_1 = -1, \quad b_2 = 1, \quad b_n = 3b_{n-1} - 2b_{n-2}$;
9. $b_1 = -10, \quad b_2 = 2, \quad b_n = |b_{n-2}| - 6b_{n-1}$;
10. $b_1 = 2, \quad b_2 = 4, \quad b_n = 6b_{n-1} - b_{n-2}$;
11. $b_1 = 5, \quad b_n = \frac{b_{n-1}}{n^2 + n + 1}$;
12. $b_1 = 0.5, \quad b_2 = 0.2, \quad b_{n+1} = b_n^2 + \frac{b_{n-1}}{n}$;
13. $b_1 = 1, \quad b_n = \frac{1}{4}\left(3b_{n-1} + \frac{1}{3b_{n-1}}\right)$;
14. $b_1 = 2, \quad b_2 = 1, \quad b_n = \frac{2}{3}b_{n-2} - \frac{1}{3}b_{n-1}^2$;
15. $b_1 = 1, \quad b_2 = 2, \quad b_n = \frac{b_{n-2}}{4} + \frac{5}{b_{n-1}^2}$;
16. $b_1 = 1, \quad b_2 = 2, \quad b_n = \frac{nb_{n-2} - b_{n-1}}{n+1}$;
17. $b_1 = 4, \quad b_2 = 2, \quad b_n = \frac{b_{n-2}}{n} + \frac{n^2}{b_{n-1}}$
18. $b_1 = 100, \quad b_{2n} = b_{2n-1}/10, \quad b_{2n+1} = b_{2n} + 10$;
19. $b_1 = 0, \quad b_{2n} = b_{2n-1} + 3, \quad b_{2n+1} = 2b_{2n}$;
20. $b_1 = 1, \quad b_2 = 5 \quad b_{2n} = b_{2n-1} + b_{2n-2}, \quad b_{2n+1} = b_{2n} - b_{2n-1}$.

5. ВЫЧИСЛЕНИЕ КОНЕЧНЫХ И БЕСКОНЕЧНЫХ СУММ И ПРОИЗВЕДЕНИЙ

5.1. Вычисление конечных сумм и произведений

Решение многих задач связано с нахождением суммы или произведения элементов заданной последовательности. В данном разделе мы рассмотрим основные приемы вычисления конечных сумм и произведений.

Пусть $u_1(x), u_2(x), \ldots, u_n(x)$ - произвольная последовательность n функций. Будем рассматривать конечную сумму вида $u_1(x)+u_2(x)+\ldots+u_n(x)$. Такую сумму можно записать более компактно, используя следующее обозначение: $u_1(x)+u_2(x)+\ldots+u_n(x)=\sum_{i=1}^{n} u_i(x)$. При $n \le 0$ значение суммы равно 0.

В дальнейшем будем также использовать сокращенную запись для конечного произведения данной последовательности, которая выглядит следующим образом: $u_1(x)\cdot u_2(x)\cdot\ldots\cdot u_n(x)=\prod_{i=1}^{n} u_i(x)$.

1. Написать программу, которая подсчитывает сумму натуральных чисел от 1 до n ($n \ge 1$).

Указания по решению задачи. Пусть s_n - сумма натуральных чисел от 1 до n. Тогда $s_n=1+2+\ldots+(n-1)+n=(1+2+\ldots+(n-1))+n=s_{n-1}+n$, $s_0=0$. Мы пришли к рекуррентному соотношению $s_0=0$, $s_n=s_{n-1}+n$, которым мы можем воспользоваться для подсчета суммы. Соотношение $s_n=s_{n-1}+n$ говорит о том, что сумма на n-ном шаге равна сумме, полученной на предыдущем шаге, плюс очередное слагаемое.

```
#include <iostream>
using namespace std;
int main()
{ int n, s=0;
  cout <<"n="; cin>>n;
  for (int i=1; i<=n; i++) //выполняем n шагов и на каждом i-том шаге
    s+=i; //используем полученное рекуррентное соотношение
  cout<<"s="<<s<<endl;
  return 0; }
```

Рассмотрим пошаговое выполнение программы:

№ шага	*Значение счетчика*	*Результат (значение s)*
1	i=1	0+1=1
2	i=2	1+2=3
3	i=3	1+2+3=6
…	…	
n	i=n	1+2+3+4+ … +n

2. Написать программу, которая считает x^n для вещественного x и натурального n.

Указание по решению задачи. Из свойства степенной функции ($x^0=1$, $x^n=x^{n-1}*x$) следует, что ее можно вычислять, используя рекуррентное соотношение $b_0=1$, $b_n=b_{n-1}*x$.

```
#include <iostream>
using namespace std;
int main()
{ int n;
  float x, b=1;
```

```
cout <<"x="; cin>>x;
cout <<"n="; cin>>n;
for (int i=1; i<=n; i++) b*=x;
cout<<"x^n="<<b<<endl;
return 0;}
```

Рассмотрим пошаговое выполнение программы:

№ шага	*Значение счетчика*	*Результат (значение b)*
1	i=1	$1 \cdot x$
2	i=2	$x \cdot x = x^2$
3	i=3	$x^2 \cdot x = x^3$
…	…	…
n	i=n	$x^{n-1} \cdot x = x^n$

3. Написать программу для подсчета суммы: $-\sin x+\sin 2x-\sin 3x+\ldots+(-1)^n \sin nx$ ($n \geq 1$).

Указания по решению задачи. Если пронумеровать слагаемые, начиная с номера 1, то мы увидим закономерность: знак минус ставится перед слагаемыми с нечетными номерами, а знак плюс – с четными, при этом сумму можно выразить рекуррентным соотношением: $s_0=0$, $s_n=s_{n-1}+(-1)^n \sin(n*x)$. Введем вспомогательную переменную *singl*, которая будет использоваться для хранения знака очередного слагаемого. Т.к. первое слагаемое отрицательно, то начальное значение переменной *singl* равно -1, а затем после каждой итерации цикла значение переменной *singl* будем заменять на противоположное с помощью унарного минуса. В этом случае рекуррентное соотношение можно преобразовать к виду $s_0=0$, $singl_0 = -1$, $s_n=s_{n-1}+singl*\sin(n*x)$, $singl_n=-singl_{n-1}$.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ int n, singl=-1;
 float x, s=0;
 cout <<"x="; cin>>x;
 cout <<"n="; cin>>n;
 for (int i=1; i<=n; i++)
  {s+=singl*sin(i*x); singl=-singl;}          //1
 cout<<"s="<<s<<endl;
 return 0;}
```

Рассмотрим пошаговое выполнение программы:

№ шага	*Значение счетчика*	*Результат (значение s)*
1	i=1	0-sinx=sinx
2	i=2	-sinx+sin2x
3	i=3	-sinx+ sin2x-sin3x
…	…	
n	i=n	-sinx+ sin2x-sin3x+…+(-1)sinnx

Замечание. Подумайте, что произойдет, если поменять местами операторы присваивания в *строке 1.*

4. Написать программу для подсчета суммы $S_n = \frac{\cos x}{1} + \frac{\cos x + \cos 2x}{2} + \frac{\cos x + cox2x + \cos 3x}{3} + \ldots + \frac{\cos x + \ldots + \cos nx}{n}$, где x – вещественное число, n – натуральное число.

Указания по решению задачи. Если пронумеровать слагаемые, начиная с 1, то мы увидим, что номер слагаемого совпадает со значением знаменателя. Рассмотрим каждый числитель отдельно: $b_1 = \cos x$, $b_2 = \cos x + \cos 2x$, $b_3 = \cos x + \cos 2x + \cos 3x \ldots$ Эту последовательность можно представить рекуррентным соотношением $b_0=0$, $b_n=b_{n-1}+\cos nx$ *(1).* Теперь сумму можно предста-

вить следующим образом $S_n = \frac{b_1}{1} + \frac{b_2}{2} + \frac{b_3}{3} + \ldots + \frac{b_n}{n}$, а для нее справедливо рекуррентное соотношение $S_0=0$, $S_n = S_{n-1} + \frac{b_n}{n}$ *(2)*. При составлении программы будем использовать формулы (1-2).

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ int n;
  float x, s=0, b=0;
  cout <<"x="; cin>>x;
  cout <<"n="; cin>>n;
  for (int i=1; i<=n; i++) {b+=cos(i*x); s+=b/i;}
  cout<<"s="<<s<<endl;
  return 0;}
```

Рассмотрим пошаговое выполнение программы:

№ шага	*Значение счетчика*	*Значение b*	*Значение s*
1	i=1	$\cos x$	$\frac{\cos x}{1}$
2	i=2	$\cos x + \cos 2x$	$\frac{\cos x}{1} + \frac{\cos x + \cos 2x}{2}$
3	i=3	$\cos x + \cos 2x + \cos 3x$	$\frac{\cos x}{1} + \frac{\cos x + \cos 2x}{2} + \frac{\cos x + cox2x + \cos 3x}{3}$
…	…		
n	i=n	$\cos x + \cos 2x + \cos 3x$	$\frac{\cos x}{1} + \frac{\cos x + \cos 2x}{2} + \ldots + \frac{\cos x + \ldots + \cos nx}{n}$

5. Написать программу для подсчета суммы $S_n = \sum_{i=1}^{n} \frac{(-1)^{i+1} x^i}{i!}$, где x – вещественное число, n – натуральное число.

Указания по решению задачи. Перейдем от сокращенной формы записи к развернутой, получим $S_n = \frac{x}{1!} - \frac{x^2}{2!} + \frac{x^3}{3!} - \ldots + \frac{(-1)^{n+1} x^n}{n!}$. Каждое слагаемое формируется по формуле $a_n = \frac{(-1)^{n+1} x^n}{n!}$. Если в эту формулу подставить n=0, то получим $a_0 = \frac{(-1)^1 x^0}{0!} = -1$.

Запись n! читается как «n факториал». По определению факториала: 0!=1!=1, n!=1*2*3*…*n, n!=(n-1)!n. Таким образом, факториал выражается рекуррентным соотношением.

Чтобы не вводить несколько рекуррентных соотношений (отдельно для числителя, отдельно для знаменателя), выразим последовательность слагаемых рекуррентным соотношением вида $a_n = a_{n-1} q$, где q для нас пока не известно. Найти его можно из выражения $q = \frac{a_n}{a_{n-1}}$. Произведя расчеты мы получим, что $q = -\frac{x}{i}$. Следовательно, для последовательности слагаемых мы получили рекуррентное соотношение $a_0 = -1$, $a_i = -a_{i-1} \cdot \frac{x}{i}$ *(3)*. А всю сумму, по аналогии с предыдущими примерами, можно представить рекуррентным соотношением: $S_0=0$, $S_n = S_{n-1} + a_n$ *(4)*. Таким образом, при составлении программы будем пользоваться формулами (3-4).

```
#include <iostream>
using namespace std;
int main()
{ int n;
  float x, s=0, a=-1;
  cout <<"x="; cin>>x;
  cout <<"n="; cin>>n;
  for (int i=1; i<=n; i++)  {a*=-x/i; s+=a;}
  cout<<"s="<<s<<endl;
  return 0;}
```

Рассмотрим пошаговое выполнение программы:

№ шага	*Значение счетчика*	*Значение a*	*Значение s*
1	i=1	$-(-1)\frac{x}{1}=\frac{x^1}{1!}$	$0+\frac{x^1}{1!}=\frac{x^1}{1!}$
2	i=2	$-\frac{x}{1}\cdot\frac{x}{2}=-\frac{x^2}{2!}$	$\frac{x^1}{1!}-\frac{x^2}{2!}$
3	i=3	$-(-\frac{x^2}{2!})\cdot\frac{x}{3}=\frac{x^3}{3!}$	$\frac{x^1}{1!}-\frac{x^2}{2!}+\frac{x^3}{3!}$
…	…	…	…
n	i=n	$-\frac{(-1)^n x^{n-1}}{(n-1)!}\cdot\frac{x^n}{n}=\frac{(-1)^{n+1}x^n}{n!}$	$\frac{x^1}{1!}-\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots+\frac{(-1)^{n+1}x^n}{n!}$

6. Написать программу для подсчета произведения $P_k=\prod_{n=1}^{k}(1+\frac{x^{2n}+x^n}{n})$, где x – вещественное число, n – натуральное число.

Указания по решению задачи. Преобразуем заданное выражение к виду $P_k=\prod_{n=1}^{k}(1+\frac{x^n(x^n+1)}{n})$ и перейдем от сокращенной формы записи к развернутой $P_k=(1+\frac{x^1(x^1+1)}{1})(1+\frac{x^2(x^2+1)}{2})\ldots(1+\frac{x^k(x^k+1)}{k})$. В числителе каждой дроби встречается x^n (см.пример 2), его можно вычислить по рекуррентному соотношению $b_0=1$. $b_n=b_{n-1}*x$ *(5)*. Тогда произведение можно представить как $P_k=(1+\frac{b_1(b_1+1)}{1})(1+\frac{b_2(b_2+1)}{2})\ldots(1+\frac{b_k(b_k+1)}{k})$, что в свою очередь можно выразить рекуррентным соотношением $P_0=1$. $P_k=P_{k-1}*(1+\frac{b_k(b_k+1)}{k})$ *(6)*. При составлении программы будем пользоваться формулами (5-6).

```
#include <iostream>
using namespace std;
int main()
{ int n;
  float x, p=1, b=1;
  cout <<"x="; cin>>x;
  cout <<"n="; cin>>n;
  for (int i=1; i<=n; i++)  {b*=x; p*=(1+b*(b+1)/i);}
  cout<<"p="<<p<<endl;
  return 0;}
```

Рассмотрим пошаговое выполнение программы:

№ шага	*Значение счетчика*	*Значение b*	*Значение p*
1	i=1	x	$1*(1+\frac{x(x+1)}{1})$
2	i=2	x^2	$1*(1+\frac{x(x+1)}{1})(1+\frac{x^2(x^2+1)}{2})$
3	i=3	x^3	$1*(1+\frac{x(x+1)}{1})(1+\frac{x^2(x^2+1)}{2})(1+\frac{x^3(x^3+1)}{3})$
…	…	…	…
N	i=k	x^k	$1*(1+\frac{x(x+1)}{1})(1+\frac{x^2(x^2+1)}{2})...(1+\frac{x^k(x^k+1)}{k})$

5.2. Вычисление бесконечных сумм

Будем теперь рассматривать бесконечную сумму вида $u_1(x)+u_2(x)+\ldots+u_n(x)+\ldots=\sum_{i=1}^{\infty}u_i(x)$. Это выражение называется функциональным рядом. При различных значениях x из функционального ряда получаются различные числовые ряды $a_1+a_2+\ldots+a_n+\ldots=\sum_{i=1}^{\infty}a_i$. Числовой ряд может быть сходящимся или расходящимся. Совокупность значений x, при которой функциональный ряд сходится, называется его областью сходимости.

Числовой ряд называется сходящимся, если сумма n первых его членов $S_n=a_1+a_2+\ldots+a_n$ при $n\to\infty$ имеет предел, в противном случае, ряд называется расходящимся. Ряд может сходиться лишь при условии, что общий член ряда a_n при неограниченном увеличении его номера стремится к нулю: $\lim_{n\to\infty}a_n=0$. Это необходимый признак сходимости для всякого ряда.

В случае бесконечной суммы будем вычислять ее с заданной точностью *e*. Считается, что требуемая точность достигается, если вычислена сумма нескольких первых слагаемых и очередное слагаемое оказалось по модулю меньше чем *e*.

1. Написать программу для подсчета суммы $\sum_{i=1}^{\infty}\frac{(-1)^i}{i!}$ с заданной точностью e (e>0).

Указание по решению задачи. Рассмотрим, что представляет из себя заданный ряд: $\sum_{i=1}^{\infty}\frac{(-1)^i}{i!}=-\frac{1}{1}+\frac{1}{2}-\frac{1}{6}+\frac{1}{24}-\frac{1}{120}+\ldots+\frac{1}{\infty}$. Общий член ряда с увеличением значения i стремится к нулю, следовательно, данную сумму будем вычислять с определенной точностью *e*. Заметим также, что последовательность слагаемых можно выразить с помощью рекуррентного соотношения $a_1=-1$, $a_i=\frac{-a_{i-1}}{i}$, а всю сумму - с помощью рекуррентного соотношения $S_0=0$, $S_n=S_{n-1}+a_n$. (Данные рекуррентные соотношения выведите самостоятельно.)

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

```
int main()
{ int n,i=1;
  float e, s=0, a=-1;
  cout <<"e="; cin>>e;
  while (fabs(a)>=e)  // до тех пор, пока очередное слагаемое больше e
   {s+=a;             // добавляем его к сумме
    i++; a/=-i;}      // вычисляем номер очередного слагаемого и его значение
  cout<<"s="<<s<<endl;
  return 0;}
```

Рассмотрим подробно, как выполняется цикл при e = 0.01:

№ шага	*Значения переменных до выполнения цикла*			*Условие abs(a)>=e*	*Значения переменных после выполнения цикла*		
	i	*a*	*s*		*i*	*a*	*s*
1	1	-1	0	TRUE	2	0.5	-1
2	2	0.5	-1	TRUE	3	-0.166...	-0.5
3	3	-0.166...	-0.5	TRUE	4	0.046...	-0.666...
4	4	0.046...	-0.666...	TRUE	5	0.0083...	-0.625
5	5	0.0083...	-0.625	FALSE	5	0.0083...	-0.625

После выполнения программы на экране будет выведено следующие сообщение «Сумма с заданной точностью равна - 0.625».

2. Вычислить значение функции $F(x)= -\frac{1}{(x+1)}+\frac{(x-1)^2}{2(x+1)^2}-\frac{(x-1)^4}{4(x+1)^3}+\frac{(x-1)^6}{8(x+1)^4}-\ldots$ на отрезке *[a,b]* с шагом *h=0.1* и точностью ε. Результат работы программы представить в виде таблицы, которая содержит номер аргумента, значение аргумента, значение функции и количество просуммированных слагаемых.

Указания по решению задачи. Разработаем вспомогательную функцию Fun(x,e,n), которая по заданным значениям *x* и *e* вычисляет значение функции и количество слагаемых *n*, при суммировании которых была достигнута заданная степень точности. Для этого перейдем от развернутой формы записи функции к сокращенной, получим $F(x)=\sum_{i=1}^{\infty}\frac{(-1)^i(x-1)^{2i-2}}{2^{i-1}(x+1)^i}$ и воспользуемся приемом, рассмотренным в примере 5 раздела 5.1. Получим, что слагаемые данной функции определяются с помощью рекуррентного соотношения $a_1=-\frac{1}{(x+1)}$, $a_i=\frac{-a_{i-1}(x-1)^2}{2(x+1)}$.

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;
float fun(float x, float e, int &n)    //вспомогательная функция
{ float s=0, a=-1/(x+1);      // определяем начальное значение суммы и первое слагаемое
  n=0;                        //определяем количество просуммированных слагаемых
  while (fabs(a)>=e)          //пока не достигнута заданная степень точности
   { s+=a;                    //добавляем слагаемое к сумме
     a*=-pow(x-1,2)/(2*x+2);      //формируем очередное слагаемое
     n++;}                    //увеличиваем количество просуммированных слагаемых
  return s;}                  //возвращаем в качестве значения функции значение s
int main() //главная функция
{ float a, b, e, h, f, x;
  cout <<"a="; cin>>a; //вводим необходимые значения
```

```
cout <<"b="; cin>>b;
cout <<"h="; cin>>h;
cout <<"e="; cin>>e;
int n, i;
cout<<setprecision(3); //устанавливаем количество цифр после запятой для вещ.чисел
cout <<"i\t x\t f(x) \t n\n"; //ввыводим заголовок таблицы
for (x=a, i=1; x<=b; x+=h, i++) // строим таблицу на отрезке [a, b]
 { f=fun(x,e,n); //вызываем вспомогательную функцию
   cout <<i <<"\t" <<x <<"\t" <<f <<"\t" <<n <<endl; } //выводим полученные данные на экран
 return 0;}
```

Результат работы программы для отрезка [1,2], h=0.3, e=0.00001:

i	x	f(x)	n
1	1	-0.5	1
2	1.3	-0.426	2
3	1.6	-0.36	4
4	1.9	-0.303	6

5.3. Упражнения

I. Для заданного натурального n и действительного x подсчитать следующие суммы:

1. $S = 1^2 + 2^2 + 3^2 + \ldots + n^2$;
2. $S = \sqrt{1} + \sqrt{2} + \sqrt{3} + \ldots + \sqrt{n}$;
3. $S = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$;
4. $S = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{n^2}$;
5. $S = 1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}} + \ldots + \frac{1}{\sqrt{n}}$;
6. $S = \frac{1}{\sin 1} + \frac{1}{\sin 2} + \ldots + \frac{1}{\sin n}$;
7. $S = 1+2+2^2+2^3+\ldots+2^n$;
8. $S = \cos 1 - \cos 2 + \cos 3 - \ldots + (-1)^{n+1} \cos n$;
9. $S = 1!+2!+3!+\ldots+n!$;
10. $S = 1 - 3 + 3^2 - 3^3 + \ldots (-1)^n 3^n$;
11. $S = 1!-2!+3!-\ldots+(-1)^{n+1} n!$;
12. $S = \sin x + \sin x^2 + \sin x^3 + \ldots + \sin x^n$;
13. $S = 1 + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$;
14. $S = -\frac{1}{2} + \frac{1}{2^2} - \frac{1}{2^3} + \ldots + \frac{(-1)^n}{2^n}$;
15. $S = 1^3 - 2^3 + 3^3 \ldots + (-1)^{n+1} n^3$;
16. $S = x+3x^3+5x^5+7x^7+\ldots+(2n-1)x^{2n-1}$;
17. $S = \frac{\cos x}{1} + \frac{\cos^2 x}{2} + \frac{\cos^3 x}{3} + \ldots + \frac{\cos^n x}{n}$;
18. $S = \frac{1}{3^2} - \frac{1}{5^2} + \frac{1}{7^2} - \ldots + \frac{(-1)^{n+1}}{(2n+1)^2}$
19. $S = \frac{1}{\sin 1} + \frac{1}{\sin 1 + \sin 2} + \ldots + \frac{1}{\sin 1 + \sin 2 + \ldots + \sin n}$;
20. $S = \sin x + \sin\sin x + \sin\sin\sin x + \ldots + \underbrace{\sin\sin\sin\ldots\sin x}_{n\ раз}$;

II. Для заданного натурального k и действительного x подсчитать следующие выражения:

1. $S = \sum_{n=1}^{k} \frac{x^n}{n}$;
2. $S = \sum_{n=1}^{k} \frac{2^n \cdot n!}{n^2}$
3. $S = \sum_{n=1}^{k} \frac{(-1)^{n+1}}{n^2}$;
4. $S = \sum_{n=1}^{k} \frac{x^{2(n-1)}}{(2+4(n-1))^2}$;

5. $S=\sum_{n=1}^{k}\frac{(-1)^{n+1}x^{2n-1}}{(2n-1)!}$;

6. $S=\sum_{n=1}^{k}\frac{1}{n\cdot n!}$;

7. $S=\sum_{n=0}^{k}\frac{(-1)^{n+1}x^{2n+1}}{(2n+1)!}$;

8. $S=\sum_{n=1}^{k}\frac{(-1)^{n-1}x^{n}}{(2n)!}$;

9. $S=\sum_{n=1}^{k}\frac{(-1)^{n-1}x^{2n}}{(2n)!}$;

10. $S=\sum_{n=1}^{k}\frac{(-1)^{n}x^{n}}{2^{n}7n}$;

11. $P=\prod_{n=1}^{k}(1+\frac{x^{n}}{n^{2}})$;

12. $P=\prod_{n=1}^{k}(1+\frac{x^{2n-1}}{n(n+1)})$;

13. $P=\prod_{n=1}^{k}(1-\frac{x^{n}}{n!})$

14. $P=\prod_{n=1}^{k}(1+\frac{(-1)^{n}x^{2n}}{n^{3}})$;

15. $P=\prod_{n=1}^{k}(1+\frac{(-1)^{n+1}x^{n}}{n!})$:

16. $P=\prod_{n=1}^{k}(1+\frac{x^{2n}}{n(n+4)})$;

17. $P=\prod_{n=1}^{k}(1+\frac{(-1)^{n}x^{2n+1}}{n^{3}+n^{2}})$;

18. $P=\prod_{n=1}^{k}(1+\frac{x^{n}}{2n!})$;

19. $P=\prod_{n=2}^{k}(1+\frac{(-1)^{n}x^{2n-1}}{n^{3}-1})$;

20. $P=\prod_{n=0}^{k}(1+\frac{(-1)^{n-1}x^{2n}}{(n+2)(n+1)})$;

III. Вычислить бесконечную сумму ряда с заданной точностью e (e>0).

1. $\sum_{i=1}^{\infty}\frac{1}{i^{2}}$

2. $\sum_{i=1}^{\infty}\frac{1}{(i+1)^{3}}$

3. $\sum_{i=2}^{\infty}\frac{(-1)^{i}}{i^{2}-1}$

4. $\sum_{i=1}^{\infty}\frac{1}{i(i+1)}$

5. $\sum_{i=2}^{\infty}\frac{5}{(i+1)(i-1)}$

6. $\sum_{i=1}^{\infty}\frac{(-2)^{i+1}}{i(2i+1)}$

7. $\sum_{i=1}^{\infty}\frac{2}{i!}$

8. $\sum_{i=1}^{\infty}\frac{1}{(2i)!}$

9. $\sum\frac{(-1)^{i}}{(2i-1)!}$

10. $\sum_{i=1}^{\infty}\frac{(-1)^{i+1}}{2i!}$

11. $\sum_{i=1}^{\infty}\frac{(-1)^{2i}}{i(i+1)(i+2)}$

12. $\sum_{i=3}^{\infty}\frac{(-1)^{2i-1}}{i(i-1)(i-2)}$

13. $\sum_{i=1}^{\infty}\frac{(-3)^{2i}}{3i!}$

14. $\sum_{i=1}^{\infty}\frac{(-5)^{2i-1}}{5(2i-1)!}$

15. $\sum_{i=1}^{\infty}\frac{1}{2^{i}}$

16. $\sum_{i=1}^{\infty}\frac{1}{3^{i}+4^{i}}$

17. $\sum_{i=1}^{\infty}\frac{1}{5^{i}+4^{i+1}}$

18. $\sum_{i=1}^{\infty}\frac{(-1)^{i}}{2^{2i}}$

19. $\sum_{i=1}^{\infty}\frac{(-1)^{i+1}}{3^{2i-1}}$

20. $\sum_{i=1}^{\infty}\frac{1}{\sqrt{3^{i}}}$

IV. Вычислить и вывести на экран значение функции F(x) на отрезке *[a,b]* с шагом *h=0.1* и точностью ε. Результат работы программы представить в виде следующей таблицы:

№	Значение x	Значение функции F(x)	Количество просуммированных слагаемых n
1			
2			
…			

Замечание. При решении задачи использовать вспомогательную функцию.

1. $F(x) = 1 + \frac{x^2}{4} + \frac{x^3}{4^2} + \frac{x^4}{4^3} + \frac{x^5}{4^4} + \ldots, \quad x \in [0.1; 0.9].$

2. $F(x) = 1 - \frac{x}{2 \cdot 7} + \frac{x^2}{4 \cdot 14} - \frac{x^3}{8 \cdot 21} + \frac{x^4}{16 \cdot 28} - \ldots, \quad x \in [0; 0.9].$

3. $F(x) = 1 - \frac{x^2}{1 \cdot 3 \cdot 4} + \frac{x^4}{2 \cdot 4 \cdot 5} - \frac{x^6}{3 \cdot 5 \cdot 6} + \frac{x^8}{4 \cdot 6 \cdot 7} - \ldots, \quad x \in [0.2; 0.7].$

4. $F(x) = 1 + \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 2^2} + \frac{x^7}{7 \cdot 2^3} + \ldots, \quad x \in [0; 0.99].$

5. $F(x) = 1 + \frac{x}{1 \cdot 4} - \frac{x^2}{2 \cdot 5} + \frac{x^3}{3 \cdot 6} - \frac{x^4}{4 \cdot 7} + \ldots, \quad x \in [0.1; 0.9].$

6. $F(x) = 1 - \frac{x^3}{3 \cdot 4^2} + \frac{x^5}{4 \cdot 5^2} - \frac{x^7}{5 \cdot 6^2} + \ldots, \quad x \in [0; 0.8].$

7. $F(x) = 1 + \frac{x}{3} + \left(\frac{x}{3}\right)^2 + \ldots, \quad x \in [0; 0.9].$

8. $F(x) = 1 + \frac{x^2}{2 \cdot 4} + \frac{x^3}{4 \cdot 6} + \frac{x^4}{6 \cdot 8} + \ldots, \quad x \in [0.2; 0.6].$

9. $F(x) = 1 + \frac{x}{1 \cdot 3} + \frac{x^2}{1 \cdot 3 \cdot 5} + \frac{x^3}{1 \cdot 3 \cdot 5 \cdot 7} + \ldots, \quad x \in [0.05; 0.95].$

10. $F(x) = -\frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \frac{1}{7x^7} - \ldots., \quad x \in [-3; -2].$

11. $F(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots, \quad x \in [0, 1].$

12. $F(x) = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \ldots, \quad x \in [2, 3].$

13. $F(x) = 1 + x^2 + \frac{x^4}{2!} + \frac{x^6}{3!} + \frac{x^8}{4!} + \ldots, \quad x \in [-1, 0].$

14. $F(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots, \quad x \in [1, 2].$

15. $F(x) = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \frac{x^8}{9!} - \ldots, \quad x \in [0, 1].$

16. $F(x) = -\frac{x}{2!} + \frac{x^3}{4!} - \frac{x^5}{6!} + \frac{x^7}{8!} - \ldots, \quad x \in [-1, 0].$

17. $F(x) = 2\left(\frac{x-1}{x+1} + \frac{(x-1)^3}{3(x+1)^3} + \frac{(x-1)^5}{5(x+1)^5} + \ldots\right), \quad x \in [1; 2].$

18. $F(x)=\frac{x-1}{x}+\frac{(x-1)^2}{2x^2}+\frac{(x-1)^3}{3x^3}+\ldots, \quad x\in[1;2].$

19. $F(x)=(x-1)-\frac{(x-1)^2}{2}+\frac{(x-1)^3}{3}-\frac{(x-1)^4}{4}+\ldots, \quad x\in[0.5;1.5].$

20. $F(x)=\frac{\pi}{2}-\left(x+\frac{x^3}{2\cdot3}+\frac{3x^5}{2\cdot4\cdot5}+\frac{3\cdot5x^7}{2\cdot4\cdot6\cdot7}+\frac{3\cdot5\cdot7x^9}{2\cdot4\cdot6\cdot8\cdot9}+\ldots\right), \quad x\in[-0.9;0.9].$

6. МАССИВЫ

6.1. Указатели

Когда компилятор обрабатывает оператор определения переменной, например, *int a=50;* , то он выделяет память в соответствии с типом *int* и записывает в нее значение *50*. Все обращения в программе к переменной по ее имени заменяются компилятором на адрес области памяти, в которой хранится значение переменной. Программист может определить свои собственные переменные для хранения адресов области памяти. Такие переменные называются *указателями*. В C++ различают три вида указателей – *указатели на объект, на функцию* и *на void*, которые отличаются друг от друга свойствами и допустимыми операциями. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим – базовым типом.

Указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа (простого или составного). Простейшее объявление указателя на объект имеет следующий вид:

<базовый тип> [<модификатор>] * <имя указателя>;

где *базовый тип* – имя типа переменной, адрес которой будет содержать переменная указатель; тип может быть любым, кроме ссылки (см. следующий раздел) и битового поля (см. справочники по C++);

модификатор необязателен и может иметь значение: *near, far* или huge (см. справочники по C++).

Замечание. По умолчанию устанавливается модификатор *near*. Нам этого будет достаточно для решения задач, поэтому при описании указателей модификатор явным образом указывать не будем.

Указатель может быть переменной или константой, указывать на переменную или константу, а также быть указателем на указатель. Например:

```
int i;                  //целочисленная переменная
const int j=10;         //целочисленная константа
int *a;                 //указатель на целочисленное значение
int **x;                //указатель на указатель на целочисленное значение
const int *b;           //указатель на целочисленную константу
int * const c=&i;       //указатель-константа на целочисленную переменную
const int *const d=&j;        //указатель-константа на целую переменную
```

Как видно из примера, модификатор *const*, находящийся между именем указателя и звездочкой, относится к самому указателю и запрещает его изменение, а *const* слева от звездочки задает постоянство значения, на которое он указывает.

Величины типа указатель, подчиняются общим правилам определения области действия, видимости и времени жизни. Память под указатели выделяется в сегменте данных или в стеке (в зависимости от места описания и спецификатора класса памяти), а область памяти, связанная с указателем, обычно выделяется в динамической памяти (куче) (см. раздел 2.5).

Указатель на функцию содержит адрес в сегменте кода, по которому передается управление при вызове функции. Указатели на функцию используются для косвенного вызова функции (не через ее имя, а через переменную, хранящую ее адрес), а также для передачи функции в другую функцию в качестве параметра.

Замечание. Более подробно данный вид указателей будет рассмотрен в следующей части пособия.

Указатель типа void применяется в тех случаях, когда конкретный тип объекта, адрес которого нужно хранить, не определен. Указателю на void можно присвоить значение указателя любого типа, а также сравнить его с любым указателем, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом.

Перед использованием любого указателя надо выполнить его *инициализацию*, т.е. произвести присваивание начального значения. Существуют следующие способы инициализации указателя:

1) присваивание указателю адреса существующего объекта:
   - с помощью операции получения адреса:

```
int a=50;        //целая переменная
int *x=&a;       //указателю присваивается адрес целой переменной a
int *y (&a);     // указателю присваивается адрес целой переменной a
```

   - с помощью значения другого инициализированного указателя:

```
int *z=x;        //указателю присваивается адрес, хранящийся в x
```

   - с помощью имени массива или функции (рассмотрим позже).
2) присваивание указателю адреса области памяти в явном виде:

```
int *p=(int *) 0xB8000000;
```

   где 0xB8000000 – шестнадцатеричная константа, *(int *)* – операция явного приведения типа к типу указатель на целочисленное значение.
3) присваивание пустого значения:

```
int *x=NULL;
int *y=0;
```

   где NULL стандартная константа, определенная как указатель равный 0
4) выделение участка динамической памяти и присваивание ее адреса указателю:

```
int *a = new int;        //1
int *b = new int (50);   //2
```

В *строке 1* операция *new* выполняет выделение достаточного для размещения величины типа *int* участка динамической памяти и записывает адрес начала этого участка в переменную *a*. Память под переменную *a* выделяется на этапе компиля-

ции. В *строке 2*, кроме действий описанных выше, производится инициализация выделенной динамической памяти значением 50.

Освобождение памяти, выделенной с помощью операции *new*, должно выполняться с помощью операции *delete*. При этом переменная-указатель сохраняется и может инициализироваться повторно. Пример использования операции *delete*:

```
delete a; delete []b;
```

С указателями можно выполнять следующие операции: разадресации или косвенного обращения к объекту, получения адреса, присваивания, сложения с константой, вычитание, инкремент, декремент, сравнение и приведение типов.

Операция разадресации (*) предназначена для доступа к значению, адрес которой хранится в указателе. Эту операцию можно использовать как для получения значения, так и для его изменения. Рассмотрим пример:

```
#include <iostream>
using namespace std;
int main()
{ int *a=new int(50);          //инициализация указателя на целочисленное значение
  int b=*a;     //переменной b присваивается значение, хранящееся по адресу указателя a
  cout <<"adress  \t *a\t b\n";
  /*выводим: адрес, хранящийся в указателе a; значение, хранящееся по адресу
  указателя  a; значение переменной b*/
  cout <<a <<"\t" <<*a<<"\t" <<b<<endl;
  *a=100;      // изменяем значение, хранящееся по адресу указателя a
  cout <<a <<"\t" <<*a<<"\t" <<b<<endl;
  return 0;}
```

Результат работы программы:

adress	*a	b
00355900	50	50
00355900	100	50

Замечания

1) При запуске данной программы на вашем компьютере будет выведен другой адрес, записанный в указателе *a*. Это связано с текущим состоянием динамической памяти.
2) При попытке выполнить команду *a=100*; возникнет ошибка, т.к. *a* – переменная-указатель, в ней может храниться только адрес ОП, но не целочисленное значение.
3) При попытке выполнить команду **b=100*; также возникнет ошибка, т.к. операция разадресации может применяться только к указателям, а не к обычным переменным.

Операция получения адреса (&) применима к величинам, имеющим имя и размещенным в ОП. Рассмотрим пример:

```
#include <iostream>
using namespace std;
int main()
{ int b=50;
  int *a=&b;   //в указатель a записали адрес переменной b
  cout <<"adress  \t *a\t b\n";
  cout <<a <<"\t" <<*a<<"\t" <<b<<endl;
  b+=10;      //1
  cout <<a <<"\t" <<*a<<"\t" <<b<<endl;
  *a=100;     //2
  cout <<a <<"\t" <<*a<<"\t" <<b<<endl;
  return 0; }
```

Результат работы программы:

adress	*a	b
0012FF60	50	50
0012FF60	60	60
0012FF60	100	100

Замечание. После того. как в указатель *a* был записан адрес переменной *b*, мы получили доступ к одной и той же области памяти через имя обычной переменной и через указатель. Поэтому, изменяя в *строке 1* значение переменной *b*, мы фактически изменяем значение по адресу, записанному в указателе *a*. И наоборот (см. *строку 2*).

Арифметические операции с указателями (сложение с константой, вычитание, инкремент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл при работе со структурами данных, последовательно размещенными в памяти, например, с массивами.

Более подробно эти операции будут рассмотрены при изучении массивов. Однако хотелось обратить внимание на следующий пример:

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

int main()
{ int *a =new int(50);
  cout <<"adress  \t *a\n";
  cout <<a <<"\t" <<*a<<endl;
  (*a)++;       // 1  увеличивается на 1 значение, хранящееся по адресу указателя a
  cout <<a <<"\t" <<*a<<endl;
  *(a++);       //2  значение указателя изменяется на величину sizeof (int)
  cout <<a <<"\t" <<*a<<endl;
  *a++;         //3 с учетом приоритета операций * и ++ аналог строки 2
  cout <<a <<"\t" <<*a<<endl;
  return 0;}
```

Результат работы программы:

adress	*a
00355900	50
00355900	51
00355904	-31686019
00355908	393224

Замечание. В *строке 2* значение указателя изменилось на величину sizeof (int) – в нашем случае на 4 байта, в результате чего от адреса 00355900 мы перешли к адресу 00355904. Т.к. данные по новому адресу нами не были инициализированы, то на экран вывелось случайное число. Аналогичным образом была выполнена *строка 3*.

Операция присваивания используется для изменения значения – указателя или значения, которое хранится по адресу, записанному в указателе. Использование данной операции мы рассмотрели в ходе изучения других операций.

6.2. Ссылки

Ссылка представляет собой синоним имени, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который разыменовывается неявным образом. Формат объявления ссылки:

<базовый тип> & <имя ссылки>

Например:

```
int a;          //целочисленная переменная
int &b=a;       //ссылка на целочисленную переменную a
```

Рассмотрим следующий пример:

```
#include <iostream>
using namespace std;
int main()
{ int a=50;    //целочисленная переменная a
  int &b=a;    //ссылка b – альтернативное имя для переменной a
  cout <<"a\t b\n";
  cout <<a <<"\t" <<b<<endl;
  a++;                          //1
  cout <<a <<"\t" <<b<<endl;
  b++;                          //2
  cout <<a <<"\t" <<b<<endl;
  return 0;}
```

Результат работы программы:

a	b
50	50
51	51
52	52

Замечание. Переменная *a* и ссылка *b* обращаются к одному и тому же участку памяти, поэтому, при изменении значение *a*, изменяется значение *b*, и наоборот. Однако, в отличие от указателей для обращения к значению по ссылке (см. строку 2) нам не надо использовать операцию разадресации, т.к. она выполняется неявным образом.

Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых значений. В отличие от указателей, они не занимают дополнительного пространства в памяти и являются просто другим именем величины. Однако при работе со ссылками нужно помнить следующие правила:

1) Тип ссылки должен совпадать с типом величины, на которую она ссылается.
2) Переменная-ссылка должна быть явно инициализирована при описании, кроме случаев, когда она является параметром функции, имеет спецификатор extern или ссылается на поле данных класса.
3) Повторная инициализация ссылки невозможна.
4) Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки ссылок.

6.3. Одномерные массивы

Одномерный массив – это фиксированное количество элементов одного и того же типа, объединенных общим именем, где каждый элемент имеет свой номер. Нумерация элементов массива в C++ начинается с нулевого элемента, то есть, если массив состоит из 10 элементов, то его элементы будут иметь следующие номера: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Элементы массива могут быть любого типа, в том числе и структурированного (кроме файлового).

Между указателями и массивами существует взаимосвязь: любое действие над элементами массивов, которое достигается индексированием, может быть выполнено и с помощью указателей. Вариант программы с указателями будет работать быстрее, но для понимания он сложнее. Рассмотрим различные варианты реализации одномерных массивов.

Статические одномерные массивы

Формат объявления статического массива:

<базовый тип> <имя массива>[<размер массива>]={<список инициализации>}

Замечание. В данном случае [] являются операцией индексации, а не признаком необязательного элемента в формате объявления массива.

Одновременно с объявлением массива может проводиться его инициализация, например:

```
float v[10];    // массив из 10 элементов с плавающей точкой, не инициализирован
int  a[]={1,2,3};      // массив из 3 элементов целого типа, инициализирован
                       // списком чисел 1, 2, 3
int b[5]={1,2,3};      // массив из 5 элементов целого типа,  инициализирован
                       // списком чисел 1, 2, 3, 0, 0
```

Память под статический массив выделяется на этапе объявления массива в стеке или сегменте данных (в зависимости от спецификатора класса памяти, указанного явно или используемого по умолчанию).

Для обращения к элементу массива указывается имя массива, а затем в квадратных скобках индекс (номер). В примере, рассмотренном выше, для обращения к элементу массива *a* с номером 2 нужно записать *a[2]*, к элементу с номером ноль – *a[0]*. Индексом может быть любое целое выражение, образуемое целыми переменными и целыми константами.

Ввод и вывод массивов производится поэлементно. В следующем фрагменте программы производится потоковой ввод и вывод 10 элементов целочисленного массива:

```
int b[10];      //объявление массива
for (int i = 0; i<10; i++)      //ввод элементов массива
  { cout<<"b["<<i<<"]="; cin>>b[i];}
...
for (int i = 0; i<10; i++)      //вывод элементов массива на экран
   cout<<"b["<<i<<"]="<<b[i]<<"\n";
```

Замечание. При работе со статическими одномерными массивами можно использовать указатели. Например:

```
int b[10];      //объявление массива b
int *p=&b[0];   //определяем указатель на тип int и устанавливаем его на нулевой элемент
                //массива b
for (int i=0;i<10; i++)  //ввод элементов массива
 {cout<<"b["<<i<<"]=";
  cin>>*p++;}           //введенное значение помещаем по адресу указателя p и сдвигаем
                        //указатель на следующий элемент массива
...
p=&b[0];        //повторно устанавливаем указатель на нулевой элемент массива b
for (int i=0;i<10; i++)  //вывод элементов массива
 cout<<"b["<<i<<"]="<<*p++<<endl; //выводим на экран значение, хранящееся по адресу
                        // указателя p, и сдвигаем указатель на следующий элемент массива
```

Команда *p++ с учетом приоритетов операций эквивалентна команде (*p)++. При этом вначале будет произведено обращение к значению, которое хранится по адресу, записанному в указатель p, а затем этот адрес изменится на столько байт, сколько требуется для размещения в памяти базового типа указателя. Для одномерного массива, в котором элементы располагаются последовательно друг за другом, произойдет переход к адресу следующего элемента массива.

Динамические одномерные массивы

Динамические одномерные массивы отличаются от статических тем, что:

1) имя массива является указателем-переменной;
2) память под элементы массива выделяется в куче с помощью специальной команды new.

Формат объявления динамического массива:

<базовый тип> * <имя массива>= new <базовый тип> [<размерность массива>];

Замечание. В данном случае [] являются операцией индексации, а не признаком необязательного элемента в формате объявления массива.

Например:

```
float *v=new float [10]; //вещественный массив из 10 элементов, не инициализирован
int  *a=new int [3];   // массив из 3 элементов целого типа, не инициализирован
```

Обращаться к элементам массива можно через индексы, как и для статических массивов, а можно и через указатели. Например, чтобы обратиться к нулевому элементу массива можно написать *a[0]* или **a*, а для обращения к элементу массива *a* с номером i нужно написать *a[i]* или **(a+i)*.

Рассмотрим более подробно обращение к элементам массива через указатели. Указатель *a* хранит адрес нулевого элемента массива, поэтому, когда мы выполняем операцию **a*, то мы обращаемся к значению, которое хранится по данному адресу. Выражение **(a+2)*, говорит о том, что вначале мы сдвинемся по отношению к нулевому элементу массива *a* на столько байт, сколько занимают 2 элемента, а затем обратимся к значению, которое там хранится.

Рассмотренные способы обращения к элементам массива эквивалентны друг другу. Однако первый способ более понятен для восприятия, а во втором случае программа будет выполняться быстрее, т.к. компилятор автоматически переходит от индексного способа обращения к массиву к обращению через указатели.

Ввод и вывод массивов также как и в случае со статическими массивами производится поэлементно. В следующем фрагменте программы производится потоковой ввод и вывод 10 элементов целочисленного массива:

```
int *b=new int [10]; //объявили динамический массив, для хранения 10 целых чисел
for (int i=0;i<10; i++)  // ввод элементов массива
 {cout<<"b["<<i<<"]=";
 cin>>*(b+i);}  // введенное значение записываем в i-тый элемент массива b
...
for (int i=0;i<10; i++)
  cout<<"b["<<i<<"]="<<*(b+i)<<endl;     //выводим на экран значение i-того элемента
                                         // массива  b
```

Освободить память, выделенную под динамический массив можно с помощью операции *delete*. Например, освобождение памяти, выделенной под элементы массива b из предыдущего примера, производится следующим образом:

```
delete [] b;
```

Замечание. Если в данной записи пропустить квадратные скобки, то ошибки не будет, однако освобождена будет память, выделенная под нулевой элемент массива, а с остальными элементами массива «связь» будет потеряна и они не будут доступны для дальнейших операций. Такие ячейки памяти называются мусором. Кроме того, операция *delete* освобождает память, выде-

ленную в куче под элементы массива, а сам указатель *b* при этом сохраняется и может повторно использоваться для определения другого массива.

Передача одномерных массивов в качестве параметров

При использовании в качестве параметра одномерного массива в функцию передается указатель на его нулевой элемент. Напомним, что указателем на нулевой элемент является имя одномерного массива. Операцию разадресации внутри функции при обращении к элементам массива выполнять не надо, т.к. запись вида *a[i]* автоматически заменяется компилятором на выражение **(a+i)*.

При передаче массива в качестве параметра функции информация о количестве элементов теряется, поэтому размерность массива следует передавать в качестве отдельного параметра. Рассмотрим несколько примеров.

1. Передача статического массива:

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

/* в функцию в качестве параметров передаются статический массив mas и его размерность n*/
void print (int mas[], int n)
{ for (int i=0; i<n; i++) cout<<mas[i]<<"\t";
  cout<<endl; }

int main()
{int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
 print(a,10); cout<<endl;
 print (a, 7);
 print(a, 13);
 return 0; }
```

Результат работы программы:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7 8 9 -858993406 1245112 4269558
```

Замечание. При втором вызове функции *print* в качестве параметра *n* мы передали значение меньшее реального размера массива, поэтому мы смогли распечатать не весь массив, а его часть. Если же в функцию в качестве параметра n передать значение больше реального размера массива, то на экран будут выведены случайные числа. Это объясняется тем, что будут задействованы участки памяти, которые не закреплены за массивом. Подумайте, чем опасен для программы выход за границы массива.

2. Передача динамического массива

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

/* в функцию в качестве параметров передаются динамический массив mas и его размерность n*/
void print (int *mas, int n)
{ for (int i=0; i<n; i++)  cout<<mas[i]<<"\t";}

int main()
{ int n=10;
  int *a=new int [n];
  for (int i=0; i<n;i++) a[i]=i*i;
  print(a,10);
  return 0; }
```

Результат работы программы:

```
0 1 4 9 16 25 36 49 64 81
```

Поскольку массив всегда передается в функцию по адресу, то все изменения, внесенные в формальный параметр массив, отразятся на фактическом параметре массиве. Если необходимо запретить изменение массива в функции, то его нужно передавать как параметр константу. Например:

```
int sum (const int *mas, const int n)
{ ... }
```

Массив как возвращаемое значение функции

Массив может возвращатся в качестве значения функции только как указатель:

```
#include <iostream>
using namespace std;
int * creat( int n)
{int *mas=new int [n];
 for (int i=0; i<n; i++)mas[i]=i*i;
 return mas;}
void print (int *mas, int n)
{ for (int i=0; i<n; i++)  cout<<mas[i]<<"\t"; }
int main()
{ int n=5;
  int *a=creat(n);
  print(a,n);
  return 0; }
```

Результат работы программы:

0 1 4 9 16

Замечание. В общем случае нельзя возвращать из функции указатель на локальную переменную, поскольку память, выделенная под локальную переменную, освобождается после выхода из функциию. Например:

```
int *f()
{int a=5;
 return &a; } //нельзя!!!
```

Однако в случае с динамическим массивом *mas*, который является указателем на область данных в куче, такое действие возможно. Команда *return mas* возвратит в качестве значения функции адрес в куче, начиная с которого последовательно хранятся элементы массива. После этого локальная переменная mas перестанет существовать и осовободится соответствующая ей область стека (см. раздел 2.5). Но куча при этом не освобождается, поэтому элементы массива будут нам доступны.

6.4. Примеры использования одномерных массивов

Одномерные массивы удобно использовать тогда, когда данные можно представить в виде некоторой последовательности, элементы которой проиндексированы. Напомним, что индексация элементов в одномерном массиве в C++ начинается с нуля. При решении задач мы будем использовать статические массивы и обращаться к элементам через индексы. Однако мы советуем вам попробовать модифицировать данные задачи для работы с динамическими массивами.

1. Дан массив из n целых чисел ($n \le 20$). Написать программу, которая заменяет в данном массиве все отрицательные элементы нулями.

```
#include <iostream>
using namespace std;
int main()
{  int a[20];    // объявляем статический массив, задав максимальную размерность
   int n; cout<<"n="; cin>>n;                  //ввели количество элементов массива
   for (int i=0;i<n; ++i)          //ввод и обработка данных
   {cout<<"a["<<i<<"]="; cin>>a[i]; // ввод очередного элемента
```

```
  if (a[i]<0)          //если i-ый элемент массива отрицательный,
    a[i]=0;}           //то заменяем его нулем
 for (int i=0;i<n;++i) cout<<a[i]<<"\t"; //вывод массива на экран
return 0; }
```

Результат работы программы:

Исходные данные	Измененные данные
2 –4 1 2 –2 0 23 –12 1 -1	2 0 1 2 0 0 23 0 1 0

2. Дан массив из n действительных чисел (n≤100). Написать программу для подсчета суммы этих чисел.

```
#include <iostream>
using namespace std;
int main()
{ float a[100];
  int n; cout<<"n="; cin>>n;
  float s=0;
  for (int i=0;i<n; ++i)
   {cout<<"a["<<i<<"]="; cin>>a[i]; //ввод очередного элемента в массив
    s+=a[i];}                       //добавление значения элемента массива к сумме
  cout <<"s="<<s<<endl;
return 0; }
```

Результат работы программы:

n	Исходные данные	Ответ
5	2.3 0 2.5 1.7 -1.5	S = 5

Замечание. Обратите внимание на то, что при подсчете суммы мы использовали уже известный нам из раздела 5 прием накопления суммы *s+=a[i]*.

3. Дан массив из n целых чисел (n≤50). Написать программу для подсчета среднего арифметического четных значений данного массива.

```
#include <iostream>
using namespace std;
int main()
{  int a[50];
   int n, k=0;
   cout<<"n="; cin>>n;
   float s=0;
   for (int i=0;i<n; ++i)
    {cout<<"a["<<i<<"]="; cin>>a[i];        //ввод очередного элемента в массив
     if (!(a[i]%2))      // если остаток при делении  элемента на 2 равен 0
       {s+=a[i];++k;}  //то элемент четный  - добавить его к сумме и увеличить
     }                 // количество четных  элементов на 1
  if (k)          //если k не нулевое, то четные числа в последовательности есть
    cout <<"sr="<< s/k<<endl;   //и можно вычислить их среднее арифметическое значение
    else cout<<" четных чисел в последовательности нет "<<endl;
  return 0; }
```

Результат работы программы:

n	Исходные данные	Ответ
5	1 3 7 –41 9	четных чисел в последовательности нет
4	2 4 6 4	sr = 4.00

Замечание. Выражение *a[i]%2* будет давать 0, если *a[i]* четное число. В C++ 0 трактуется как *ложь*, поэтому в операторе *if* мы ставим операцию логического отрицания (!) перед этим выражением.

4. Дан массив из n целых чисел (n≤30). Написать программу, которая определяет наименьший элемент в массиве и его порядковый номер.

```
#include <iostream>
using namespace std;
int main()
{ int a[30];
  int n; cout<<"n="; cin>>n;
  for (int i=0;i<n; ++i) {cout<<"a["<<i<<"]="; cin>>a[i];}
  int min=a[0];    //в качестве наименьшего значения полагаем нулевой элемент массива
  int nmin=0;      //соответственно его порядковый номер равен 0
  for (int i=1;i<n; ++i) //перебираем все элементы массива с первого по последний
    if (a[i]<min)  //если очередной элемент окажется меньше значения min, то в качестве
     {min=a[i];    //нового наименьшего значения запоминаем значение текущего элемента
      nmin=i;}     //массива и, соответственно, запоминаем его номер
  cout <<"min="<< min<<"\t nmin=" << nmin<<endl;
return 0;}
```

Результат работы программы:	n	Исходные данные	Наименьшее значение	Его номер
	5	*1 3 7 -41 9*	*-41*	*4*

5. Дан массив из n действительных чисел (n≤20). Написать программу, которая меняет местами в этом массиве наибольший и наименьший элемент местами (считается, что в последовательности только один наибольший и один наименьший элементы).

```
#include <iostream>
using namespace std;
int main()
{ float a[20];
  int n; cout<<"n="; cin>>n;
  for (int i=0;i<n; ++i) {cout<<"a["<<i<<"]="; cin>>a[i];}
  //первоначально полагаем элемент с номером 0 минимальным и максимальным
  // и запоминаем их номера
  float min=a[0], max=a[0];
  int nmin=0, nmax=0;
  for (int i=1;i<n; ++i) //поиск наибольшего и наименьшего значения в массиве и их номеров
    {if (a[i]<min){min=a[i];nmin=i;}
     if (a[i]>max){max=a[i];nmax=i;}}
  a[nmax]=min; //в позицию наименьшего элемента записываем значение наибольшего
  a[nmin]=max; //в позицию наибольшего элемента записываем значение наименьшего
  for (int i=0;i<n; ++i) cout<<a[i]<<"\t"; //выводим измененный массив на экран
return 0;}
```

Результат работы программы:	n	Исходные данные	Измененные данные
	4	1.1 3.4 –41.2 9.9	1.1 3.4 9.9 -41.2

6. Дан массив из n действительных чисел (n≤20). Написать программу, которая подсчитывает количество пар соседних элементов массива, для которых предыдущий элемент равен последующему.

```
#include <iostream>
using namespace std;
int main()
{ float a [20];
  int n, k=0; cout<<"n="; cin>>n;
```

```
for (int i=0;i<n; ++i) {cout<<"a["<<i<<"]="; cin>>a[i];}
for (int i=0;i<n-1; ++i)
  //если соседние элементы равны, то количество искомых пар увеличиваем на 1
  if (a[i]==a[i+1]) ++k;
cout<<"k="<<k<<endl;
return 0; }
```

Результат работы программы:

n	Исходные данные	Ответ
6	1.1 3.4 –41.2 9.9 3.1 -3.7	k=0
6	1.1 1.1 –41.2 -3.7 -3.7 -3.7	k=3

Замечание. Обратите внимание на то, что в последнем цикле параметр *i* принимает значения от *0* до *n-2*, а не до *n-1*. Это связано с тем, что для *i=n-2* существует пара с номерами *(n-2, n-1)*, а для *i=n-1* пары с номерами *(n-1, n)* не существует, так как в массиве всего n элементов и последний элемент имеет номер *n-1*.

6.5. Двумерные массивы

Двумерные массивы (матрицы, таблицы) – представляют собой фиксированное количество элементов одного и того же типа, объединенных общим именем, где каждый элемент определяется номером строки и номером столбца, на пересечении которых он находится. Нумерация строк и столбцов начинается с нулевого номера. Поэтому если массив содержит три строки и четыре столбца, то строки нумеруются: 0, 1, 2; а столбцы: 0, 1, 2, 3. В C++ двумерный массив реализуется как одномерный, каждый элемент которого также массив. При этом массивы бывают статические и динамические.

Статические двумерные массивы

Формат объявления статического двумерного массива:

<базовый тип> <имя массива>[число строк][число столбцов]

Замечания

1) В данном случае [] являются операцией индексации, а не признаком необязательного элемента в формате объявления массива.
2) Память под статический массив выделяется на этапе объявления массива в стеке или сегменте данных (в зависимости от спецификатора класса памяти, указанного явно или используемого по умолчанию).

Одновременно с объявлением массива может проводиться его инициализация, например:

```
//вещественный массив из 4 строк и 3 столбцов, массив не инициализирован
float a[4][3];
/*целочисленный массив из 3 строк и 2 столбцов, в нулевой строке записаны
 единицы, в первой строке  - двойки, во второй строке – тройки */
int b[3][2]={1,1, 2, 2, 3, 3} ;
//аналог массива b, но так как количество строк и столбцов не указано явным образом,
то содержимое каждой строки помещается в дополнительные фигурные скобки*/
int c[][]={{1,1}, {2, 2}, {3, 3}};
```

Для обращения к элементу двумерного массива нужно указать имя массива, номер строки и номер столбца (каждый в отдельной паре квадратных скобок), на пересечении которых находится данный элемент массива. Например, для массива *a* обратиться к элементу, расположенному на пересечении строки с номером 0 и столбца с номером 3, можно так: *a[0][3]*.

Ввод и вывод двумерного массива осуществляется поэлементно. В следующем фрагменте программы производится потоковой ввод и вывод элементов целочисленного массива, содержащего 4 строки и 5 столбцов:

```
int b[4][5];    //объявление массива
for (int i = 0; i<4; ++i)        //ввод элементов массива
 for (int j = 0; j<5; ++j)
  {cout<<"b["<<i<<"]["<<j<<"]="; cin>>b[i][j];}
...
for (int i = 0; i<4; ++i, cout<<endl)//вывод элементов массива на экран
 for (int j = 0; j<5; ++j)
   cout<<"b["<<i<<"]["<<j<<"]="<<b[i][j]<<"\t";
```

Замечание. При работе со статическими двумерными массивами можно использовать указатели. Например:

```
int b[4][5];     //объявление массива
int *p=&b[0][0]; //определяем указатель на тип int и устанавливаем его на элемент b[0][0]
for (int i = 0; i<4; ++i) //ввод элементов массива
 for (int j = 0; j<5; ++j)
   cin>>*p++; //введенное значение помещаем по адресу указателя p и сдвигаем
              //указатель на следующий элемент массива
...
p=&b[0][0];    //повторно устанавливаем указатель на элемент b[0][0]
for (int i = 0; i<4; ++i, cout<<endl)   //вывод элементов массива на экран
 for (int j = 0; j<5; ++j)
   cout<<*p++<<"\t"; //выводим на экран значение, хранящееся по адресу
                     // указателя p, и сдвигаем указатель на следующий элемент массива
```

Напомним, что команда *p++ с учетом приоритетов операций эквивалентна команде (*p)++. При этом вначале будет произведено обращение к значению, которое хранится по адресу, записанному в указатель p, а затем этот адрес изменится на столько байт, сколько требуется для размещения в памяти базового типа указателя. Для двумерного массива, в котором элементы располагаются последовательно друг за другом (вначале элементы нулевой строки, затем первой строки и т.д.), произойдет переход к адресу следующего элемента массива.

Динамические двумерные массивы

Замечание. Напомним, динамические массивы отличаются от статических тем, что имя массива является указателем-переменной, а память под элементы массива выделяется программистом в куче с помощью специальной команды new.

Формат объявления динамических массивов:

<базовый тип> ** <имя массива>= new <базовый тип> *[<размерность массива>];

Замечание. В данном случае [] являются операцией индексации, а не признаком необязательного элемента в формате объявления массива.

Например:

```
int **a= new int *[10];
```

В данном случае переменная *a* является указателем на массив указателей целого типа. Чтобы из него получить двумерный массив нам потребуется выделить память под каждый из десяти указателей. Это можно сделать с помощью цикла:

```
for (int  i=0; i<10; ++i)
 a[i]=new int [5];
```

В этом случае мы получим двумерный массив целых чисел, который содержит 10 строк и 5 столбцов. В общем случае вместо констант 10 и 5 можно поставить переменные и тогда получим массив необходимой размерности для данной задачи.

Кроме того, для каждого значения *i* мы можем создавать одномерные массивы различной размерности. В результате у нас получится не прямоугольный массив, а свободный (количество элементов в каждой строке различно).

Обратиться к элементам динамического массива можно с помощью индексации, например, *a[i][j]* , или с помощью указателей **(*(a+i)+j)*.

Рассмотрим более подробно второе выражение. Указатель *a* хранит адрес указателя на нулевую строку. Операция *a+i* обратится к указателю на i-тую строку массива. Выражение **(a+i)* позволит обратиться к значению, хранящемуся по адресу *a+i* – фактически к адресу нулевого столбца *i*-той строки двумерного массива. И, соответственно, выражение **(*(a+i)+j)* позволит обратиться к значению *j*-того столбца *i*-той строки.

Замечание. Как и в случае с одномерными массивами, рассмотренные способы обращения к элементам массива эквивалентны друг другу. Однако первый способ более понятен для восприятия, а во втором случае программа будет выполняться быстрее, т.к. компилятор автоматически переходит от индексного способа обращения к массиву к обращению через указатели.

Ввод и вывод массивов также как и в случае со статическими массивами производится поэлементно. В следующем фрагменте программы производится потоковой ввод и вывод целочисленного массива, который содержит 4 строки и 5 столбцов:

```
int **b=new int *[4]; //объявили двумерный динамический массив
for (int  i=0; i<2; ++i) b[i]=new int [5];
for (int i=0;i<4; ++i)  //ввод элементов массива
 for (int j=0;j<5; ++j)
  {cout<<"b["<<i<<"]["<<j<<"]=";
   cin>>*(*(b+i)+j);}  // введенное значение записываем в b[i][j]
...
for (int i=0;i<4; ++i, cout<<endl)  // вывод элементов массива
 for (int j=0;j<5; ++j)
  cout<<"b["<<i<<"]["<<j <<"]="<<*(*(b+i)+j)<< "\t"; //выводим на экран значение b[i][j]
```

Освободить память, выделенную под динамический массив, можно с помощью операции *delete*. Например, освобождение памяти, выделенной под элементы массива *b* из предыдущего примера, производится следующим образом:

```
// освобождаем память, выделенную под i-тую строку двумерного массива
for (int i=0;i<4; ++i)  delete [] b[i];
delete [] b; // освобождаем память, выделенную под массив указателей
```

Замечание. Данный фрагмент программы освобождает память, выделенную в куче под двумерный массив, сам указатель b при этом сохраняется и может повторно использоваться для определения другого массива.

Передача двумерных массивов в качестве параметров

При использовании в качестве параметра двумерного массива в функцию передается указатель на элемент, стоящий в нулевой строке и нулевом столбце. Этим указателем является имя массива. Также, как и в случае с одномерными массивами, операцию разадресации внутри функции при обращении к элементам массива выполнять не надо, т.к. запись вида *a[i][j]* автоматически заменяется компилятором на выражение **(*(a+i)+j)*.

При передаче двумерного массива в качестве параметра функции информация о количестве строк и столбцов теряется, поэтому размерности массива следует передавать в качестве отдельных параметров. Рассмотрим несколько примеров:

1. Передача статического массива:

```
#include <iostream>
using namespace std;
/* в функцию в качестве параметров передаются статический массив mas и его размерности: n – количество строк, m- количество столбцов*/
void print (int mas[3][2], int n, int m)
{ for (int i=0; i<n; ++i, cout<<endl)
  for (int j=0; j<m; ++j) cout<<mas[i][j]<<"\t";}
int main()
{int a[3][2]={{1, -2}, {-3, 4}, {-5,6} };
 print(a,3,2); cout<<endl;
 print (a, 2,2)
 return 0;}
```

Результат работы программы:

```
 1  -2
-3   4
-5   6
 1  -2
-3   4
```

Замечание. При втором вызове функции *print* в качестве параметра *n* мы передали значение меньше реального количества строк в массиве, поэтому мы смогли распечатать не весь массив, а его часть. Подумайте, что произойдет, если в качестве n или m передать значение большее реальной размерности массива.

2. Передача динамического массива:

```
#include <iostream>
using namespace std;
/* в функцию в качестве параметров передаются динамический массив mas и его размерности: n – количество строк, m- количество столбцов*/
void print (int **mas, int n, int m)
{ for (int i=0; i<n; ++i, cout<<endl)
   for (int j=0; j<m; ++j) cout<<mas[i][j]<<"\t";}
int main()
{int n=3, m=4;
 int **a=new int *[n];
 for (int i=0; i<n; ++i) a[i]=new int [m];
 for (int i=0; i<n; ++i)
   for (int j=0; j<m; ++j)a[i][j]=i+j;
 print(a,3,4);
 for (int i=0;i<n; ++i)  delete [] a[i]; //освобождение памяти
  delete [] a;
 return 0;}
```

Результат работы программы:

```
0 1 2 3
1 2 3 4
2 3 4 5
```

Поскольку массив всегда передается в функцию по адресу, то все изменения, внесенные в формальный параметр массив, отразятся на фактическом параметре массиве. Если необходимо запретить изменение массива в функции, то его нужно передавать как параметр константу. Например:

```
int sum (const int **mas, const int n)
{ ... }
```

Напомним, что двумерный массив – это одномерный массив указателей, каждый из который в свою очередь ссылается на одномерный массив. Поэтому двумерный массив можно передавать во вспомогательную функцию для обработки построчно:

```
#include <iostream>
using namespace std;
/* в функцию в качестве параметров передаются одномерный массив mas и его размерность n*/
void print (int *mas, int n)
{ for (int i=0; i<n; ++i) cout<<mas[i]<<"\t";
  cout<<endl; }
int main()
{int n=3, m=4;
 int **a=new int *[n];
 for (int i=0; i<n; ++i) a[i]=new int [m];
 for (int i=0; i<n; ++i)
   for (int j=0; j<m; ++j) a[i][j]=i+j;
 for (int i=0; i<n; ++i) // обработка двумерного массива построчно: в качестве аргумента
  print(a[i],m);        / функции print передаем указатель на i-тую строку
 for (int i=0;i<n; ++i)  delete [] a[i];//освобождение памяти
  delete [] a;
 return 0; }
```

Замечание. Результат работы программы будет соответствовать предыдущему примеру.

Массив как возвращаемое значение функции

Двумерный массив может возвращаться в качестве значения функции только как указатель:

```
#include <iostream>
using namespace std;
int ** creat(int n, int m)
{int **mas=new int *[n];
 for (int i=0; i<n; ++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j) mas[i][j]=i+j;
 return mas;}
int main()
{int n=5, m=5; int **a=creat(n,m);
 print(a,n,m);
 for (int i=0;i<n; ++i)  delete [] a[i]; //освобождение памяти
  delete [] a;
 return 0;}
```

Результат работы программы:

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

6.6. Примеры использования двумерных массивов

Двумерные массивы находят свое применение тогда, когда исходные данные представлены в виде таблицы, или когда для хранения данных удобно использовать табличное представление. В данном разделе мы рассмотрим примеры использования двумерных массивов. При решении задач мы будем использовать динамические массивы, но обращаться к элементам через индексы. Однако вы можете обращаться к элементам массива и через указатели. Для этого нужно будет заменить выражение *a[i][j]* на **(*(a+i)+j)*.

1. В двумерном массиве, элементами которого являются целые числа, подсчитать среднее арифметическое четных элементов массива.

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

//Функция создает и заполняем двумерный массив
int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[n];
 for (int i=0; i<n; ++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j) {cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}

int main()
{ int n,m, k=0;
  float s=0;
  int **a=creat(n,m);
  for (int i=0;i<n; ++i)  // обработка элементов массива
  for (int j=0;j<m; ++j)
    {if (!(a[i][j]%2))          //если элемент массива четный,  то добавляем его к сумме и
      {s+=a[i][j]; k++;}}       //увеличиваем количество четных элементов на 1
  if (k) cout <<s/k;
   else cout<<" Четных элементов в массиве нет'";
 for (int i=0;i<n; i++)  delete [] a[i]; //освобождаем память, выделенную под массив
 delete [] a;
 return 0;}
```

Результат работы программы:

n m	Массив $A_{n\times m}$	Ответ
2 3	2 1 3 1 3 6	4.00
3 2	1 3 5 1 7 9	Четных элементов в массиве нет

2. Дан двумерный массив, элементами которого являются целые числа. Найти значение максимального элемента массива.

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

int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[n];
 for (int i=0; i<n; ++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j) {cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}

int main()
{int n,m;
 cout <<"n="; cin >>n; cout <<"m="; cin >>m;//ввели размерность массива
 int **a=creat(n,m);
 int max=a[0][0]; //первоначально в качестве максимального элемента берём a[0][0]
 for (int i=0;i<n; ++i)  // просматриваем все элементы массива
  for (int j=0;j<m; ++j)
```

```
  if (a[i][j]>max)     //если очередной элемент больше значения максимального,
   max=a[i][j];        //то в качестве максимального запоминаем этот элемент
cout<<"max="<<max;
for (int i=0;i<n; i++)  delete [] a[i];//освобождаем память, выделенную под массив
delete [] a;
return 0; }
```

Результат работы программы:

n m	Массив $A_{n\times m}$	Ответ
2 3	2 1 3	6
	1 3 6	

3. Дана квадратная матрица, элементами которой являются вещественные числа. Подсчитать сумму элементов главной диагонали.

Указания по решению задачи. Для элементов, стоящих на главной диагонали характерно то, что номер строки совпадает с номером столбца. Этот факт будем учитывать при решении задачи.

```
#include <iostream>
using namespace std;
float ** creat(int &n)
{cout <<"n="; cin >>n;
 float **mas=new int *[n];
 for (int i=0; i<n; ++i) mas[i]=new int [n];
 for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j) {cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}
int main()
{ int n;
  float **a=creat(n);
  float s=0;
  for (int i=0;i<n; i++) //просматриваем все строки массива.
    s+=a[i][i]; //добавляем к сумме значение элемента, стоящего на главной диагонали
  cout<<" Сумма элементов главной диагонали ="<<s;
  for (int i=0;i<n; i++)  delete [] a[i]; //освобождаем память, выделенную под массив
  delete [] a;
  return 0;}
```

Результат работы программы:

n	Массив $A_{n\times n}$	Ответ
3	2.4 -1.9 3.1	Сумма элементов главной диагонали = 4.300
	1.1 3.6 -1.2	
	-2.1 4.5 -1.7	

4. Дана прямоугольная матрица, элементами которой являются вещественные числа. Поменять местами ее строки следующим образом: первую строку с последней, вторую с предпоследней и т.д.

Указания по решению задачи. Если в массиве *n* строк, то 0-ую строку нужно поменять с *n*-1, 1-ую строку – с *n-2*, *i*-ую строку - с *n-i-1*.

```
#include <iostream>
using namespace std;
float ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 float **mas=new int *[n];
 for (int i=0; i<n; ++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j) {cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
```

```
 return mas;}
int main()
{ int n, m;
  cout <<"n="; cin >>n; cout <<"m="; cin >>m;
  float **a=creat(n,m);
  float *z;
  for (int i=0;i<(n/2); ++i)  //меняются местами  i-ая и (n-i-1)-ая строки
    { z=a[i]; a[i]=a[n-i-1]; a[n-i-1]=z;}
  for (int i=0;i<n; ++i, cout<<endl)  //вывод измененного массива
   for (int j=0;j<m; ++j)   cout<<a[i][j]<<"\t";
  for (int i=0;i<n; ++i)  delete [] a[i];
  delete [] a;
  return 0;}
```

Замечание. Обратите внимание на то, что во время перестановки строк цикл перебирает не все строки, а только половину. Если бы перебирались все строки, то никакой перестановки не произошло, и массив оставался бы неизменным. Подумайте почему.

Результат работы программы:

n m	Массив $A_{n,m}$	Ответ
3 4	2.4 -1.9 3.1 0.0	-2.1 4.5 -1.7 4.9
	1.1 3.6 -1.2 3.7	1.1 3.6 -1.2 3.7
	-2.1 4.5 -1.7 4.9	2.4 -1.9 3.1 0.0

5. Дана прямоугольная матрица, элементами которой являются целые числа. Для каждого столбца подсчитать среднее арифметическое его нечетных элементов и записать полученные данные в новый массив.

Указания по решению задачи. Массив результатов будет одномерный, а его размерность будет совпадать с количеством столбцов исходной матрицы.

```
#include <iostream>
using namespace std;
int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[n];
 for (int i=0; i<n; ++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j) {cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}
int main()
{int n, m;
 int **a=creat(n,m);
 float *b=new float [m];      //создание массива результатов
 for (int j=0;j<m; ++j)       // для каждого j-го столбца
  {b[j]=0;int k=0;      //обнуляем сумму и количество нечетных элементов столбца
   for (int i=0;i<n; i++)       // перебираем все элементы j-того столбца и находим
    if (a[i][j]%2) {b[j]+=a[i][j]; k++;} //сумму и количество нечетных элементов
   if (k) //если количество нечетных элементов положительное,
    b[j]/=k;} //то вычисляем среднее арифметическое
 for (int i=0;i<m; i++)  //вывод массива результатов
  cout<<b[i]<<"\t";
 for (int i=0;i<n; i++)  delete [] a[i];//удаление массива a
 delete [] a;
 delete [] b; //удаление массива b
 return 0;}
```

Результат работы программы:

n m	Массив $A_{n\times m}$	Ответ
2 3	2 1 3	Вывод результата: 0.00 2.00 3.00
	4 3 6	

6. Дана матрица А и вектор Х соответствующих размерностей. Нечетные строки матрицы заменить элементами вектора Х.

```
#include <iostream>
using namespace std;
int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[n];
 for (int i=0; i<n; ++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j) {cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}
int main()
{cout <<"n="; cin >>n;  cout <<"m="; cin >>m;
 int **a=creat(n, m);
 int *b=new int [m]; //создаем массив b
 for (int i=0;i<m; ++i)  cin>>b[i]; //заполняем массив b
 for (int i=1;i<n; i+=2) //каждую нечетную строку массива a заменяем на массив b
  for (int j=0; j<m; ++j) a[i][j]=b[j];
 for (int i=0;i<n; ++i, cout<<endl)  //вывод измененного массива
  for (int j=0;j<m; ++j) cout<<a[i][j]<<"\t";
 for (int i=0;i<n; ++i)  delete [] a[i];
 delete [] a; delete []b;
 return 0; }
```

Результат работы программы:

n m	Массив $A_{n\times m}$	Вектор Х	Ответ
5 3	-2 1 13	0 4 7	-2 1 13
	1 -3 6		0 4 7
	3 5 1		3 5 1
	3 15 6		0 4 7
	12 4 -3		12 4 -3

7. Даны две матрицы $A_{m\times n}$ и $B_{n\times v}$ соответствующих размерностей, элементами которых являются целые числа. Найти произведение этих матриц.

Указания по решению задачи. Операция произведения двух матриц определена только для матриц соответствующих размерностей $A_{m\times n}$ и $B_{n\times v}$, то есть количество столбцов первой матрицы должно совпадать с количеством строк второй. В качестве результата операции получается матрица $C_{m\times v} = A_{m\times n} * B_{n\times v}$. Как видим, количество строк матрицы *C* совпадает с количеством строк матрицы *A*, а количество столбцов матрицы *C* равно количеству столбцов матрицы *B*. Каждый элемент искомой матрицы *C* определяется по формуле $c[i,j] = \sum_{k=1}^{n} a[i,k] * b[k,j]$, где индекс *i* принимает значения от 1 до *m*, а индекс *j* – от 1 до *v*.

```
#include <iostream>
using namespace std;
int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[n];
 for (int i=0; i<n;++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
```

```
 for (int j=0; j<m; ++j){cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}
/*функция в качестве результата возвращает произведение двух матриц или NULL,
если таков произведение вычислить невозможно*/
int **mult(int **a, int na,int ma, int **b, int nb, int mb)
{if (ma!=nb) return NULL; //если размерности матриц не совпадают, то возвращается 0
 else {int **c=new int *[na];//иначе создается матрица размерностью na строк  и
       for (int i=0; i<na;++i) c[i]=new int [mb]; // mb столбцов
       for (int i=0; i<na;++i) //выполняем подсчет произведения матриц
        for (int j=0; j<mb;++j)
         {c[i][j]=0;
          for (int k=0; k<ma;k++) c[i][j]+=a[i][k]*b[k][j];}
        return c; } //в качестве результата возвращаем матрицу c
}
//функция выводит двумерную матрицу на экран в виде  таблицы
void print (int **mas, int n, int m)
{ for (int i=0; i<n; i++, cout<<endl)
   for (int j=0; j<m; j++) cout<<mas[i][j]<<"\t";}
//функция освобождает память, выделенную под параметр массив
void deleteMas(int **mas, int n)
{for (int i=0;i<n; i++)  delete [] mas[i];
 delete [] mas; }
int main()
{int na, ma, nb,mb;
 int **a=creat(na,ma); //создаем и заполняем матрицу a
 int **b=creat(nb, mb); //создаем и заполняем матрицу b
 int **c=mult(a, na,ma, b, nb,mb); //вычисляем произведение матриц a и b
 if (c) print (c,na,mb) ;//если указатель с не пустой, то выводим на экран матрицу c
   else cout <<" к данным матрицам операция неприменима, не совпадает размерность";
 //освобождаем память, выделенную под динамические массивы
 deleteMas(a, na); deleteMas(b, nb); deleteMas(c, na);
 return 0; }
```

Результат работы программы:

na	ma	Массив $A_{na \times ma}$	nb	mb	Массив $B_{nb \times mb}$	Ответ $C_{na \times mb}$
3	2	-2 1 1 -3 3 5	2	3	0 4 7 1 -3 6	1 -11 -8 -3 13 -11 5 -3 51
3	3	-2 1 1 1 -3 3 3 5 4	2	3	0 4 7 1 -3 6	к данным матрицам операция неприменима, не совпадает размерность

Замечание. Обратите внимание на то, что использование вспомогательных функций повышает читаемость программы. Поэтому если вы видите, что один и тот же фрагмент программы будет выполняться несколько раз, то оформите его в виде функции.

6.7. Вставка и удаление элементов в массивах

При объявлении массива мы определяем его максимальную размерность, которая в дальнейшем изменена быть не может. Однако с помощью вспомогательной переменной можно контролировать текущее количество элементов, которое не может быть больше максимального.

Пример. Рассмотрим фрагмент программы:

```
int *a=new int [10];
int n=5;
for (int i=0; i<5;i++) a[i]:=i*i;
```

В этом случае массив можно представить следующим образом:

n=5	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
a	0	1	4	9	16	случайное число	случайное число	случайное число	случайное число	случайное число

Так как во время описания был определен массив из 10 элементов, а заполнено только первые 5, то оставшиеся элементы будут заполнены случайными числами.

Что значит *удалить из одномерного массива* элемент с номером 3? Удаление должно привести к физическому «уничтожению» элемента с номером 3 из массива, при этом общее количество элементов должно быть уменьшено. В этом понимании удаления элемента итоговый массив должен выглядеть следующем образом

	0	*1*	*2*	*4*	*5*	*6*	*7*	*8*	*9*	*недопустимое*
a	0	1	4	16	случайное число	случайное число	случайное число	случайное число	случайное число	*состояние*

Такое удаление для массивов *невозможно,* поскольку элементы массива располагаются в памяти последовательно друг за другом, что позволяет организовать индексный способ обращения к массиву.

Однако «удаление» можно смоделировать сдвигом элементов влево и уменьшением значения переменной, которая отвечает за текущее количество элементов в массиве, на единицу:

n=4	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
a	0	1	4	16	случайное число	случайное число	случайное число	случайное число	случайное число	случайное число

В общем случае, если мы хотим удалить элемент массива с номером *k* (всего в массиве *n* элементов, а последний элемент имеет индекс *n-1*), то нам необходимо произвести сдвиг элементов, начиная с *k+1*-го на одну позицию влево. Т.е. на *k*-ое место поставить *k+1*-й элемент, на место *k+1* – *k+2*-й элемент, ..., на место *n-2* – *n-1*-й элемент. После чего значение *n* уменьшить на 1. В этом случае размерность массива не изменится, изменится лишь текущее количество элементов, и у нас создастся ощущение, что элемент с номером *k* удален. Рассмотрим данный алгоритм на примере:

```
#include <iostream>
using namespace std;
int main()
{int n; cout<<"n="; cin>>n; //ввели размерность массива
 int *a=new int [n]; //создали и заполнили массив
 for (int i=0; i<n;i++) {cout<<"a["<<i<<"]="; cin>>a[i];}
 int k; cout<<"k="; cin>>k; //ввели номер удаляемого элемента
 if (k<0 || k>n-1) // если номер отрицательный или больше номера последнего элемента,
  cout<<"error"; //то выводим сообщение об ошибке, иначе
 else {for (int i=k; i<n-1;i++) a[i]=a[i+1]; //выполняем сдвиг элементов массива
       n--; //уменьшаем текущую размерность массива
       for (int i=0; i<n;i++) cout<<a[i]<<"\t";} //выводим измененный массив на экран
return 0;}
```

Рассмотрим теперь операцию *удаления в двумерном массиве*. Размерность двумерного массива также зафиксирована на этапе объявления массива. Однако при необходимости можно «смоделировать» удаление целой строки в массиве, выполняя сдвиг всех строк, начиная с k-ой на единицу вверх. В этом случае размерность массива не изменится, а текущее количество строк будет уменьшено на единицу. В качестве примера удалим из двумерного массива, строку с номером k.

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

//функция создает и замолняем двумерный массив
int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[n];
 for (int i=0; i<n;++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j){cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}

//функция выводит двумерную матрицу на экран в виде  таблицы
void print (int **mas, int n, int m)
{ for (int i=0; i<n; i++, cout<<endl)
   for (int j=0; j<m; j++) cout<<mas[i][j]<<"\t";}

//функция освобождает память, выделенную под параметр массив
void deleteMas(int **mas, int n)
{for (int i=0;i<n; i++)  delete [] mas[i];
 delete [] mas; }

int main()
{int n, m, k;
 int **a=creat(n,m);
 print(a,n,m); //выводим первоначальный массив
 cout<<"k="; cin>>k; //вводим номер строки для удаления
 if (k<0 || k>n-1) cout<<"error"; /*если номер строки отрицательный или больше номера
  последнего элемента, то выводим сообщение об ошибке*/
 else {delete [] a[k]; //освобождаем память, выделенную для k-ой строки массива
  for (int i=k; i<n-1; ++i)a[i]=a[i+1]; //передресуем указатели, начиная с k-ой строки
  a[n-1]=NULL; //указатель на n-1 строку обнуляем
  --n; //уменьшаем текущее количество строк
  print(a,n,m);  //выводим измененный массив
 }
 deleteMas(a, n); //освобождаем память
 return 0; }
```

Результат работы программы:	n m k	Исходный массив $A_{4\times4}$	Измененный массив $A_{3\times4}$
	4 4 1	0 0 0 0	0 0 0 0
		1 1 1 1	2 2 2 2
		2 2 2 2	3 3 3 3
		3 3 3 3	

Аналогичным образом можно удалить *k*-ый столбец в двумерном массиве. Однако, т.к. у нас нет указателей на столбцы массива, сдвиг столбцов нужно будет проводить поэлементно.

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

```
int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[n];
 for (int i=0; i<n;++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j){cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}
void print (int **mas, int n, int m)
{ for (int i=0; i<n; i++, cout<<endl)
   for (int j=0; j<m; j++) cout<<mas[i][j]<<"\t";}
void deleteMas(int **mas, int n)
{for (int i=0;i<n; i++)  delete [] mas[i];
 delete [] mas; }
int main()
{int n, m, k;
 int **a=creat(n,m); //создаем и заполняем матрицу a
 print(a,n,m); //выводим первоначальный массив
 cout<<"k="; cin>>k; //вводим номер столбца для удаления
 if (k<0 || k>m-1) cout<<"error";
 else {for (int j=k; j<m-1; ++j) //выполняем поэлементный сдвиг столбцов
        for( int i=0; i<n; ++i)a[i][j]=a[i][j+1];
       --m; //уменьшаем текущее количество столбцов в массиве
       print(a,n,m); } //выводим измененный массив
 deleteMas(a, n);
 return 0; }
```

Результат работы программы:	n m k	Исходный массив $A_{4\times4}$	Измененный массив $A_{4\times3}$
	4 4 1	0 1 2 3	0 2 3
		0 1 2 3	0 2 3
		0 1 2 3	0 2 3
		0 1 2 3	0 2 3

Вернемся к массиву, определенному в самом первом примере. И подумаем теперь, что значит *добавить элемент в одномерный массив* в позицию с номером k? В этом случае все элементы, начиная с k-ого, должны быть сдвинуты вправо на одну позицию. Однако сдвиг нужно начинать с конца, т.е. на первом шаге на n-е место поставить n-1-ый элемент, потом на n-1-ое место поставить n-2-й элемент, …, наконец, на k+1 место вставить k-й элемент. Таким образом, копия k-го элемента будет на k+1-м месте и на k-е место можно поставить новый элемент. Затем необходимо увеличить текущее количество элементов на 1.

Рассмотрим массив из примера 1 и в качестве k зададим значение равное 3. В этом случае массив будет выглядеть следующим образом:

k=3	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
a	0	1	4	9	9	16	случайное число	случайное число	случайное число	случайное число

Теперь в позицию с номером 3 можно поместить новое значение. А текущее количество элементов в массиве становится равным 6. Подумайте, почему сдвиг нужно выполнять с конца массива, а не с начала, как мы это делали в случае удаления элемента из массива.

Рассмотрим программную реализацию данного алгоритма:

```
#include <iostream>
```

```
using namespace std;
int main()
{int n; cout<<"n="; cin>>n;          //ввели  текущую размерность массива
 int m=2*n;                          //определили максимальную размерность массива
 int *a=new int [m];  //создали и заполнили массив
 for (int i=0; i<n;i++) {cout<<"a["<<i<<"]="; cin>>a[i];}
 // ввели номер элемента, в позицию которого будем производить вставку
 int k; cout<<"k="; cin>>k;
 if (k<0 || k>n-1 || n+1>m) cout<<"error"; /*если введенный номер отрицательный или
больше номера последнего значимого элемента в массиве, или после вставки размер-
ность  массива станет больше допустимой, то выдаем сообщение об ошибке, иначе */
  else { int x; cout<<"x="; cin>>x; //вводим значение для вставки
         for (int i=n; i>k;i--) a[i]=a[i-1]; //выполняем сдвиг в массиве
          a[k]=x; //записываем в позицию k значение x
          n++; //увеличиваем текущую размерность массива
          for (int i=0; i<n;i++) cout<<a[i]<<"\t";} //выводим измененный массив
 return 0;
}
```

Теперь рассмотрим *добавление столбца в двумерный массив*. Для этого все столбцы после столбца с номером k передвигаем на 1 столбец вправо. Затем увеличиваем количество столбцов на 1. После этого копия столбца с номером k будет находиться в столбце с номером k+1. И, следовательно, k-ый столбец можно заполнить новыми значениями. Рассмотрим программную реализацию алгоритма:

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

int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[n];
 for (int i=0; i<n;++i) mas[i]=new int [2*m];//определили максимальную количество столбцов
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j){cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}

void print (int **mas, int n, int m)
{ for (int i=0; i<n; i++, cout<<endl)
   for (int j=0; j<m; j++) cout<<mas[i][j]<<"\t";}

void deleteMas(int **mas, int n)
{for (int i=0;i<n; i++)  delete [] mas[i];
 delete [] mas; }

int main()
{int n, m, k;
 int **a=creat(n,m); //создаем и заполняем матрицу a
 int m2=2*m; //m- текущее количество столбцов, m2 –максимально возможное
 print(a,n,m); //выводим первоначальный массив
 cout<<"k="; cin>>k; //вводим номер столбца для добавления
 if (k<0 || k>m-1 ||m+1>m2) cout<<"error"; /*если введенный номер столбца
  отрицательный или больше номера последнего значимого столбца в массиве, или
  после  вставки столбца размерность  массива станет больше допустимой, то
  выдаем сообщение об ошибке, */
  else {for (int j=m; j>k; --j) // иначе выполняем позлементный сдвиг столбцов
       for( int i=0; i<n; ++i)a[i][j]=a[i][j-1];
      ++m; //увеличиваем текущее количество столбцов в массиве
```

```
    for (int i=0; i<n; ++i) //вводим новые данные в k-ый столбец
      {cout <<"a["<<i<<"]["<<k<<"]="; cin>>a[i][k];}
    print(a,n,m);} //выводим измененный массив
deleteMas(a, n);
return 0; }
```

Результат работы программы:	n m k	Исходный массив $A_{4\times4}$	Содержимое нового столбца	Измененный массив $A_{4\times5}$
	4 4 1	0 1 2 3	9	0 9 1 2 3
		0 1 2 3	9	0 9 1 2 3
		0 1 2 3	9	0 9 1 2 3
		0 1 2 3	9	0 9 1 2 3

Аналогичным образом можно провести вставку новой строки в двумерный массив. Однако, если вспомнить, что двумерный массив – это одномерный массив указателей на одномерные массивы, например, целых чисел, то поэлементный сдвиг строк можно заменить на сдвиг указателей.

```
#include <iostream>
using namespace std;
int ** creat(int &n, int &m)
{cout <<"n="; cin >>n; cout <<"m="; cin >>m;
 int **mas=new int *[2*n]; // определяем максимальное количество строк
 for (int i=0; i<n;++i) mas[i]=new int [m];
 for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j){cout<<"mas["<<i<<"]["<<j<<"]="; cin>>mas[i][j];}
 return mas;}
void print (int **mas, int n, int m)
{ for (int i=0; i<n; i++, cout<<endl)
   for (int j=0; j<m; j++) cout<<mas[i][j]<<"\t";}
void deleteMas(int **mas, int n)
{for (int i=0;i<n; i++)  delete [] mas[i];
 delete [] mas; }
int main()
{int n, m, k;
 int **a=creat(n,m);  //создаем и заполняем матрицу a
 int n2=2*n;          //n- текущее количество строк, n2 –максимально возможное
 print(a,n,m);        //выводим первоначальный массив
 cout<<"k="; cin>>k; //вводим номер строки для добавления
 if (k<0 || k>n-1 ||n+1>n2) cout<<"error";
 else {for (int i=n; i>k; --i)  a[i]=a[i-1]; //выполняем сдвиг строк
       ++n; //увеличиваем текущее количество строк в массиве
       a[k]=new int [m]; //выделяем память под новую строку массива и заполняем ее
       for (int j=0; j<m; ++j)  {cout <<"a["<<k<<"]["<<j<<"]="; cin>>a[k][j];}
     print(a,n,m); } //выводим измененный массив
 deleteMas(a, n);
 return 0; }
```

Результат работы программы:	n m k	Исходный массив $A_{4\times4}$	Содержание новой строки	Измененный массив $A_{5\times4}$
	4 4 1	0 0 0 0	9 9 9 9	0 0 0 0
		1 1 1 1		9 9 9 9
		2 2 2 2		1 1 1 1
		3 3 3 3		2 2 2 2
				3 3 3 3

6.8. Упражнения

I. Дана последовательность целых чисел.

Замечание. Задачи из данного пункта решить двумя способами, используя одномерный массив, а затем двумерный.

1. Заменить все положительные элементы противоположными им числами.
2. Заменить все элементы, меньшие заданного числа, этим числом.
3. Заменить все элементы, попадающие в интервал [a, b], нулем.
4. Заменить все отрицательные элементы, не кратные 3, противоположными им числами.
5. Все элементы, меньшие заданного числа, увеличить в два раза.
6. Подсчитать среднее арифметическое элементов.
7. Подсчитать среднее арифметическое отрицательных элементов.
8. Подсчитать количество нечетных элементов.
9. Подсчитать сумму элементов, попадающих в заданный интервал.
10. Подсчитать сумму элементов, кратных 9.
11. Подсчитать количество элементов, не попадающих в заданный интервал.
12. Подсчитать сумму квадратов четных элементов.
13. Вывести на экран номера всех элементов больших заданного числа.
14. Вывести на экран номера всех нечетных элементов.
15. Вывести на экран номера всех элементов, которые не делятся на 7.
16. Вывести на экран номера всех элементов, не попадающих в заданный интервал.
17. Определить, является ли произведение элементов трехзначным числом.
18. Определить, является ли сумма элементов двухзначным числом.
19. Вывести на экран элементы с четными индексами (для двумерного массива – сумма индексов должны быть четной).
20. Вывести на экран положительные элементы с нечетными индексами (для двумерного массива – первый индекс должен быть нечетным).

II. Дана последовательность из n действительных чисел.

Замечание. Задачи из данного пункта решить, используя одномерный массив.

1. Подсчитать количество максимальных элементов.
2. Вывести на экран номера всех минимальных элементов.
3. Заменить все максимальные элементы нулями.
4. Заменить все минимальные элементы на противоположные.
5. Поменять местами максимальный элемент и первый.
6. Вывести на экран номера всех элементов, не совпадающих с максимальным.
7. Найти номер первого минимального элемента.
8. Найти номер последнего максимального элемента.

9. Подсчитать сумму элементов, расположенных между максимальным и минимальным элементами (минимальный и максимальный элементы в массиве единственные). Если максимальный элемент встречается позже минимального, то выдать сообщение об этом.
10. Найти номер первого максимального элемента.
11. Найти номер последнего минимального элемента.
12. Подсчитать сумму элементов, расположенных между первым максимальным и последним минимальным элементами. Если максимальный элемент встречается позже минимального, то выдать сообщение об этом.
13. Поменять местами первый минимальный и последний максимальный элементы.
14. Найти максимум из отрицательных элементов.
15. Найти минимум из положительных элементов.
16. Найти максимум из модулей элементов.
17. Найти количество пар соседних элементов, разность между которыми равна заданному числу.
18. Подсчитать количество элементов, значения которых больше значения предыдущего элемента.
19. Найти количество пар соседних элементов, в которых предыдущий элемент кратен последующему.
20. Найти количество пар соседних элементов, в которых предыдущий элемент меньше последующего.

III. Дан массив размером $n \times n$ (если не оговорено иначе), элементы которого целые числа.

1. Подсчитать среднее арифметическое нечетных элементов, расположенных выше главной диагонали.
2. Подсчитать среднее арифметическое четных элементов, расположенных ниже главной диагонали.
3. Подсчитать сумму элементов, расположенных на побочной диагонали.
4. Подсчитать среднее арифметическое ненулевых элементов, расположенных над побочной диагональю.
5. Подсчитать среднее арифметическое элементов, расположенных под побочной диагональю.
6. Поменять местами столбцы по правилу: первый с последним, второй с предпоследним и т.д.
7. Поменять местами две средних строки, если количество строк четное, и первую со средней строкой, если количество строк нечетное.
8. Поменять местами два средних столбца, если количество столбцов четное, и первый со средним столбцом, если количество столбцов нечетное.
9. Если количество строк в массиве четное, то поменять строки местами по правилу: первую строку со второй, третью – с четвертой и т.д. Если количество строк в массиве нечетное, то оставить массив без изменений.
10. Если количество столбцов в массиве четное, то поменять столбцы местами по правилу: первый столбец со вторым, третий – с четвертым и т.д. Если количество столбцов в массиве нечетное, то оставить массив без изменений.
11. Вычислить A^n, где n – натуральное число.

12. Подсчитать норму матрицы по формуле $\|A\| = \sum_i \max_j a_{i,j}$.

13. Подсчитать норму матрицы по формуле $\|A\| = \sum_j \max_i a_{i,j}$.

14. Вывести элементы матрицы в следующем порядке: 

15. Выяснить, является ли матрица симметричной относительно главной диагонали.

16. Заполнить матрицу числами от 1 до n (где n=m×k, а m – количесто строк, а k – количество столбцов прямоугольной матрицы) следующим образом:

17. Определить, есть ли в данном массиве строка, состоящая только из положительных элементов.

18. Определить, есть ли в данном массиве столбец, состоящий только из отрицательных элементов.

19. В каждой строке найти максимум и заменить его на противоположный элемент.

20. В каждом столбце найти минимум и заменить его нулем.

IV. Дан массив размером n×n, элементы которого целые числа.

1. Найти максимальный элемент в каждой строке и записать данные в новый массив.
2. Найти минимальный элемент в каждом столбце и записать данные в новый массив.
3. Четные столбцы таблицы заменить на вектор X.
4. Нечетные строки таблицы заменить на вектор X.
5. Вычислить A*X, где A – двумерная матрица, X – вектор.
6. Для каждой строки подсчитать количество положительных элементов и записать данные в новый массив.
7. Для каждого столбца подсчитать сумму отрицательных элементов и записать данные в новый массив.
8. Для каждого столбца подсчитать сумму четных положительных элементов и записать данные в новый массив.
9. Для каждой строки подсчитать количество элементов, больших заданного числа, и записать данные в новый массив.
10. Для каждого столбца найти первый положительный элемент и записать данные в новый массив.
11. Для каждой строки найти последний четный элемент и записать данные в новый массив.
12. Для каждого столбца найти номер последнего нечетного элемента и записать данные в новый массив.
13. Для каждой строки найти номер первого отрицательного элемента и записать данные в новый массив.
14. Для каждой строки найти сумму элементов с номерами от k1 до k2 и записать данные в новый массив.
15. Для каждого столбца найти произведение элементов с номерами от k1 до k2 и записать данные в новый массив.

16. Для каждой строки подсчитать сумму элементов, не попадающих в заданный интервал, и записать данные в новый массив.
17. Подсчитать сумму элементов каждой строки и записать данные в новый массив. Найти максимальный элемент нового массива.
18. Подсчитать произведение элементов каждого столбца и записать данные в новый массив. Найти минимальный элемент нового массива.
19. Для каждой строки найти номер первой пары неравных элементов. Данные записать в новый массив.
20. Для каждого столбца найти номер первой пары одинаковых элементов. Данные записать в новый массив.

V. В одномерном массиве, элементы которого – целые числа, произвести следующие действия:

1. Удалить из массива все четные числа.
2. Удалить из массива все максимальные элементы.
3. Удалить из массива все числа, значения которых попадают в данный интервал.
4. Удалить из массива все элементы, последняя цифра которых равна данной.
5. Удалить из массива элементы с номера k1 по номер k2.
6. Вставить новый элемент перед первым отрицательным элементом.
7. Вставить новый элемент после последнего положительного.
8. Вставить новый элемент перед всеми четными элементами.
9. Вставить новый элемент после всех элементов, которые заканчиваются на данную цифру.
10. Вставить новый элемент после всех элементов, кратных своему номеру.
11. Удалить из массива все элементы, в записи которых все цифры различны.
12. Удалить из массива повторяющиеся элементы, оставив только их первые вхождения.
13. Вставить новый элемент после всех максимальных.
14. Вставить новый элемент перед всеми элементами, в записи которых есть данная цифра.
15. Вставить новый элемент между всеми парами элементов, имеющими разные знаки.

VI. В двумерном массиве, элементы которого – целые числа, произвести следующие действия:

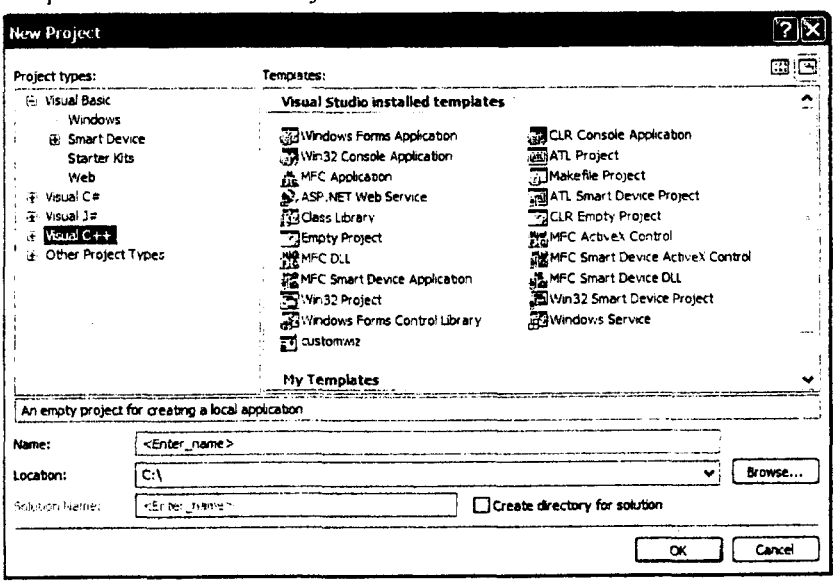
1. Вставить новую строку после строки, в которой находится первый встреченный минимальный элемент.
2. Вставить новый столбец после столбца, в котором нет ни одного отрицательного элемента.
3. Вставить новую строку после всех строк, в которых нет ни одного четного элемента.
4. Вставить новый столбец перед всеми столбцами, в которых встречается заданное число.
5. Вставить строку из нулей после всех строк, в которых нет ни одного нуля.
6. Удалить все строки, в которых нет ни одного четного элемента.
7. Удалить все столбцы, в которых первый элемент больше последнего.
8. Удалить все строки, в которых сумма элементов не превышает заданного числа.

9. Удалить все столбцы, в которых четное количество нечетных элементов.
10. Удалить все строки, в которых каждый элемент попадает в заданный интервал.
11. Удалить все столбцы, в которых все элементы положительны.
12. Удалить все строки, в которых среднее арифметическое элементов является двузначным числом.
13. Удалить строку и столбец, на пересечении которых стоит минимальный элемент (минимальный элемент встречается в массиве только одни раз).
14. Удалить из массива k-тую строку и j-тый столбец, если их значения совпадают.
15. Уплотнить массив, удалив из него все нулевые строки и столбцы.

ПРИЛОЖЕНИЕ 1.
РАБОТА В СРЕДЕ MICROSOFT VISUAL STUDIO

Для разработки, отладки и тестирования программы на C++ могут использоваться различные среды программирования. В данном приложении мы рассмотрим основные приемы работы в среде Microsoft Visual Studio, которые позволят вам создавать и запускать консольные приложения.

1. Запустите среду;
2. В среде выполните команду *File – New Project*
3. В открывшимся окне *New Project*:

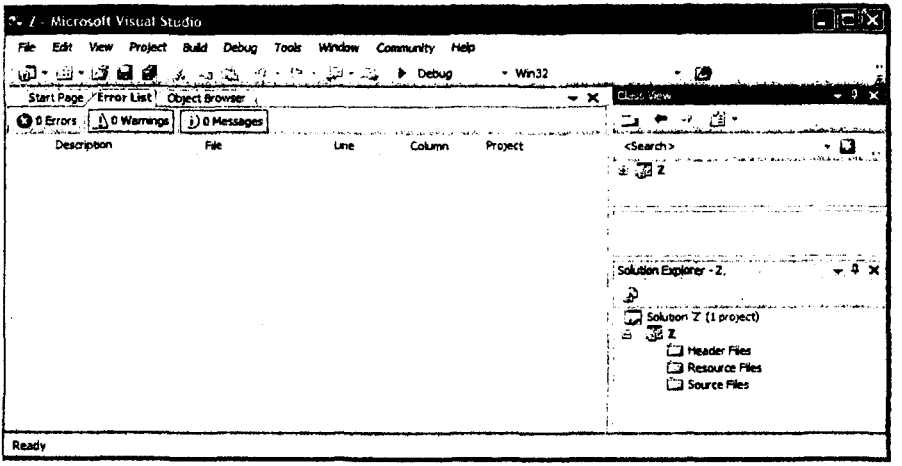


- в разделе *Project types* выберите тип *Visual C++;*
- в разделе *Templates* выберите *Empty Project*
- в разделе *Name* укажите имя папки, в которую будут сохраняться все файлы проекта (мы в качестве примера введем имя *z*);
- в разделе *Location* укажите местоположение папки на диске;

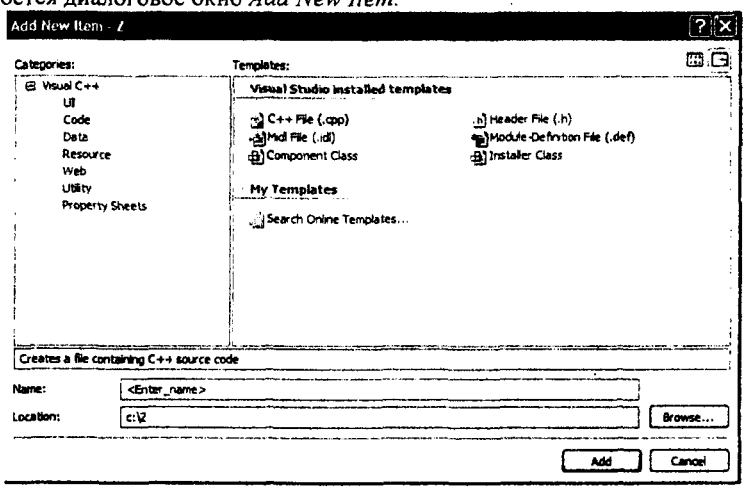
- проследите, чтобы был снят флаг в разделе *Create directory for solution.*

После чего нажмите кнопку *OK*.

4. Откроется окно проекта:



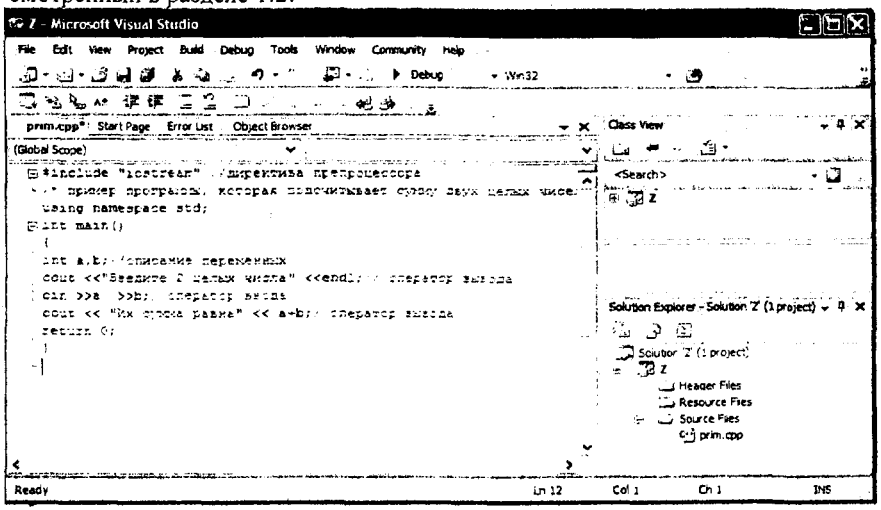
5. Во вкладке *Solution Explorer* щелкните правой кнопкой мыши на имени проекта, в нашем случае на *z*.
6. Откроется вспомогательное меню, в котором нужно выполнить последовательность действий *Add – New Item*.
7. Откроется диалоговое окно *Add New Item*:



- в разделе *Categories* выберите *Code*;
- в разделе *Templates* выберите *C++File (.cpp)*
- в разделе *Name* укажите имя файла (мы введем название *prim*)

После чего необходимо нажать кнопку *Add*.

8. Откроется окно файла *prim.cpp*, куда мы с вами и введем код программы, рассмотренный в разделе 1.2:



9. Для компиляции кода щелкните правой кнопкой по имени файла во вкладке *Solution Explorer*, в открывшемся вспомогательном меню выберите команду *Compile*.
10. Для запуска программы из среды Visual Studio нажмите *CTRL+F5*.

Замечания:

1) на Visual C++ директива препроцессора помещается в кавычках;
2) если во время компиляции будут обнаружены ошибки, то информация о них будет отображена на вкладке *Error List*;
3) после компиляции в папке проекта создастся папка *Debug*, в которой будет находиться *exe*-файл проекта, готовый к использованию;
4) самостоятельно изучите другие возможности *Visual Studio* с помощью справочной системы.

ПРИЛОЖЕНИЕ 2.
ОШИБКИ, ВОЗНИКАЮЩИЕ ПРИ РАЗРАБОТКЕ ПРОГРАММ

Отладка является непременным этапом при создании любой программы, так как при ее написании обычно допускаются различные ошибки. Ошибки могут быть трех типов: синтаксические, семантические и логические.

Синтаксические ошибки возникают в результате нарушения правил написания команд, и выявляются на этапе компиляции. При выявлении ошибки компилятор создает сообщение о ее характере, а курсор указывает место в тексте, где эта ошибка обнаружена. В Visual Studio ошибки выводятся на отдельной вкладке *Error List*. Наиболее часто встречающиеся синтаксические ошибки приведены в следующей таблице:

Сообщение об ошибке	*Причины возникновения*
indeclared identifier – (необъявленный идентификатор)	1) Использование не объявленного идентификатора. 2) Не указан тип идентификатора.

	3) Ошибочное написание служебного слова.
redefinition (повторное описание)	Повторное описание одного и того же идентификатора (переменной, константы, функции).
syntax error: missing ... before ... (синтаксическая ошибка: пропущен ... перед ...)	1) Пропущен апостроф при формировании символьной константы. 2) Пропущены кавычки при формировании строковой константы. 3) Пропущена точка с запятой в конце оператора. И т.д.
cannot convert from ... to .. (невозможно выполнить неявное преобразование из типа ... в тип ...)	1) Несовместимы типы переменной и выражения в операторе присваивания. 2) Несовместимы типы фактического и формального параметров при обращении к функции. 3) Тип выражения несовместим с типом индекса при индексировании массива. 4) Несовместимы типы операндов в выражении.
end of file fount befor the left brace ... (конец файла должен находиться слева от позиции ...)	Возможно в вашей программе несбалансированны фигурные скобки.
... : illegal. left/right operand has type .. (для операции ... недопустим тип ... левого/правого операнда)	Данная операция не может быть применена к операндам указанного типа, например, 10.7 % 2.

Замечание. Более подробно со списком ошибок можно познакомится в справочной системе и технической документации по C++.

Семантические ошибки возникают в результате использования недопустимых значений параметров или выполнения недопустимых действий над параметрами. Например, при выходе за границу массива, введении недопустимых значений. Выявляются эти ошибки на этапе выполнения программы, о чем тзкже выдается сообщение, например: ***Z.exe - обнаружена ошибка. Приложение будет закрыто. Приносим извинения за неудобства***. После этого сообщения программа завершает свою работу.

Замечание. Программные средства обработки семантических ошибок (или обработка исключений) будут рассмотрены в следующей части пособия.

Логические ошибки возникают в связи с ошибочной реализацией алгоритма. Самые простые примеры логических ошибок: вместо операции + указали операцию –, вместо операции сравнения == указали операцию присваивания =. Эти ошибки не приводят к прерыванию программы, но при этом выдаются неверные результаты. Отладка таких ошибок зависит от качества тестовых примеров, составленных вами.

Мы рекомендуем к каждой задаче составить набор тестов, в которых указаны входные значения и определены выходные значения, просчитанные аналитически (без участия программы). Количество тестовых примеров может быть различным в зависимости от сложности программы. Чем сложнее задача, тем больше разрабатывается тестов.

ПРИЛОЖЕНИЕ 3.
ОПЕРАЦИИ ЯЗЫКА С++

Операции приведены в порядке убывания приоритетов. Операции с разными приоритетами разделены чертой.

Операция	Краткое описание
Унарные операции	
::	доступ к области видимости
.	выбор
–>	выбор
[ ]	индексация
( )	вызов функции
<тип>( )	конструирование
++	постфиксный инкремент
--	постфиксный декремент
typied	идентификация типа
dynamic_cast	преобразование типа с проверкой на этапе выполнения
static_cast	преобразование типа с проверкой на этапе компиляции
reinterpret_cast	преобразование типа без проверки
const_cast	константное преобразование типа
sizeof	размер объекта или типа
++	префиксный инкремент
--	префиксный декремент
~	поразрядное отрицание
!	логическое отрицание
+	унарный плюс
-	унарный минус
&	взятие адреса
*	разадресация
new	выделение памяти
delete	освобождение памяти
(<тип>)	преобразование типа
.*	выбор
–>*	выбор
Бинарные и тернарные операции	
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево или извлечение из потока
>>	сдвиг вправо или помещение в поток
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
==	равно
!=	не равно
&	поразрядное И
^	поразрядное исключающее ИЛИ
|	поразрядное ИЛИ
&&	логическое И

||	логическое ИЛИ
? :	условная операция (тернарная)
=	присваивание
*=	присваивание с умножением
/=	присваивание с делением
%=	остаток от деления с присваиванием
+=	присваивание со сложением
-=	присваивание с вычитанием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное И с присваиванием
^=	поразрядное исключающее ИЛИ с присваиванием
|=	поразрядное ИЛИ с присваиванием
throw	исключение
,	последовательное вычисление

ПРИЛОЖЕНИЕ 4.
МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

C++ содержит большое количество встроенных математических функций, описание которых содержится в заголовочном файле *math* или *cmath* (в зависимости от версии компилятора). Рассмотрим краткое описание некоторых математических функций. Особое внимание следует обратить на типы операндов и результатов, т.к. каждая функция имеет несколько перегруженных версий.

Замечание. Использование нескольких функций с одним и тем же именем, но с различными типами параметров, называется перегрузкой функции. Например, функция pow имеет 7 перегруженных версий: *pow(double x, double y), pow(double x, int y), pow(float x, float y)* и т.д.

Название	Описание
abs(x)	Модуль аргумента, x - целое
acos(x)	Арккосинус (угол в радианах)
asin(x)	Арксинус (угол в радианах)
atan(x)	Арктангенс (угол в радианах)
atan(x,y)	Арктангенс отношения параметра x/y в радианах
ceil(x)	Ближайшее большее целое, >= x
cos(x)	Косинус (угол в радианах)
exp(x)	Экспонента e^x
fabs(x)	Модуль аргумента, x - вещественное
floor(x)	Ближайшее меньшее целое, <= x
log(x)	Логарифм натуральный
log10(x)	Логарифм десятичный
pow (x, y)	Значение x в степени y
sin(x)	Синус (угол в радианах)
sqrt(x)	Корень квадратный аргумента
tan(x)	Тангенс (угол в радианах)

ЛИТЕРАТУРА

1. International Standart ISO/IEC 14882:2003(E), Programming languages – C++.
2. *Климова Л.М.* Си++. Практическое программирование. Решение типовых задач. М.:КУДИЦ-ОБРАЗ, 2001.
3. *Лафоре Р.* Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е изд. – СПб.: Питер, 2007.
4. *Павловская Т.А.* C/C++. Программирование на языке высокого уровня. – СПб: Питер, 2006.
5. *С.Б.Липпман, Ж. Лажойе,* Язык программирования C++. Вводный курс. 4-е изд-е. Изд. дом "Вильямс", 2007 г.
6. *Страуструп Бьерн.* Язык программирования C++. \ Пер. с англ., 3-е изд. – СПб.; М.: «Невский диалект» - «Издательство БИНОМ», 1999.
7. *Шилдт Г.* Самоучитель C++: Пер. с англ. — 3-е изд. — СПб.: БХВ-Петербург, 2005.

СОДЕРЖАНИЕ