ДУМАЙ ЯРКО!

Издательский Дом
НИТУ «МИСиС»

store.misis.ru

№ 4579

МИСиС
Национальный исследовательский
технологический университет

О.И. Ремизова

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ (C++)

Методические указания

№ 4579 МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСиС»
ЦЕНТР ДОВУЗОВСКОЙ ПОДГОТОВКИ И ОРГАНИЗАЦИИ ПРИЕМА
Проект «Инженерный класс в московской школе»

О.И. Ремизова

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ (C++)

Методические указания

Рекомендовано редакционно-издательским советом университета

МИСиС

Москва 2021

УДК 004.4
Р38

Рецензент
канд. техн. наук, доц. *Д.В. Калитин*

Ремизова, Ольга Игоревна.
Р38 Алгоритмизация и программирование (C++): метод. указания / О.И. Ремизова. – Москва : Издательский Дом НИТУ «МИСиС», 2021. – 76 с.

В методических указаниях рассматриваются вопросы разработки программ для решения разнообразных задач с использованием языка программирования C++. Рассмотрены основные типы данных и алгоритмические конструкции, показано, как создавать простые программы из этих конструкций. Никаких предварительных знаний в программировании для усвоения материала не требуется.

Предназначены для учащихся московских школ в рамках городского образовательного проекта «Инженерный класс в московской школе».

УДК 004.4

Содержание

Введение

Зачем изучать программирование?

Во-первых, это очень интересно.

Во-вторых, владение программированием здорово облегчает жизнь специалистам многих профессий. Математик, физик, химик, биолог, социолог, экономист, лингвист и любой другой специалист может взять готовую библиотеку функций, написать на ее основе несложную программу и быстро решить свою задачу. И пока коллеги ведут подсчеты в Excel или на калькуляторе, специалист с навыками программирования уже будет отдыхать или думать над новой проблемой.

Ну и в-третьих, изучив программирование, можно стать программистом. Работать программистом не только приятно и прибыльно, но и полезно для общества. Компьютерные программы используются, например, для создания лекарств, для фундаментальных исследований.

Как изучать программирование?

Самое сложное в нашей области – это первые шаги, и в программировании начинать лучше с практики. Именно ей и будет посвящен наш курс. Мы напишем сотни несложных программ, не углубляясь в теорию. С таким опытом вы сможете потом самостоятельно справиться и с более сложными задачами.

Чему учит курс?

Мы будем изучать практические основы языка C++. Научимся работать с числами, их последовательностями, таблицами, а также освоим несложные структуры данных и алгоритмы: сортировку, ассоциативные массивы, множества.

Почему C++?

Это один из самых популярных языков программирования. Он хорош своей быстротой и универсальностью. С языка C++ легко перейти на любой другой, так как все они в целом очень похожи. Можно заняться чистым C, чтобы писать операционные системы, драйверы и распределенные программы. Можно перейти на Java или C# и писать софт для организаций. Или

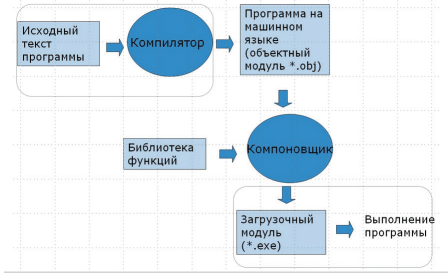
можно выучить немного другой вариант Java или Objective-C и писать приложения для Android и iPhone. В любом случае после C++ осваивать новые языки вам будет гораздо легче.

Интегрированная среда разработки (IDE). Язык C++ является языком высокого уровня. Это означает, что программа пишется на языке, приближенном к естественному человеческому языку, а затем автоматически переводится на язык машинных кодов. Перевод с языка высокого уровня на машинный язык выполняет ***компилятор***. В результате получается файл с расширением *.obj*, который можно сохранить на диске.

В программе, как правило, используются различные компоненты из библиотек. Большие программы состоят из нескольких исходных файлов, и каждый такой файл компилируется в отдельный объектный модуль. Объединить все это в одну программу может компоновщик. Результат работы компоновщика – это исполняемая программа (например, файл с расширением *.exe*, который иначе называется загрузочным модулем). Такой файл уже может выполняться автономно, без участия исходной программы и компилятора.

В принципе, можно использовать обычный текстовый редактор для ввода кода программы, а затем – отдельные программы компилятора и компоновщика, вызвав их из командной строки операционной системы. Но гораздо удобнее использовать специальное приложение, в котором объединены все инструменты для разработки и отладки программ.

Интегрированная среда разработки (IDE – Integrated Development Environment) – это специальное приложение, которое позволяет упростить разработку программ на языке высокого уровня (рисунок).



Интегрированная среда разработки

Она обычно включает:

- текстовый редактор – чтобы набрать текст программы, сохранить его на диске, редактировать. Такой редактор обычно еще и выделяет элементы кода разными цветами, подсказывает возможные ошибки синтаксиса, а иногда и создает фрагменты кода автоматически;
- компилятор и компоновщик – чтобы перевести программу в машинный код;
- средства отладки и запуска программ – чтобы обеспечить удобство поиска ошибок;
- стандартные библиотеки, содержащие многократно используемые элементы программ;
- справочную систему и др.

Для языка C++ существует достаточно много различных сред разработки. Мы будем использовать Microsoft Visual Studio. Эту систему можно установить на компьютеры с операционной системой (ОС) Windows. Для других ОС можно использовать, например, Code::Blocks или CLion. Они обладают похожим функционалом.

1 Переменные и типы данных

1.1 Основные понятия языка

В тексте на любом естественном языке можно выделить четыре основных элемента: символы, слова, словосочетания и предложения. Подобные элементы содержит и язык программирования, только слова называют лексемами, словосочетания – выражениями, а предложения – операторами.

Все тексты на языке пишутся с помощью его алфавита. ***Алфавит*** языка C++ включает:

- прописные и строчные латинские буквы, а также символ подчеркивания, который употребляется наряду с буквами;
- арабские цифры от 0 до 9;
- специальные символы: , . ; : ? ! ‘ « | / \ ~ ^ ( ) { } [ ] < > # % & – = + *
- пробельные символы: пробел, символы табуляции и перевода строки.

Алфавит языка в стандарте называется базовым набором символов.

Обратите внимание, что символы национальных алфавитов (в частности, русские) не входят в базовый набор символов по стандарту! Именно поэтому не рекомендуется использовать русские буквы при задании имен в программе (хотя Visual Studio это позволяет).

Кроме того, существует понятие «набор символов реализации» – все множество символов, доступных на данном компьютере. Этот набор содержит базовый набор в качестве подмножества. Из символов базового набора составляются лексемы языка и директивы препроцессора. Символы из набора реализации используются для написания комментариев.

Лексемы языка программирования аналогичны словам естественного языка. Это минимальные единицы языка, которые компилятор отличает и обрабатывает как единое целое. Например, лексемами являются константа 128 (но не ее часть 12), имя Vasia, но не часть этого имени. Существуют следующие виды лексем:

- имена (идентификаторы);
- ключевые слова;
- знаки операций;

- разделители;
- литералы (константы).

Идентификатор – это имя программного объекта (переменной, функции, константы и т.д.). Правила составления идентификаторов следующие:

- в идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания;
- первым символом идентификатора должна быть буква (со знака подчеркивания обычно начинаются служебные идентификаторы);
- прописные и строчные буквы различаются;
- идентификатор не должен совпадать с ключевыми словами и именами используемых стандартных объектов языка;
- длина идентификатора по стандарту не ограничена.

Не жалейте времени на придумывание подходящих имен. Имя должно отражать смысл переменной или функции, отвечать принятой нотации (правилам составления имен), и желательно, чтобы оно не содержало символов, которые можно перепутать друг с другом, например 1, l и I (единица, строчная L и прописная i). Примеры удачных идентификаторов: *ageOfStudents, sumOfPositive, backColor*.

Ключевые слова – это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в одном определенном смысле, например do, int, void и т.д. В Visual Studio ключевые слова выделяются синим цветом.

Список ключевых слов C++ приведен в приложении 1.

Знак операции – это один или более символов, определяющих действие над операндами.

Операнд (англ. *operand*) в математике и в языках программирования – аргумент операции; данные, которые обрабатываются командой; математическое выражение, задающее значение аргумента операции.

Внутри знака операции пробелы не допускаются. Символы, составляющие знак операции, могут быть как специальными, например &&, | и <, так и буквенными, такими как ***new*** или ***sizeof***.

Литералы – это фиксированные значения (константы), которые не имеют имени. Но при этом литералы имеют тип,

который компилятор определяет по их внешнему виду. Например, если в программе записано число 456 – то это литерал целого типа, если число 45.6 – то это литерал вещественного типа. *Строковый литерал* – это набор символов в двойных кавычках.

Из лексем составляются выражения и операторы.

Выражение задает правило вычисления некоторого значения. Например, выражение a + b задает правило вычисления суммы величин a и b. Выражение содержит знак операции (+) и операнды (a и b).

Оператор задает законченное описание некоторого действия. Операторы делятся на исполняемые и неисполняемые, простые и составные. *Исполняемые* операторы задают действия над данными. *Неисполняемые* операторы служат для описания данных, поэтому их часто называют операторами описания, например *int a;* – это оператор описания целочисленной переменной *a*.

Составной оператор ограничивается фигурными скобками, а все остальные операторы должны завершаться точкой с запятой (;).

1.2 Представление чисел в памяти компьютера

Вся информация в памяти компьютера представляется в двоичной системе счисления.

Один разряд двоичного числа называется ***битом***. Бит может принимать значения 0 или 1 в зависимости от состояния электронной схемы (вкл/выкл). Однако отдельный бит памяти компьютера не имеет своего адреса, т.е. к нему нельзя обратиться как к самостоятельной единице.

Минимальная адресуемая единица памяти называется ***байтом***. Байт состоит из 8 бит. С помощью такого количества двоичных разрядов можно закодировать 256 различных целых чисел.

Чтобы понять, почему это именно так, представим, что у нас два бита. Сколько различных комбинаций из двух нулей и единиц можно придумать? Очевидно, что 4 (таблица 1.1).

Таблица 1.1 – Возможные комбинации двоичных цифр в трех разрядах

Двоичное число	Соответствующее десятичное число
00	0
01	1
10	2
11	3

При этом $4 = 2^2 = 2^{\text{количество разрядов}}$.

Теперь представим, что у нас три бита. Возможные комбинации нулей и единиц будут соответствовать приведенным в таблице 1.2.

Таблица 1.2 – Возможные комбинации двоичных цифр в трех разрядах

Двоичное число	Соответствующее десятичное число
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

И их всего $8 = 2^3 = 2^{\text{количество разрядов}}$.

Таким образом, легко заметить закономерность: если у нас n бит, то с их помощью можно закодировать 2^n различных двоичных чисел. Причем максимальное из этих чисел будет равно $2^n - 1$ (и оно представляет собой единицы во всех разрядах).

Именно поэтому в байте можно разместить 256 различных целых чисел: $2^8 = 256$. Причем это числа от 0 до 255. Но это в том случае, если мы собираемся хранить только неотрицательные числа.

Для хранения чисел со знаком (положительных и отрицательных) придумали первый (старший) бит числа отводить для кодирования знака: 0 – число неотрицательное, 1 – отрицательное. Тогда для хранения собственно значения (модуля) числа остается на один бит меньше, т.е. $n - 1$. (в байте 7 бит). А значит, максимальное положительное число, которое можно так закодировать, равно $2^{n-1} - 1$ (рисунок 1.1).

Рисунок 1.1 – Принцип размещения целого числа со знаком в памяти компьютера

Вещественным называется число, которое может иметь дробную часть. Представим именно десятичное дробное число, например 0,058. Перед размещением в памяти такого числа компьютер его нормализует, т.е. представляет в виде

$$0{,}58*10^{-1}$$

При этом число 0,58 называется ***мантиссой*** (первая цифра после запятой должна быть отличной от нуля), а число -1 – ***порядком***. Кстати, почти в таком виде число можно вывести и на экран: 0.58E-1. Такая форма записи называется научной (*scientific*).

В памяти компьютера ячейка, которая отводится под хранение вещественного числа, делится на две части (рисунок 1.2): в одной хранится мантисса (со знаком), а в другой – порядок (со знаком). Какое именно количество бит отводится под то и другое, определяется реализацией компилятора.

Рисунок 1.2 – Принцип размещения вещественного числа в памяти компьютера

Если же нужно данное число вывести на экран в привычном нам виде (с фиксированной точкой), то производится обратное преобразование: мантисса умножается на $10^{\text{порядок}}$.

1.3 Адресация данных в памяти компьютера

Минимальной адресуемой единицей памяти является один байт. Это означает, что для выполнения операции процессор может обратиться только к байту целиком, а не к какой-то его части.

Чтобы иметь доступ к байту памяти (или набору байтов), процессор должен знать его адрес. Адрес хранится в специальном регистре процессора. В 32-разрядных операционных системах под регистр адреса отводится 32 бита. Поэтому диапазон возможных адресов составляет $0...2^{32} - 1$. Максимальный адрес, таким образом, $2^{32} - 1 = 4\ 294\ 967\ 295$ байта ≈ 4 Гбайт.

Поэтому если на компьютере с 32-разрядной ОС установить микросхемы оперативной памяти с объемом более 4 Гбайт, то система просто «не увидит» эту память и не сможет с ней работать.

2 Переменные и константы

Переменная – это именованная область памяти, предназначенная для хранения данных, которые могут быть изменены. Каждая переменная имеет *имя*, *тип* и *значение*.

Можно представлять себе переменную как ящик, на который наклеена этикетка с именем. Тип определяет размер ящика, т.е. диапазон значений, которые может принимать переменная. А значение – это то, что в ящике лежит.

Любая переменная или константа должна быть описана в программе перед ее использованием. Причем в языке C++ описание переменной может располагаться в любом месте программы! В отличие, например, от Паскаля.

При описании указывается тип и имя переменной. Можно также после знака = указать начальное значение переменной (или значение константы), например:

```
int a=70; // a - имя переменной, int -  целочисленный тип,
          //70 -  начальное значение
```

Когда компилятор обрабатывает оператор описания переменной, он выделяет память для нее. Если начальное значение не задано, то невозможно предугадать, чем заполнена эта память (фактически она заполняется мусором, который остался на этом месте в памяти от каких-то предыдущих данных).

Значение переменной далее в программе можно изменить, например можно ввести в нее новые данные:

```
cin>>a;
```

Здесь *cin* – это имя стандартного потока ввода с клавиатуры (сокращение от *console in*), >> – оператор вставки. То, что пользователь наберет на клавиатуре, попадет в ячейку с именем a.

Также можно изменить переменную оператором присваивания:

```
a=25;
```

В этом операторе значение выражения справа от знака равно записывается в переменную, которая задана слева. Старое значение переменной a при этом исчезает («затирается»).

Аналогично можно ввести понятие именованной ***константы*** – это область памяти, предназначенная для хранения постоянных данных (значение константы нельзя изменять в тексте программы).

Для константы дополнительно указывается ключевое слово ***const***. Константа обязательно должна быть инициализирована (ей должно быть дано значение при объявлении). Имена констант принято записывать прописными буквами, например:

```
const double PI=3.1415; //объявление вещественной константы
        //PI - имя, double - вещественный тип, 3.1415 - значение
```


3 Типы данных языка C++

Тип данных определяет, сколько памяти выделяется компилятором для переменной или константы и как интерпретируется выделенная память. Как следствие, от типа зависит возможный диапазон значений переменной и то, какие операции над ней разрешены.

Типы языка C++ делятся на базовые и составные. Базовые типы данных являются неделимыми и позволяют описывать целые, вещественные, символьные и логические величины. На основе этих типов программист может конструировать составные типы. К составным типам относятся массивы, структуры, объединения, перечисления, ссылки, указатели.

3.1 Целочисленные типы данных

Целочисленные типы данных предназначены для хранения целых чисел. Такая переменная подойдет, например, для хранения счетчика количества цифр в числе или для номера дня недели, количества выигранных матчей за сезон и т.д.

В приложении 2 приведен список всех целочисленных типов с указанием соответствующих диапазонов значений.

Отличаются целочисленные типы размером (количеством байт памяти, выделяемых для переменной) и тем, используется ли знаковый разряд для представления числа (т.е. можно ли хранить в такой переменной число со знаком или же только неотрицательные числа). Для явного указания того, что число используется без знака, служит модификатор *unsigned*. По умолчанию все целые типы считаются типами со знаком. Но если это нужно указать специально, то можно использовать модификатор *signed*.

Пусть число имеет n разрядов. Если оно *unsigned*, то все n разрядов значимые. Поэтому диапазон возможных значений от 0 до $2^n - 1$.

При хранении целого числа со знаком в n битах старший разряд интерпретируется как знаковый. На представление собственно числа остается $n - 1$ бит. Поэтому диапазон значений такого числа от -2^{n-1} до $2^{n-1} - 1$.

Тип ***char*** предназначен для хранения кодов символов. В языке C++ эти коды символов могут также рассматриваться как целые числа и участвовать в арифметических операциях.

Обычно в языках программирования есть основной целочисленный тип *int*, занимающий 4 байта. Тип *short* (короткий) должен быть в 2 раза меньше, а тип *long* (длинный) – в 2 раза больше. Но исторически сложилось так, что в языке C++ тип *long* занимает столько же памяти, что и *int*, и фактически ничем не отличается от него. Для исправления этой ситуации в стандарте C++11 появился тип *long long*.

3.2 Потоковый ввод и вывод языка C++

Для использования потокового ввода и вывода необходимо подключить заголовочный файл библиотеки *<iostream>* и открыть соответствующее пространство имен:

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

Для ***вывода*** используется поток, связанный с экраном консоли, который называется cout (сокращение от Console out – консоль для вывода). Оператор вставки в поток можно использовать несколько раз в одной строке:

```
cout<<"Значение суммы: "<<sum<<"\n";
```

В этом примере на консоль выводится сначала строковый литерал, затем значение переменной sum, а потом снова строковый литерал, содержащий один символ – перевод курсора на новую строку. В поток можно выводить не только значение переменной, но и результат вычисления выражения:

```
cout<<"Значение суммы: "<<a+b<<"\n";
```

Для **ввода** используется оператор вставки в противоположном направлении (>>). Стандартный поток ввода, связанный с клавиатурой, называется cin (сокращение от Console in – консоль для ввода). Опять же можно вводить несколько значений последовательно:


```
cin>>a>>b;
```

Вводимые данные должны разделяться одним или несколькими пробелами, а ввод завершается после нажатия клавиши *Enter*. Дробная часть вещественных чисел отделяется десятичной точкой, как и в литералах программы.

При вводе значения в переменную ее старое значение «затирается», перестает существовать.

Задача 3.1

Ввести с клавиатуры два вещественных числа и вывести их сумму на консоль. При вводе чисел принято сначала выводить приглашение для пользователя:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL,"rus");
double a,b;
cout<<"Введите два числа: ";
cin>>a>>b;
cout<<"Сумма чисел равна: "<<a+b<<"\n";
system("pause");
return 0;
}
```

Результат работы этой программы показан на рисунке 3.1.

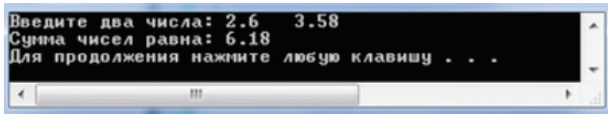


Рисунок 3.1 – Результат работы программы

4 Операции и выражения

4.1 Операции присваивания

Задача этого урока – научиться делать какие-то вычисления в программе. И первое, о чем мы поговорим, – это операция присваивания. Выглядит она просто как знак «равно» (=). Суть этой операции в том, что вычисляется выражение справа от знака равенства и результат помещается в переменную, записанную слева от него. При этом старое значение переменной теряется, «затирается» новой информацией.

Например, такая запись в математике неверна:

```
a=a+1;
```

А в программировании это оператор присваивания, который означает, что берется старое значение переменной *a*, увеличивается на 1 и результат записывается опять в переменную *a*.

Задача 4.1

Пусть есть две переменные *a* и *b* и нужно поменять местами их значения. Как это сделать, используя операцию присваивания?

Давайте представим, что у нас не две переменные, а два стакана. В стакане № 1 налита вода, а в стакане № 2 – кофе. А я хочу, чтобы было наоборот: в стакане № 1 – кофе, а в стакане № 2 – вода. Что нужно для того, чтобы перелить напитки? Правильно, нужен третий стакан! В него мы временно сольем, например, кофе. Тогда стакан № 2 освободится, и можно будет перелить в него воду из стакана № 1. А в освободившийся стакан № 1 зальем кофе из вспомогательного стакана, который после этого станет не нужен.

Также и с переменными. Для перестановки их значений нам потребуется третья, вспомогательная переменная. Назовем ее, например, *tmp* (от английского *temporary* – временный). Последовательность действий будет следующая:

```
tmp=a; // во вспомогательную ячейку запишем копию значения a
a=b; // теперь можем «портить» ячейку a, записав туда значение b
b=tmp; //в b записываем копию a, которую ранее сохранили в tmp
```


Наглядно этот алгоритм представлен на рисунке 4.1.

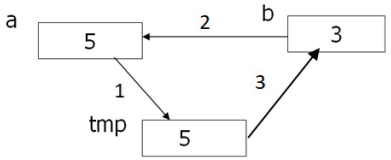


Рисунок 4.1 – Алгоритм перестановки двух значений

4.2 Выражения и операции

В переменную можно записать не только значение другой переменной, но и результат вычисления выражения. ***Выражения*** состоят из операндов (переменных, констант или литералов), знаков операций и скобок. Примеры выражений:

```
(a + 0.12 ) / 6

x && y || ! z

a = b = c
```

Операции задают действия, которые необходимо выполнить. В зависимости от количества операндов выделяют три вида операций:

1) ***унарные*** – один операнд (например, a++; -b), причем знак операции может стоять и слева, и справа от операнда;

2) ***бинарные*** – два операнда (например, a+b, x/y);

3) ***тернарная*** – три операнда (например, a>b?a:b).

Список основных операций языка C++ приведен в порядке убывания приоритета в таблице (приложение 3). Это не полный список, здесь только те операции, которые мы рассмотрим в данном курсе. Группы операций с одинаковым приоритетом отделяются двойными линиями.

Приоритет задает порядок вычисления операций в выражении, т.е., например, в выражении *a+b*3* сначала выполняется умножение, так как его приоритет выше.

Если в одном выражении записано несколько операций одинакового приоритета, то большинство операций выполня-

ются ***слева направо***. Исключение составляют унарные операции, условная (тернарная) операция и операции присваивания, которые выполняются ***справа налево***.

Например, $a = b = c$ означает, что сначала b присваивается значение c, а потом то, что записалось в b, присваивается переменной a (т.е. $a = (b = c)$).

А в выражении $a + b + c$ сначала выполняется сложение a и b, а потом к результату прибавляется c (т.е. $(a + b) + c$).

Для изменения порядка выполнения операций используются круглые скобки.

4.3 Арифметические операции

Арифметические операции в C++:

-x – унарный минус: меняет знак числа на противоположный;

+x – унарный плюс: ничего не делает;

x+y – сложение;

x-y – вычитание;

x*y – умножение;

x/y – деление;

x%y – остаток от деления (только для целых операндов).

Выполнение операции деления зависит от типа операндов. Если хотя бы один операнд имеет вещественный тип, то результат тоже будет вещественным (с дробной частью). Если же операнды целые, то результат будет иметь целый тип (дробная часть результата просто отбрасывается).

Таким образом, результат операции 1/2 равен 0 (литералы 1 и 2 по умолчанию целые, поэтому и результат целый – дробная часть будет отброшена). А результат операции 1.0/2 равен 0.5 (литерал 1.0 по умолчанию имеет вещественный тип *double*, поэтому и результат – вещественное число).

Операция взятия остатка от деления (деление по модулю) в языке C++ разрешена только для целых операндов (если операнды не целые, выводится сообщение об ошибке на этапе компиляции). Остаток от деления – это то число, которое нужно прибавить к произведению делителя и частного, чтобы получилось делимое.

Пусть числа a, b, k и c – целые, причем a – делимое, b – делитель k – частное, c – остаток от деления:

```
k=a/b;

c=a%b;
```

Тогда по определению остатка от деления должно быть истинно выражение

```
a=b*k+c.
```

Например, пусть $a = 7$, $b = 2$.

```
7/2=3  7%2=1, так как 2*3+1=7 - истинно.
```

Еще примеры:

```
19/5=3  19%5=4 , так как 3*5+4=19;

34/6=5  34%6= 4,  так как 6*5+4=34.
```

Обратите внимание, что остаток от деления всегда меньше делителя.

Для вещественных чисел операция деления на ноль не является ошибкой. В результате получается значение «бесконечность» (inf). Для целых чисел операции деления на ноль и взятия остатка от деления запрещены. Это означает, что во время выполнения такой операции возникает ошибка (исключение), выполнение программы прерывается.

Нужно учитывать, что в языке C++ операция взятия остатка от деления для отрицательных операндов выполняется следующим образом: берутся операнды по модулю и вычисляется остаток от деления. К нему приписывается знак делимого:

```
#include <iostream>
using namespace std;
int main()
{
        cout << 7 % 2 << endl; //выводится 1
        cout << -7 % 2 << endl; //выводится -1
        cout << 7 % -2 << endl; //выводится 1
        cout << -7 % -2 << endl; //выводится -1
        system("pause");
        return 0;
}
```

4.4 Инкремент и декремент

Инкремент – это увеличение на 1, а ***декремент*** – уменьшение на 1:

```
a++;    //эквивалентно   a=a+1;
b--;    //эквивалентно     b=b-1;
```

Эти операции имеют две формы:

1) *префиксная* форма (знак операции перед именем переменной), в этом случае сначала выполняется увеличение (уменьшение), а затем используется значение переменной:

```
int a=3;
cout<<++a;  //выводится 4, значение a равно 4
```

2) *постфиксная* форма (знак операции после имени), в постфиксной форме сначала используется значение переменной, а потом выполняется увеличение (уменьшение):

```
int a=3;
cout<<a++;  //выводится 3, значение a равно 4
```

4.5 Сокращенные формы операции присваивания

Эти операции позволяют коротко записать выполнение какой-либо арифметической операции над значением переменной и записать результат опять в эту же переменную:

```
a+=10;        //  эквивалентно a=a+10;
a-=10;        //  эквивалентно  a=a-10;
a*=10;       //   эквивалентно  a=a*10;
a/=10;       //   эквивалентно  a=a/10;
a%=10;       //   эквивалентно  a=a%10;
```

4.6 Условная (тернарная операция)

Условная (тернарная операция) имеет три операнда и следующий формат:

операнд_1 ? операнд_2 : операнд_3

Сначала вычисляется значение операнда 1. Результат должен быть типа, который можно преобразовать к типу bool. Если он равен true, то результатом выполнения всей условной операции будет значение операнда_2, иначе – операнда_3.

Например:

```
c=(a>b)?a:b; // максимум из двух чисел записывается в переменную c
i=(i<n)?i+1:1; //если i<n, i принимает значение на единицу
               //больше, а если нет, то i становится равно 1
```

5 Математические функции

Описания математических функций находятся в библиотеках <stdlib.h>, <math.h> и <cmath>. В Visual Studio эти библиотеки подключаются по умолчанию. В других средах разработки необходимо подключать эти библиотеки директивой препроцессора #include <...>.

По умолчанию значения функций и их аргументы имеют тип double. Вещественные литералы также по умолчанию double. Поэтому, если нет особых причин экономить память, всегда используйте для вещественных чисел тип double.

Список некоторых математических функций приведен в приложении 4.

Пример 5.1

Написать программу вычисления значения функции

$$y = \frac{\sqrt{2}}{2} \sin \frac{x}{2}.$$

```
#include <iostream>
using namespace std;
int main()
{       double x,y;
        setlocale(LC_ALL,"rus");
        cout<<"Введите аргумент: ";
        cin>>x;
        y=sqrt(2.)/2*sin(x/2);
        cout<<"Результат функции: "<<y<<"\n";
        system("pause");
        return 0;
}
```

В C++ нет стандартной константы для числа ϖ. Это значение легко высчитать, если взять арктангенс единицы и умножить его на 4. Записывается это как *atan(1) * 4*.

Отдельно рассмотрим функции для округления значений вещественных чисел (их результат – вещественное число, хотя дробная часть нулевая) (таблица 5.1).

Таблица 5.1 – Функции для округления значений вещественных чисел

ceil(a)	округление в большую сторону Примеры: ceil(2.3)=3.0 ceil(-2.3)=-2.0
floor(a)	округление в меньшую Примеры: floor(2.3)=2.0 floor(-2.3)=-3.0
trunc(a)	отбрасывается дробная часть Примеры: trunc(2.3)=2 trunc(-2.3)=-2
round(a)	округление по правилам Примеры: round(2.3)=2.0 round(2.7)=3.0 round(-2.3)=-2.0 round(-2.7)=-3

6 Преобразование типов данных

Если операнды некоторой операции имеют различные типы, то они приводятся к одному типу. Такое преобразование типа может быть неявным (автоматическим) и явным (заданным программистом).

Неявное преобразование типов в выражении (справа от знака «=») всегда является ***расширяющим:*** более короткие типы приводятся к более длинным. Если в выражении участвуют типы bool, char и short, то они всегда приводятся к типу int.

При этом true преобразуется в 1, а false – в 0. Преобразование char в int означает, что код символа становится тем числом, над которым производятся действия.

Ниже показана иерархия типов, согласно которой компилятор выбирает тип, к которому приводятся операнды (и соответственно тип результата):

(bool,char,short)->int->unsigned int->long->unsigned long->float->double->long double

Например:

```
int a=8;
long b=56;
long c=a+b;   //int +long приводится к long+long
```

```
float x=7.8;
double y=x+3.14; //литерал с точкой по умолчанию double
                 //float+double приводится к double+double
```

В операции присваивания может неявно выполняться ***сужающее*** преобразование: если в левой части оператора присваивания более короткий тип, то правая часть преобразуется к этому типу, возможно, с потерей данных! При этом компилятор не сообщает об ошибке.

Например:

```
int a=3.14*2; //сначала в правой части выполняется расширяющее
   //преобразование: double*int->double
   //но потом это число присваивается целой переменной  и
   //выполняется сужающее преобразование: отбрасывается дробная часть
   // в результате a получает значение 6
```

```
int q=-5;
unsigned int h;
h=q;  //число int записывается в переменную unsigned как набор
      //двоичных цифр. При этом не только теряется знак, но и само число
      //будет записано неправильно: h равно 4294967291
```

Поэтому программисту нужно быть очень внимательным при выполнении сужающих преобразований: вся ответственность за потерю и искажение данных лежит на нем.

В сложном выражении преобразование типа выполняется непосредственно перед каждой операцией. Таким образом, ***последовательность преобразований зависит от порядка выполнения операций***.

Например:

```
int a;
float b;
float c=3.5;
b=a=c;
```

Чему станут равны значения переменных a и *b*?

Порядок выполнения операции присваивания – справа налево, поэтому первой будет выполнена операция $a = c$. При этом произойдет сужающее преобразование и дробная часть значения *c* будет отброшена, *a* получит значение 3. Затем уже результат этой операции будет присвоен переменной *b*, т.е. она тоже станет равна 3!

В случае такой записи оператора присваивания:

```
a=b=c;
```

сначала выполняется $b = c$ без потери данных (так как они обе имеют тип float), а уже затем – сужающее преобразование перед присвоением 3,5 целой переменной. В результате *b* равно *3,5*, *a* равно *3*.

7 Операторы ветвления

7.1 Логические выражения

Логические выражения – это выражения, которые могут принимать значения *true* (истина) или *false* (ложь), т.е. имеют логический тип (*bool*).

7.2 Операции сравнения

Такой результат дают операции сравнения (другое их название – операции отношения). В языке С++ можно использовать знаки операций сравнения, приведенные в таблице 7.1.

Таблица 7.1 – Знаки операций сравнения

Операция	Утверждение	Пример	Результат
<	Левый операнд меньше, чем правый	3 < 6	true
>	Левый операнд больше, чем правый	3 > 6	false
<=	Левый операнд меньше правого или равен ему	3 <= 3	true
>=	Левый операнд больше правого или равен ему	3 >= 6	false
==	Левый операнд равен правому	3 == 6	false
!=	Левый операнд не равен правому	3 != 6	true

При записи знаков операций из нескольких символов между ними не должно быть пробела (<= верно, а < = неверно!). Знак > или < должен стоять первым (>= верно, а => неверно).

Обратите внимание на операцию проверки на равенство! Нужно использовать два знака равенства, потому что один такой знак означает присваивание. Начинающие программисты очень часто путают эти операции!

7.3 Логические операции

Иногда нужно записать более сложное условие, чем простое сравнение. Например, нужно проверить, что число ***x*** находится в диапазоне от 0 до 10. Записать это как ***0<=x<=10*** нельзя!

Тут на помощь приходят логические операции (***И, ИЛИ, НЕ***). Принято описывать действие этих операций с помощью таблиц истинности, в которых показаны все возможные значения аргументов и результат операции для каждого случая.

Операция «*Логическое И» (&&)* дает результат *true*, если оба операнда *true* (т.е. истинно и то, и другое утверждение).

Например, условие **0<=x<=10** в языке С++ правильно записать так: **(x>=0)&&(x<=10)**.

Операция «*Логическое ИЛИ» (||)* дает результат *true*, если хотя бы один из операндов *true* (т.е. истинно одно или другое утверждение или оба сразу).

Например, нужно записать условие, что число *x* не принадлежит диапазону от 0 до 10. Это означает, что **либо x<0, либо x>10**. На С++ это можно записать так: **(x<0)||(x>10)**.

Операция *«Логическое НЕ»* дает результат, противоположный значению аргумента (меняет *true* на *false* и наоборот).

Например, условие неотрицательности числа *x* можно записать просто: x>=0, либо с использование отрицания: !(x<0).

В сложном логическом выражении эти операции выполняются слева направо в порядке их приоритетов. Наивысший приоритет имеет унарная операция ! (НЕ), затем операция && (И – логическое умножение), а затем || (ИЛИ – логическое сложение).

Операции сравнения имеют приоритет ниже, чем !, но выше, чем && и ||. Поэтому выражение **x<y&&k==2||k>10** эквивалентно **(x<y)&&(k==2)||(k>10)**.

Таким образом, скобки здесь не обязательны. Но лучше их поставить во избежание ошибок и для лучшего понимания кода.

Если при выполнении логической операции значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

Пример 7.1

(4<10)||(k>3)

Поскольку первый операнд операции ИЛИ *true*, то результат будет *true* независимо от значения второго операнда. Поэтому он даже не вычисляется.

Пример 7.2

(i<n)&&(3*i-2>0)

Если первое условие (*i*<*n*) нарушено (его результат *false*), то второе даже не вычисляется, поскольку ясно, что результат операции И тоже будет *false*.

7.4 Условный оператор *if*

Оператор ***if*** позволяет изменить последовательность выполнения инструкций кода в зависимости от какого-то условия.

Можно использовать две формы этого оператора – полную:

if (выражение) оператор_1; else оператор_2;

и сокращенную:

if (выражение) оператор;

Если значение выражения в круглых скобках *true* (истина), то выполняется оператор_1. Если значение этого выражения *false* (ложь), то выполняется оператор_2 после слова *else* (либо ничего не выполняется в случае сокращенной формы оператора *if*). Блок-схемы полной и сокращенной формы условного оператора *if* приведены на рисунке 7.1.

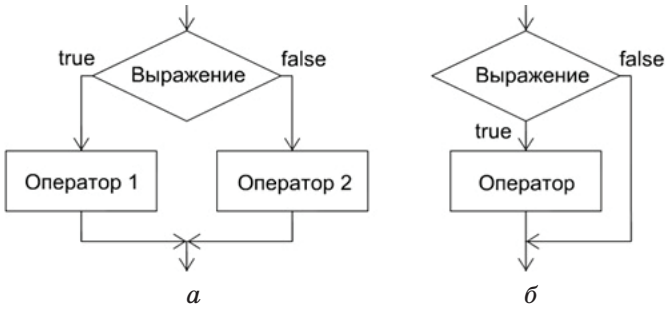


Рисунок 7.1 – Блок-схемы полной (*а*)
и сокращенной (*б*) формы условного оператора *if*

Пример 7.3

Ввести два целых числа и разделить первое на второе.

Поскольку целочисленное деление на 0 запрещено (активируется исключение и программа прерывается), то после ввода чисел проверим, что делитель не равен 0. В случае правильности данных выводим частное, а иначе – выводим предупреждение:

```
int a, b;
cin >> a >> b;
if (b != 0)
cout << a / b;
else
cout << "На ноль делить нельзя!\n";
```

Если нужно, чтобы по какой-то ветке (когда выражение истинно или когда ложно) выполнялся не один оператор, а несколько, то их объединяют в составной оператор с помощью *фигурных скобок*.

Допустим, в нашем примере мы хотим сначала записать результат деления в переменную *c*, а потом вывести ее на консоль:

```
int a, b, c;
cin >> a >> b;
if (b != 0) {
c = a / b;
cout << c;
}
else {
cout << "На ноль делить нельзя!\n";
}
```

Выражение в круглых скобках оператора *if* может иметь не только тип *bool*, но и любой целый тип. В этом случае выполняется неявное приведение результата выражения к логическому типу по следующему **правилу:** *любое целое число, не равное 0 (даже отрицательное), считается true. Если же выражение имеет нулевое значение, то результат – false.*

Поэтому записать условие, что сумма чисел не равна нулю, можно так:

```
if (a+b) {…}
```

А можно и так:

```
if (a+b!=0) {…}
```

7.5 Вложенные операторы *if*

Операторы *if* могут быть «вложены» друг в друга.

Пример 7.4

Пользователь вводит сумму, на которую он собирается открыть счет в банке. Процент по вкладу зависит от его размера следующим образом:

- при сумме до 1000 $ (включительно) вкладчику начисляется 2 % в год;
- если сумма от 1001 до10 000 (включительно), то 3 % в год;
- если же сумма вклада более 10 000 $, то вкладчик получит 5 % годовых.

Проценты начисляются однократно в конце года. Нужно вывести на консоль сумму, которую пользователь получит через год.

Нарисуем числовую прямую и отметим на ней контрольные точки, в которых изменяется процентная ставка (рисунок 7.2).

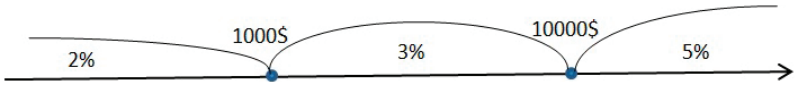


Рисунок 7.2 – Контрольные точки на числовой прямой

Получается три интервала значений суммы вклада, для которых будут разные процентные ставки.

Блок-схема основной части программы приведена на рисунке 7.3.

Решим эту задачу разными способами.

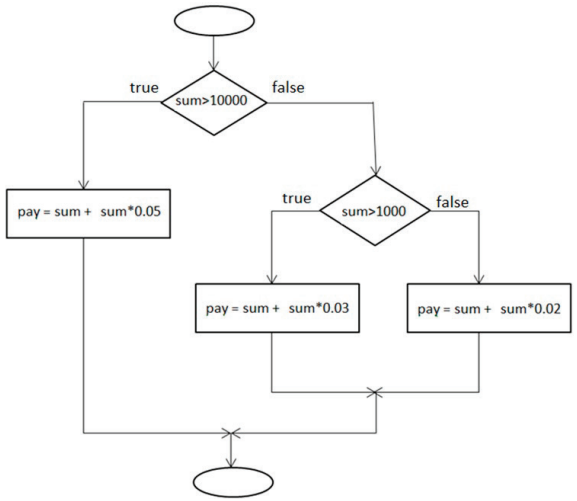


Рисунок 7.3 – Блок-схема основной части программы

Первый состоит в том, что мы начнем проверку с самой большой границы суммы. Следующее условие будем проверять только в том случае, если предыдущая проверка оказалась неудачной:

```
#include <iostream>
using namespace std;
int main() {
setlocale(LC_ALL, "rus");
int sum, pay;
cout << "Введите сумму вклада: ";
cin >> sum;
if (sum > 10000) {
pay = sum + sum*0.05; //5% ставка при максимальной сумме
}
else {
if (sum > 1000){ //>1000, но <=10000
pay = sum + sum*0.03; //3% ставка по депозиту
}
else { //<1000
pay = sum + sum*0.02; //2% при минимальной сумме
}
}
cout << "Получите через год: " << pay << " $\n";
system("pause");
return 0;
}
```

Обратите внимание, что слово *else* всегда относится к предыдущему *if*. В данном примере это подчеркивается тем, что *else* записывается строго под своим *if*.

Однако есть и другой способ оформления вложенных инструкций *if – else*:

- если по ветке *else* только один вложенный *if*, то скобки можно не ставить;

- все *else* записываются друг под другом (при этом все равно каждый *else* относится к ближайшему к нему *if*):

```
#include <iostream>
using namespace std;
int main() {
setlocale(LC_ALL, "rus");
int sum, pay;
cout << "Введите сумму вклада: ";
cin >> sum;
if (sum < 0) {
cout << "Ошибочная сумма!!\n";
system("pause");
return 0;
}
if (sum > 10000) {
pay = sum + sum*0.05; //5% ставка при максимальной сумме
}
else if (sum > 1000){
pay = sum + sum*0.03; //3% ставка по депозиту
}
else {
pay = sum + sum*0.02; //2% при минимальной сумме
}
cout << "Получите через год: " << pay << " $\n";
system("pause");
return 0;
}
```

В эту программу добавлена еще проверка корректности вводимых данных. Если пользователь ввел отрицательное число, то выводится сообщение об ошибке. Затем выполняется задержка экрана до нажатия любой клавиши (чтобы он успел это прочитать). И после этого программа прекращает выполнение (*return 0*).

Таким образом, если программа продолжается и выходит на следующий *if*, то можно быть уверенными в корректности данных.

Запомните этот прием отсекания неверных исходных данных, при которых продолжение программы бессмысленно! Он очень упрощает код программы, позволяя избежать многих вложенных операторов ветвления.

Следует отметить, что использование вложенных операторов *if* очень сильно усложняет программу, а программист всегда должен стремиться к простоте и очевидности своего кода. Попробуем упростить этот пример. Самый очевидный способ – это сделать несколько операторов *if*, которые выполняются последовательно. Но при этом для любой введенной пользователем суммы условие ветвления станет истинным только один раз, т.е. условия должны быть взаимоисключающими. Это легко сделать, используя *логические операции*. Центральная часть программы тогда выглядит так:

```
if (sum > 10000) {
pay = sum + sum*0.05; //5% ставка при максимальной сумме
}
if ((sum > 1000)&&(sum<=10000)){
pay = sum + sum*0.03; //3% ставка по депозиту
}
if(sum<=1000){
pay = sum + sum*0.02; //2% при минимальной сумме вклада
}
cout << "Получите через год: " << pay << " $\n";
```

Третий способ написать эту программу состоит в том, чтобы *начать проверку с самой маленькой суммы*. Количество денег, которое будет получено через год (переменная *pay*), будет перезаписываться (с «затиранием» предыдущего результата) каждый раз, когда мы обнаруживаем, что сумма оказалась подходящей под следующий уровень процента по вкладу:

```
pay = sum + sum*0.02; //2% по вкладу изначально
if (sum>1000){
pay = sum + sum*0.03; //отменяем 2%, применяем 3%
}
if (sum>10000) {
pay = sum + sum*0.05; //отменяем все предыдущее, применяем 5%
}
cout << "Получите через год: " << pay << " $\n";
```

Нам удалось избавиться от одного *if*, и мы обеспечили инициализацию переменной *pay* в любом случае, ошибка времени выполнения не возникнет!

Очень важно уже на таких простых задачах учиться писать не просто работоспособный, а наиболее простой код! Этот навык очень пригодится, когда ваши программы станут значительно больше и сложнее. В качественно написанном коде обычно не бывает более трех уровней вложенных операторов *if*.

7.6 Оператор выбора *switch*

Оператор выбора (***switch***) используется в тех случаях, когда необходимо разветвить процесс выполнения программы на несколько веток. Часто он является удобной альтернативой вложенным операторам *if*.

Формат оператора:

```
switch (выражение){
   case значение_1: действия_1; break;
   case значение_2: действия_2; break;
   …
   case значение_n: действия_n; break;
   default: действия по умолчанию;
}
```

Выражение в скобках должно иметь целочисленный тип (тип *char* тоже целочисленный). После *case* указываются константы, с которыми последовательно сравнивается результат вычисления выражения. Если есть совпадение, то управление передается операторам, указанным после двоеточия на соответствующей ветке.

Если совпадений нет, то выполняются операторы после слова *default*. Ветка *default* может отсутствовать, тогда не выполняются никакие действия.

В конце каждой ветки следует указывать оператор *break* (выход). Он выводит за пределы конструкции выбора. Если оператор *break* не написать, то после выбранной ветки будут выполняться также все последующие действия до конца оператора *switch* (включая и действия по умолчанию). После операторов последней группы (после *default* или после последнего *case*) *break* можно не указывать.

Блок-схема оператора выбора приведена на рисунке 7.4.

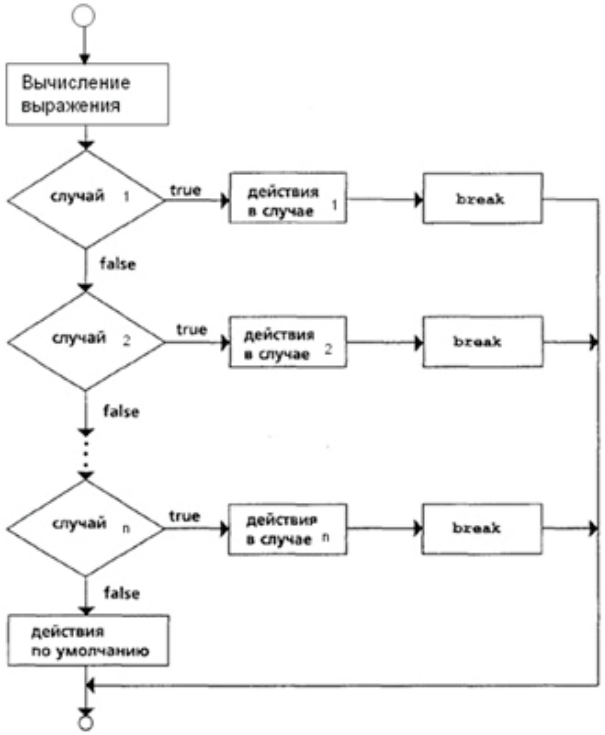


Рисунок 7.4 – Блок-схема оператора выбора

Обратите внимание на оформление программы: действия по каждой ветке не обязательно заключать в фигурные скобки! Если количество операторов, выполняемых по каждой ветке, мало, то их можно записать в одну строку. Если же по каждой ветке операторов много или они имеют сложную структуру, то лучше их размещать друг под другом, используя сдвиг вправо на три-четыре позиции (для указания вложенности).

Пример 7.5

Требуется реализовать простейший калькулятор на четыре действия.

Знак операции имеет тип *char*, и по нему производится выбор. Если введен не предусмотренный знак, то выводит-

ся сообщение «*Неизвестная операция*» (действия по ветке *default*). В случае если это деление, проверяется второй операнд. При равенстве его нулю выводится сообщение о невозможности выполнения операции:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "rus");
int a, b, res;
char op;
cout << "\nВведите 1й операнд : ";
cin >> a;
cout << "\nВведите знак операции : ";
cin >> op;
cout << "\nВведите 2й операнд : ";
cin >> b;
bool flag = true; //признак вывода результата
switch (op)
{
case '+': res = a + b; break;
case '-': res = a - b; break;
case '*': res = a * b; break;
case '/':
if (b != 0) res = a / b;
else
{
cout << "Деление на 0 невозможно!\n";
flag = false; //результат выводить не нужно
}
break;
default:
cout << "\nНеизвестная операция\n";
flag = false; //результат выводить не нужно
}
if (flag) cout << "\nРезультат : " << res << "\n";
system("pause");
return 0;
}
```

В этой программе применен один из известных приемов программирования: использование флага.

Флаг – это переменная типа *bool*. В нашей программе она означает необходимость вывода результата операции на экран. Перед началом оператора *switch* флаг устанавливается в *true*. Если по ходу анализа знака операции выясняется, что результат не может быть рассчитан (неизвестный знак или деление на 0), то флаг устанавливается в *false* (сбрасывается).

Вывод результата на экран выполняется только в случае установленного флага. Такой прием позволяет избежать дублирования кода (например, вывода результата в каждой ветке оператора *switch*).

В предыдущем примере использовалась переменная типа *char* в операторе *switch*. Но выбор можно делать и по переменной другого целочисленного типа (например, *int*).

Пример 7.6

В здании 10 этажей. На первых трех этажах располагается большой торговый центр, на четвертом этаже – фудкорт. С пятого по седьмой этаж занимает компания EPAM Systems, на восьмом этаже находится крупное агентство недвижимости «Столица». Последние два этажа занимают офисы различных мелких компаний.

Пользователь вводит номер этажа, а программа должна вывести информацию о том, что на этом этаже находится («Торговый центр», «Фудкорт», «EPAM Systems», «Агентство недвижимости «Столица»», «Офисы различных компаний»). Если же введен неправильный номер этажа, то должно быть выведено: «В здании только 10 этажей!».

```
#include <iostream>
using namespace std;
int main() {
setlocale(LC_ALL, "rus");
int number;
cin >> number;
switch (number) {
case 1: case 2: case 3:
cout << "Торговый центр" << endl;  break;
case 4:
cout << "Фудкорт" << endl;  break;
case 5: case 6: case 7:
cout << "EPAM Systems" << endl; break;
case 8:
cout << "Агенство недвижимости \"Столица\"" << endl; break;
case 9:case 10:
cout << "Офисы различных компаний" << endl; break;
default:
cout << "В здании только 10 этажей!" << endl;
}
system("pause");
return 0;
}
```

Обратите внимание, что значения рядом с ***case*** приводятся без кавычек, поскольку это числа, а не символы!

7.7 Операторы цикла

Цикл – это повторение каких-то однотипных действий. Чтобы реализовать это повторение, в языке С имеется три оператора:

- ***while***;
- ***for***;
- ***do-while***.

В принципе, можно обойтись только одним циклом *while*. Просто другие два оператора немного упрощают код программы.

Все виды циклов покажем на одной задаче. Это продемонстрирует тот факт, что решений всегда может быть несколько.

Задача 7.1

Ввести 10 чисел с клавиатуры и найти их сумму.

В этой задаче не нужно заводить 10 переменных для ввода чисел, а достаточно одной переменной (назовем ее *x*). В эту переменную мы вводим очередное число, прибавляем его к накапливаемой сумме (переменная *sum*), и после этого хранить уже использованное значение *x* нет смысла! Поэтому мы эту же переменную используем для ввода следующего числа.

Таким образом, нам необходимо организовать повторение 10 раз следующих действий:

```
Ввод x;
sum+=x;
```

Эти действия, которые нужно повторять многократно, называются ***телом цикла***, а один шаг повторения – ***итерацией***.

Очевидно, что в начале программы нужно переменную *sum* очистить, подготовить для накопления суммы: *sum*=0;.

Чтобы контролировать, сколько повторений мы сделали, заведем переменную-счетчик *k*. В начале программы мы еще не выполнили ни одного повторения, поэтому присвоим этой переменной значение 0, а на каждом шаге повторения счетчик будем увеличивать на 1.

Повторения продолжаются, пока k<10, т.е. если условие истинно, то нужно войти в цикл и выполнить ввод и увеличение суммы еще раз.

Блок-схема этого алгоритма приведена на рисунке 7.5.

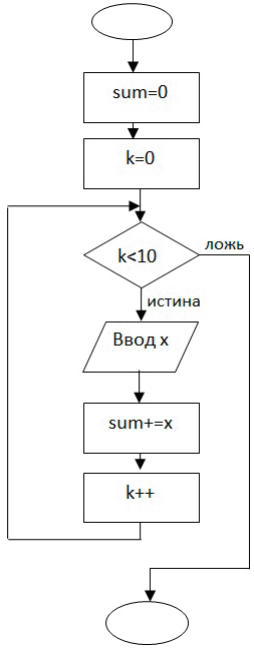


Рисунок 7.5 – Блок-схема алгоритма

7.7.1 Цикл *while*

Обычно этот цикл применяется тогда, когда число повторений заранее неизвестно. Формат оператора:

while (выражение) оператор;

Сначала вычисляется выражение и проверяется его истинность, т.е. выражение должно иметь тип *bool* (или тип, который неявно приводится *к bool* аналогично выражению в операторе *if*). Если выражение истинно, то выполняется оператор, т.е. тело цикла, и происходит возврат на проверку выражения. Если выражение ложно, то выполнение цикла за-

канчивается и управление передается на следующий оператор после цикла. Обратите внимание, что если условие ложно с самого начала, то тело цикла не выполнится ни разу (рисунок 7.6)!

Рисунок 7.6 – Блок-схема алгоритма,
реализующего работу цикла while

Если тело цикла должно состоять более чем из одного оператора, то они объединяются в составной оператор с помощью фигурных скобок.

Код для вычисления суммы 10 чисел:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL,"rus");
int x,sum,k;
sum=0;
k=0;
while(k<10)
{
cin>>x;
sum+=x;
k++;
}
cout<<"Сумма чисел= "<<sum<<"\n";
system("pause");
    return 0;
}
```


7.7.2 Цикл с параметром *for*

Этот вид цикла используется в тех случаях, когда мы заранее знаем число повторений. В нем используется параметр цикла (переменная, которая служит для контроля числа повторений). Формат оператора:

for (инициализация; выражение; модификации) оператор;

Сначала выполняются действия, указанные в разделе «инициализация». Затем вычисляется значение выражения (условие выполнения цикла) и проверяется на истинность. Если выражение истинно, то выполняется тело цикла, затем действия из раздела «модификации» и происходит возврат на проверку выражения. Если выражение ложно, то происходит выход из цикла и управление передается на следующий оператор после него (рисунок 7.7).

Рисунок 7.7 – Блок-схема алгоритма,
реализующего работу цикла *for*

Так же, как и в случае цикла *while*, тело цикла может не выполниться ни разу (если выражение ложно с самого начала).

Используем этот вид цикла для записи программы определения суммы 10 чисел:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL,"rus");
int x,sum,k;
sum=0;
for(k=0;k<10;k++)
{
cin>>x;
sum+=x;
}
cout<<"Сумма чисел= "<<sum<<"\n";
system("pause");
        return 0;
}
```

Обратите внимание, что в теле цикла нам уже не нужно заботиться об увеличении счетчика: цикл *for* выполнит это автоматически.

Особенности использования цикла for:

- любая из трех компонент цикла *for* может отсутствовать, но точки с запятыми остаются;

- если пропущено условие выполнения цикла, то оно по умолчанию считается истинным:

```
for( ; ; ) {…}  //бесконечный цикл
```

- если переменная объявлена (создана) в разделе инициализации, то после выхода из цикла она перестает существовать:

```
for(int k=0;k<10;k++)
{
cin>>x;
sum+=x;
}
//после выхода из цикла переменная k не существует
```

7.7.3 Цикл с постусловием *do-while*

Этот вид цикла также применяется в том случае, когда мы заранее не знаем число повторений цикла. Формат оператора:

do оператор while (выражение);

Сначала выполняется тело цикла, потом вычисляется выражение и проверяется его истинность. Если выражение истинно, то происходит возврат и тело цикла выполняется еще раз. Если выражение ложно, то происходит выход из цикла и управление передается следующему за циклом оператору.

Таким образом, тело цикла в этом случае обязательно выполнится хотя бы один раз (рисунок 7.8)!

Рисунок 7.8 – Блок-схема алгоритма, реализующего работу цикла *do - while*

Пример программы определения суммы 10 чисел с помощью цикла с постусловием:

```
#include <iostream>
using namespace std;
int main() {
setlocale(LC_ALL, "rus");
int x, sum;
sum = 0;
int k = 0;
do {
cin >> x;
sum += x;
k++;
} while (k < 10);
cout << "Сумма= " << sum << endl;
system("pause");
return 0;
}
```

Блок-схема программы уже будет несколько иная, хотя результат работы программы абсолютно тот же (рисунок 7.9).

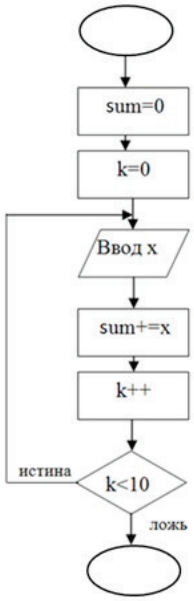


Рисунок 7.9 – Блок-схема программы

7.8 Операторы передачи управления из тела цикла

Для экстренного выхода из цикла применяются два оператора:

- **break** – досрочно завершает выполнение цикла (или оператора *switch*), передавая управление на следующий за ним оператор;

- **continue** – выполняет переход к следующей итерации цикла, пропуская все оставшиеся действия до конца текущего шага. Если это цикл *for*, то перед проверкой условия выполняются модификации.

Схема работы этих операторов приведена на рисунке 7.10.

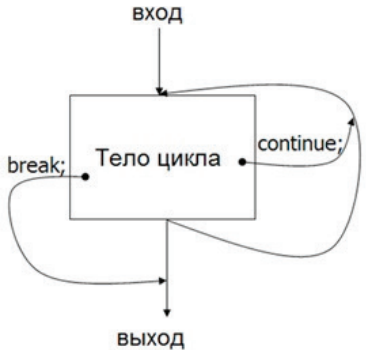


Рисунок 7.10 – Схема работы операторов *break* и *continue*

Оператор *break* выполняет выход только из одного цикла (или оператора *switch*), в котором он непосредственно находится! С его помощью нельзя выйти сразу из всех вложенных циклов!

Хотя операторы *break* и *continue* нарушают принципы структурного программирования (о которых мы поговорим позже), часто они дают изящное решение задачи, выигрыш в производительности и улучшают читабельность программ.

Пример 7.7

Пользователь вводит неотрицательные целые числа. Ввод любого отрицательного числа означает, что он желает закончить процесс ввода и получить сумму введенных чисел.

Если реализовать эту задачу аналогично примеру предыдущего урока, то решение будет неправильное: последнее отрицательное число прибавится к сумме и изменит ее!

Поэтому после ввода числа проверим его на отрицательность. И если это отрицательное, т.е. «служебное» число, то выйдем из цикла оператором *break*. При этом оператор *sum*+=x; который стоит далее в теле цикла, выполнен уже не будет. А цикл можно сделать бесконечным, задав конструкцию *while(true)*{...} или *for* (;;){...}.

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "rus");
int x, sum = 0;
for (;;)
{
cout << "Введите число (<0  означает окончание ввода )\n";
cin >> x;
if (x<0) break;
sum += x;
}
cout << "Сумма чисел равна= " << sum << "\n";
system("pause");
return 0;
}
```

7.9 Операторы *goto* и *return*

Оператор ***goto*** позволяет передать управление в любую точку программы. Для этого в нужном месте программы перед оператором ставим метку (идентификатор, после которого идет двоеточие). А в *goto* указываем эту метку:

```
          goto newstep;
          …
newstep:  k++;
```

Здесь *newstep* – это метка, т.е. обычный идентификатор, который подчиняется всем правилам создания имен в языке С.

С точки зрения принципов структурного программирования необходимо отказаться от использования этого оператора в принципе, поскольку он:

- делает программу сложной для понимания;
- повышает вероятность ошибок;
- не дает возможность компилятору оптимизировать код.

Однако никакие принципы (в том числе и принципы структурного программирования) не нужно возводить в абсолют. Существуют ситуации, когда использование оператора *goto* вполне оправдано, например в случае, если нужно выйти из вложенных циклов:


```
for(int i=0;j<m;i++){
   for(int j=0;j<n;j++)
   {    …
if (…) goto myexit;
        …
   }
}
myexit: cout<<"Продолжение программы…\n";
```

Оператор *break* тут не поможет, поскольку выведет только из внутреннего цикла.

В любом случае с помощью *goto* управление не должно передаваться назад по тексту программы. Использовать этот оператор нужно только в крайних случаях, когда другие средства языка программу не упрощают, а запутывают.

Оператор ***return*** служит для выхода из функции (передачи управления назад той функции, которая вызвала текущую). Поскольку мы пока используем только одну функцию (*main*), оператор *return* означает прекращение программы (возврат управления Visual Studio).

Использование оператора *return* позволяет избежать вложенных конструкций языка и упростить текст программы.

8 Массивы

8.1 Одномерные массивы

8.1.1 Объявление и инициализация одномерного массива

Массив – это набор данных одного типа. Рассмотрим сначала одномерные массивы (с одним индексом). При объявлении массива указывается тип элемента, затем имя массива и в квадратных скобках – число элементов:

```
double x[3]; //массив из трех вещественных элементов
```

Вся используемая программой оперативная память делится на две части: *автоматическая память*, которая выделяется на этапе компиляции, и *динамическая память* (*heap*, или «куча»), которая выделяется в процессе выполнения программы.

Мы будем рассматривать массивы, которые размещаются в автоматической памяти. *На этапе компиляции число элементов такого массива должно быть однозначно известно.* Поэтому число элементов – это литерал (как в примере выше), или целочисленная константа, или константное выражение (результат такого выражения известен на этапе компиляции):

```
const int SIZE=5; //константа SIZE - число элементов массива
int a[SIZE];   //массив из пяти элементов типа int
char s[SIZE+2]; //константное выражение: массив из семи элементов char
```

Использование констант при объявлении массива является признаком хорошего стиля программирования. Если потом станет нужно изменить количество элементов массива, то не придется просматривать и изменять всю программу, достаточно будет изменить значение константы в одном месте.

Чтобы обратиться к элементу массива, нужно указать имя массива и индекс (порядковый номер) в квадратных скобках, например a[3].

Нумерация элементов в массиве начинается с 0. Таким образом, если, например, объявлен массив из 10 элементов, то используются индексы от 0 до 9.

Элементы массива располагаются подряд друг за другом. Например, пусть массиву a из элементов типа *int* выделена память, начиная с адреса 40 (рисунок 8.1). Тогда с этого адреса размещается элемент a[0], с адреса 40+4=44 – элемент a[1] и т.д. (поскольку каждый элемент типа *int* занимает 4 байта).

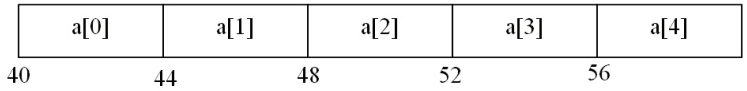


Рисунок 8.1 – Размещение элементов массива

Имя массива – это фактически адрес начала массива (адрес нулевого элемента), т.е. значение a и &a[0] – это одно и то же.

Общее число байт, занятое под массив, можно рассчитать, умножив количество элементов на размер одного элемента, а можно воспользоваться операцией *sizeof( )*, которая выдает количество занятых байт памяти.

```
int a[5];
cout << sizeof(a) << endl; //выводит 20, т.е.5*4
```

При объявлении можно сразу инициализировать элементы массива, если указать знак «=» и список значений элементов в фигурных скобках. Если при этом не указывать количество элементов, то оно будет определено автоматически по количеству инициализирующих значений:

```
int a[]={3,1,5,6}; //объявление и инициализация массива из 4 элементов
```

Если же задать меньше значений, чем размер массива, то будут инициализированы первые значения, а остальные получат значение 0:

```
int b[5]={3,4,2}; //элементы получили значения 3,4,2,0,0
```

Если же инициализирующих значений больше, чем указанная размерность массива, то возникнет ошибка на этапе компиляции:

```
//double f[3]={0.1, 0.3, 0.4, 0.5}; //ошибка
```

8.1.2 Ввод и вывод одномерного массива

При работе с массивами часто нужно перебрать все элементы массива. Для этого удобнее всего использовать цикл *for*. Причем в роли параметра цикла будет выступать индекс элемента в массиве.

Пусть нужно ввести значения всех элементов массива. Самый простой способ – прочитать с консоли вводимые элементы через пробел:

```
const int SIZE=6;
int a[SIZE];
for(int i=0;i<SIZE;i++)
{
     cin>>a[i];
}
```

Обратите внимание, что индекс изменяется от 0 до SIZE-1, поскольку элемента с номером SIZE не существует.

Выводить элементы одномерного массива удобно в строку (если, конечно, этих элементов не очень много). После вывода очередного элемента выводим знак табуляции, а после вывода всех элементов переводим курсор на новую строку:

```
cout<<"Исходный массив:\n";
for(int i=0;i<SIZE;i++)
    cout<<a[i]<<"\t";
cout<<endl;
```

8.1.3 Сумма элементов массива

Сумма накапливается в переменной *sum*. Перебираем все элементы массива и прибавляем каждый из них к сумме:

```
int sum=0;
for(int i=0;i<SIZE;i++)
     sum=sum+a[i];
cout<<"Сумма элементов массива = "<<sum<<endl;
```


8.1.4 Подсчет количества элементов в массиве, удовлетворяющих некоторому условию

Пусть, например, нужно подсчитать количество ненулевых элементов в массиве. Заведем счетчик этих элементов (*kol*), который будем увеличивать на единицу каждый раз, как встречается ненулевой элемент при просмотре массива:

```
int kol=0;
for(int i=0;i<SIZE;i++)
     if(a[i]!=0) kol++;
cout<<"Количество ненулевых элементов= "<<kol<<endl;
```

8.1.5 Поиск максимального значения в массиве

Для нахождения максимального значения в массиве заведем переменную max, в которой будем хранить максимум из просмотренных элементов. Сначала в эту переменную записываем нулевой элемент массива. Потом просматриваем все элементы начиная с первого и сравниваем их с текущим максимумом. Если очередной элемент больше, максимум получает новое значение (старое значение при этом «затирается»). Если же текущий элемент меньше или равен предыдущему, то значение переменной *max* не изменяется:

```
int  max=a[0];
for(int i=1;i<SIZE;i++){
    if(a[i]>max) max=a[i];
}
cout<<"Максимальное значение= "<<max<<"\n";
```

Часто бывает необходимо найти не только само максимальное значение, но и его место в массиве (т.е. индекс максимального элемента). Для того чтобы запомнить индекс максимума, заведем еще одну переменную: imax, которая будет изменяться одновременно с максимумом:

```
max=a[0];
int imax=0; //текущий максимум на месте 0
for(int i=1;i<SIZE;i++){
   if(a[i]>max)
   {
   max=a[i];
   imax=i; //текущий максимум на месте i
   }
}
cout<<"Максимальное значение= "<<max<<"\n";
cout<<"Индекс максимума= "<<imax<<"\n";
```

8.1.6 Генератор случайных чисел в языке C++

Для заполнения массивов большого размера произвольными числами удобно генерировать эти числа случайным образом. Для этого можно использовать функцию *rand( )*, которая в языке C++ содержится в библиотеке *<cstdlib>*.

Функция ***rand( )*** генерирует целое случайное число от 0 до RAND_MAX согласно равномерному закону распределения. Это означает, что вероятность появления любого из чисел диапазона одинаковая. RAND_MAX – это константа, определенная в файле <cstdlib>. В настоящее время она имеет значение 32767.

Пример использования функции *rand( ):*

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "rus");
int x;
x = rand();
cout << "Первое случайное число= " << x << "\n";
x = rand();
cout << "Второе случайное число= " << x << "\n";
system("pause");
    return 0;
}
```

8.1.7 Линейный поиск в массиве

Линейный поиск предполагает перебор всех элементов массива до тех пор, пока не будет найден нужный нам элемент. Пусть, например, необходимо найти первый нулевой элемент в массиве и вывести его индекс. Рассмотрим два способа решения этой задачи. Массив при этом объявлен:

```
const int SIZE = 10;
int a[SIZE];
```

Первый способ состоит в использовании цикла ***while***. Заведем переменную i, которая будет хранить индекс просматриваемого элемента. Заранее мы не знаем, сколько элементов

нужно перебрать до момента, когда встретится нулевой элемент. На самом деле существуют две причины закончить просмотр массива:

1) если все элементы уже просмотрены, а ноль так и не найден, тогда будет нарушено условие i<SIZE;

2) если нашли первый нулевой элемент, тогда будет нарушено условие a[i]!=0.

Таким образом, просмотр элементов продолжается, если выполняется условие (i<SIZE)&&(a[i]!=0). В самом цикле просто увеличиваем индекс текущего элемента:

```
int i=0;
while(i<SIZE&&a[i]!=0) i++;
if(i==SIZE)
    cout<<"В массиве нет нулевых элементов\n";
else
    cout<<"Индекс первого нулевого элемента= "<<i<<"\n";
```

После выхода из цикла проверяем причину окончания просмотра: если весь массив просмотрели, а нужный элемент не нашли, то i==SIZE. Тогда нужно вывести сообщение об отсутствии нулевых элементов в массиве. В противном случае индекс найденного элемента находится в переменной i.

Отметим, что важен порядок условий цикла *while*: сначала проверяется условие, что индекс находится в пределах массива (i<SIZE), и только если это условие выполняется, происходит обращение к элементу a[i] в условии a[i]!=0. *Таким образом, никогда не произойдет обращение к памяти за пределами массива.*

Второй способ – это использование цикла ***for***, который перебирает все элементы массива. Если нужный элемент найден, то используется оператор экстренного выхода из цикла *break*.

Для хранения найденного индекса заведем переменную *inull*. В начале алгоритма присвоим ей такое значение, которое не может быть индексом в массиве (например, –1). Если нужный элемент найден, то переменная *inull* получит значение его индекса. После выхода из цикла также нужно проверить причину окончания перебора: если *inull* так и осталось равно –1, то нужного элемента в массиве не было:

```
int inull = -1; //такого индекса в массиве быть не может
for (int i = 0; i < SIZE; i++) {
if (a[i] == 0) {
inull = i; //запомнить индекс
break;     //выйти из цикла
}
}
if (inull == -1) {
cout << "В массиве нет нулевых элементов\n";
}
else {
cout << "Индекс первого нулевого элемента: " << inull << endl;
}
```

Если далее в программе требуются какие-то действия с участием найденного индекса (например, нужно на место первого нуля записать какое-то число), то при отсутствии такого элемента продолжать программу нет смысла. Поэтому лучше после вывода сообщения об отсутствии найденного элемента произвести задержку экрана (чтобы прочитать это сообщение), а затем прекратить выполнение программы:

```
if (inull == -1) {
cout << "В массиве нет нулевых элементов\n";
        system("pause");
return 0;
}
else {
cout << "Индекс первого нулевого элемента: " << inull << endl;
}
```

8.1.8 Бинарный поиск в массиве

Бинарный (двоичный) поиск применяется только в отсортированном массиве. Он позволяет значительно ускорить процесс поиска.

Допустим, массив упорядочен по возрастанию (неубыванию).

Идея бинарного поиска состоит в том, что массив разбивается пополам и искомое значение сравнивается со срединным элементом. Если искомое значение меньше элемента в середине массива, то дальнейший поиск нужно проводить в левой части массива (поскольку справа все элементы заведомо будут больше). Если же искомое значение больше серединного

элемента, то поиск продолжается в правой части. Таким образом, за один шаг алгоритма область поиска сужается в 2 раза!

Процесс продолжается делением оставшейся части массива пополам и выполнением тех же действий до тех пор, пока не будет найден искомый элемент либо обнаружено его отсутствие.

Обозначим границы области поиска: сначала *low=0* – нижняя граница и *up=N-1* – верхняя граница. Поиск продолжается, пока *low<=up*.

Пусть *middle* – индекс серединного элемента в рассматриваемой части массива (с учетом округления).

Переменная *find* хранит индекс найденного элемента. До начала поиска ей присваивается значение –1. Если по окончании процесса оно таким и останется, то поиск не дал результатов (в массиве нет такого значения).

```
int low=0, up=N-1, middle;
int find=-1; //индекс найденного элемента
do
{
middle=(low+up)/2; //серединный элемент
if(elem==a[middle]) //элемент найден
{
find= middle;
break;
}
if(elem<a[middle]) up=middle-1; //ищем далее в левой половине
if(elem>a[middle]) low=middle+1; //ищем далее в правой половине
}
while(low<=up);
if (find == -1) {
cout << "Искомый элемент не найден\n";
}
else {
cout << "Индекс искомого элемента: " << find << endl;
}
```

8.2 Двумерные массивы

Кроме одномерных массивов (с одним индексом) в языке C++ можно объявить и использовать многомерные массивы (с несколькими индексами). В принципе, количество индексов не ограничено, но наиболее часто используются двумерные массивы.

Можно представлять себе двумерный массив (матрицу) как таблицу, имеющую определенное количество строк и столбцов. При этом первый индекс означает номер строки, а второй – номер столбца. При объявлении массива после имени отдельно в квадратных скобках указывается количество строк, а затем количество столбцов:

```
int a[3][4];  //массив целых чисел из 3-х строк и 4-х столбцов
```

В этом массиве строки нумеруются от 0 до 2, а столбцы – от 0 до 3 (рисунок 8.2).

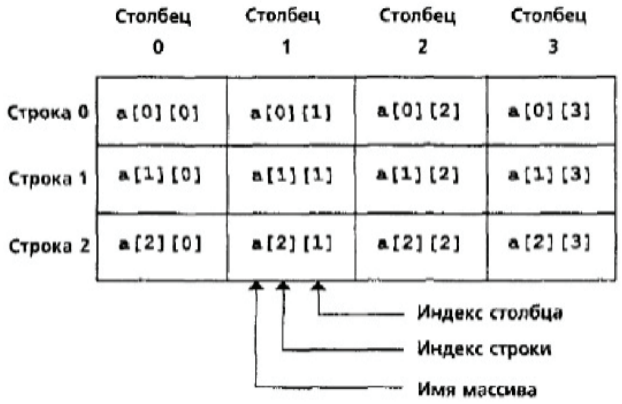


Рисунок 8.2 – Расположение элементов в двумерном массиве

Чтобы обратиться к отдельному элементу двумерного массива, после имени массива каждый индекс указывают в отдельных квадратных скобках:

```
a[2][1]=10; //присваивание значения элементу во второй строке и первом
            // столбце
```

Хороший стиль программирования предполагает, что количество строк и столбцов при описании двумерного массива задается с помощью констант:


```
const int ROW=3;  //под эти константы
const int COL=4;  //выделяется память
int a[ROW][COL];
```

```
#define ROW 3    //команда препроцессора
#define COL 4   //место в памяти не резервируется
int main()
{
     int x[ROW][COL];  //Препроцессор до компиляции
          …           //вместо ROW и COL подставляет 3 и 4
}
```

Описанные двумерные массивы располагаются в автоматической памяти. Элементы такого массива расположены подряд по строкам (последний индекс всегда изменяется быстрее). Можно считать, что двумерный массив – это массив из элементов, которыми являются строки (массив из массивов) (рисунок 8.3).

Рисунок 8.3 – Массив, элементами которого являются строки

Элементы двумерного массива можно инициализировать при объявлении двумя способами:

1) каждая строка заключается в отдельные фигурные скобки:

```
int a1[2][3]={{1,2,3},{4,5,6}};
```

2) значения указываются подряд и построчно вписываются в массив:

```
int a1[2][3]={1,2,3,4,5,6};
```

Недостающие элементы в списках инициализируются нулями:

```
int a2[2][3]={1,2,3,4,5};
```

Массив в результате будет таким:

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

Простой способ инициализировать весь массив нулевыми элементами – указать один ноль в списке инициализирующих значений (остальные нули будут добавлены по умолчанию):

```
int a4[2][3]={0};
```

Как и в случае одномерного массива, нельзя задать больше инициализирующих элементов, чем определено размерностью (компилятор выдаст ошибку).

8.2.1 Типовые алгоритмы работы с двумерными массивами

Построчный вывод двумерного массива:

```
int a[ROW][COL]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
cout<<"Исходный массив:\n";
for (int i=0;i<ROW;i++) { // цикл по строкам
    for(int j=0;j<COL;j++) { //цикл по столбцам
        cout<<a[i][j]<<"\t";
    }
    cout<<"\n"; //перевод курсора после строки
}
```

Определение суммы элементов двумерного массива:

```
int sum=0;
for (int i=0;i<ROW;i++){   //цикл по строкам
   for(int j=0;j<COL;j++) { //цикл по столбцам
      sum+=a[i][j];
   }
}
cout<<"Сумма элементов массива= "<<sum<<"\n";
```

Отметим, что никто не заставляет называть номер строки *i*, а номер столбца – *j*. Это просто традиция, которой рекомендуется придерживаться.

Определение минимального элемента в каждом столбце двумерного массива:


```
for(int j=0;j<COL;j++){ //цикл по столбцам
    int min=a[0][j];   //нач.значение минимума - нулевой элемент столбца
    for(int i=1;i<ROW;i++){   //цикл по строкам
        if(a[i][j]<min) min=a[i][j];
    }
    cout<<"Минимальный элемент в "<<j<<"-м столбце= "<<min<<"\n";
}
```

Для каждого столбца двумерного массива организуется поиск минимального элемента (как в обычном одномерном массиве), а затем вывод этого значения на консоль.

8.2.2 Методы сортировки

Сортировкой называется упорядочивание элементов в массиве данных. ***Ключом*** сортировки называется поле, по которому происходит упорядочивание.

Различные методы сортировки различаются скоростью выполнения, затрачиваемой памятью, эффективностью использования для различных наборов данных.

Рассмотрим три простых метода сортировки:

1) метод выбора;
2) метод обмена;
3) метод вставок.

Метод выбора. *Метод выбора* заключается в том, что из массива выбирается максимальный или минимальный элемент и ставится на нужное место. Пусть, например, необходимо упорядочить массив чисел по возрастанию (неубыванию) методом выбора минимального элемента, т.е. должен быть получен массив, для которого $(a1 \le a2 \le \ldots \le an)$.

Суть алгоритма. В исходном массиве выбирается минимальный элемент. Этот элемент меняется местами с первым элементом массива. Таким образом, один элемент занял свое место в массиве.

Далее рассматривается массив без учета этого первого элемента, и в нем повторяются те же действия (выбирается минимальный элемент и меняется местами со вторым элементом).

Аналогично процесс продолжается и далее, последний раз минимальный элемент выбирается из двух последних элементов массива.

```
for(int k=0;k<N-1;k++) // к - левая граница части массива
{
   int imin=k;// считаем первый элемент подмассива минимальным
   for(int i=k+1;i<N;i++){  // перебираем элементы от k+1 до конца
        if(a[i]<a[imin]){
            imin=i; //обновляем индекс минимума
        }
   }
   //меняем местами a[k] и a[imin]
   int tmp=a[k];
   a[k]=a[imin];
   a[imin]=tmp;
}
```

Данный алгоритм не использует никакой дополнительной памяти. Однако при большом количестве элементов массива время его реализации значительно возрастает. Можно использовать этот алгоритм для массивов небольшого размера.

Метод пузырька. Идея метода пузырька состоит в просмотре массива от начала до конца, в процессе которого сравниваются соседние элементы. Если они стоят не по порядку, то они обмениваются местами. В результате одного просмотра элемент, который должен стоять на последнем месте, там и оказывается («всплывает», как пузырек).

Пусть необходимо упорядочить массив по возрастанию (неубыванию), проходы должны выполняться слева направо (от начала к концу массива).

Суть алгоритма. Сравниваются элементы a[0] и a[1]. Если они не упорядочены (т.е. не выполняется условие a[0]<=a[1]), то меняем местами эти элементы.

Затем сравниваются элементы a[1] и a[2] и при необходимости переставляются. Так просматриваются все пары соседних чисел. Последняя пара будет a[N-2] и a[N-1]. В результате максимальный элемент окажется на последнем месте.

Второй просмотр массива также начинается со сравнения элементов a[0] и a[1]. А последней будет пара a[N-3] и a[N-2] (поскольку последний элемент массива уже учитывать не нужно). В результате на месте предпоследнего элемента окажется второй по величине элемент («всплыл» второй пузырек).

Продолжаются просмотры, причем количество рассматриваемых элементов все время уменьшается. При последнем просмотре будут сравниваться только элементы a[0] и a[1].


```
//k- правая граница части массива
for (int k = N-1; k >0; k--) { //правая граница части массива уменьшается
//просмотреть пары с индексом первого элемента от 0 до k-1
for (int i = 0; i < k; i++) { //i-индекс первого элемента из пары
if (a[i] > a[i + 1]) {
//переставить местами элементы пары
int tmp = a[i];
a[i] = a[i + 1];
a[i + 1] = tmp;
}
}
}
```

Метод вставок. Идея метода вставок состоит в том, что у нас всегда есть два подмассива: один отсортированный (в самом начале он состоит из одного первого элемента), а второй – еще не отсортированный. Из начала второго подмассива берется первый элемент и вставляется в отсортированную часть массива так, чтобы порядок сохранился. И так продолжается до тех пор, пока все элементы не будут перемещены из неотсортированной части в отсортированную.

Пусть, например, по неубыванию упорядочены элементы от a[0] до a[i]: a[0]≤a[1]≤...≤a[i]. Нужно вставить элемент a[i+1] так, чтобы последовательность элементов a[0]...a[i+1] осталась упорядоченной.

Процесс вставки будем проводить, «погружая» вставляемый элемент справа налево до тех пор, пока он не займет свое место.

Для этого заведем вспомогательную переменную tmp, в которой сохраним значение вставляемого элемента. Также заведем текущий индекс движения влево: сначала j=i, а затем уменьшается (использовать переменную i нельзя: она нужна, чтобы продолжать выбор неотсортированных элементов). Проверяем условие a[j]>tmp и в случае его выполнения подвигаем j-й элемент вправо (освобождаем место для вставки). После чего уменьшаем индекс j.

Процесс продолжается до левого края массива или до нахождения места, подходящего для вставляемого значения.

```
//i - индекс последнего из упорядоченных элементов
for (int i = 0; i < N - 1; i++) {
//элемент a[i+1] вставить на нужное место от 0 до i+1
int tmp = a[i+1]; //сохранить копию вставляемого элемента
int j = i;//j - текущий индекс для движения к началу массива
while (j >= 0 && a[j] > tmp) {//пока не вышли за пределы и не найдено место
a[j + 1] = a[j]; //сдвинуть a[j] на место j+1
j--; //продвигаемся к началу
}
//вставка сохраненной копии после найденного индекса  j
a[j + 1] = tmp;
}
```


9 Функции

В упрощении разработки программ важную роль играет принцип ***модульного программирования***. Суть его состоит в том, что программа конструируется из небольших частей, называемых модулями. Преимущества такого подхода:

- маленький фрагмент кода легче написать и отладить;
- модули можно использовать повторно в разных местах программы, это уменьшает объем кода и время разработки;
- модули можно объединить в библиотеки и предоставить возможность их использования другим программистам;
- в целом модульное проектирование повышает качество кода.

В языке C++ модули реализуются с помощью функций.

9.1 Описание функции

Функция – именованный фрагмент кода, который может быть вызван многократно.

Мы уже использовали стандартные функции (например, функции ввода/вывода *printf( )* и *scanf( )*, математические функции *sqrt( ), sin( )* и т.д.) Для их применения нужно подключить соответствующий заголовочный файл библиотеки командой препроцессора #include.

Однако программист может написать и свою функцию, а затем вызывать ее из функции *main* (или из других функций) столько раз, сколько потребуется.

Описание функции состоит из заголовка и следующего за ним тела функции в фигурных скобках. В заголовке указывается тип данных результата, который возвращает функция, имя функции и в круглых скобках список формальных параметров функции (не обязательный):

```
тип_результата имя_функции([список_параметров]){
        … //тело функции
       return выражение;  //возврат результата
}
```

Тело функции – это собственно фрагмент кода, который выполняет некоторую определенную небольшую задачу. Вну-

три тела функции должен быть оператор return, который прекращает работу функции и возвращает некоторое значение в место ее вызова. Тип этого значения должен совпадать с типом результата функции (или неявно к нему преобразовываться).

Пример 9.1

Функция возведения в квадрат может иметь следующий вид:

```
double sqr(double x){
    return x*x;
}
```

Такое описание означает, что в функцию передается один параметр типа *double* (внутри функции он носит имя *x*). Сама функция называется *sqr*. По этому имени ее можно вызывать из других функций. А результат, который она возвращает, имеет тип *double*. Возврат значения происходит в тот момент, когда выполнение кода функции доходит до оператора *return*. Таким образом, если после оператора *return* находятся какие-то другие операторы, то они уже выполнены не будут.

При вызове функции указывается ее имя и список аргументов (иногда их называют фактическими параметрами). Список аргументов может отсутствовать:

```
имя_функции([список_аргументов]);
```

Вызов функции может стоять отдельным оператором. Но можно использовать результат, который возвращает функция, например присвоить его какой-либо переменной, передать на консоль и т.д.

```
int main() {
     double z;
     z=sqr(3.5);  // первый вызов
     cout<<z;
     cout<<sqr(48); //второй вызов
     return 0;
}
```

Операторов *return* в теле функции может быть несколько.

Пример 9.2

Функция signum возвращает 1, если аргумент положителен, –1 – если отрицателен, и 0 – если равен 0.

```
int signum (int x){
   if (x>0) return 1;
   if (x<0) return -1;
   return 0;
}
```

Очевидно, что если в процессе выполнения этой функции окажется, что параметр положительный, то произойдет возврат по первому оператору *return*, а код, который записан далее, игнорируется.

Если тип функции объявлен *void*, то функция никакого значения не возвращает. Внутри такой функции оператор *return* может быть без параметра или вообще отсутствовать. В этом случае возврат в вызвавшую функцию происходит, когда процесс выполнения функции доходит до последней закрывающейся скобки.

Пример 9.3

Функция, которая выводит на печать переданное ей значение:

```
void print(int x){
    cout<<"Результат= "<<x<<"\n";
}
```

Функция должна быть описана в программе до первого ее вызова. Например, если мы хотим использовать функцию возведения в квадрат в функции *main*, то сначала в файле программы размещаем описание функции *sqr*, а потом – функции *main*:

```
#include <iostream>
using namespace std;
double sqr(double x) {
return x*x;
}
int main() {
setlocale(LC_ALL, "rus");
cout << "Квадрат 1.5 = " << sqr(1.5) << endl; //вызов функции
system("pause");
return 0;
}
```

Если же удобнее поместить описание функции после вызова, то можно использовать ***прототип функции*** – заголовок

функции, после которого ставится точка с запятой. При этом в прототипе могут указываться только типы формальных параметров, без имен. Прототип сообщает компилятору интерфейс функции и дает возможность проверить правильность ее вызова (соответствие типа и количества параметров в вызове и в описании функции, соответствие типа результата). Прототип размещается в начале программы (перед первым вызовом функции), а само описание функции может тогда располагаться в любом месте программы.

```
#include <iostream>
using namespace std;
double sqr(double); //прототип функции
int main() {
setlocale(LC_ALL, "rus");
cout << "Квадрат 1.5 = " << sqr(1.5) << endl;// вызов функции
system("pause");
return 0;
}
double sqr(double x) { //описание функции
return x*x;
}
```

Когда в процессе выполнения программы должен быть выполнен вызов функции, выполняются следующие действия:

1) создаются временные переменные для каждого формального параметра, который приведен в заголовке функции (под них выделяется место в памяти);

2) устанавливается соответствие между аргументами в вызове функции (фактическими параметрами) и формальными параметрами в ее заголовке.

Аргументы ставятся в соответствие параметрам по порядку в списке. Типы соответствующих аргументов и параметров должны совпадать.

Сформулируем правила описания функций:

1) функция должна выполнять только ***одну***, небольшую задачу. Хорошо, если объем функции не превышает половины страницы кода;

2) имя функции должно иметь ясный смысл, отражать назначение функции. Например, sumOfArray() – функция для определения суммы всех элементов массива. Если вы не можете выразить суть функции одним словом, то, возможно, она выполняет более чем одну задачу. В этом случае проанализируйте, можно ли ее разбить на несколько функций;

3) если тип результата или параметров не указан, компилятор предполагает *int*:

```
double someFun(double x,y){...} //y имеет тип int
double func(double x, double y){...} //оба параметра double
```

4) все функции в языке C++ являются равноправными. Нельзя описать одну функцию внутри другой.

Пример 9.4

Рассмотрим две функции – возведения в квадрат и печати результата. Используются прототипы этих функций:

```
#include <iostream>
using namespace std;
double sqr(double);
void print(double);
int main() {
setlocale(LC_ALL, "rus");
print(1.5); // первый вызов print
double x;
cout << "Введите вещественное число: ";
cin >> x;
print(x);//второй вызов
system("pause");
return 0;
}
double sqr(double x) {
return x*x;
}
void print(double x) {
cout << "Квадрат " << x << " = " << sqr(x) << endl;
}
```

9.2 Вызов функции и передача параметров

Когда в процессе выполнения программы должен быть выполнен вызов функции, выполняются следующие действия:

1) создаются временные переменные для каждого формального параметра, который приведен в заголовке функции (под них выделяется место в памяти);

2) устанавливается соответствие между аргументами в вызове функции (фактическими параметрами) и формальными параметрами в ее заголовке.

Аргументы ставятся в соответствие параметрам по порядку в списке. Типы соответствующих аргументов и параметров должны совпадать.

В языке C++ по умолчанию ***аргументы в функцию передаются по значению***, т.е. каждый параметр получает временную копию соответствующего аргумента. Таким образом, функция не может изменить оригинальный аргумент в вызвавшей программе;

3) управление передается в функцию. Код функции выполняется до тех пор, пока не встретится оператор return. Записанное в нем значение передается в место вызова, где каким-то образом используется (или игнорируется). При выходе из функции временные копии формальных параметров уничтожаются (память освобождается, они становятся недоступны).

Пример 9.5

Пример вызова функции:

```
#include <iostream>
using namespace std;
//описание функции
double fun(int a, double b){
return a + b;
}
int main(){
int k = 5;
double d, l;
//вызов 1
d = fun(3, 7.6);
cout << "d= " << d << "\n";
//вызов 2
l = fun(k, d);
cout << "l= " << l << "\n";
system("pause");
return 0;
}
```

При вызове функции происходит автоматическое приведение аргументов к соответствующему типу по общим правилам преобразования типов. Например, если вызвать функцию *sqr*, которая была описана ранее, с целым аргументом: *sqr*(4), то целый аргумент автоматически преобразуется к типу формального параметра (*double*), т.е. функция получит вещественное число 4.000.

Приложения

Приложение 1

Список ключевых слов C++

alignas (начиная с C++11)
alignof (начиная с C++11)
and
and_eq
asm
auto(1)
bitand
bitor
bool
break
case
catch
char
char16_t(начиная с C++11)
char32_t(начиная с C++11)
class
compl
const
constexpr(начиная с C++11)
const_cast
continue
decltype(начиная с C++11)
default(1)
delete(1)
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
noexcept(начиная с C++11)
not
not_eq
nullptr (начиная с C++11)
operator
or
or_eq
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_assert(начиная с C++11)
static_cast
struct
switch
template
this
thread_local(начиная с C++11)
throw
true
try
typedef
typeid
typename
union
unsigned
using(1)
virtual
void
volatile
wchar_t
while
xor
xor_eq

Приложение 2

Целочисленные типы языка C++

Типы данных	Выделяемое количество байт памяти	Диапазон значений
char	1	От –128 до 127
unsigned char	1	От 0 до 255
short	2	От –32 768 до 32 767
unsigned short	2	От 0 до 65535
int	4	От –2 147 483 648 до 2 147 483 647
unsigned int	4	От 0 до 4 294 967 296
long	4	От –2 147 483 648 до 2 147 483 647
unsigned long	4	От 0 до 4 294 967 296
long long	8	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
unsigned long long	8	От 0 до $2^{64} - 1$

Приложение 3

Операции языка C++

Операция	Краткое описание
Унарные операции	
++	Инкремент
--	Декремент
!	Логическое отрицание
sizeof	Размер объекта или типа
-	Арифметическое отрицание (унарный минус)
+	Унарный плюс
(<тип>)	Преобразование типа
Бинарные и тернарные операции	
*	Умножение
/	Деление
%	Остаток от деления
+	Сложение
-	Вычитание

Операция	Краткое описание
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно
&&	Логическое «И»
‖	Логическое «ИЛИ»
? :	Условная операция (тернарная)
=	Присваивание
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Остаток от деления с присваиванием
+=	Сложение с присваиванием
-=	Вычитание с присваиванием

Приложение 4

Список некоторых математических функций

Функция	Описание
abs(a)	Абсолютное значение (модуль) a
sqrt(a)	Корень квадратный из a
pow(a,b)	Возведение a в степень b
fmod(a,b)	Вычисление остатка от деления a на b
exp(a)	Вычисление экспоненты
sin(a)	Синус a (a задается в радианах)
cos(a)	Косинус a (a задается в радианах)
log(a)	Натуральный логарифм a
log10(a)	Десятичный логарифм a
asin(a)	Арксинус a
atan(a)	Арктангенс a

Приложение 5

Логические «И», «ИЛИ», «НЕ»

Логическое «И»

a	b	a&&b
true	true	true
true	false	false
false	true	false
false	false	false

Логическое «ИЛИ»

a	b	a‖b
true	true	true
true	false	true
false	true	true
false	false	false

Логическое «НЕ»

a	!a
true	false
false	true

Учебное издание

Ремизова Ольга Игоревна

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ (С++)

Методические указания

Технический редактор *Т.В. Суханова*
Корректор *Е.Н. Леонова*
Компьютерная верстка *А.Л. Бабабекова*

Подписано в печать 03.12.21 Уч.-изд. л. 4,75

Формат 60 × 90 $^{1}/_{16}$

Национальный исследовательский
технологический университет «МИСиС»,
119049, Москва, Ленинский пр-т, 4

Издательский Дом НИТУ «МИСиС»,
119049, Москва, Ленинский пр-т, 4
Тел. 8 (495) 638-44-06

Отпечатано в типографии
Издательского Дома НИТУ «МИСиС»,
119049, Москва, Ленинский пр-т, 4
Тел. 8 (495) 638-44-16, 8 (495) 638-44-43