

Министерство науки и высшего образования Российской Федерации
Ярославский государственный университет им. П. Г. Демидова

О. В. Власова, Н. П. Федотова, О. П. Якимова

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Ярославль
ЯрГУ
2019

УДК 004.42(075.8)
ББК 3973.2-018я73
В58

*Рекомендовано
Редакционно-издательским советом ЯрГУ
в качестве учебного издания. План 2019 года*

*Рецензенты:
кафедра теории и методики обучения информатике ЯГПУ им. К. Д. Ушинского;
д-р физ.-мат. наук, проф. В. Л. Дольников*

Власова, Ольга Владимировна.
В58 **Основы программирования** : учебное пособие / О. В. Власова,
Н. П. Федотова, О. П. Якимова ; Яросл. гос. ун-т им. П. Г. Демидова. — Ярославль : ЯрГУ, 2019. — 84 с.

ISBN 978-5-8397-1180-8

В учебное пособие, состоящее из двух частей, включен основной материал, изучаемый студентами первого курса математического факультета в рамках дисциплин «Основы программирования» и «Информатика». Данное пособие будет полезно также при изучении дисциплин «Практикум по основам программирования», «Практикум по информатике», «Языки программирования», «Практикум по языкам программирования».

В учебном пособии рассматриваются основы синтаксиса языков C# и C++, система типов, основные конструкции языков и базовые структуры данных, различные алгоритмы поиска. Разобрано множество примеров задач различного уровня сложности.

УДК 004.42(075.8)
ББК 3973.2-018я73

ISBN 978-5-8397-1180-8

© ЯрГУ, 2019

Оглавление

Введение	5
1. Типы данных	10
Типы данных	10
Приведение типов и другие операции	15
Перечисляемый тип	17
Математические функции	19
2. Ввод и вывод на консоль	21
Ввод и вывод данных в языке C#	21
Ввод и вывод данных в языке C++	24
3. Операторы ветвления	30
Простой условный оператор	30
Логические операции и сложные условия	31
Вложенный оператор ветвления	34
Оператор выбора	36
Тернарная операция	38
4. Операторы цикла	39
Цикл for	40
Циклы с пред- и постусловием	42
Операторы прерывания цикла	43
Задачи на анализ цифр числа	44

Задачи на обработку последовательности	45
5. Указатели и ссылки	50
Указатели	50
Операции с указателями в C++	52
Ссылки	54
6. Массивы	56
Одномерные массивы	57
Применение датчика случайных чисел	59
Линейный поиск	61
Динамические массивы	65
Сортировка массива	66
Бинарный поиск	69
7. Функции	72
Объявление и определение функции (метода)	72
Вызов функции (метода).	75
Способы передачи параметров	77
Заключение	82
Литература	83



Введение

Программирование – это процесс создания компьютерных программ, он сочетает в себе элементы искусства, науки, математики и инженерии. Работа программиста интересна, трудна и востребована в современном мире. Это учебное пособие предназначено для студентов первого курса математического факультета, делающих первые шаги в освоении искусства написания красивого кода.

В пособии рассматриваются основы синтаксиса языков C# и C++ ([5],[6]), система типов, основные конструкции языков и базовые структуры данных, различные алгоритмы поиска и сортировки, приводятся полезные ссылки. Разобрано множество примеров задач различного уровня сложности. Поэтому пособие может быть полезно студентам других факультетов, начинающим изучать языки программирования C# и C++.

Особенностью данного пособия является изучение одновременно двух языков программирования. Это обусловлено разными причинами, в том числе тем, что языки имеют схожий синтаксис. Оба языка регистрозависимы, операторы присваивания, арифметические операторы, конструкции ветвления и цикла устроены практически одинаково. Однако они имеют и существенные отличия.

Язык C# :

- безопасный и надежный, есть автоматическая сборка мусора (управление памятью);
- код программы легко читаем;
- полностью объектно-ориентированный;
- современный, идеально подходит Visual Studio, т. к. создавался параллельно с ней.

Язык C++:

- быстрый, эффективный, поддерживает указатели и адресную арифметику;
- часто код программы короткий и красивый;
- поддерживает объектно-ориентированное и обобщенное программирование.

Разрабатывать программы на языках C# и C++ можно с помощью разных сред разработки. Например, можно использовать codeblocks или онлайн компиляторы языков. У каждого из них есть свои достоинства и недостатки. Данное пособие в основном описывает синтаксис языков C# и C++ и подходит для любой среды разработки. Однако первый пункт – программа "Hello, World!" и некоторые другие примеры – рассчитаны на использование MS Visual Studio версии

community, доступной для скачивания и использования в некоммерческих целях на официальном сайте: <https://visualstudio.microsoft.com/ru/vs/community/>. Данная среда была выбрана по ряду причин:

- она является профессиональным, мощным, современным, часто используемым средством разработки программного обеспечения;
- поддерживает оба рассматриваемых языка программирования;
- используется при изучении специальных дисциплин на более старших курсах.

Следует обратить внимание на то, что при создании приложений, в зависимости от версии Visual Studio и выбранного типа приложения, автоматически сгенерированный код меняется. Особенно значимыми оказываются изменения для языка C++. Поэтому код программы, приведенный в пособии, может несколько отличаться от того кода, который будет соответствовать более новой версии среды разработки. Это может доставить читателю определенные неудобства. За это авторы просят их простить, а также с благодарностью примут все замечания.

Программа "Hello, World!"

Часто изучение языка начинают с программы, которая выводит сообщение "Hello, world!", или с какой-то другой простейшей программы. Так делают для того, чтобы объемный вводный теоретический материал проиллюстрировать примером и иметь возможность добавлять к примеру новые детали. Надеемся, что читатель тоже напишет первую программу на изучаемом языке программирования и будет модифицировать ее, стараясь понять очередной теоретический материал.

Оформление кода в языках C# и C++ производится практически одинаковым образом. Содержимое (или тело) пространства имен, класса, метода или оператора заключается в фигурные скобки { }, что позволяет структурировать код. Для пояснения текста программ используются комментарии, которые могут быть однострочными и многострочными:

- однострочный комментарий начинается с двух символов косой черты. Все, что находится в однострочном комментарии — от // до конца строки — игнорируется компилятором;
- многострочный комментарий расположен между /* и */. Многострочные комментарии не могут быть вложены друг в друга;
- в C# есть еще специальные комментарии, начинающиеся с трех символов косой черты. Они необходимы для генерации документации к коду.

Создадим консольное приложение на языке C# в среде разработки Visual Studio. Назовем его Hello. На экране мы увидим следующий код:

```
using System;
namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

В первой строке кода подключается пространство имен **System**, в котором определены основные типы данных и класс, осуществляющий консольный ввод-вывод (то есть ввод данных с клавиатуры и вывод на экран). Ниже, в пространстве имен **Hello**, которое соответствует имени приложения, описан класс **Program**, а в нем – статический метод **Main**, с которого начинается исполнение программы. Для вывода на экран строки "Hello, world!" внутрь метода **Main** добавим оператор **Console.WriteLine** и передадим ему эту строку. Первая программа готова:

```
using System;
namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            // вывод на экран сообщения
            Console.WriteLine("Hello, world!");
            // задержка экрана до нажатия клавиши
            Console.ReadKey();
        }
    }
}
```

Запуск программы на выполнение в среде Visual Studio производится нажатием клавиши F5.

Теперь выполним аналогичную работу на языке C++. Создадим пустой проект C++ в среде разработки Visual Studio. Назовем его hello. Добавим к проекту файл исходного кода программы. Для этого в меню выберем пункт «Проект», в нем выберем "Добавить новый элемент". В появившемся окне выберем "Файл C++". Поскольку проект пустой, ничего сгенерировано не будет. Набираем код программы:

```
#include <iostream> // подключаем заголовочный файл для ввода-вывода
using namespace std; // открываем пространство имен std

int main() { // исполнение программы начинается с функции main
    // вывод приветствия и переход на новую строку
```

```
cout << "Hello, world!" << endl;
system("pause"); // задержка экрана
return 0; // функция main возвращает значение, равное нулю
}
```

Как и в C#, для запуска программы на выполнение необходимо нажать клавишу F5.

Заметим, что:

- для вывода используется предопределенный стандартный выходной поток **cout**;
- функция **main** возвращает значение 0(**return 0**;). Это код нормального завершения программы. Когда язык C только создавался, функция **main** должна была возвращать код ошибки, если она возникла. Чем больше значение, тем серьезнее ошибка.

Переменные

Для хранения данных в программе используются переменные. Переменная – это именованная область памяти, в которой хранится значение определенного типа, то есть переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная (подробно про типы данных рассказывается в следующей главе).

Перед использованием любую переменную надо описать. Синтаксис описания переменной выглядит следующим образом:

```
<тип> <имя_переменной>;
```

Именем переменной может быть любая последовательность символов, которая удовлетворяет следующим требованиям:

- имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания. Наличие пробелов или не буквенно-цифровых символов в имени приведет к ошибке;
- имя не может быть ключевым словом языка C# (C++) – **for**, **if**, **class**, **switch**, **while** и т. д. Таких слов немного, и при работе в Visual Studio среда разработки подсвечивает ключевые слова другим цветом.

Языки программирования C# и C++ являются регистрозависимыми, поэтому, допустим, **number** и **Number** – это две различные переменные.

Правила хорошего стиля программирования предполагают, что имя переменной должно быть содержательным, то есть отражать ее предназначение.

Например, определим простейшую переменную целого типа:

```
int number;
```

В дальнейшем мы сможем присвоить этой переменной только то значение, которое соответствует ее типу, то есть в данном случае целое число. С помощью имени переменной мы сможем обращаться к той области памяти, в которой хранится

ее значение. Кроме того, мы можем сразу при описании присвоить переменной значение. Данный прием называется инициализацией: `int number = 12;`

В программе можно многократно менять значения переменных. Изменим нашу первую программу, добавив туда описание, изменение и вывод переменной.

На языке C#:

```
using System;
namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            int number = 12; // описание и инициализация переменной
            number = number + 8; // изменение переменной
            Console.WriteLine(number); // вывод значения на экран
            // задержка экрана до нажатия клавиши
            Console.ReadKey();
        }
    }
}
```

На языке C++:

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

int main()
{
    int number = 12; // описание и инициализация переменной
    number = number - 2; // изменение переменной
    cout << number << endl; // вывод значения на экран
    system("pause"); // задержка экрана
    return 0;
}
```



Глава 1.

Типы данных

Основная цель любой программы состоит в обработке данных. Данные программы хранятся в ячейках оперативной памяти. Компилятору для формирования машинных команд необходимо точно знать, сколько места занимают данные, как именно они закодированы и какие операции с ними можно выполнять. Все это задается при описании данных с помощью типа.

1.1. Типы данных

Каждая константа, переменная, результат вычисления выражения или функции должны иметь конкретный тип.

Тип данных однозначно определяет:

- объем выделяемой памяти;
- внутреннее представление данных в памяти компьютера;
- множество возможных значений (связанное с внутренним представлением данных в памяти компьютера);
- допустимые действия над данными (операции и функции).

Программист задает тип каждой переменной, используемой в программе, исходя из характеристик реальных объектов, которые представляют эти переменные. Обязательное указание типа позволяет компилятору проверять, правильно ли используется объект в конструкциях языка.

Память, которую используют программы, состоит из нескольких *сегментов*:

- сегмент, где во время выполнения находится скомпилированная программа, – *сегмент кода* (только чтение);
- сегмент, где хранятся параметры функций, локальные переменные, адреса возврата и многое другое, связанное с функциями, – *стек вызовов* или просто *стек*;
- сегмент, где программист может получить дополнительную память уже во время работы программы (динамические переменные), – *куча*;
- сегмент, где хранятся глобальные и статические переменные, – *сегмент данных*.

В этом пособии мы кратко рассмотрим последние три сегмента.

Стек (stack) работает по принципу LIFO (Last In, First Out), то есть последний добавленный в стек элемент будет первым в очереди на освобождение. Каждый раз, когда мы внутри функции или блока объявляем новую переменную, она добавляется в стек, когда функция завершает свою работу, переменная автоматически удаляется из стека, а занимаемая ею область памяти становится доступной для других стековых переменных. Такой принцип управления памятью обеспечивает высокую скорость доступа к данным. Время цикла обновления ячейки стека очень мало, так как она чаще всего привязана к кэшу процессора. К недостаткам работы со стеком можно отнести его фиксированный размер. Размер стека задаётся при создании потока, превышение объёма выделенной на стеке памяти приведёт к его переполнению. Переменные, расположенные на стеке, всегда являются локальными. Использование стека для передачи параметров в подпрограммы обсуждается в главе 7 данного пособия.

Куча (heap) предназначена для динамического выделения памяти в процессе работы приложения. Когда в куче выделяется участок памяти для хранения данных, то обратиться к этому участку можно не только в потоке, но и во всем приложении. По завершении приложения все выделенные участки памяти в куче освобождаются. Размер кучи задаётся при запуске приложения, но, в отличие от стека, он ограничен лишь физически.

Взаимодействие с кучей выполняется с помощью указателей в C++ и ссылочных переменных в C#. Инициализируя указатель (ссылочную переменную), вы указываете на местоположение данных в куче и говорите программе, как получить доступ к этим данным. В языке C# предусмотрен автоматический сборщик мусора, поэтому разработчику не нужно вручную освобождать участки памяти, которые больше не нужны. В языке C++ программист должен сам следить за освобождением выделенных участков памяти. В сравнении со стеком куча работает медленнее, так как переменные могут быть разбросаны по несмежным участкам памяти.

Сегмент данных предназначен для хранения переменных, которые объявлены вне всякого блока, а также для статических переменных. Эти переменные создаются при запуске программы, а уничтожаются при её завершении.

Типы языка C++ можно разделить на две группы: **элементарные** (базовые, основные) и **составные**.

Элементарные типы данных являются неделимыми и позволяют описывать целые (*int*), вещественные (*float*, *double*), символьные (*char*) и логические (*bool*) величины, а также тип *void*. На основе этих типов программист может конструировать составные типы. **Составной тип** – это тип, определенный в терминах другого типа. К составным типам относятся массивы, структуры, объединения, перечисления, ссылки, указатели и классы. Внутреннее представление данных всех типов, их размеры и диапазоны величин определяются конкретной платформой. Вещественные типы часто называют типами с плавающей точкой. Код, который формирует компилятор для обработки целых величин, отличается от кода, формируемого для величин с плавающей точкой.

Ключевые слова *short* (короткий), *long* (длинный), *signed* (знаковый) и *unsigned* (беззнаковый) уточняют внутреннее представление и диапазон значений стандартных типов.

При написании переносимых на различные платформы программ нельзя делать предположений о размере типа. Для его получения необходимо пользоваться операцией *sizeof()*, результатом которой является размер типа в байтах.

В стандарте ANSI диапазоны значений для основных типов не задаются, определяются только соотношения между их размерами, например:

$$\begin{aligned} \text{sizeof(float)} &\leq \text{sizeof(double)} \leq \text{sizeof(long double)} \\ \text{sizeof(char)} &\leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)} \end{aligned}$$

Минимальные и максимальные допустимые значения для целых типов зависят от реализации и приведены в заголовочном файле `<limits.h>` (`<climits>`), характеристики типов с плавающей точкой — в файле `<float.h>` (`<cfloat>`).

В C# имеются две общие категории встроенных типов данных: **типы значений** и **ссылочные типы**. Разница между ними состоит в том, что тип значения (*value type*) хранит данные непосредственно, в то время как ссылочный тип (*reference type*) хранит ссылку на значение. Когда вы присваиваете новое значение переменной типа значения, это значение копируется. Когда вы присваиваете новое значение переменной ссылочного типа, копируется не сам объект, а ссылка на него.

Эти типы сохраняются в разных местах памяти: типы значений сохраняются в **стеке**, а ссылочные типы — в **куче**. К типам значений относятся целые, вещественные, символьные и логические величины, структуры. К ссылочным типам — массивы, строки, классы. Заметим, что все типы данных, относящихся к типам значений, на самом деле являются структурами, а ссылочные типы — классами.

Для представления вещественных чисел используются два формата:

с фиксированной точкой: [знак][целая часть].[дробная часть]

Например: -8.13; .168 (аналогично 0.168); 183. (аналогично 183.0).

с плавающей точкой (экспоненциальной форме): [знак] мантисса E/e порядок

Например: 3.535e+01 ($3.535 \times 10^1 = 35.35$); -1.7E-03 ($-1.7 \times 10^{-03} = -0.0017$)

В большинстве случаев следует использовать тип *double*, он обеспечивает более высокую точность, чем тип *float*. Максимальную точность и наибольший диапазон значений достигается с помощью типа *long double* в языке C++ и *decimal* в языке C#.

Обращаем внимание, что константы с плавающей точкой имеют по умолчанию тип *double*. Если требуется явно указать тип константы, используйте F, f для *float* и L, l для *long*, в C# для типа *decimal* — M, m. Например, константа 56.8E+6F будет иметь тип *float*, а константа -10.78M — тип *decimal*.

Таблица 1.1.

Диапазон значений и объем памяти основных типов Microsoft C++

C++		
Тип данных	Диапазон	Размер в байтах
bool	true и false	1
signed char	-128 ... 127	1
unsigned char	0 ... 255	1
signed short int	-32 768 ... 32 767	2
unsigned short int	0 ... 65 535	2
signed int	-2 147 483 648 ... 2 147 483 647	4
unsigned int	0 ... 4 294 967 295	4
signed long int	-2 147 483 648 ... 2 147 483 647	4
unsigned long int	0 ... 4 294 967 295	4
long long	-9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807	8
unsigned long long	0 ... 18 446 744 073 709 551 615	8
float	$\pm 3.4e - 38 \dots \pm 3.4e + 38$	4
double	$\pm 1.7e - 308 \dots \pm 1.7e + 308$	8
long double	$\pm 3.4e - 4932 \dots \pm 3.4e + 4932$	10

Таблица 1.2.

Диапазон значений и объем памяти основных типов Microsoft C#

C#				
Ключевое слово	Тип данных	Что это?	Диапазон	Размер в байтах
bool	System.Boolean	структура	true и false	1
byte	System.Byte	структура	0 ... 255	1
char	System.Char	структура	Символ Юникода	2
decimal	System.Decimal	структура	$\pm 1.0 \times 10^{-28} \dots \pm 7.9228 \times 10^{1028}$	16
double	System.Double	структура	$\pm 5.0 \times 10^{-324} \dots \pm 1.7 \times 10^{308}$	8
float	System.Single	структура	$\pm 1,5 \times 10^{-45} \dots \pm 3,4 \times 10^{38}$	4
int	System.Int32	структура	-2 147 483 648 ... 2 147 483 647	4
long	System.Int64	структура	-9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807	8
object	System.Object	класс		—
sbyte	System.SByte	структура	-128 ... 127	1
short	System.Int16	структура	-32 768 ... 32 767	2
string	System.String	класс		—
uint	System.UInt32	структура	0 ... 4 294 967 295	4
ulong	System.UInt64	структура	0 ... 18 446 744 073 709 551 615	8
ushort	System.UInt16	структура	0 ... 65 535	2

Тип *void* также относится к основным типам данных, его множество значений пусто. Объявить переменную такого типа нельзя. Обычно этот тип используют, если необходимо показать, что функция не возвращает значения. Еще одно применение *void* – это базовый тип для указателей и целевой тип в операциях приведения типов (C++).

Рассмотрим общий вид оператора описания переменных:

[const] <тип> <имя> [<инициализатор>];

– ключевое слово *const* применяется к типу данных, показывая, что данные неизменяемы. Такую переменную называют именованной константой, или просто константой;

– с помощью инициализатора (константа или выражение) для переменной или константы можно задать начальное значение.

Заметим, что каждая константа должна обязательно быть проинициализирована при объявлении, а каждая переменная должна получить значение перед своим использованием (путем инициализации, присваивания или чтения).

Инициализатор можно записывать, используя знак равенства (оба языка) или в круглых скобках (только C++)(пример см. ниже). Выражение должно быть вычисляемым на момент выполнения оператора.

```
int iValue; // без инициализации,
long lValue; // значение не определено

int iValue2 = 0; // инициализация константой
bool flag = true, isFound = false;
float fValue = 2.5F;
int iValue3(-100); // инициализация константой (только C++)

int sum = iValue2 + iValue3; // инициализация выражением
const double delta = 0.0001; // константа
```

Та часть программы, в которой имя переменной можно использовать для доступа к связанной с ним области памяти, называется *областью действия имени*. В зависимости от области действия переменная может быть *локальной* или *глобальной*. Если переменная определена внутри тела цикла, внутри составного оператора или внутри тела функции, т. е. внутри блока (блок ограничен фигурными скобками), она называется локальной, область ее действия – от точки описания до конца блока, включая все вложенные блоки. Если переменная определена вне любого блока, она называется глобальной и область ее действия считается модуль, в котором она определена, от точки описания до его конца. Разделение переменных на глобальные и локальные соответствует одному из главных правил программирования, а именно принципу наименьших привилегий, то есть переменные, объявленные внутри блока, должны быть доступны только в нем.

Каждое имя переменной имеет свою *область видимости* – это область, в которой мы можем работать с этой переменной. Чаще всего область видимости совпадает с областью действия. Исключением является ситуация, когда во вложенном блоке определена переменная с таким же именем, что и у переменной вне блока. В этом случае внешняя переменная во вложенном блоке невидима, хотя он и входит в ее область действия.

Имя переменной должно быть уникальным в своей области видимости.

Оператор присваивания обозначается одиночным знаком равенства (=).

<имя_переменной> = <выражение>;

Здесь тип переменной должен быть совместим с типом выражения.

Оператор присваивания, как и любой другой оператор, имеет возвращаемое значение. Оно равно значению выражения, стоящего справа. Поэтому у оператора присваивания имеется одна интересная особенность: он позволяет создавать цепочку операций присваивания. Рассмотрим следующий фрагмент кода:

```
int valueX, valueY, valueZ;  
valueX = valueY = valueZ = 100;
```

В этом фрагменте кода одно и то же значение 100 заносится в переменные *valueX*, *valueY* и *valueZ* с помощью единственного оператора присваивания. Такой способ удобен для задания общего значения группе переменных.

1.2. Приведение типов и другие операции

В программировании часто возникают ситуации, когда необходимо значения переменных одного типа присвоить переменным другого типа.

```
int iValue = 10;  
float fValue; fValue = iValue;
```

В приведенном фрагменте кода целое значение типа *int* присваивается переменной с плавающей точкой типа *float*.

Если в одной операции присваивания смешиваются *совместимые* типы данных, то значение в правой части оператора присваивания автоматически преобразуется в тип, указанный в левой его части. Поэтому в приведенном выше фрагменте кода значение переменной *iValue* сначала преобразуется в тип *float*, а затем присваивается переменной *fValue*.

Всегда ли возможно преобразование типов?

Вследствие выполнения контроля типов далеко не все типы данных оказываются полностью совместимыми и, следовательно, не все преобразования типов разрешены в неявном виде.

Например, типы *bool* и *int* совместимы в C++ (0 – ложь, != 0 – истина), но несовместимы в C#. Правда, преобразование несовместимых типов все-таки может быть осуществлено путем выполнения операции *приведения*. Приведение типов, по существу, означает явное их преобразование.

Автоматическое (неявное) приведение типов происходит при следующих условиях:

- 1) оба типа совместимы;
- 2) диапазон представления чисел целевого типа шире, чем у исходного типа.

Если оба эти условия выполняются, то происходит так называемое *расширяющее преобразование*. Например, тип *int* достаточно крупный, чтобы вмещать в себя все значения типа *short*, а кроме того, оба типа, *int* и *short*, являются совместимыми целочисленными типами и поэтому для них вполне возможно неявное преобразование. Числовые типы, как целочисленные, так и с плавающей точкой, совместимы друг с другом для выполнения расширяющих преобразований.

Приведение несовместимых типов. Несмотря на всю полезность неявных преобразований типов, они не способны удовлетворить все потребности в программировании, поскольку допускают лишь расширяющие преобразования совместимых типов. Во всех остальных случаях приходится выполнять явное приведение типов. Явное приведение — это команда компилятору преобразовать результат вычисления выражения в указанный тип. Общая форма оператора приведения типов:

(**<целевой_тип>**) **<выражение>**;

Здесь **<целевой_тип>** обозначает тот тип, в который желательно преобразовать указанное выражение.

Если приведение типов приводит к *сужающему преобразованию*, то часть информации может быть потеряна. Например, в результате приведения типа *long* к типу *int* старшие разряды этого числового значения отбрасываются. Когда же значение с плавающей точкой приводится к целочисленному типу, то в результате усечения теряется дробная часть этого числового значения.

Таблица 1.3.

Таблица основных операций и операторов C++ и C#

Операции		
Приведение типа	(<целевой_тип>) <выражение>	double dValue = 7.98845; int iValue = (int)dValue;
Простое присваивание	<имя_переменной> = <выражение>;	int num = 11; num = num*25 - 14;
Составное присваивание	+=, -=, *=, /= %=, =, >>=	int a = 3, b = 6, c = 10; a += b; // a = 9 b %= a; c >>= 1; // b = 6, c = 5
Арифметические операции	++ (приращение, инкремент) -- (уменьшение, декремент) + (сложение) и - (вычитание) % (остаток от деления) * (умножение), / (деление)	int a = 3; int b = a++; // b = 3, a = 4 a = 5; a++; int c = --a; // c = 5, a = 5 c = c + a - 11; a = 8 % 2; // a = 0 a = b * (1 + a) / 2;
Логические операции	! (логическое отрицание) && (логическое И) и (логическое ИЛИ)	bool f = !(a < b); bool f = (x > 0) (y < 0);
Поразрядные	& (логическое И), (логическое ИЛИ) ^ (логическое исключающее ИЛИ)	int a = 5 2; // a = 7 int a = 7 ^ 2; // a = 5
Сдвиг	>> (сдвиг вправо, деление на 2^k) << (сдвиг влево, умножение на 2^k)	int a = 17 >> 2; // a = 4 int a = 17 << 2; // a = 68
Сравнение	< (меньше), > (больше) <= (меньше или равно) >= (больше или равно), == (равно)	bool f = (5 == (10 >> 1));

Действие арифметических операций не требует особых пояснений, за исключением деления. Когда операция **/** применяется к целому числу, то любой остаток от деления отбрасывается, результатом будет целое число. Например, $5/2$ равно 2. Для получения остатка от деления применяется операция **%**: $5 \% 2$ равно единице. В C# операцию **%** можно применять как к целочисленным типам данных, так и к типам с плавающей точкой, в C++ только к целочисленным типам данных. Например, следующий код будет верен для обоих языков:

```
int number = 17;
int result = number / 3;    // result = 5
int mod = number % 3;      // mod = 2
```

Пример ниже демонстрирует применение остатка от деления для чисел с плавающей точкой (C#):

```
float number = 17.5f;
float result = number / 3.0f;    // result = 5.83333
float mod = number % 3.0f;      // mod = 2.5
```


1.3. Перечисляемый тип

Перечисляемый тип – это тип данных, чьё множество значений представляет собой ограниченный список именованных констант, называемых *перечислителями*. Перечисления определяются с помощью ключевого слова **enum**, которое указывает на начало перечисляемого типа.

Стандартный вид перечислений следующий:

```
enum <имя_перечисления> {список перечислений} <список переменных>;  
enum <имя_перечисления> { список перечислений }; // C#
```

Определим простейшее перечисление:

```
enum Colors  
{  
    WHITE,  
    RED,  
    BLUE  
};
```

Каждому перечислителю автоматически присваивается целочисленное значение в зависимости от его позиции в списке перечисления. По умолчанию, первому перечислителю присваивается целое число 0, а каждому следующему — на единицу больше, чем предыдущему. Программист может сам определять значения перечислителей. Они могут быть как положительными, так и отрицательными. Любые неопределённые перечислители будут иметь значения на единицу больше, чем значения предыдущих перечислителей. Мы можем определить переменную типа *enum* и присвоить этой переменной значение одной из констант, объявленной в перечислении. Но фактически это будет числовое значение. В то же время перечисления – это отдельный тип, поэтому мы не можем присвоить переменной напрямую числовое значение. Поскольку значениями перечислителей являются целые числа, то их можно присваивать целочисленным переменным. В C++ в этом случае будет выполнено неявное преобразование, однако в C# необходимо явное приведение, чтобы преобразовывать из типа *enum* в целочисленный тип. При выводе на консоль в C++ мы увидим числовой эквивалент, а в C# – именованную константу.

Рассмотрим это на примерах.

В примере 1 объявлено перечисление *Seasons*. Один перечислитель явно преобразуется в целое число и присваивается целочисленной переменной. Описана переменная *season* типа *Seasons* и показан её вывод на консоль (C#).

```
public class Enum  
{  
    enum Seasons { Winter = 1, Spring, Summer, Autumn };  
  
    static void Main()  
    {  
        int y = (int)Seasons.Spring;  
        Console.WriteLine("Spring = {0}", y); // вывод: Spring = 2
```

```

    Seasons season = Seasons.Summer;
    Console.WriteLine(season); // вывод: Summer
    Seasons s1 = 3; // Ошибка
    Console.ReadKey();
}
}

```

В примере 2 три арифметические операции (сложение, вычитание и умножение) объявлены в перечислении *Operations*. В зависимости от выбранной пользователем операции в конструкции *switch* производятся определенные действия (C++):

```

#include <iostream>
using namespace std;
enum Operations
{
    ADD = 1,
    SUB = 2,
    MUL = 4
};
int main()
{
    int operation;
    int number1, number2;
    int result;
    bool isValue = true;
    cout << "Input number1:";
    cin >> number1;
    cout << "Input number2 :";
    cin >> number2;
    cout << "ADD: 1" << endl << "SUB: 2" << endl << "MUL: 4" << endl;
    cout << "Input operation number: ";
    cin >> operation;
    switch (operation)
    {
        case Operations::ADD :
            result = number1 + number2;
            break;
        case Operations::SUB :
            result = number1 - number2;
            break;
        case Operations::MUL :
            result = number1 * number2;
            break;
        default :
            cout << "Invalid operation " << endl;
            isValue = false;
    }
}

```

```

    }
    if(isValue)
        cout << "Result = " << result << endl;
        system("pause");
        return 0;
}

```

Использование перечислений позволяет сделать исходные коды программ более читаемыми, так как позволяют заменить числа, кодирующие определённые значения, на читаемые имена. Перечисляемый тип может использоваться в объявлениях переменных и формальных параметров функций (методов). Практически всегда поддерживается сравнение значений перечислимого типа. Результат сравнения двух перечислимых значений определяется порядком следования этих значений в объявлении типов: значение, которое в объявлении типа встречается раньше, считается меньше значения, встречающегося позже.

1.4. Математические функции

Практически в любом языке программирования есть встроенные функции для вычисления логарифмов, косинусов, степеней с произвольными показателями. Такие методы в языке C# предоставляет класс *Math*, а в C++ математические функции определяются в заголовочном файле `<cmath>`.

При использовании математических функций и методов нужно знать несколько нюансов:

1. Результат будет вещественный, даже если параметры целые. Например, при возведении числа 5 в квадрат с помощью функции *pow* мы получим вещественный результат, который необходимо будет явно преобразовать в целое число:

```

// C++
int r = pow(5, 2); // неявное преобразование типов
// C#
int r = (int) Math.Pow(5, 2);

```

2. Аргументы тригонометрических функций задаются в радианах. Для того, чтобы вычислить синус 45°, сначала нужно 45° преобразовать в радианы и только после этого считать синус:

```

// C++
// для работы с константами, помещаем до подключения cmath
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double degrees = 15.0;

```

```

double radian = degrees * M_PI / 180;
double sinAngle = sin(radian);
cout << "sin = " << sinAngle << endl;
system("pause");
return 0;
}

```

Аналогичный код на C#

```

double degrees = 15.0;
double angle = Math.PI * degrees / 180.0;
double sinAngle = Math.Sin(angle);
double cosAngle = Math.Cos(angle);

```

Задача 1.1. Для точек $p \in [0; 5]$ с шагом 0.2 вычислить значение $y = \sqrt{t}$ (C++).

```

#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    for (double p = 0; p <= 5; p += 0.2)
    {
        cout << p << '\t';
        cout << sqrt(p) << endl;
    }
    system("pause");
    return 0;
}

```

Задача 1.2. Дано два числа a и b . Найдите гипотенузу треугольника с заданными катетами (C#).

```

using System;
namespace Hypotenuse
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = int.Parse(Console.ReadLine()); // ввод длины первого катета
            int b = int.Parse(Console.ReadLine()); // ввод длины второго катета
            double c = Math.Sqrt(a * a + b * b); // вычисление длины гипотенузы
            // вызов метода Sqrt класса Math, возвращающего тип double
            Console.WriteLine("Гипотенуза равна {0:f2}", c);
            Console.ReadKey();
        }
    }
}

```



Глава 2.

Ввод и вывод на консоль

Ввод и вывод данных в языках C# и C++ имеют общие черты, а именно:

- 1) Осуществляется потоковый ввод и вывод (поток – последовательность символов, которая записывается на устройство ввода-вывода или считывается с него);
- 2) Применяются управляющие последовательности. Например, последовательность `'\n'` интерпретируется как перевод на новую строку, `'\t'` – табуляция. Если именно эти символы надо напечатать, то необходимо удвоить обратную косую черту, т. е. вместо `'\n'` написать `"\\n"`;
- 3) Есть возможность форматированного вывода. Полноценный форматированный вывод есть в C#, форматированный вывод в C++ основан на функциях языка C. Далее мы разберем его подробно.

2.1. Ввод и вывод данных в языке C#

Для вывода информации на экран мы уже использовали метод **WriteLine()**, который передает в поток вывода строку вместе с символом перевода строки. Существует еще метод **Write()** того же класса **Console**, который выставляет в поток вывода текст без перевода строки. Например, выполнение фрагмента кода:

```
Console.Write("Как тебя зовут? ");  
string name = "Вася";  
Console.Write(name);
```

приведет к появлению на экране слов:

```
Как  тебя зовут? Вася
```

Для вывода нескольких значений переменных используют прием, который называется интерполяцией. Перед выводимой строкой помещается знак доллара, внутри строки имена переменных заключаются в фигурные скобки, например:

```
string color = "красный";  
int      size = 44;  
double height = 1.7;  
Console.WriteLine($"Костюм цвет: {color}, размер: {size}, рост: {height} м.");
```

При выводе на консоль вместо помещенных в фигурные скобки выражений будут выводиться их значения, а именно:

Костюм цвет: красный, размер: 44, рост: 1,7 м.

Существует и другой способ – через метки-заполнители. Будем предполагать, что переменные определены так же, как и выше. Тогда для вывода их значений на экран используем команду:

```
Console.WriteLine("Костюм цвет:{0}, размер:{1}, рост:{2}м.", color, size, height);
```

Этот способ подразумевает, что первый параметр в методе **Console.WriteLine()** представляет выводимую строку ("Костюм цвет: {0} размер: {1} рост: {2} м.") с метками-заполнителями. Все последующие параметры представляют переменные, значения которых встраиваются в выводимую строку. При этом важен порядок параметров. Вместо метки **{0}** вставляется значение параметра **color**, он самый первый после строки, а нумерация начинается с нуля, вместо **{2}** вставляется значение параметра **height**.

В метки-заполнители можно включить различные символы форматирования в виде суффикса после двоеточия, например **{0:f2}**. Наиболее полезные символы форматирования приведены в таблице ниже.

Таблица 2.1.

Символы для форматирования числовых данных C#

Символ форматирования	Описание
D или d	Применяется при выводе целых чисел, указывает минимальное количество цифр
E или e	Применяется для экспоненциального представления числа, указывает количество десятичных разрядов после запятой
F или f	Применяется для представления чисел с фиксированной точкой, указывает количество десятичных разрядов после запятой
X или x	Применяется для представления числа в шестнадцатеричном формате

Приведем пример форматированного вывода.

```
Console.WriteLine("Целое число: {0} \t {0:d4} \t {0:f2}", 123);
Console.WriteLine("Действ. число: {0} \t {0:f1} \t {0:e4}", 123.456);
```

На экране увидим:

```
Целое число:    123      0123      123,00
Действ. число: 123,456    123,4      1,2346e+002
```

Для ввода данных используется метод **Console.ReadLine()**. Он позволяет получить введенную информацию из потока ввода вплоть до нажатия клавиши Enter, а именно:

```
string name = Console.ReadLine();
```

Метод **Console.ReadLine()** считывает информацию именно в виде строки. Поэтому ее можно присвоить только переменной типа **string**. Для получения числовых значений введенную строку нужно преобразовать. Для этого у каждого числового типа есть встроенные методы **Parse** и **TryParse**. Кроме того, для преобразований можно использовать класс **Convert**.

```
// читаем строку и преобразуем в вещественное число
double height = double.Parse(Console.ReadLine());
// ввод целого числа
int size = int.Parse(Console.ReadLine());
// метод TryParse возвращает значение, указывающее,
// успешно ли выполнена операция
byte number;
if( byte.TryParse(Console.ReadLine(), out number))
{
    Console.WriteLine($"Число {number} успешно введено");
}
else
{
    Console.WriteLine("Не удалось преобразовать строку в число");
}
```

Допустим, на вход программе подается строка, в которой через пробел записаны три числа. В этом случае строку надо разбить на подстроки, используя метод **Split**, а затем преобразовать каждую к числу.

```
static void Main()
{
    Console.WriteLine("Введите три числа ");
    string[] numbers = (Console.ReadLine()).Split(' ');
    int num1 = int.Parse(numbers[0]);
    int num2 = Convert.ToInt32(numbers[1]);
    int num3 = Convert.ToInt32(numbers[2]);
    Console.WriteLine("Сумма этих чисел = {0}", num1+num2+num3);
    Console.ReadKey();
}
```

Методу **Split** можно передать множество символов-разделителей. Тогда строка будет разделена на подстроки, которые расположены между каждыми подряд идущими разделителями. Для удаления пустых подстрок, которые могут появиться в результате деления строки, используется специальный параметр, например:

```
string line = Console.ReadLine();
char[] razd = new char[] { ' ', ',', '.', '!', '?', '\n', '\r' };
string[] s = line.Split(razd, StringSplitOptions.RemoveEmptyEntries);
```

2.2. Ввод и вывод данных в языке C++

В стандартном C++ существует два основных способа ввода-вывода информации: с помощью потоков, реализованных в STL (Standard Template Library), и посредством традиционной системы ввода-вывода, унаследованной от C. На самом деле, и потоки, и традиционная система ввода-вывода для осуществления необходимых действий используют вызовы операционной системы.

Для использования традиционного ввода-вывода в стиле C в программу необходимо включить заголовочный файл `<cstdio>`. Эта библиотека содержит функции:

`printf()` — для вывода информации;
`scanf()` — для ввода информации.

При запуске консольного приложения неявно открываются три потока: `stdin` — для ввода с клавиатуры, `stdout` — для буферизованного вывода на монитор и `stderr` — для небуферизованного вывода на монитор сообщений об ошибках. Эти ключевые слова определены в `<cstdio>`.

Функция `printf()` предназначена для форматированного вывода. Она переводит данные в символьное представление и выводит полученные изображения символов на экран. При этом у программиста имеется возможность форматировать данные, то есть влиять на их представление на экране.

Общая форма записи функции `printf()`:

```
int printf ( const char * format, ... );
```

Строка `format` состоит из следующих элементов:

- управляющих символов;
- текста, представленного для непосредственного вывода;
- форматов, предназначенных для вывода значений переменных различных типов.

Форматы нужны для того, чтобы указывать вид, в котором информация будет выведена на экран. Отличительной чертой формата является наличие символа `'%'` перед ним. Перечислим основные:

`%d` — целое число типа `int` со знаком в десятичной системе счисления;

`%u` — целое число типа `unsigned int`;

`%ld` — целое число типа `long int` со знаком в десятичной системе счисления;

`%f` — вещественный формат (числа с плавающей точкой типа `float`);

`%lf` — вещественный формат двойной точности (числа с плавающей точкой типа `double`);

`%e` — вещественный формат в экспоненциальной форме (числа с плавающей точкой типа `float` в экспоненциальной форме);

`%c` — символьный формат;

`%s` — строковый формат.

После строки форматов через запятую указываются имена переменных, которые необходимо вывести. Количество символов `'%'` в строке формата должно совпадать с количеством переменных для вывода. Тип каждого формата должен совпадать с типом переменной, которая будет выводиться на это место. Замещение форматов вывода значениями переменных происходит в порядке их следования.


```
#include <stdio>
int main()
{
    int a = 5;
    double x = 2.78;
    printf("a = %d \n", a);
    printf("x = %lf \n", x);
    return 0;
}
```

На экране увидим:

```
a = 5
x = 2.7800
```

При указании формата можно явным образом определить общее количество знаковых и количество знаковых, занимаемых дробной частью:

```
#include <stdio>
int main()
{
    float x = 1.2345;
    printf("x=%10.5f\n", x);
    int data = 8;
    printf("data = %10d", data);
    return 0;
}
```

На экране увидим:

```
x =      1.23450
data =              8
```

В приведенном примере 10 — общее количество знаковых, отводимое под значение переменной; 5 — количество позиций после разделителя целой и дробной части (после десятичной точки). В указанном примере количество знаковых в выводимом числе меньше 10, поэтому свободные знаковые слева от числа заполняются пробелами.

Функция форматированного ввода данных с клавиатуры *scanf()* выполняет чтение данных, вводимых с клавиатуры, преобразует их во внутренний формат и передает вызывающей функции. При этом программист задает правила интерпретации входных данных с помощью строки форматов.

Общая форма записи функции *scanf()*:

```
int scanf ( const char * format, ... );
```

Строка *format* аналогична используемой в функции *printf()*. После строки формата через запятую идут адреса переменных для ввода. Для формирования адреса переменной используется символ амперсанд '&': адрес = &объект. При вводе строки символ '&' не ставится, ввод идет до первого разделителя.

Строка форматов и список аргументов для функции обязательны. Функция возвращает количество прочитанных переменных.

```

#define _CRT_SECURE_NO_WARNINGS // для использования scanf
#include <stdio.h>
#include <locale.h>
int main()
{
    float y;
    setlocale(LC_ALL, "Rus"); // для вывода русских букв в консоли
    system("cls"); // очищаем окно консоли
    printf("Введите y: "); // выводим сообщение
    // вводим значение переменной y
    if( scanf("%f", &y) == 1)
        printf("Значение переменной y =%f", y); // вывод на экран
    else
        printf("Ошибка ввода \n");
    system("pause");
    return 0;
}

```

Для использования объектно-ориентированного консольного ввода-вывода с помощью потоков (stream) STL в программу необходимо включить заголовочный файл `<iostream>`. При запуске консольного приложения неявно открываются потоки: *cin* — для ввода с клавиатуры, *cout* — для буферизованного вывода на монитор, *cerr* — для небуферизованного вывода на монитор сообщений об ошибках. Эти ключевые слова определены в `<iostream>`.

Потоки *cin*, *cout* и *cerr* соответствуют потокам *stdin*, *stdout* и *stderr*. Для их использования необходимо прописать строку:

```
using namespace std;
```

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

>> — получить из входного потока

<< — поместить в выходной поток

Вывод информации:

`cout << значение; или`

`cout << значение1 << значение2 << ... << значениеN;`

Ввод информации:

`cin >> <переменная>; или`

`cin >> переменная1 >> переменная2 >> ... >> переменнаяN;`

Приведем пример программы, осуществляющей ввод и вывод данных.

```

#include <iostream>
using namespace std;
int main()
{
    int age;
    double weight;

```

```

    cout << "Input age: ";
    cin >> age;
    cout << "Input weight: ";
    cin >> weight;
    cout << "Your age: " << age << "\t your weight: " << weight << endl;
    return 0;
}

```

Если читать строку с использованием *cin*, то она будет введена до первого пробела. Прочитать строку целиком можно с помощью метода **getline**.

```

#include <iostream>
#include <string>
using namespace std;
int main() {
    int count; float price;
    cin >> count >> price; // ввод двух чисел
    string name; cin >> name; // ввод строки до пробела
    string address;
    getline (cin, address); // ввод оставшейся строки полностью
    // вывод чисел и переход на новую строку
    cout << "Price: " << price << " Count: " << count << endl;
    cout << "Name: " << name << endl; // вывод строк
    cout << "Address: " << address << endl;
    system("pause" );
    return 0;
}

```

Заметим, что при написании программ строки могут встречаться в двух различных кодировках в следующих местах: в исходных текстах программ в виде литералов, в выводе на консоль, в вводе с консоли, при вводе из файла и выводе в файл.

Основное правило при работе с национальными алфавитами: все строки должны быть в единой кодировке. Также необходимо в настройках консоли установить для вывода шрифт True Type (Lucida Console).

При несоблюдении этих правил будет невозможно сравнение и сортировка строк (а также и символов) и будет затруднён корректный ввод и вывод строк на консоль или в файл.

Если необходим только вывод кириллицы на консоль советуем использовать `setlocale (LC_ALL, "Russian");`

Например:

```

#include <iostream>
using namespace std;
int main() {
    setlocale (LC_ALL, "Russian"); // русские буквы
    cout << "Привет, мир! ";
    system("pause" );
}

```

```
    return 0;
}
```

Для корректного ввода и вывода кириллицы на консоль надо использовать пару функций: *SetConsoleOutputCP()* и *SetConsoleCP()* из библиотеки `<Windows.h>`. Например:

```
#include <iostream>
#include <string>
#include <Windows.h>
using namespace std;
int main() {
    SetConsoleOutputCP(1251); // устанавливает кодировку вывода на консоль;
    //устанавливает кодировку ввода из консоли и из редактора кода;
    SetConsoleCP(1251);
    string name;
    cin >> name;
    cout << "Привет, " << name << "!";
    system("pause");
    return 0;
}
```

При этом иногда приходится использовать оба способа. Например, если нужно обрабатывать ввод русских букв с помощью функций проверки класса символов языка C. Вот такой код, при вводе русских букв, не будет корректно проверять, введена ли буква:

```
#include <iostream>
#include <cstdlib>
#include <cctype>
#include <Windows.h>
using namespace std;
int main()
{
    // setlocale (LC_ALL, "Rus");
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    char ch;
    cin >> ch;
    if (isalpha((unsigned char)ch)) cout << "is alpha" << endl;
    else cout << "no alpha" << endl;
    system("pause");
    return 0;
}
```

Для его корректной работы необходимо убрать знак комментария у строки с *setlocale*.

Для управления вводом-выводом в C++ используются :

- флаги форматированного ввода-вывода;

– манипуляторы форматирования.

Флаги позволяют установить параметры ввода-вывода, которые будут действовать во всех последующих операторах ввода-вывода до тех пор, пока не будут отменены. Приведем примеры их использования.

Для установки флага используется конструкция: `cout.setf(ios :: flag);`

```
double p = 123.45678;
cout.width(15); // количество знакомест
cout.precision(10); // количество цифр в дробной части
    // фиксированный формат, вывод "+" для положительных чисел,
    // выравнивание по левому краю
cout.setf(ios :: fixed | ios :: showpos | ios :: left );
cout << "p = " << p << endl;
```

Для снятия флага используется конструкция: `cout.unsetf(ios :: flag);`

```
//снятие флага вывода чисел с плавающей точкой в экспоненциальной форме
cout.unsetf(ios :: scientific );
```

Манипуляторы вставляются в операторы `cin(cout)` и устанавливают параметры текущего оператора ввода-вывода.

Таблица 2.2.

Манипуляторы форматирования C++

Манипулятор	Описание
<code>setw(n)</code>	ширина поля вывода в n символов
<code>setprecision(n)</code>	количество цифр (n-1) в дробной части числа
<code>left/right</code>	выравнивание по границе
<code>boolalpha</code>	вывод логических величин в текстовом виде
<code>showpos/noshowpos</code>	вывод знака + для положительных чисел
<code>scientific/fixed</code>	экспоненциальная/фиксированная форма вывода вещественных чисел
<code>endl</code>	помещение в выходной поток символа конца строки '\n'

Для корректного использования манипуляторов необходимо подключить библиотеку `<iomanip>`.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    double p = 123.45678;
    cout << fixed << setw(15) << setprecision(10) << left << "p = " << p << endl;
    system("pause");
    return 0;
}
```



Глава 3.

Операторы ветвления

Ветвление — одна из базовых конструкций структурного программирования. Оператор ветвления применяется в случаях, когда выполнение того или иного набора команд зависит от условия. К условным инструкциям можно отнести условный оператор (оператор *if*) и оператор выбора (*switch*).

3.1. Простой условный оператор

Общий вид условного оператора:

```
if (<условие>)  
{  
    <Оператор1>;  
}  
else  
{  
    <Оператор2>;  
}
```

Оператор1 выполняется, если условие истинно. Оператор2 выполняется, если условие ложно. И тот и другой операторы могут состоять из нескольких команд. Условие может быть сложным (составным) и обязательно заключается в скобки. Операторы ветвления могут быть вложены друг в друга.

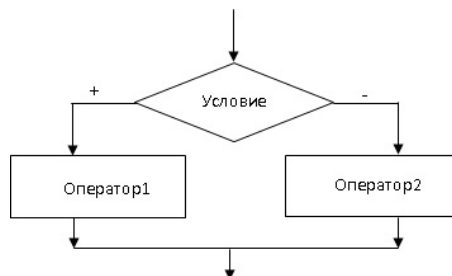


Рис. 3.1. Блок-схема условного оператора

Рассмотрим несколько задач, для решения которых требуется применить условный оператор.

Задача 3.1. Требуется написать программу, которая считывает натуральное число и сообщает, является ли число четным(C++).

```

#include <iostream>
using namespace std;
int main()
{
    int number;
    cin >> number; // вводим число
    // проверяем, четное ли число (остаток от деления на 2 равен 0)
    if (number % 2 == 0) // (1)
        cout << "even" << endl; // выводим even (четное)
    else
        cout << "odd" << endl; // выводим нечетное
    system("pause");
    return 0;
}

```

Заметим, что условие (1) можно было написать так: *if(!(number % 2))* ..., при этом знак отрицания ставится перед скобками, чтобы сначала выполнялась операция взятия остатка.

Задача 3.2. Даны два целых числа, каждое число записано в отдельной строке. Выведите наибольшее из данных чисел(C#).

```

static void Main(string[] args)
{
    int a = int.Parse(Console.ReadLine()); // вводим первое число
    int b = int.Parse(Console.ReadLine()); // вводим второе число
    if (a > b)
        Console.WriteLine(a);
    else
        Console.WriteLine(b);
    Console.ReadKey();
}

```

3.2. Логические операции и сложные условия

Для написания сложного условия, то есть условия, состоящего из нескольких логических выражений, используются логические операции, представленные в таблице 3.1.

Таблица 3.1.

Логические операции

Операция	Обозначение
НЕ (отрицание)	!
И (конъюнкция)	&&
ИЛИ (дизъюнкция)	

При вычислении логического выражения учитывается приоритет логических операций: в первую очередь выполняется отрицание, потом конъюнкция, затем дизъюнкция. Для наглядности при записи выражения лучше использовать скобки.

Иногда запись можно сократить:

```
bool isEqual;
// . . .
if(isEqual) // эквивалентно записи isEqual == true
{
    // действия, если условие выполнено
}
```

Следует упомянуть про типичные ошибки начинающих программистов C++. Допустим, даны два числа и надо проверить, равны ли они.

```
int num1, num2;
cin >> num1 >> num2;
if (num1 = num2) // (2)
    cout << "numbers are equal" << endl;
else
    cout << "numbers aren't equal" << endl;
```

В записи условия (2) допущена ошибка – для сравнения используется "=" вместо правильного "==", – но программа будет исполняться, поскольку операция присваивания возвращает результат, равный num2, преобразует его в bool, при этом любое число, кроме нуля будет приведено к true.

Другой типичной ошибкой является использование в логических выражениях одиночных символов & и | вместо удвоенных && и ||. Результат побитовой операции & (|) может быть приведен к логическому типу, и программа будет работать неверно.

Одно и то же число, полученное в результате вычислений разными способами, может быть представлено в памяти компьютера по-разному, поэтому вещественные числа сравнивают с учетом погрешности.



Рис. 3.2. Сравнение вещественных чисел

Обозначим погрешность *eps*, и числа, принадлежащие *eps*-окрестности заданного числа *A*, будем считать равными этому числу.

Задача 3.3. На вход программе подается два вещественных числа. Требуется сравнить их с учетом погрешности (C#).


```

static void Main(string[] args)
{
    double a, b, c, eps = 0.000000001;
    a = double.Parse(Console.ReadLine());
    b = double.Parse(Console.ReadLine());
    if (Math.Abs(a - b) < eps)
        Console.WriteLine("YES");
    else Console.WriteLine("NO");
    Console.ReadKey();
}

```

Рассмотрим ряд задач на использование условного оператора.

Задача 3.4. На вход программе подается целое число – номер года. Требуется определить, является ли данный год високосным, и вывести слово *YES*, если является, и *NO* – в противном случае. Напомним, что год является високосным, если его номер кратен 4, но не кратен 100, а также если он кратен 400 (C++).

```

#include <iostream>
using namespace std;
int main()
{
    int year;
    cin >> year;
    if ((year % 400 == 0) ||
        ((year % 4 == 0) && !(year % 100 == 0)))
    {
        cout << "YES" << endl;
    }
    else
        cout << "NO" << endl;
    system("pause");
    return 0;
}

```

Задача 3.5. Требуется написать программу, которая определяет, попала ли точка с заданными координатами в выделенную область (см. рис. 3.3) (C#).

Во всех задачах такого типа принадлежность области имеется ввиду вместе с границей (если не оговорено иное).

Известно, что точка с координатами (x, y) лежит внутри или на границе круга с центром в точке $O(a, b)$ тогда и только тогда, когда выполнено условие

$$(x - a)^2 + (y - b)^2 \leq r^2.$$

Точка с координатами (x, y) лежит под прямой, заданной уравнением $y = ax + b$, тогда и только тогда, когда выполнено условие $y < ax + b$. На прямой: $y = ax + b$. Над прямой: $y > ax + b$.

Заметим, что для всех точек, принадлежащих выделенной области, верно:

- 1) $y \leq 1$ (точки не выше прямой $y = 1$);

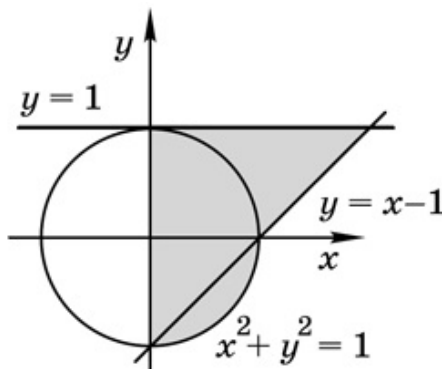


Рис. 3.3

- 2) $x \geq 0$ (точки правее оси OY);
- 3) $x^2 + y^2 \leq 1$ или $y \geq x - 1$ (точки внутри или на границе круга $x^2 + y^2 = 1$ или не ниже прямой $y = x - 1$).

Таким образом, если выполнены все три условия одновременно, то точка принадлежит выделенной области.

```
static void Main()
{
    double x = double.Parse(Console.ReadLine());
    double y = double.Parse(Console.ReadLine());
    if ((y <= 1) && (x >= 0) && ((x * x + y * y <= 1) || (y >= x - 1)))
    {
        Console.WriteLine("т.({0},{1}) принадлежит выделенной области", x, y);
    }
    else
    {
        Console.WriteLine("т.({0},{1}) не принадлежит выделенной области", x, y);
    }
    Console.ReadKey();
}
```

3.3. Вложенный оператор ветвления

В задачах этого параграфа используется вложенный оператор ветвления. Заметим, что в таком случае блок **else** относится к ближайшему **if**. Так, после выполнения приведенного ниже фрагмента кода переменная x примет значение -12.

```
int x = 12;
if ( x > 0)
    if ( x < 10)
        x = x * 2;
```

```
else x = -1 * x;
```

Настоятельно рекомендуется использовать фигурные скобки для выделения блоков кода, что позволит избежать многих ошибок.

```
int x = 12;
if( x > 0)
{
    if( x < 10)
        x = x * 2;
}
else x = -1 * x;    // x = 12
```

Задача 3.6. Даны действительные числа $a(a \neq 0)$, b , c . Найдите все решения квадратного уравнения $ax^2 + bx + c = 0$. Выведите два действительных числа, если уравнение имеет два корня, одно действительное число – при наличии одного корня. При отсутствии действительных корней ничего выводить не нужно (C++).

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    double a, b, c, d, x1, x2;
    cin >> a >> b >> c; // ввод коэффициентов
    d = b*b - 4 * a * c; // вычисляем дискриминант
    if (d == 0){          // если корень один, выводим его
        x1 = -b / (2 * a);
        cout << x1 << endl;
    }
    else if (d > 0) {     // два корня, только если дискриминант > 0
        x1 = (-b + sqrt(d)) / (2 * a);
        x2 = (-b - sqrt(d)) / (2 * a);
        cout << x1 << " " << x2 << endl;
    }
    system("pause");
    return 0;
}
```

*Задача 3.7. Определите тип треугольника (остроугольный, тупоугольный, прямоугольный) с данными сторонами. Необходимо вывести одно из слов: *right* – для прямоугольного треугольника, *acute* – для остроугольного треугольника, *obtuse* – для тупоугольного треугольника или *impossible*, если входные числа не образуют треугольника(C#).*

```
static void Main()
{
    // ввод длин сторон треугольника
```

```

    double a = double.Parse(Console.ReadLine());
    double b = double.Parse(Console.ReadLine());
    double c = double.Parse(Console.ReadLine());
    // если не выполнено неравенство треугольника, то его не существует
    if ((a >= b + c) || (b >= a + c) || (c >= a + b))
    {
        Console.WriteLine("impossible");
    }
    //если треугольник удовлетворяет теореме Пифагора, то он прямоугольный
    else if ((a * a == b * b + c * c) || (b * b == a * a + c * c)
        || (c * c == b * b + a * a))
    {
        Console.WriteLine("right");
    }
    // если есть угол больше 90 градусов, то треугольник тупоугольный
    else if ((a * a > b * b + c * c) || (b * b > a * a + c * c)
        || (c * c > b * b + a * a))
    {
        Console.WriteLine("obtuse");
    }
    // иначе треугольник остроугольный
    else Console.WriteLine("acute");
    Console.ReadKey();
}

```

3.4. Оператор выбора

Оператор выбора выполняет одну ветвь из нескольких в зависимости от значения вычисляемого выражения. Принципиальным отличием от условного оператора является тип выражения. Здесь выражение может быть целым числом, символом или строкой(только в C#), тогда как в условном операторе должен быть исключительно логический тип или тип, приводимый к логическому(C++).

Общий вид оператора выбора:

```

switch (<выражение>)
{
    case <значение1>: <оператор1>; break;
    case <значение2>: <оператор2>; break;
    . . .
    default: <оператор>; break;
}

```

В первой строке оператора находится сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора *case*. И если совпадение найдено, то будет выполняться блок кода после найденного совпадения. Любой блок <операторN> может состоять из нескольких

команд. Если в нем будет описана переменная, то этот блок заключается в фигурные скобки.

В конце каждого блока *case* должен ставиться один из операторов перехода: *break*, *goto case*, *return*. Как правило, используется оператор *break*. При его применении другие блоки *case* выполняться не будут.

В конце конструкции *switch* может стоять блок *default*. Он необязателен и выполняется в том случае, если значение после *switch* не соответствует ни одному из операторов *case*.

Приведем фрагмент кода, обрабатывающий ввод пользователя(C#).

```
Console.WriteLine("Нажмите Y или N");
string selection = Console.ReadLine();
switch (selection)
{
    case "Y":
        Console.WriteLine("Вы нажали букву Y");
        break;
    case "N":
        Console.WriteLine("Вы нажали букву N");
        break;
    default:
        Console.WriteLine("Вы нажали неизвестную букву");
        break;
}
```

Задача 3.8. Напишите программу, которая один раз выводит меню из 3 пунктов (1. Познакомиться. 2. Попрощаться. 3. Выход) и выполняет действия пользователя согласно выбранному пункту меню(C++).

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    int option; // пункт меню, выбранный пользователем
    string name; // имя пользователя
    setlocale(LC_ALL, "Rus"); // вывод русских букв
    //-----вывод меню -----
    cout << "1. Познакомиться" << endl;
    cout << "2. Попрощаться" << endl;
    cout << "3. Выход" << endl;
    cout << "Введите пункт меню: ";
    cin >> option; // ввод пункта меню
    //-----обработка выбора пользователя
    switch (option)
    {
        case 1: // если пользователь выбрал пункт 1
            cout << "Как Вас зовут? Введите имя: ";
```

```

        cin >> name;
        cout << "Приятно познакомиться, " << name << "!" << endl;
        break;
    case 2: // если пользователь выбрал пункт 2
        cout << "До встречи!" << endl;    break;
    case 3: // если пользователь выбрал пункт 3
        break;
    default: // если пользователь ввел число, отличное от 1, 2, 3
        cout << "Нет такого пункта меню" << endl;
        break;
    }
    // конец обработки выбора пользователя
    system("pause");
    return 0;
}

```

3.5. Тернарная операция

Тернарная операция $?$: позволяет сократить определение простейших условных конструкций *if* и имеет следующую форму:

$\langle \text{первый операнд (условие)} \rangle ? \langle \text{второй операнд} \rangle : \langle \text{третий операнд} \rangle;$

В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно *true*, то возвращается второй операнд; если условие равно *false*, то третий. Например:

```

int x = 3;
int y = 2;
Console.WriteLine("Нажмите + или -");
string selection = Console.ReadLine();

int z = selection == "+" ? (x + y) : (x - y);
Console.WriteLine(z);

```

Здесь результатом тернарной операции является переменная *z*. Если мы выше вводим "+", то *z* будет равно второму операнду — (*x+y*). Иначе *z* будет равно третьему операнду.



Глава 4.

Операторы цикла

Операторы цикла позволяют многократно исполнять один и тот же набор инструкций. Если известно, сколько раз должен выполняться цикл, то обычно используют цикл со счетчиком *for*. В противном случае применяют цикл с пред- или постусловием: *while*, *do – while*. В языках C++/C# есть специальные циклы для перебора элементов из массива, списка или другого контейнера/коллекции – *foreach*. Пример использования такого оператора будет приведен позднее. В таблице ниже представлена краткая информация о различных видах циклов.

Таблица 4.1.

Краткое описание циклов

Вид цикла / язык	C++	C#
Цикл со счетчиком (пример: сумма квадратов чисел от 1 до n)	<pre>int sum = 0; for(int i = 1; i <= n; i++) sum += i*i;</pre>	
	Счетчиков цикла может быть несколько, они могут быть любого числового типа и изменяться с любым шагом, например: <pre>float sum = 0; for(float num = 2; num > 0; num = num - 0.2) { sum = sum + num * num; }</pre>	
Цикл с предусловием (пример: произведение цифр числа k)	<pre>int mult = 1; while((k != 0) && (mult != 0)) // условие пишется в скобках { mult *= k % 10; k /= 10; }</pre>	
	Цикл продолжается, пока условие истинно. Условие может быть составным. В рассмотренном примере цикл продолжается, пока число и произведение его цифр не равны нулю (если произведение стало равным нулю, то оно уже не изменится и можно не продолжать цикл)	
Цикл с постусловием (пример: сумма последовательности чисел, заканчивающейся 0)	<pre>do { cin >> ai; sum += ai; }while(ai != 0);</pre>	<pre>do { ai = int.Parse(Console.ReadLine()); sum += ai; }while(ai != 0);</pre>
	Цикл продолжается, пока условие истинно	

4.1. Цикл `for`

Цикл `for` используют, как правило, когда число повторений известно заранее, т. е. в задачах, связанных с перебором. Для языков C++/C# цикл `for` имеет следующее формальное определение:

```
for (<инициализация счетчика>; <условие>; <изменение счетчика>)
{
    // операторы
}
```

В первом блоке задается начальное значение счетчика(параметра цикла), во втором – условие, при котором будет выполняться цикл, в третьем – изменение параметра цикла. Заметим, что любой из блоков может быть пустым. Например,

```
int i = 0;
for (; i < 11;)
{
    Console.WriteLine($"Квадрат числа {++i} равен {i * i}");
}
```

Здесь изменение параметра цикла происходит внутри цикла, а инициализация – до него. Аналогичный код верен и для языка C++, только надо использовать другой оператор вывода.

Рассмотрим еще несколько примеров, где продемонстрируем различные варианты использования цикла `for`.

Задача 4.1. Выведите все натуральные делители числа в порядке возрастания (включая 1 и само число, $2 \leq number \leq 30000$) (C++).

```
int main()
{
    int number;
    cin >> number;
    for (int i = 1; i <= number; i++)
    { // если число делится на i без остатка
        if (x % i == 0)
            cout << i << " "; //вывод делителя
    }
    system("pause");
    return 0;
}
```

*Задача 4.2. Подсчитайте количество натуральных делителей числа (включая 1 и само число; $1 \leq number \leq 2 * 10^9$)(C#).*

Для решения этой задачи воспользуемся некоторыми сведениями из теории чисел: если число имеет делитель до корня этого числа, то он имеет ещё один делитель после корня этого числа. Алгоритм для решения этой задачи таков: циклом проходим от 1 до корня из введенного числа и проверяем на делимость; если

делится, то увеличиваем счетчик. После цикла удваиваем счетчик. Отдельно проверяем, является ли число полным квадратом, и если да, то увеличиваем на один счетчик. Выводим ответ.

```
static void Main()
{
    int number = int.Parse(Console.ReadLine());
    if (number == 1)
    {
        Console.Write(1);
        Console.ReadKey();
        return;
    }
    int bound = (int)Math.Sqrt(number);
    int count = 0;
    for (int i = 1; i < bound; i++)
    {
        if (number % i == 0)
            count++;
    }
    count = count * 2;
    if (bound * bound == number)
    {
        count++;
    }
    Console.Write(count);
    Console.ReadKey();
}
```

Задача 4.3. Проверьте, есть ли среди данных n чисел нули. Введите число n , а затем n чисел. Выведите YES, если среди введенных чисел есть хотя бы один ноль, или NO в противном случае (C++).

При решении этой задачи будем использовать составное условие в записи цикла *for*: цикл продолжается, пока мы не ввели n чисел или пока не встретили среди чисел ноль.

```
int main()
{
    int n;
    cin >> n;
    int number;
    bool noZero = true;

    for (int i = 1; i <= n && noZero; i++)
    {
        cin >> number;
        if (number == 0)
        {
```

```

        noZero = false;
    }
}
if (noZero)
    cout<<"NO"<<endl;
else
    cout << "YES"<<endl;
system("pause");
return 0;
}

```

4.2. Циклы с пред- и постусловием

Иногда необходимо повторять одни и те же действия, пока выполнено или не выполнено какое-либо условие. Например, кассир в магазине выполняет одни и те же операции (пробивает товары, получает деньги и печатает чек), пока есть покупатели в очереди. Такой алгоритм действий можно представить в виде блок-схемы (см. рис. 4.1).

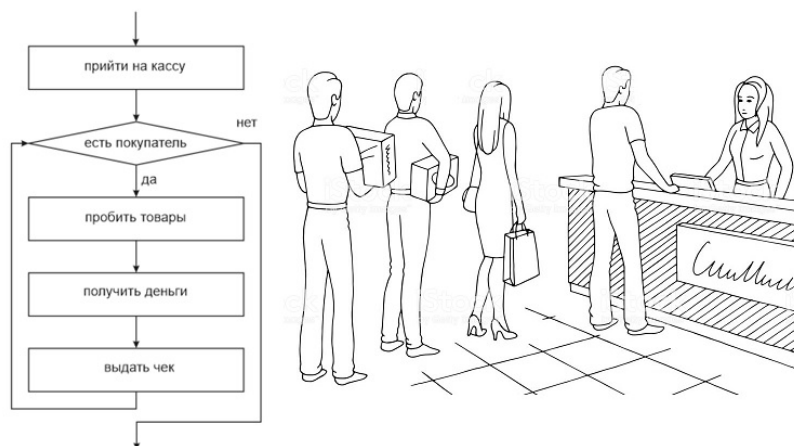


Рис. 4.1. Блок-схема циклических действий кассира

Во многих задачах требуется подобный алгоритм действий, например:

- анализируем очередную цифру числа и уменьшаем число, пока в числе есть цифры;
- обрабатываем очередной элемент последовательности, пока последовательность не закончится;
- читаем очередную строку из файла, пока файл не закончится.

Общей особенностью рассмотренных примеров является то, что мы не знаем, сколько раз выполнится цикл. Мы знаем только условие, в случае выполнения которого вычисления продолжают (условие продолжения цикла). Поэтому такой цикл и называется циклом с условием.

В цикле *do – while* сначала выполняется код цикла, а потом происходит проверка условия в инструкции *while*. И пока это условие истинно, цикл повторяется. Например:

```
int number = 5; //(*)
do
{
    Console.WriteLine(number*number);
    number--;
} while (number > 0);
```

Здесь код цикла сработает 5 раз, пока *number* не станет равным нулю. Важно отметить, что цикл *do* гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции *while* не будет истинно. Если в строке (*) будет присваивание

```
int number = -5;
```

то все равно код цикла выполнится один раз.

Цикл с предусловием *while* сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

```
int number = 1;
while(number < 10)
{
    cout << number * number << endl;
    number++;
}
```

4.3. Операторы прерывания цикла

При решении задач иногда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором *break*. Если в цикле встретился оператор *break*, цикл прерывается сразу и выполнение переходит к первому оператору после тела цикла. Рассмотрим пример. На вход программе поступает последовательность целых чисел, оканчивающаяся нулем. Необходимо посчитать число отрицательных элементов последовательности.

```
int number, count = 0;
while(true) //бесконечный цикл
{
    number = int.Parse(Console.ReadLine());
    if(number == 0)
        break; // выход из цикла
    if(number < 0)
        count++;
}
```

Оператор *continue* обеспечивает прерывание текущего шага цикла и выполняет переход к следующему шагу. Например, приведем фрагмент кода для вычисления произведения всех ненулевых цифр числа.

```
int mult = 1;
while (number != 0)
{
    // получаем последнюю цифру числа
    int digit = number % 10;
    // отсекаем последнюю цифру от числа с помощью целочисленного деления
    number /= 10;
    // обрабатываем полученную цифру, согласно условию задачи
    if ( digit == 0 )           // (**)
        continue;
    mult *= digit;
}
```

Здесь предполагается, что описание и ввод числа уже произведены. Условие (**) позволяет пропустить обработку цифры нуль в таких числах, как 3207, 100 или 5020.

4.4. Задачи на анализ цифр числа

Разберем два примера задач на эту тему: обращение числа (C++), автоморфные числа (C#).

Задача 4.4. Напишите программу, которая получает новое число из исходного путем перестановки цифр исходного числа в обратном порядке.

```
int main()
{
    int number, digit, reverse = 0;
    cin >> number;
    while (number > 0)
    {
        digit = number % 10; // получаем последнюю цифру числа
        reverse = reverse * 10 + digit; // формируем перевернутое число
        number /= 10; // отсекаем последнюю цифру от числа
    }
    cout << reverse << endl;
    system("pause");
    return 0;
}
```

Задача 4.5. Натуральное число называется автоморфным, если оно равно последним цифрам своего квадрата. Например, $25^2 = 625$. Напишите программу, которая находит все автоморфные числа на отрезке $[a, b]$. Если ни одного числа не будет найдено, программа выводит -1.

Для решения этой задачи будем в цикле рассматривать каждое число из заданного диапазона. Нам необходимо посчитать количество цифр k в текущем числе. Число будет автоморфным, если остаток от деления на 10^k его квадрата совпадает с самим числом. В коде ниже мы сразу будем считать 10^k (переменная *stepen*).

```
static void Main()
{
    string[] s = (Console.ReadLine()).Split(' ');
    int numberA = int.Parse(s[0]);
    int numberB = int.Parse(s[1]);
    bool exist = false; // флаг существования автоморфного числа
    for (int currentNum = numberA; currentNum <= numberB; currentNum++)
    {
        int stepen = 1, number = currentNum;
        do
        {
            stepen *= 10;
            number = number / 10;
        } while (number > 0);
        if (currentNum == (currentNum * currentNum) % stepen)
        {
            Console.Write(currentNum + " ");
            exist = true;
        }
    }
    if (!exist)
        Console.Write(-1);
    Console.ReadKey();
}
```

4.5. Задачи на обработку последовательности

Общий алгоритм решения задач этого типа таков: читаем и обрабатываем элементы до тех пор, пока очередной прочитанный элемент не будет равен нулю или мы не введем заранее заданное число элементов последовательности. Два примера (подсчета суммы элементов такой последовательности и числа отрицательных элементов были рассмотрены при изучении синтаксиса цикла *while*). Рассмотрим еще несколько примеров.

Задача 4.6. Дана последовательность натуральных чисел, завершающаяся числом 0. Определите, какое наибольшее число подряд идущих элементов этой последовательности равны друг другу (C++).

```
int main()
{
    int prev, next, count = 1, maxCount = 0;
    cin >> prev;
```

```

while (prev != 0)
{
    cin >> next;
    if (prev == next) count++;
    else
    {
        if (count > maxCount)
            maxCount = count;

        count = 1;
    }
    prev = next;
}
cout << maxCount << endl;
system("pause");
return 0;
}

```

Задача 4.7. Последовательность состоит из n ($n \leq 10000$) натуральных чисел. Определите, сколько элементов этой последовательности равны ее наибольшему элементу (C++).

Идея решения этой задачи состоит в следующем: примем за максимальный первый элемент последовательности. Будем вводить в цикле остальные элементы. Если введенный элемент равен максимальному, то увеличиваем счетчик на единицу; если больше, то меняем значение максимума и устанавливаем счетчик равным 1. Если меньше, то ничего не делаем.

```

int main()
{
    int item; // текущий элемент
    int max, // текущий максимум
        count, // текущее количество максимумов
        n; // число элементов
    cout << "n = ";
    cin >> n;
    cin >> item;
    max = item; count = 1;
    for(int i = 2; i <= n; i++)
    {
        cin >> item;
        if (item > max)
        {
            max = item;
            count = 1;
        }
        else if (item == max) count++;
    }
    cout << endl << count << endl;
}

```

```
    system("pause");  
    return 0;  
}
```

Задача 4.8. Элемент последовательности называется локальным максимумом, если он строго больше предыдущего и последующего элемента последовательности. Первый и последний элемент последовательности не являются локальными максимумами. Дана последовательность натуральных чисел, признаком конца которой является число 0. Определите количество локальных максимумов в этой последовательности(C#).

```
static void Main()  
{  
    int prev, curr, next;  
    prev = int.Parse(Console.ReadLine());  
    curr = int.Parse(Console.ReadLine());  
    int count = 0;  
    if ((prev == 0) || (curr == 0))  
    {  
        Console.WriteLine(0);  
        return;  
    }  
    do{  
        next = int.Parse(Console.ReadLine());  
        if ((next != 0) && (curr > prev) && (curr > next))  
        {  
            count++;  
        }  
        prev = curr;  
        curr = next;  
    } while (next != 0);  
    Console.WriteLine(count);  
    Console.ReadKey();  
}
```

Задача 4.9. Имеется набор данных, состоящий из пар положительных целых чисел. Необходимо выбрать из каждой пары ровно одно число так, чтобы сумма всех выбранных чисел не делилась на 4 и при этом была максимально возможной. Если получить требуемую сумму невозможно, в качестве ответа нужно выдать 0. Необходимо написать эффективную по времени и по памяти программу, решающую эту задачу(C#).

Программа считается эффективной по времени, если время работы программы пропорционально количеству пар чисел N , т. е. при увеличении N в k раз время работы программы должно увеличиваться не более чем в k раз. Программа считается эффективной по памяти, если размер памяти, использованной в программе для хранения данных, не зависит от числа N и не превышает 1 килобайта.

Входные данные: на вход программе в первой строке подаётся количество пар N ($1 \leq N \leq 10^5$). Каждая из следующих N строк содержит два натуральных числа, не превышающих 10 000.

Пример входных данных:

6	
1	3
5	12
6	8
5	4
3	3
1	1

Пример выходных данных для приведённых выше входных данных: 31

Сначала решим задачу, не обращая внимание на условие «сумма не должна делиться на 4». Тогда ответом будет просто сумма наибольших чисел из каждой пары, которую легко найти:

```
static void Main()
{
    Console.Write("n = ");
    int n = int.Parse(Console.ReadLine());
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        var nums = Console.ReadLine().Split(' ');
        int num1 = int.Parse(nums[0]);
        int num2 = int.Parse(nums[1]);
        if (num1 > num2)
        {
            sum += num1;
        }
        else sum += num2;
    }
    Console.WriteLine(sum);
}
```

Допустим, полученная таким образом сумма делится на 4. Тогда попробуем заменить одно число. Например, если сумма (sum) делится на 4 и в эту сумму входит число X , которое тоже делится на 4, то достаточно заменить его на другое число, которое не делится на 4. Если же число X НЕ делится на 4, его нужно заменить числом, которое при делении на 4 даёт другой остаток. В этом случае вся сумма не будет делиться на 4.

По условию задачи сумма должна быть максимально возможной, поэтому для замены нужно выбрать такую пару из входных данных, в которой разница между двумя числами минимальна. Отсюда, получая входные данные, надо искать пару чисел, для которой выполняются два условия:

- 1) числа в паре дают разные остатки при делении на 4;
- 2) разность *delta* между двумя числами в паре минимальна.

Первое из этих условий можно записать в виде $num1 \% 4 \neq num2 \% 4$. Отметим, что по условию все числа положительные и не больше 10000, поэтому их разность всегда будет меньше 10000. Следовательно, начальное значение для *delta* можно взять 10000, а в процессе корректировать.

Значение суммы будем считать как и выше. Если в итоге сумма не делится на 4, следует просто вывести эту сумму. Если делится, то при $delta < 10000$ можно уменьшить сумму на разность между двумя числами в паре, в которой эта замена делается, то есть на *delta*. Если разность осталась равна 10000, то замену произвести нельзя.

```
static void Main()
{
    Console.Write("n = ");
    int n = int.Parse(Console.ReadLine());
    int sum = 0;
    int delta = 10000;
    for (int i = 0; i < n; i++)
    {
        var nums = Console.ReadLine().Split(' ');
        int num1 = int.Parse(nums[0]);
        int num2 = int.Parse(nums[1]);
        if (num1 > num2)
        {
            sum += num1;
        }
        else sum += num2;

        if (num1 % 4 != num2 % 4 && Math.Abs(num1 - num2) < delta)
        {
            delta = Math.Abs(num1 - num2);
        }
    }
    if (sum % 4 == 0)
    {
        if (delta == 10000)
        {
            sum = 0;
        }
        else sum = sum - delta;
    }
    Console.WriteLine("Сумма = {0}", sum);
    Console.ReadKey();
}
```



Глава 5.

Указатели и ссылки

В предыдущих главах мы рассмотрели вопросы, связанные с понятием *переменная*. Переменную, объявленную внутри блока, мы назвали локальной переменной и выяснили, что память для неё выделяется на стеке, а время её жизни и область видимости начинается от места объявления и продолжается до конца блока. Давайте подумаем, а что нам делать, если например, необходимо, чтобы переменная существовала и после выхода из области её видимости, или для хранения данных нужен большой объем памяти, который может переполнить стек, или, что очень часто случается, размер массива, который придется использовать, заранее неизвестен. Во всех этих случаях нам придется в процессе работы приложения запрашивать дополнительные участки памяти и выполнять с выделенными ячейками определенные действия. Память, запрашиваемая в процессе работы приложения, выделяется в куче (heap). Обращение к таким ячейкам памяти, называемым *динамическими переменными*, возможно только по адресу. Время жизни динамических переменных — от момента выделения памяти до конца программы или до явного освобождения памяти. В языках программирования существует специальный тип данных, предназначенный для хранения адресов ячеек памяти, *указатель* (см. рис. 5.1).

5.1. Указатели

Указатель (pointer) — переменная, диапазон значений которой состоит из адресов ячеек памяти или специального значения — нулевого адреса (NULL, nullptr). Нулевой адрес используется для указания того, что в данный момент указатель не ссылается ни на одну из допустимых ячеек. С помощью ненулевого указателя можно получить доступ к определенной ячейке памяти и произвести определенные действия со значением, хранящимся в этой ячейке.

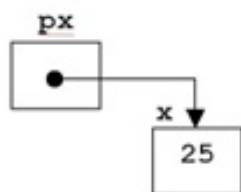


Рис. 5.1. Указатель на переменную

В языке C# указатели очень редко используются и рассматриваться в данном пособии не будут.

Величины типа указатель подчиняются общим правилам определения области действия, видимости и времени жизни. Размер указателя зависит от разрядности вашего приложения: на 32-битной версии – 4 байта на 64-битной версии – 8 байт. При определении указателя лучше всегда выполнять его инициализацию. Непреднамеренное использование неинициализированных указателей – распространенный источник ошибок в программе. Указатель не является самостоятельным типом данных, он всегда связан с каким-либо другим конкретным типом. Стандарт языка C++ определяет три вида указателей, отличающихся свойствами и набором допустимых операций:

- типизированные указатели,
- бестиповые указатели (void),
- указатели на функции.

Типизированный указатель содержит адрес области памяти, в которой хранятся данные определенного типа. Объявление типизированного указателя:

```
<тип> *имя;
```

где <тип> может быть любым типом данных, кроме ссылки и битового поля, причем тип может быть к этому моменту только объявлен, но еще не определен. *Звездочка относится непосредственно к имени*, поэтому ставьте ее перед именем каждого указателя.

```
int x=25, *px = NULL, *ptr = nullptr;
```

Бестиповый указатель применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов). Объявление бестипового указателя:

```
void * имя;
```

Бестиповому указателю можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями. Однако перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется явным образом преобразовать его к конкретному типу.

```
int x = 5;
void * vptr = &x;
int *ptr = (int *)vptr;
```

Указатель на функцию (function pointer) хранит адрес функции, т. е. адрес первого байта в памяти, по которому располагается выполняемый код функции. Самым распространенным указателем на функцию является ее имя. С помощью имени функции можно ее вызвать и получить результат работы.

```
<тип> (*имя_указателя) (<параметры>);
```

Более подробно рассмотрим этот тип указателей, когда введем понятие функции.

5.2. Операции с указателями в C++

Как правило, две основные операции над указателями – это **присваивание** (**взятие адреса (&)**) и **разыменование (*)**. Первая присваивает указателю некоторый адрес. Вторая служит для обращения к значению в памяти, на которое указывает указатель (см. рис. 5.2).

```
int x = 25; // целая переменная
int *px = &x; // в указатель записывается адрес x – взятие адреса
int y = *px; // y = 25, обращение по адресу – разыменование
```

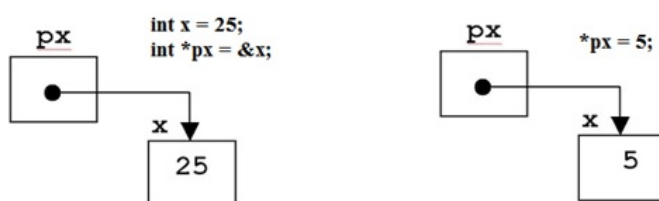


Рис. 5.2. Операции для работы с указателем

Присвоить значение указателю можно с помощью значения другого инициализированного указателя:

```
int *ptr = px;
```

или указатель может быть проинициализирован нулевым значением:

```
int * ptrNull = 0, *ptr = NULL, *p = nullptr;
```

Операция выделения памяти (*new*) выполняется следующим образом:

// без инициализации выделенного участка под переменную типа float

```
float* ptr = new float;
```

// выделение с начальной инициализацией

```
float* ptr10 = new float (10.7);
```

// выделение блока памяти для 10 переменных типа float

```
float* ptrArray = new float[10]; // (одномерный массив)
```

Освобождение памяти, выделенной с помощью операции *new*, должно выполняться с помощью *delete*. При этом переменная-указатель сохраняется и может инициализироваться повторно. Приведенные выше динамические переменные уничтожаются следующим образом:

```
delete ptr;
delete ptr10;
delete [] ptrArray;
```

Квадратные скобки после *delete[]* говорят о том, что надо освободить блок памяти, начинающийся по этому адресу (т. е. весь массив). Размерность массива при этом не указывается. Если квадратных скобок нет, то никакого сообщения об

ошибке не выдается, но помечен как свободный будет только первый элемент массива, а остальные окажутся недоступными для дальнейших операций. Такие ячейки памяти называются *мусором*. Если переменная-указатель выходит из области своего действия, отведенная под нее память освобождается, а память, на которую она указывала, – нет. Следовательно, динамическая переменная, на которую ссылался указатель, становится недоступной. Ещё один случай появления «мусора» – когда инициализированному указателю присваивается значение другого указателя. При этом старое значение бесследно теряется.

```
float* ptr = new float[10]; // выделили блок
ptr = new float (10.7);    // предыдущее значение потеряно – мусор
```

Кроме того, указатели одного вида можно *сравнивать друг с другом*, а с типизированными указателями можно выполнять *арифметические операции* (сложение с целым, вычитание, инкремент и декремент).

Если указатель на определенный тип увеличивается (уменьшается) на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа.

```
int *ptr = new int[10];
int* newptr = ptr + 2; // указывает на элемент массива ptr[2].
```

Суммирование двух указателей не допускается.

Разность двух указателей – это разность их значений, деленная на размер типа в байтах (в применении к массивам разность указателей – это количество элементов массива между этими адресами).

```
int cntElement = &ptr[6] - &ptr[3]; //cntElement = 3
```

Инкремент (декремент) перемещает указатель к следующему (предыдущему) элементу массива. Фактически значение указателя изменяется на величину `sizeof(тип)`.

```
int *ptr = new int[5];
ptr++; //значение увеличится на 4, хранится адрес ptr[1]
char *cptr = new char[5];
cptr++; //значение увеличится на 1, хранится адрес cptr[1]
```

Арифметические операции в основном имеют смысл при работе со структурами данных, последовательно размещенными в памяти, например с массивами.

Операции сравнения (`>`, `>=`, `<`, `<=`, `==`, `!=`.) применяются только к указателям одного типа и к значениям `NULL` и `nullptr`.

Задача 5.1. Выполнить реверс массива array из size элементов, используя арифметику указателей.

```
int *left = array, // указатель на первый элемент массива
    *right = array + size - 1, // указатель на последний элемент массива
    tmp;
while( left < right ) // сравнение указателей
{
    tmp = *left; // обмен с использованием операции разыменования
```

```

    *left = *right;
    *right = tmp;
    left++; // инкремент — движение вправо
    right--; // декремент — движение влево
}

```

Указатели могут хранить адреса как переменных, так и констант. Чтобы определить указатель на константу, он тоже должен объявляться с ключевым словом `const`:

```

int i; // целая переменная
const int ci = 1; // целая константа
int * ptri = &i; // указатель на переменную
const int * ptrci = &ci; // указатель на константу
int * const cptri = &i; // указатель-константа на переменную
const int * const cptrci = &ci; // указатель-константа на константу

```

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

5.3. Ссылки

Ссылка является альтернативным именем (синонимом или псевдонимом) объекта. Сама ссылка это не объект, поэтому память для нее не выделяется. Поскольку ссылка является другим именем объекта, все операции выполняются на самом деле с самим объектом. Ссылки подчиняются общим правилам определения области видимости, действия и времени жизни.

Ссылки в основном используются при передаче параметров в функции и методы или для возврата значения из них. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

```
<тип> & <имя>;
```

где **тип** — это тип величины, на которую указывает ссылка, & — оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа, например:

```

int value;
int& ref = value; // ссылка ref — альтернативное имя для value

```

Разрешается объявлять константную ссылку, то есть ссылку на константу, например:

```
const char& Tab = '\t'; // ссылка на константу
```

Инициализация ссылки осуществляется во время выполнения программы. Операция над ссылкой приводит к изменению величины, на которую она ссылается. Кроме того:

- переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции;

- после инициализации ссылке не может быть присвоена другая переменная;
- тип ссылки должен совпадать с типом величины, на которую она ссылается;
- не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Задача 5.2. Создать ссылку на целочисленную переменную. Изменить значение переменной путем изменения ссылки на неё.

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

int main()
{
    int value = 2;      // целочисленная переменная
    int &ref = value;   // ссылка на переменную value
    value = 5;          // value = 5
    ref = 7;            // value = 7

    cout << value;      // выведется 7
    ++ref;
    cout << value;      // выведется 8
    system("pause");
    return 0;
}
```



Глава 6.

Массивы

Массив – это структура данных для хранения множества **однотипных** элементов. Массивы бывают одномерные и многомерные, причем чаще используются двумерные.

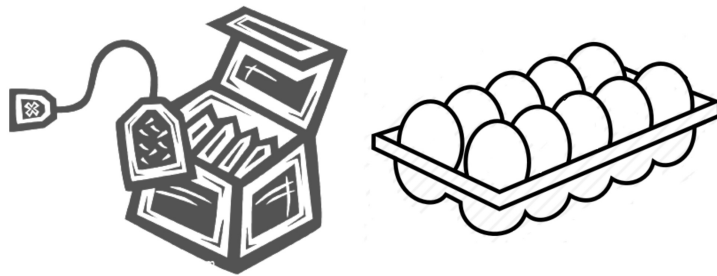


Рис. 6.1. Массивы в повседневной жизни

Аналогом одномерного массива может служить коробка с чайными пакетиками, двумерного – упаковка яиц (см. рис. 6.1). Заметим, что:

- элементы массива всегда одного типа;
- каждый массив должен иметь свое имя;
- массив может в себе содержать от одного элемента до бесконечности (это в теории, на практике размер массива ограничивается памятью компьютера), но количество элементов задается и не меняется в процессе работы программы;
- обращение к элементам массива идет по порядковому номеру (индексу).

Прежде всего рассмотрим одномерные массивы (см. рис. 6.2).



Рис. 6.2. Одномерный массив

6.1. Одномерные массивы

Перед использованием любой структуры данных ее надо описать. И тут в каждом языке программирования есть свои нюансы. В C# массивы динамические: допускается как сразу указать количество элементов в массиве, так и задать во время выполнения программы, например:

```
int [] nums = new int[6]; // целочисленный массив из шести элементов
char [] sb = new char[10]; // массив из 10 символов
int size = int.Parse(Console.ReadLine());
int [] arr = new int[size]; // массив из size элементов, его размер size
                        // определен во время выполнения программы
```

В языке C++ поддерживаются два типа массивов: статические и динамические. Вначале речь пойдет о статических массивах. Например:

```
int nums[6]; // целочисленный массив из шести элементов
char sb[10]; // массив из 10 символов
```

При описании массива в любом языке программирования обязательно указывается тип его элементов (не забываем, что массив – это набор однотипных данных). В языке C# все элементы массива после описания инициализируются значением по умолчанию для конкретного типа, например нулем для числовых типов; в языке C++ инициализируются только статические глобальные массивы, для остальных начальные значения надо задавать. При явной инициализации элементов массива значения задаются прямо в тексте программы, при этом размер массива определяет компилятор по количеству элементов:

```
// C++
int nums[] = { 1, 2, 3, 4 };
// C#
int [] nums = { 1, 2, 3, 4 };
```

В C++ допустимы и другие варианты:

```
// задает нулевое начальное значение для всех элементов массива
int numbers[6] = {0};
// первые два элемента массива будут равны 1 и 2,
int numbers[6] = {1, 2}; // остальные получат значение 0
```

Доступ к элементу массива происходит по индексу, т. е. по его порядковому номеру внутри массива. В языках C++ и C# номера начинаются с нуля. Обратившись к элементу по индексу можно задать, получить или изменить его значение.

Заметим, что индекс элемента должен быть строго меньше длины массива, иначе возникнет ошибка "выход за границы массива".

Приведем несколько примеров.

Задача 6.1. Заполните элементы массива значениями, введенными с клавиатуры(C++).

В C++ возможен любой формат ввода для числовых данных – все элементы в одной строке или каждый элемент в новой строке.

```
int main()
{
    int numbers[10];
    cout << "Input array: ";
    for(int i = 0; i < 10; i++)
        cin >> numbers[i];
    cout << "Result array :";
    for(int i = 0; i < 10; i++)
        cout << numbers[i] << endl; // печать массива по одному в строке
    system("pause");
    return 0;
}
```

Решая аналогичную задачу на языке C#, также можно вводить каждый элемент массива по отдельности. Есть и другой способ: записать все значения в одну строку через пробел (например, 17 -3 22 76 1) и потом ее разобрать для заполнения массива.

```
static void Main()
{
    Console.WriteLine("Введите количество чисел ");
    int n = int.Parse(Console.ReadLine());
    int[] arr = new int[n];
    Console.Write("Введите числа через пробел");
    string line = Console.ReadLine();
    var nums = line.Split(' ', StringSplitOptions.RemoveEmptyEntries);
    for(int i = 0; i < n; i++)
    {
        arr[i] = int.Parse(nums[i]);
    }
    // печать массива в одной строке
    foreach( int item in arr) // (*)
    {
        Console.Write("{0} ", item);
    }
    Console.ReadKey();
}
```

В строке (*) используется цикл **foreach** для перебора всех элементов массива.

6.2. Применение датчика случайных чисел

При тестировании программ массив нередко заполняют случайными числами. Случайные числа – это такая последовательность чисел, в которой невозможно назвать следующее число, зная сколько угодно предыдущих. Получить случайные числа на компьютере достаточно сложно. Однако математики придумали универсальный и удобный способ – *псевдослучайные числа*.

Псевдослучайные числа – это последовательность чисел, обладающая свойствами, близкими к свойствам случайных чисел, в которой каждое следующее число вычисляется на основе предыдущих по некоторой математической формуле. Таким образом, эта последовательность ведет себя так же, как и последовательность случайных чисел, хотя, зная формулу, мы можем получить следующее число в последовательности. Большинство стандартных датчиков псевдослучайных чисел (то есть формул, по которым они вычисляются) дают равномерное распределение в некотором интервале. Поскольку случайные числа в компьютере вычисляются по формуле, то для того, чтобы повторить в точности какую-нибудь случайную последовательность, достаточно просто взять то же самое начальное значение.

В языке программирования встроен генератор псевдослучайных чисел. Покажем, как его использовать.

1. C#. Заполнение массива. Фрагмент программы.

```
// сначала необходимо создать новый экземпляр класса Random
```

```
Random rnd = new Random();
```

или так

```
// каждый раз последовательность начинается с нового числа
```

```
Random rnd = new Random(DateTime.Now.Milliseconds);
```

```
// получение целого числа в заданном диапазоне
```

```
int size = rnd.Next(7, 11); // число от 7 до 10
```

```
int[] a = new int[size];
```

```
double[] b = new double[size];
```

```
for(int i = 0; i < size; i++)
```

```
{
```

```
    a[i] = rnd.Next(); // получение случайного целого числа
```

```
    // получение случайного вещественного числа от 0 до 1
```

```
    b[i] = rnd.NextDouble();
```

```
}
```

Заметим, что функция *rnd* возвращает значение с типом *int*, и, если нам нужно значение другого типа, необходимо сообщить компилятору о преобразовании явным образом:

```
uint someValue = (uint)rnd.Next( 20); // целое число от 0 до 19
```

2. C++. Для работы со случайными числами необходимо подключить заголовочный файл `<cstdlib>` в начале программы. Для получения целого числа в интервале от 0 до *RAND_MAX* (это число – 32767) используется команда

```
int number = rand();
```

Команда `srand(m)`; позволяет установить начальное значение случайной последовательности, равное m . Это необходимо делать, чтобы при повторном запуске программы последовательность случайных чисел каждый раз начиналась с иного случайного числа.

```
srand(time(NULL)); // библиотека ctime
```

Для практических задач чаще всего надо получать случайные числа в заданном интервале $[a, b]$. Если интервал начинается с нуля ($a = 0$), можно использовать свойство операции взятия остатка от деления: остаток от деления числа на некоторое N всегда больше нуля или равен ему, но меньше N , то есть находится в интервале $[0, N - 1]$. Можно написать такую функцию

```
int random ( int N )
{
    return rand() % N; // случайное число в интервале [0, N-1]
}
```

С ее помощью (вызывая ее много раз подряд) можно получать последовательность случайных чисел в интервале $[0, N - 1]$ с равномерным распределением. Теперь попытаемся использовать эту функцию для интервала $[a, b]$. Очевидно, что формула $k = \text{random}(N) + a$; дает последовательность в интервале $[a, a + N - 1]$. Поскольку нам нужно получить интервал $[a, b]$, сразу имеем $b = a + N - 1$, откуда $N = b - a + 1$. Поэтому для получения случайных целых чисел с равномерным распределением в интервале $[a, b]$ надо использовать формулу $k = \text{random}(b - a + 1) + a$;

Более сложным оказывается вопрос о случайных вещественных числах.

Если разделить результат функции `rand()` на `RAND_MAX`:

```
x = (float) rand() / RAND_MAX;
```

мы получим случайное вещественное число в интервале $[0, 1)$ (при этом надо не забыть привести одно из этих чисел к вещественному типу, иначе деление одного целого числа на большее целое число будет всегда давать нуль). Длина интервала $[0, 1)$ такой последовательности равна 1, а нам надо получить интервал длиной $b - a$. Если теперь это число умножить на $b - a$ и добавить к результату a , мы получаем как раз нужный интервал.

Итак, для получения случайных вещественных чисел с равномерным распределением в интервале $[a, b]$ надо использовать формулу

```
x = (float) rand() * (b - a) / RAND_MAX + a;
```

Приведем пример. Массив `A` заполняется случайными целыми числами в интервале $[-5, 10]$, а массив `X` – случайными вещественными числами в том же интервале.

```
#include<stdlib.h>
#include<ctime>
const int N = 10;
int random ( int N )
{
```

```

    return rand() % N; // случайное число в интервале [0, N-1]
}
int main()
{
    srand(time(NULL)); // инициализация датчика случайных чисел
    int i, A[N], a = -5, b = 10;
    for ( i = 0; i < N; i ++ )
        A[i] = random(b - a + 1) + a;
    float X[N];
    for ( i = 0; i < N; i ++ )
    {
        X[i] = (float)rand() * ( b -a) / RAND_MAX + a;
    }
    // место для кода по обработке массивов
    // ...
    return 0;
}

```

6.3. Линейный поиск

Одной из распространенных задач является поиск в массиве какого-либо значения. Самый простой для реализации алгоритм поиска — **это линейный поиск**. Этот алгоритм перебирает все элементы в массиве, сравнивая их с заданным значением.

Задача 6.2. Найдите максимальный элемент в массиве(C++)

```

int main()
{
    int numbers[] = {11, 42, 3, 14, 77, 0, 1};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    int max = numbers[0];

    for(int i = 1; i < size; i++)
    {
        if( max < numbers[i])
            max = numbers[i];
    }
    cout << "max =" << max<<endl;
    system("pause");
    return 0;
}

```

Для нахождения длины массива применяется оператор `sizeof`. По сути, длина массива равна совокупной длине его элементов. Все элементы представляют один и тот же тип и занимают один и тот же размер в памяти. Поэтому с помощью выражения `sizeof(numbers)` находим длину всего массива в байтах, а с помощью

выражения `sizeof(numbers[0])` – длину одного элемента в байтах. Разделив одно значение на другое, получаем количество элементов в массиве. Просматриваем в цикле все элементы и выбираем среди них наибольший.

В языке C# длину массива можно узнать, обратившись к его свойству *Length*, например:

```
int [] numbers = {11, 2, 3, 4, 5};
int size = numbers.Length;      // size = 5;
```

Задача 6.3. Найти в одномерном целочисленном массиве второй максимум, то есть элемент, следующий по величине за максимальным и отличный от него. Например, если на входе задачи будет массив из 6 элементов: -2, 7, 1, 4, -8, 9, то ответом будет 7.

Легко сформулировать алгоритм, который решает эту задачу за два прохода по массиву. Во время первого прохода ищется максимальный элемент, а во время второго – наибольший элемент, не совпадающий с максимальным. Но эффективным будет алгоритм, который решает данную задачу за один проход по массиву.

Для этого нам потребуются две вспомогательные переменные *max* и *max2*. Первые два различных элемента массива сравниваются, и большее заносится в *max*, а меньшее, соответственно, – в *max2*. При просмотре массива значения этих переменных корректируются.

Приведем код соответствующего фрагмента программы на языке C#, предполагая, что описание и ввод данных уже произведены.

```
int i = 0;
// поиск двух различных элементов
while (i < a.Length - 1 && a[i] == a[i + 1])
{
    i++;
}
if (a[i] > a[i + 1]) // инициализация max и max2
{
    max = a[i]; max2 = a[i + 1];
}
else
{
    max = a[i + 1]; max2 = a[i];
}
for (int j = i; j < a.Length; j++) // основной цикл поиска
{
    if (a[j] > max)
    {
        max2 = max;
        max = a[j];
    }
    else if ((a[j] < max) && (a[j] > max2))
    {
        max2 = a[j];
    }
}
```

```

    }
}
Console.WriteLine(" Ответ : {0}", max2);

```

Задача 6.4. Даны натуральное n и последовательность из n целых чисел. Внутри данной последовательности могут быть повторяющиеся члены. Найдти число различных членов последовательности.

Как и выше, члены последовательности заносятся в массив. Можно предложить два варианта решения: первый – с использованием дополнительного массива, второй – с помощью сортировки. Код первого варианта получается более прозрачным и легким для понимания.

Все элементы дополнительного массива пометок *mark* первоначально равны нулю. Возьмем первый элемент исходного массива. Счетчик различных элементов увеличим на 1. Соответствующий элемент массива *mark* также положим 1. Далее для всех элементов исходного массива, равных первому, в соответствующие элементы массива пометок занесем единицу. Следующим текущим элементом будет элемент исходного массива с нулевой пометкой. И те же действия повторяются для него. Ниже приведен код программы(C++).

```

int main()
{
    const int n = 6;
    int a[n];
    int mark[n] = {0}; //инициализация нулями

    for (int i = 0; i < n; i++)
        cin >> a[i];
    int count = 0;
    for (int i = 0; i < n; i++)
    {
        if (mark[i] == 0)
        {
            count++;
            mark[i] = 1;
            for (int j = i + 1; j < n; j++)
            {
                if (a[i] == a[j])
                    mark[j] = 1;
            }
        }
    }
    cout << count;
    system("pause");
    return 0;
}

```

Задача 6.5. Даны два упорядоченных массива $x[0] \leq x[1] \leq \dots \leq x[k-1]$ и $y[0] \leq y[1] \leq \dots \leq y[n-1]$. Соединить их в массив $z[0] \leq \dots \leq z[m-1]$ ($m = k+n$, каждый элемент должен входить в массив z столько раз, сколько раз он входит в общей сложности в массивы x и y). Число действий в программе порядка m .

Идею решения можно пояснить следующим образом: пусть есть две стопки карточек с числами, отсортированными по возрастанию (точнее, по неубыванию). Для соединения их в одну стопку будем выбирать каждый раз ту из верхних карточек обеих стопок, чье число меньше. Если в одной стопке карточки закончились, то просто берем их из другой стопки.

```
static void Main()
{
    int    n, k;    // размерности массивов
    int[]  x,y,z;   // исходные и результирующий массивы
    int    i, j, m; // индексы

    Console.Write(" k =");
    k = int.Parse(Console.ReadLine());
    x = new int[k];    // ввод массива x
    for (i = 0; i < k; i++)
    {
        Console.Write(" x[ {0} ] = ", i);
        x[i] = int.Parse(Console.ReadLine());
    }
    Console.Write(" n =");
    n = int.Parse(Console.ReadLine());
    y = new int[n];    // ввод массива y
    for (i = 0; i < n; i++)
    {
        Console.Write(" y[ {0} ] = ", i);
        y[i] = int.Parse(Console.ReadLine());
    }
    z = new int[n+k]; // определение массива z

    i = 0; j = 0; m=0;
    while ((i < k) && (j < n)) // основной цикл
    {
        if (x[i] <= y[j])
        {
            z[m] = x[i];
            i++;
        }
        else
        {
            z[m] = y[j];
            j++;
        }
        m++;
    }
}
```



```

    }
    m++;
}
while (i < k)
{
    z[m] = x[i];
    i++; m++;
}
while (j < n)
{
    z[m] = y[j];
    j++; m++;
}
for (i = 0; i < z.Length; i++) // вывод результата
    Console.WriteLine(" z[ {0} ] = {1}", i, z[i]);
Console.ReadKey();
}

```

6.4. Динамические массивы

До сих пор в примерах на языке C++ использовались статические массивы, то есть при объявлении массива явно указывалось количество элементов в нем. Язык C++ позволяет также создавать и динамические массивы, в которых количество элементов задается во время выполнения программы. При работе с динамическими массивами используются указатели. Указатель хранит адрес первой ячейки памяти массива (см. рис. 6.3).

```

int size ;
cin >> size; // пусть size = 10
int *numbers = new int[size];

```

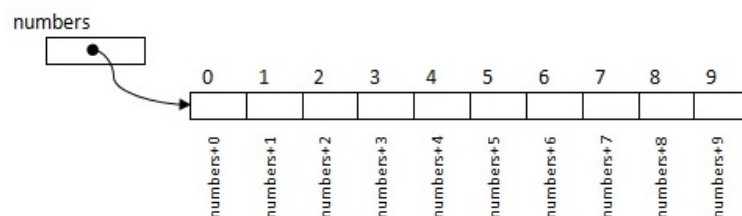


Рис. 6.3. Динамический массив

Для выделения памяти под динамический массив используется оператор *new*, после которого указывается тип элементов массива и их количество. Заметим, что оператор *new* возвращает указатель на объект заданного типа – первый элемент в созданном массиве. После создания динамического массива мы сможем с ним работать по полученному указателю, просматривать и изменять его элементы:

```

// ввод элементов массива
for (int i = 0; i < size; i++) {
    cin >> numbers[i];
}
// вывод элементов массива с использованием разыменования указателя
for (int *item = numbers; item != numbers + size; item++)
{
    cout << *item << "\t";
}

```

Для освобождения памяти применяется специальная форма оператора delete:

`delete[] <указатель на динамический массив>;`

Для примера выше: `delete[] numbers;`

6.5. Сортировка массива

Под сортировкой обычно понимают процесс перестановки элементов данного множества в определенном порядке. Цель этого процесса – облегчить в дальнейшем поиск конкретного элемента в отсортированном множестве. Сортировка – это самая распространенная алгоритмическая задача. Прежде всего будут рассмотрены простые алгоритмы сортировки.

Сортировка вставками или включениями

Алгоритм сортирует массив по мере прохождения по его элементам. На каждой итерации берется элемент и сравнивается с каждым элементом в уже отсортированной части массива, таким образом находя «свое место», после чего элемент туда вставляется. Так происходит до тех пор, пока алгоритм не пройдет по всему массиву. На выходе получим отсортированный массив.

Например, будем рассматривать такой массив:

7	3	4	4	0	6
---	---	---	---	---	---

Последовательность из одного элемента – 7 – уже отсортирована. На первой итерации рассматривается второй элемент – 3. Он меньше семи, поэтому семь сдвигается, а тройка вставляется на первое место.

3	7	4	4	0	6
---	---	---	---	---	---

Часть массива, выделенная серым, является упорядоченной. На следующей итерации берется третий элемент массива – 4. Так как семь больше четырех, то семь сдвигается на третье место, а на второе – вставляется четыре.

3	4	7	4	0	6
---	---	---	---	---	---

Четвертая итерация:

3	4	7	4	0	6
---	---	---	---	---	---



3	4	4	7	0	6
---	---	---	---	---	---

Пятая итерация:

3	4	4	7	0	6
---	---	---	---	---	---



0	3	4	4	7	6
---	---	---	---	---	---

Шестая итерация:

0	3	4	4	7	6
---	---	---	---	---	---



0	3	4	4	6	7
---	---	---	---	---	---

Описанный алгоритм показан в листинге(C#).

```
static void SortInsertion(int[] a)
{
    for (int i = 1; i < a.Length; i++)
    {
        int j = i;
        int item = a[i];

        while ((j >= 1) && (item < a[j - 1]))
        {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = item;
    }
}
```

Сортировка посредством выбора

Идея сортировки посредством выбора также элементарна. На i -м этапе сортировки выбирается наименьший элемент среди $A[i], \dots, A[n]$ и меняется местами с элементом $A[i]$. В результате после i -го этапа все записи $A[0], \dots, A[i]$ будут упорядочены.

Полный код, реализующий этот метод сортировки, приведен ниже(C++).

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

int main()
{
    const int length = 6;
    srand(time(NULL));
    int array[length];
    // заполнение случайными числами
    for (int i = 0; i < length - 1; ++i)
```

```

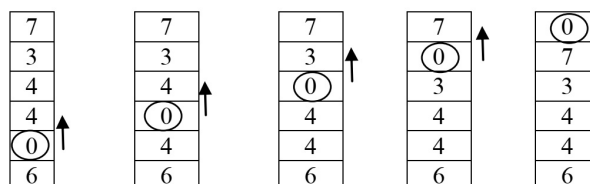
    {
        array[i] = rand() % 100;
        cout << array[i] << '\t';
    }

    // сортировка
    for (int i = 0; i < length - 1; ++i)
    {
        // в smallestIndex хранится индекс наименьшего значения
        int smallestIndex = i;
        int min = array[i];
        for (int curInd = i + 1; curInd < length; ++curInd)
        {
            if (array[curInd] < min)
            {
                smallestIndex = curInd;
                min = array[curInd];
            }
        }
        array[smallestIndex] = array[i];
        array[i] = min;
    }
    // выводим на экран отсортированный массив
    for (int index = 0; index < length; ++index)
        cout << array[index] << ' ';
    return 0;
}

```

Сортировка прямым обменом, или «пузырёк»

Самым простым методом сортировки является так называемый метод "пузырька". Для описания основной идеи этого метода представим, что массив расположен вертикально. Элементы с малыми значениями более "легкие" и "всплывают" вверх наподобие пузырька. При первом проходе вдоль массива снизу вверх берется последний элемент и поочередно сравнивается с последующими. Если встречается элемент с более "тяжелым" значением, то эти элементы меняются местами. При встрече с элементом с более "легким" значением именно он становится "эталоном" для сравнения и все последующие элементы сравниваются с этим новым, более "легким" элементом. В результате элемент с наименьшим значением окажется в самом верху массива. Например:



Во время второго прохода вдоль массива находится элемент со вторым по величине значением, который занимает вторую сверху позицию, и т. д.

0	0	0	0	0	0
7	3	3	3	3	3
3	7	7	4	4	4
4	4	4	7	4	4
4	4	4	4	7	6
6	6	6	6	6	7

Отметим, что во время второго и последующих проходов вдоль массива нет необходимости просматривать элементы, занявшие верхние места ранее, так как они имеют значения, меньшие, чем у оставшихся элементов. Другими словами, во время i -го прохода не проверяются записи, стоящие на позициях выше i . В листинге приведен код алгоритма (C#).

```
static void BubbleSort(int[] a)
{
    for ( int i = 1; i < a.Length; i++)          // (*)
    {
        for (int j = a.Length-1; j >= i; j--)    // (**)
        {
            if (a[j-1] > a[j])
            {
                int temp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Временная сложность методов сортировки. Метод "пузырька", метод вставками и метод посредством выбора имеют временную сложность $O(n^2)$ на последовательностях из n элементов.

Рассмотрим метод "пузырька". Пусть сравнение и обмен элементов происходит за c_1 единиц времени (c_1 – некоторая константа). Цикл (*) выполнится $n - 1$ раз, вложенный цикл (**) – от $n - 1$ до 1 раза. Отсюда, согласно формуле суммы арифметической прогрессии, вся программа требует $c_1 * \frac{n(n-1)}{2}$ единиц времени. Поэтому алгоритм "пузырька" имеет временную сложность $O(n^2)$.

6.6. Бинарный поиск

Для нахождения элемента в отсортированном массиве используется алгоритм двоичного(бинарного) поиска. Суть алгоритма: рассматривается средний элемент. Если искомое значение меньше, то поиск продолжается в левой части массива, от начала до середины, если больше, то в правой части массива. На следующем

шаге вновь рассматривается средний элемент левой или правой части массива и тем же методом участок для поиска сокращается вдвое. Поиск останавливается либо если встретилось заданное значение, либо если участок для поиска имеет нулевую длину, то есть определяется, что такого элемента в массиве нет.

Рассмотрим пример. Пусть дан отсортированный массив из 10 элементов, в котором надо найти число 32 или убедиться, что его там нет. Напомним, что индексы

	first			middle			last			
Индекс	0	1	2	3	4	5	6	7	8	9
Массив ar	2	11	17	23	28	32	44	51	52	60

массива начинаются с нуля ($first = 0$, $last = 9$). Каждый раз средний элемент рассматриваемой части массива будем находить по формуле:

$$middle = first + (last - first) / 2. \quad (*)$$

На первом шаге средний элемент массива будет иметь индекс 4 и этот элемент меньше искомого значения ($ar[4] < 32$), поэтому будем рассматривать только правую часть массива. Средний элемент имеет индекс 7. $ar[7] = 51 > 32$. Отсюда

	first			middle			last			
Индекс	0	1	2	3	4	5	6	7	8	9
Массив ar	2	11	17	23	28	32	44	51	52	60

следует, что искомый элемент находится слева.

	first			last						
Индекс	0	1	2	3	4	5	6	7	8	9
Массив ar	2	11	17	23	28	32	44	51	52	60

Отрезок для поиска содержит всего два элемента. По формуле (*) средний элемент имеет индекс 5 и именно этот элемент искомый.

Приведем код алгоритма бинарного поиска(C++).

```
#include <iostream>
using namespace std;
int main()
{
    // массив из 10 элементов
    int ar[10] = { 2, 11, 17, 23, 28, 32, 44, 51, 52, 60};
```

```
int key;           // переменная, где будет искомое значение
cout << endl << " key = ";
cin >> key; // считываем ключ

bool isFind = false;
int first  = 0; // левая граница
int last   = 9; // правая граница
int middle; // индекс середины отрезка [first, last]

while ( last - first > 0)
{
    middle = first + (last - first) / 2;
    if (ar[middle] == key)
    {
        isFind = true;
        break; // выход из цикла, элемент найден
    }
    if (ar[middle] > key)
    {
        last = middle - 1;
    }
    else first = middle + 1;
}
if (isFind)
{
    cout << "Index of " << key << " equals: " << middle;
}
else cout << "There is not such item";
system("pause");
return 0;
}
```

Двоичный поиск является эффективным алгоритмом — его оценка сложности $O(\log_2(n))$, в то время как у обычного последовательного поиска $O(n)$. Это значит, что, например, для массива из 1024 элементов линейный поиск в худшем случае (когда искомого элемента нет в массиве или он находится на последнем месте) обработает все 1024 элемента, в то время как бинарным поиском достаточно обработать $\log_2(1024) = 10$ элементов. Такой результат достигается за счет того, что после первого шага цикла область поиска сужается до 512 элементов, после второго — до 256 и т. д.



Глава 7.

Функции

Функция – это логически завершённый и определённым образом оформленный фрагмент программы, который может быть вызван из других частей программы. Благодаря функциям, мы большие вычислительные задачи разбиваем на более мелкие, а также получаем возможность использовать уже написанные подпрограммы, в том числе и другими разработчиками.

В языке **C#** функции чаще называют **методами**. Они определяются в рамках объявления класса. Различают статические (со спецификатором `static`) и нестатические методы (объявляются без спецификатора). Не вдаваясь в подробности объектно-ориентированного программирования, заметим, что для наших целей нужны статические методы.

В программах на **C++** не допускается вложенность функций. Каждая программа на **C++(C#)** должна содержать функцию с именем **main(Main)**. Эта функция является точкой входа во все приложение. Вызов всех остальных функций прямо или косвенно выполняется из этой главной функции. Практически всегда для правильной и корректной работы функции необходимо передать некоторый набор данных, который мы будем называть параметрами. Параметры, которые указываются при объявлении (определении) функции, называются **формальными**, а параметры, подставляемые на место формальных при вызове функции, – **фактическими**. Отработав, функция может вернуть нам результат посредством возвращаемого значения.

7.1. Объявление и определение функции (метода)

В языке **C++**, для того чтобы иметь возможность пользоваться функцией, её необходимо сначала объявить, а затем определить или только определить. Разделение объявления и определения позволяет строить многофайловые модули, состоящие из заголовочных файлов (`.h`) и файлов реализации (`.cpp`). Рассмотрим оба эти варианта. При объявлении функции используют её описание, называемое *прототипом*. Прототип функции необходим компилятору для проверки корректности её вызова: проверяется количество параметров, порядок их следования и совместимость типов. При необходимости будут проведены соответствующие преобразования типов либо выдано сообщение об их несоответствии. Приведем формат прототипа функции:

`<тип_возвращаемого_значения> <имя_функции> (<информация о параметре1>, <информация о параметре2>, ...);`

где информация о параметре может быть представлена как:

<тип> или <тип> <имя_параметра> ,

т. е. формальные параметры могут быть как именованные, так и безымянные:

```
void function(int, float, double);
void function(int arg1, float arg2, double arg3);
```

Обе эти формы записи эквивалентны, так как компилятор игнорирует имена параметров, обращая внимание только на их тип. В конце объявления ставится «точка с запятой». Когда мы даем определение функции, необходимо сначала повторить её заголовок с использованием именованных формальных параметров, а затем расположить тело функции, заключенное в фигурные скобки.

Формат определения функции имеет вид:

```
<тип_возвращаемого_значения>      <имя_функции>(<тип>
<имя_параметра1>, <тип> <имя_параметра2>, ...)
{
    // тело функции
}
```

Если мы используем при задании функции прототип, то он должен быть размещен до функции *main*, а определение после. Если используется только определение, то оно размещается до функции *main*.

Для вызова функции необходимо выполнить следующую команду:
<имя_функции>(<параметр1>, <параметр2>, ...);

В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать.

Пример функции, возвращающей наибольшую из двух целых величин:

```
int GetMax(int , int); // объявление функции – прототип
int main(){
    int numA = 2, numB = 3, numC, numD;
    numC = GetMax(numA, numB); //вызов функции
    cin >> numD;
    cout << GetMax(numC, numD); // вызов функции
    return 0;
}
int GetMax(int a, int b) { // определение функции
    int max = (a > b) ? a : b;
    return max;
}
```

В данном примере формальным параметрам *a* и *b* функции *GetMax* соответствуют фактические *numA* и *numB*, *numC* и *numD* при её вызове.

В C# определение метода выглядит следующим образом:

```
static <тип_возвращаемого_значения> <имя_метода> (<тип>
<имя_параметра1>, <тип> <имя_параметра2>, ...)
{
    // тело метода
}
```

Приведем пример метода, вычисляющего значение функции в точке.

```
static void Main()
{
    double numA = 4.3;
    // вызов метода с вещественной переменной
    double resultA = Function(numA);
    double resultC = Function(2 * Math.PI); // вызов метода с константой
    resultC = Function(resultA * Math.PI); // вызов с выражением
}
static double Function(double x)
{
    return x * Math.Sin(x);
}
```

Методы **C#** могут размещаться как до, так и после метода **Main**.

Ключевое слово *void*, стоящее перед именем функции (метода), сообщает программе о том, что функция выполняет некоторые действия, но не имеет возвращаемых значений, т. е. не передает в вызывающую программу никакого конкретного результата. В языке C++ слово *void* внутри круглых скобок, где обычно находится список параметров, сообщает компилятору, что функция не требует передачи аргументов (*void* можно не писать).

При выходе из функции используется оператор *return*. При этом если функция имеет тип *void*, то сразу после слова *return* ставится точка с запятой (или *return* может отсутствовать).

```
void PrintResult(double x)
{
    cout << "Function( " << x << ") = " << Function(x) << endl;
}
```

Все функции (методы), которые возвращают значение, должны содержать, по крайней мере, один оператор *return*, за которым следует возвращаемое значение объявленного типа.

```
// Функция проверяет наличие нечетной цифры в целом числе
bool IsOddDigit(int number)
{
    while(number > 0)
    {
        if ( (number % 10) % 2 != 0)
            return true; // нечетная цифра найдена
        number /= 10;
    }
    return false; // нечетных цифр нет
}
```

В C++ нельзя возвращать из функции указатель на локальную переменную. Например:

```

int* fWithError()
{
    int localValue= 5;
    return & localValue; // ошибка, при выходе память из-под localValue
                        // будет освобождена
}

```

7.2. Вызов функции (метода)

Вызов функции (метода) может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция (метод). Если тип возвращаемого значения не *void*, вызов может находиться в составе выражения или, в частном случае, располагаться в правой части оператора присваивания. Любые нестатические переменные, объявленные внутри функции (метода) (в том числе и в *main*), называются локальными, размещаются в оперативной памяти (в стеке) и доступны, пока функция активна. Переменные, объявленные вне всех функций, называют глобальными. Память под глобальные переменные выделяется в сегменте данных. Локальные переменные часто называют автоматическими переменными, так как они создаются и разрушаются программой автоматически.

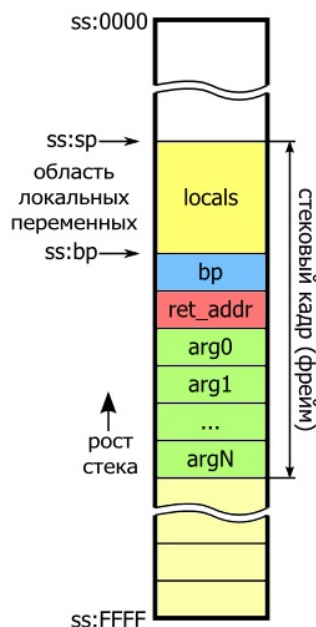


Рис. 7.1. Содержимое стека вызовов

Работа программы при наличии в ней функции (метода) происходит следующим образом. Операторы программы выполняются последовательно до тех пор, пока не встретится оператор вызова. При каждом вызове текущий адрес про-

граммы (*ret_addr*) выталкивается в стек и служит адресом возврата из функции (метода) после завершения её работы (см. рис. 7.1).

Параметры функции (*arg0, arg1, ...*) размещаются в стеке и, по сути, считаются локальными переменными. Локальные переменные, объявленные в функции, также размещаются в стеке. После возврата из функции стековая память освобождается (освобождаются все ячейки, относящиеся к функции: локальные переменные, адрес возврата и аргументы). Стек динамически изменяется, по мере того как происходят вызовы функций (методов) и возврат из них.

Этим обеспечивается:

- большее по сравнению с доступной памятью суммарное пространство, занимаемое всеми локальными переменными;
- бесконфликтное объявление одноименных локальных переменных, используемых в различных функциях одной программы.

В C++ есть возможность сохранить значения локальных переменных при выходе из области видимости, а также между вызовами одной и той же функции. Для этого необходимо использовать при объявлении локальной переменной модификатор **static**. Приведем пример использования статической переменной.

```
#include <iostream>
using namespace std;
void function(int a){
    cout << "step n  m"<<endl;
    for (int i = 0; i < a; i++)
    {
        // n – статическая, значение сохраняется между итерациями цикла
        static int n = 0;
        // m – локальная, создается заново при каждой итерации цикла
        int m = 0;
        cout << i + 1 << ' ' << ++n << ' ' << ++m << endl;
    }
}
int main(){
    function(3);
    return 0;
}
```

Статическая переменная **n** размещается в сегменте данных и инициализируется один раз при первом выполнении оператора, содержащего ее определение. Программа выведет на экран:

step	n	m
1	1	1
2	2	1
3	3	1

7.3. Способы передачи параметров

Параметры, которые мы указываем при объявлении (определении) функции, являются основным способом обмена информацией между вызывающей и вызываемой функциями.

В **C++** передача параметров осуществляется двумя способами: по значению и по адресу (ссылка или указатель). Способ передачи определяется видом объявления параметра в заголовке функции.

Если в заголовке функции перед именем параметра указан только его тип, то такой параметр передается по значению (**параметр-значение**). При вызове функции в стек помещается копия параметра и функция работает с ней (см. рис. 7.2).



Рис. 7.2. Параметр-значение

Так как функция работает с копией параметра, то любые его изменения внутри функции не видны в вызывающей функции. Для того чтобы получить измененный параметр, его передают по адресу.

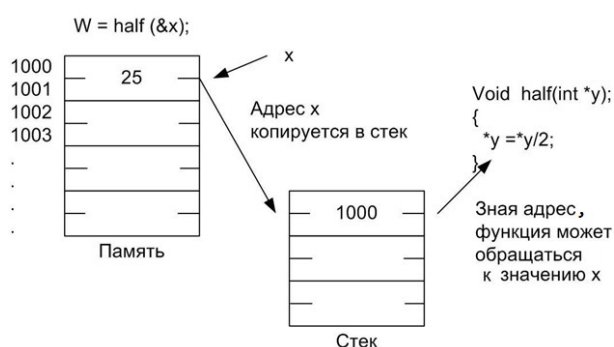


Рис. 7.3. Параметр-указатель

Если в заголовке функции перед именем параметра указан тип и символ *****, то такой параметр передается по адресу и называется **параметром-указателем**. Для передачи в функцию адреса фактического параметра используется операция взятия адреса, а для получения его значения в функции требуется операция разыменования. Так как функция работает с адресом переменной, то любые изменения по этому адресу видны в вызывающей функции (см. рис. 7.3).

Если в заголовке функции перед именем параметра указан тип и символ `&`, то такой параметр передается по адресу и называется **параметром-ссылкой**. Ссылка – это разыменованный указатель, следовательно, любые изменения ссылки видны в вызывающей функции. Приведем пример использования всех трех способов передачи параметров:

```
#include <iostream>
using namespace std;
void function(int value, int* ptr, int& ref);
// val : параметр-значение, ptr : параметр-указатель
// ref : параметр-ссылка
int main()
{
    int val = 1, ptr = 2, ref = 3;
    cout << " value  ptr  ref" << endl;
    cout << val << " << ref << " << ptr << endl;
    function(val, &ptr, ref);
    cout << val << " << ref << " << ptr << endl;
    return 0;
}
void function(int value, int* ptr, int& ref);
{
    value++; (*ptr)++; ref++;
}
```

Результат работы программы:

value	ptr	ref
1	2	3
1	3	4

В отличие от параметра-указателя доступ по параметру-ссылке не требует операции разыменования, упрощается организация тела функции – в нем можно использовать просто имена параметров-ссылок, а также упрощается вызов такой функции – на месте фактических аргументов тоже можно писать просто имена объектов (т. к. имена объектов и ссылки на них являются эквивалентами).

В качестве **фактических аргументов**, соответствующих **параметрам-значениям**, могут быть заданы любые числовые выражения (формулы):

```
double average(double x, double y)
{
    return (x + y) / 2.;
}
double w1 = average(x * cos(fi) + y * sin(fi), x * sin(fi) - y * cos(fi));
```

Одним из наиболее распространенных способов использования *параметров-указателей* является передача в качестве адреса имени массива. Например, для вывода элементов массива на экран можно воспользоваться следующей функцией:

```

    int PrintArray(int *a, int n)
    {
        for(int j = 0; j < n; j++) cout << a[j] << endl;
    }

```

Обращение к такой функции может выглядеть следующим образом:

```

int Array[20];
.....
k1 = PrintArray(Array,20);    // печать всех элементов массива
k2 = PrintArray(Array,10);    // печать первых 10 элементов массива
k3 = PrintArray(&Array[5],3); // печать Array[5], Array[6] и Array[7]
k4 = PrintArray(Array+5,3);   // печать Array[5], Array[6] и Array[7]

```

Не забывайте, что имя массива одновременно является и указателем на его первый элемент (т. е. `Array` и `&Array[0]` – это одно и то же). Приведем пример использования *параметра-ссылки* при чтении одномерного динамического массива с клавиатуры:

```

void CreateArray(int *&a, int &n)
{
    cout << "N = "; cin >> n;
    a = new int[n];
    cout << "Input array : " << endl;
    for( int i = 0; i < n; i++) cin >> a[i];
}

```

После получения от пользователя информации о размере массива выделяется необходимая память и выполняется чтение. Параметры-ссылки гарантируют доступность измененных данных. Обращение к такой функции может выглядеть следующим образом:

```

int *massiv = NULL, n;
CreateArray(massiv, n);
PrintArray(massiv, n);

```

В языке **C#** параметры в подпрограмму передаются по **ссылке** или по **значению**. Если тип параметра значимый (число, символ, булевское значение), то по умолчанию используется передача по значению, при этом передается копия объекта, а не сам объект. Поэтому изменения в аргументе не оказывают влияния на исходную копию в вызывающем методе.

```

static void ParametrValue(int number)
{
    number = 2;
}
static void Main(string[] args)
{
    int number = int.Parse(Console.ReadLine()); // number = 5
    ParametrValue(number);
    Console.WriteLine(number); // вывод: 5
}

```

```
}
```

Для того чтобы иметь возможность влиять на внешнюю переменную, передаваемую в качестве параметра, используются ключевые слова **ref** и **out**. Таким образом происходит передача параметра по ссылке. Например:

```
static void SwapRef(ref int x, ref int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
static void Main()
{
    int num1 = -5, num2 = 13;
    Console.WriteLine($"num1 = { num1}  num2 = { num2}");
    SwapRef (ref num1, ref num2);
    Console.WriteLine("num1 = {0}  num2 = {1}", num1, num2);
}
```

Результат работы программы:

```
num1 = -5    num2 = 13
num1 = 13    num2 = -5
```

Особенностью **ref** является то, что переменная, которую мы передаем в метод, обязательно должна быть проинициализирована значением, иначе компилятор выдаст ошибку «**Use of unassigned local variable**». Это является главным отличием **ref** от **out**. Модификатор **out** позволяет передать непроинициализированный параметр, но при этом в методе ему обязательно должно быть присвоено новое значение, в противном случае мы получим ошибку:

```
public static void ChangeValue(out int number)
{
    number = -5;
}
static void Main(string[] args)
{
    int number;
    ChangeValue(out number);
    Console.WriteLine(number); // вывод: -5
    Console.ReadKey();
}
```

Если тип параметра ссылочный (массив, матрица, строка), то по умолчанию происходит передача адреса объекта и поэтому все изменения в методе, кроме создания нового объекта, отражаются в вызывающей программе. Для полного контроля метода над ссылочным параметром также используется модификатор **ref**.


```
static void Change(int[] arr, ref int[] pArr)
{
    arr[0] = 88;
    arr = new int[5] {-33, -1, -2, -3, -4};
    pArr[0] = 88;
    pArr = new int[5] {-33, -1, -2, -3, -4};
    Console.Write ("Внутри метода aArray[0] = {0}", arr[0]);
    Console.WriteLine(" pArray[0] = {0}", pArr[0]);
}
static void Main()
{
    int[] aArray = {10, 40, 50};
    int[] pArray = {10, 40, 50};
    Console.Write("Main, до вызова метода aArray[0] = {0}", aArray[0]);
    Console.WriteLine(" pArray[0] = {0}", pArray[0]);
    Change(aArray, ref pArray);
    Console.Write("Main, после вызова Change aArray[0] = {0}", aArray[0]);
    Console.WriteLine(", pArray[0] = {0}", pArray[0]);
    Console.ReadKey();
}
```

Результат работы программы:

```
Main, до вызова метода aArray[0] = 10, pArray[0] = 10
Внутри метода aArray[0] = -33, pArray[0] = -33
Main, после вызова Change aArray[0] = 88, pArray[0] = -33
```

В примере массив **aArray** имеет ссылочный тип и передается в метод **Change** без параметра **ref**, т. е. по значению передается ссылка. Изменения внутри массива отражаются в исходном массиве, но изменение самой ссылки (при создании нового массива) не происходит. Таким образом, после вызова метода **Change** любые ссылки на **aArray** будут указывать на старый массив из трех элементов.

Однако все изменения, выполняемые внутри метода **Change**, влияют на исходный массив **pArray**, так как он передается с параметром **ref**. Фактически происходит замещение исходного массива с помощью оператора **new**. Таким образом, после вызова метода **Change** любые ссылки на **pArray** будут указывать на новый массив из пяти элементов.



Заключение

В первой части учебного пособия помещен материал, который пригодится начинающему программисту. Все разобранные задачи и алгоритмы не являются новыми, большинство формулировок взято из [3], [4], но код их решения написан авторами. Более подробную информацию о синтаксисе языков программирования можно посмотреть в книгах [1], [2] и на следующих сайтах:

1. <https://code.msdn.microsoft.com/> – библиотека официальной технической документации для разработчиков под ОС Microsoft Windows.
2. <https://docs.microsoft.com/ru-ru/dotnet/csharp/> – руководство по языку C#;
3. <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/> – руководство по программированию на C#;
4. <https://docs.microsoft.com/ru-ru/cpp/> – справочник по языку C++.

Полезной практикой будет решение задач и их проверка в автоматической тестирующей системе. Для этих целей рекомендуются следующие ресурсы:

1. <https://informatics.msk.ru/> – на этом сайте можно сдавать задачи по темам (ветвление, циклы...). В каждом разделе есть необходимый теоретический материал.
2. <http://acmp.ru> – кроме обилия олимпиадных задач, на этом сайте размещены курсы по программированию.
3. <http://acm.timus.ru/> – один из крупнейших архивов задач по программированию.
4. <https://codeforces.com/?locale=ru> – очень много задач и соревнований.

Бесплатное обучение языкам программирования доступно на следующих сайтах:

1. <https://stepik.org/course/363/> – введение в программирование (C++);
2. <https://stepik.org/course/4965> – основы C Sharp;
3. <https://stepik.org/course/5482> – основы программирования (C#);
4. <https://www.intuit.ru/> – сайт национального открытого университета. Содержит не один десяток курсов по программированию.

Литература

- [1] Павловская Т. А. С/С++ Программирование на языке высокого уровня. СПб.: Питер, 2019. 464 с.
- [2] Троелсен Э., Джепикс Ф. Язык программирования С# 6.0 и платформа .NET 4.6. Вильямс, 2016. 1440 с.
- [3] Шень А. Программирование. Теоремы и задачи. М.: МЦНМО, 2017. 320 с.
- [4] Дистанционная подготовка по информатике.
URL: <https://informatics.mccme.ru/>
- [5] Справочник по языку С++. URL: <https://docs.microsoft.com/ru-ru/cpp/>
- [6] Руководство по языку С#. URL: <https://docs.microsoft.com/ru-ru/dotnet/csharp/>

Учебное издание

Власова Ольга Владимировна
Федотова Наталья Петровна
Якимова Ольга Павловна

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Редактор, корректор Л. Н. Селиванова
Компьютерная верстка О. П. Якимова,
О. В. Власова, Н. П. Федотова

Подписано в печать 2.08.2019. Формат 60×84 1/8.

Усл. печ. л. 9,76. Уч.-изд. л. 7,0.

Тираж 25 экз. Заказ

Оригинал-макет подготовлен
в редакционно-издательском отделе ЯрГУ.

Ярославский государственный университет
им. П. Г. Демидова
150003, Ярославль, ул. Советская, 14