Петрозаводский государственный университет

Р. В. Воронов,
Н. А. Горинов,
А. В. Сысун

Основы программирования на языке C#

Учебное пособие
для студентов направления
Бизнес-информатика

Петрозаводск
2014

Оглавление

§1. Введение

В пособии рассматривается язык программирования C# (читается как «си шарп»), работающий на платформе Microsoft .NET (dot NET или точка NET). Данный язык успешно сочетает в себе легкость освоения и гибкость для профессионального применения; специалисты со знанием .NET и языка C# очень востребованы на рынке труда. Кроме того платформа и язык активно развиваются; в новые версии языка добавляются существенные улучшения, что дает основания предполагать, что изучаемые инструменты будут актуальны через 5 лет и полезны студентам как прикладные знания, необходимые для осуществления профессиональной деятельности.

Язык C# является наиболее популярным из языков, работающих на платформе .NET. Его невозможно рассматривать отдельно от платформы Microsoft .NET, специально для которой он был создан. Изучение других языков остается за пределами рамок данного курса, поэтому при рассмотрении возможностей будем говорить о них вместе, не уточняя к чему относится та или иная особенность: к языку или платформе.

Основной особенностью платформы является процесс трансляции и выполнения кода. Процесс перевода программы в машинный код подразделяется на компиляцию и интерпретацию. При компиляции сначала происходит трансляция текста с языка программирования в машинные коды и после этого идет выполнение. При интерпретации каждая команда транслируется непосредственно перед ее выполнением. Для лучшего понимания отличий можно представить чтение книги на иностранном языке. Если книга сначала переводится на русский язык, а потом читается — то это процесс, соответствующий компиляции. В противном случае при чтении сразу на иностранном языке и подсматривании в словаре очередного слова процесс соответствует интерпретации. Очевидно, что книгу, однажды переведенную на русский язык, намного быстрее впоследствии читать. Но при чтении книги непосредственно на языке оригинала больше возможностей по передаче тонкостей изложения от автора книги к читателю.

То, как именно происходит процесс трансляции, зависит от языка программирования и определяется его авторами при создании. Программы, написанные на компилируемых языках имеют преимущество в виде большей скорости выполнения. Интерпретируемые же языки программирования дают большую гибкость в средствах языка и обеспечивают переносимость между различными платформами. Платформа .NET применяет подход, являющийся промежуточным между вышеописанными. Компиляция происходит не в машинные коды, а в промежуточный язык (Intermediate language), который впоследствии исполняется не физическим, а эмулируемым процессором с более развитой и более гибкой системой команд.

§2. Как установить Microsoft Visual Studio .NET

Существует несколько способов установки Microsoft Visual Studio .NET. Самый простой из них – это загрузить с сайта http://www.microsoft.ru. Visual studio предлагается в нескольких вариантах. Вариант Visual Studio Express бесплатен, и далее пойдет речь об установке именно этой версии. Ее возможности ограничены по

сравнению с Visual Studio Professional или другими платными версиями, но достаточны для обучения. На сайте необходимо пройти в меню «Продукты->Сервер и Инструменты->Visual Studio».

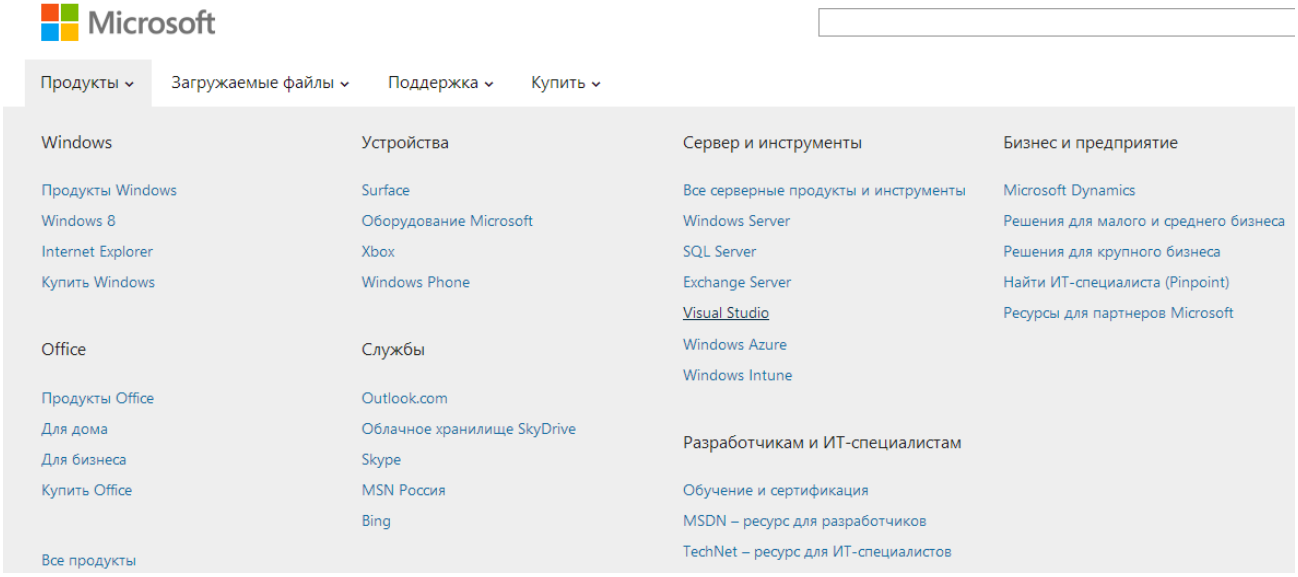


Рис 1. Сайт Microsoft

В результате откроется домашняя страница Visual Studio, где можно найти все доступные выпуски Visual Studio. У корпорации Microsoft существует семейство бесплатных выпусков под названием Visual Studio Express. Продуктами этого семейства можно пользоваться для ознакомления в течение 30 дней, а для полноценного использования достаточно пройти бесплатную регистрацию на сайте http://www.microsoft.ru. Для выполнения лабораторных работ следует выбрать в меню «Продукты->Express для Windows Desktop».

Или сразу пройти по прямой ссылке http://www.microsoft.com/visualstudio/rus/products/visual-studio-express-for-windows-desktop

и нажать «Загрузка».

Пользователю доступно два варианта загрузки:

- Загрузить. При выборе этого варианта на ваш персональный компьютер будет загружен файл до 1 МБ, который необходимо запустить для дальнейшей установки. Visual Studio будет загружаться прямо в процессе установки. Этот вариант проще для пользователя.

- Установить. При выборе этого варианта будет загружаться установка Visual Studio в формате .iso, объемом от 600 МБ. После завершения загрузки, установку необходимо монтировать. Этот вариант предпочтителен для нестабильного подключения к сети Интернет.

Рассмотрим подробнее первый вариант. В результате запуска загруженного файла будет показана информация о лицензии, которую необходимо прочитать и согласиться, рис. 2. После этого появится кнопка «Установить» (требует права администратора), нажатие на которую приведет к началу установки Visual Studio. Через некоторое время будет необходимо перезагрузить операционную систему, рис. 3.

Рис 2. Начальное положение

Рис.3. Перезагрузка.

После перезагрузки процесс установки Visual Studio будет продолжен, рис. 4. И через некоторое время успешно завершен, рис. 5. Теперь можно приступать к работе.

Рис. 4. Продолжение установки

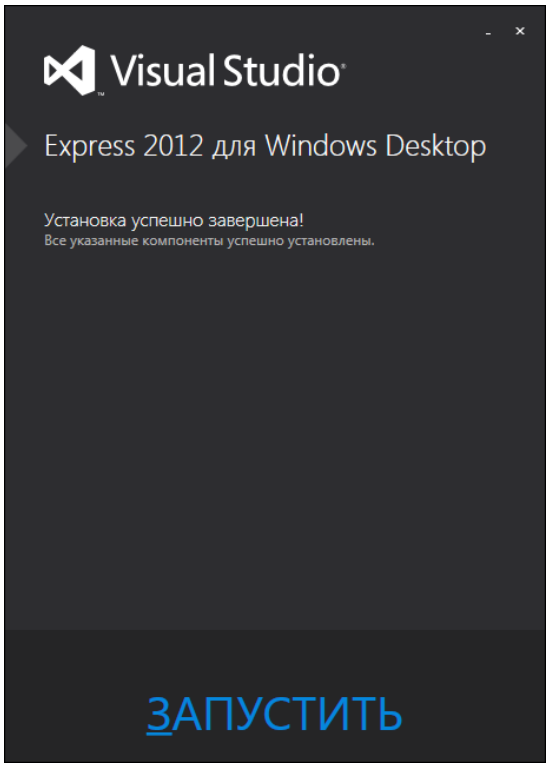
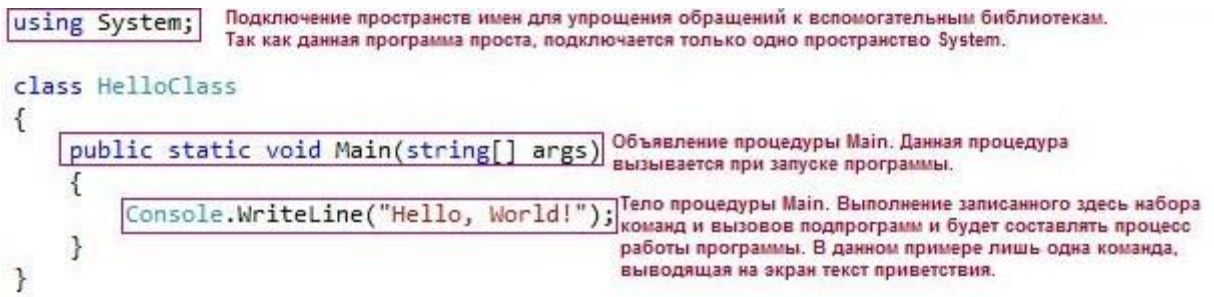


Рис. 5 Окончание

Для ознакомления с Visual Studio можно не вводить лицензионный код в течение 30 дней, но для полноценного использования код нужно получить, пройдя бесплатную регистрацию на сайте http://www.microsoft.ru.

§3. Основы языка C#

Рассмотрим код первой программы на языке C#:



Оформление комментариев

Комментарии в программе используются для пояснения программисту отдельных ее частей. При компиляции комментарии игнорируются, и они никак не влияют на работу программы.

В языке C# имеется два вида комментариев – однострочные и многострочные:

```
// такой комментарий следует до конца строки
/* а такой комментарий
можно записывать
в несколько строк */
```

Советы по комментированию программ:

1) в начале текста программы вставьте краткий комментарий, включающий автора программы, описывающий решаемую задачу и реализуемый алгоритм;
2) не поясняйте используемые инструкции языка программирования, а пишите для чего вы их используйте, поясняйте свои намерения;
3) не следует писать очевидные комментарии;
4) для лучшего понимания программы вместо некоторых комментариев используйте говорящие названия переменных, функций и классов;
5) не вставляйте комментарий в середину строки между элементами программы.

Алфавит языка

Любая программа на языке C# написана при помощи символов, входящих в его алфавит. Алфавит языка состоит из:

1) латинских прописных и строчных букв: A, B,…, Z, a, b ,…, z;
2) арабских цифр: 0, 1,…, 9;
3) специальных символов: + - * / < > = | & ! \ ~ ‘ @ # $ % ? _ : ; , . ( ) [ ] { } “

Символом разделителем является пробел.

Идентификатор – это последовательность символов из латинского алфавита, арабских цифр и символа подчеркивания, которая начинается не с цифры. Идентификаторы используются для именования различных элементов программы. Примеры идентификаторов:

a, x1, x2, mas, n_student

Операции задают некоторое действие. Элементы, над которыми выполняется операция, называются аргументами или *операндами*. Число операндов является *арностью* операции. Операция с одним операндом называется унарной, с двумя – бинарной, с тремя – тернарной.
Пример унарной операции:

-a (минус – изменение знака).

Примеры бинарных операций:

- a + b (сложение);
- a - b (вычитание);
- a * b (умножение);
- a / b (деление);
- a % b (остаток от деления).

Пример тернарной операции:

a ? b : c (если a истинно, то b, иначе – c).

Рассмотрим другие примеры операций:

- Операция простого присваивания: x = 5;
- Унарная операция инкремента (увеличение переменной на единицу): a++;
- Унарная операция декремента (уменьшение переменной на единицу): a--;
- Составные операции присваивания: a += b (то же, что и a = a + b).

Аналогично строятся составные операции присваивания для вычитания, умножения и деления.

Выражение – это последовательность знаков операций, операндов и круглых скобок, которая задает вычислительный процесс получения результата. Операндами выражения могут выступать константы, переменные, выражения и вызовы функций.
Пример выражения:

(-b + Math.Sqrt (D)) / (2 * a)

Запись действий, которые должен выполнить компьютер, состоит из операторов. Операторы:

- выполняются последовательно;
- один за другим;
- заканчиваются точкой с запятой;
- могут содержать несколько действий, которые разделены запятой.

Рассмотрим примеры некоторых операторов.

Пустой оператор содержит только точку с запятой:

;

Выражение, после которого стоит точка с запятой, – это оператор-выражение. При выполнении этого оператора вычисляется значение выражения. Оператор, в котором используется операция присваивания, называют оператором присваивания.
Пример:

c = a + b;

Операторы можно объединять в блоки (составные операторы) при помощи фигурных скобок. Пример:

```
{
        a = 0;
        b = a + 5;
}
```

Составной оператор заключается в последовательном выполнении вложенных в него операторов и рассматривается как единый оператор.

Различают также операторы объявления имен и операторы управления.

§4. Типы данных

Все данные программы хранятся в ячейках памяти компьютера. Вид данных зависит от их типа. *Тип данных* определяет множество допустимых значений, множество разрешенных операций над этими значениями и способ их хранения в памяти. Примерами типов данных могут быть «целое число», «вещественное число», «строка» и т. д. В зависимости от типа под значение отводится определенное число ячеек памяти, некоторым образом выполняются операции в процессоре компьютера. Так, одно и то же с точки зрения математики число (например, 100) будет по-разному храниться для целого и вещественного типа. Встроенные в процессор алгоритмы сложения целых чисел отличаются от алгоритмов сложения вещественных чисел.

Базовые типы данных

Базовые (встроенные или простые) типы данных перечислены в таблице 1. Эти типы предопределены и встроены в язык C#. Базовые типы делятся на числовые (целочисленные и вещественные), символьные, логические и строковые. Целочисленные типы бывают знаковые и беззнаковые.

Таблица 1. Базовые типы данных

Обозначение	Диапазон	Размер	Назначение
sbyte	-128...127	1 байт	целое со знаком
byte	0…255	1 байт	целое без знака
short	-32768…32767	2 байта	целое со знаком
ushort	0...65565	2 байта	целое без знака
int	-2147483648… 2147483647	4 байта	целое со знаком
uint	0...4294967295	4 байта	целое без знака
long	-9223372036854775808… 9223372036854775807	8 байт	целое со знаком
ulong	0…18446744073709551615	8 байт	целое без знака
char	0000…FFFF	2 байта	символ Unicode
float	±1,5*10-45… ± 3,4*1038	4 байта	число с плавающей запятой
double	± 5,0*10-324… ± 1,7*10308	8 байт	число с плавающей запятой
bool	true, false	1 байт	булевский
string		ограничено памятью	набор символов Unicode

Числовые типы в таблице 1 представлены в порядке от низших (sbyte и byte) к высшим (double). Чем больше памяти отводится под хранения значений, тем выше тип.

Далее рассмотрим подробнее базовые типы данных.

Пустой тип

Тип void (пустой тип) не имеет значений; он используется чаще всего для описания функций, не возвращающих значения.

Целочисленные типы

К целочисленным типам относятся: sbyte, byte, short, ushort, int, uint, long, ulong. Множество значений – целые числа из некоторого диапазона. Например, наиболее популярный тип int представляет из себя множество целых чисел в интервале от -2^{31} до $(2^{31}-1)$. Существует два вида целочисленных типов: знаковые и беззнаковые. Знаковые типы могут хранить положительные и отрицательные значения, а также нуль. Беззнаковые – только неотрицательные (то есть положительные и нуль).

Операции над целыми числами:

- a + b (сложение);
- a - b (вычитание);
- a * b (умножение);
- a / b (целочисленное деление);
- a % b (остаток от деления).

Побитовые операции над целыми:

- ~a (Побитовое НЕ);
- a << b (Сдвиг влево);
- a >> b (Сдвиг вправо);
- a & b (Побитовое И);
- a ^ b (Побитовое исключающее ИЛИ);
- a | b (Побитовое ИЛИ).

Вещественный тип

К вещественным типам относятся float и double. Множество значений – вещественные числа из некоторого диапазона, хранящиеся с заданной точностью.

Операции над вещественными числами:

- a + b (сложение);
- a - b (вычитание);
- a * b (умножение);
- a / b (деление).

Булевский тип

Булевский (логический) тип bool характеризуется двумя значениями: true (истина) и false (ложь).

Пример:

```
bool f1 = 5 > 1;        // f1 = true
bool f2 = 5 < 1;        // f2 = false
```

Рассмотрим операции над булевским типом. Все они заданы при помощи таблиц истинности.

Операция логическое И (конъюнкция) &&

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

Операция логическое ИЛИ (дизъюнкция) ||

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

Операция логическое исключающее ИЛИ ^

A	B	A ^ B
false	false	false
false	true	true
true	false	true
true	true	false

Операция логическое НЕ (отрицание) !

A	!A
false	true
true	false

Рассмотрим операции сравнения чисел, результат которых – булевский тип (истина или ложь):

a < b (меньше);
a > b (больше);
a<= b (меньше или равно);
a >= b (больше или равно);
a == b (равно);
a != b (не равно).

Пример логического условия:

(a > 10 || b >= 10) && (a <= 0 || b < 0)

Символьный тип

Символьный тип char используется для хранения 16-разрядных символов в коде Unicode. Константы символьного типа пишутся в одинарных кавычках, например: ‘A’. Символы ’\n’ и ’\t’ служат, соответственно, для переноса курсора при печати на новую строку и для знака табуляции.

Строковый тип

Строковый тип string используется для хранения строк, представляющих из себя последовательности символов типа char. Константы строкового типа пишутся в двойных кавычках, например: "ABC".

Пример выполнения операций над строками:

```
string s1 = "Петр";
string s2 = "ГУ";
string s3 = s1 + s2;          // s3 = "ПетрГУ"
char a = s1[0];               // a = 'П'
```

Пример преобразования строки в число:

```
string s = "125";
int n = int.Parse(s);
```

Пример преобразования числа в строку:

```
int n = 125;
string s = n.ToString();
```

Преобразование типов

Иногда возникает необходимость преобразовывать данные из одного типа в другой. Это возможно в случаях, когда в выражении используются переменные или константы разного типа либо тип переменной слева от знака присваивания отличается от типа выражения справа от знака присваивания. Преобразование типов может быть двух видов: явное и неявное.

Неявное преобразование не задается специально программистом и выполняется автоматически без явно указанной команды. Неявные преобразования не приводят к потере информации. Для числовых типов разрешены следующие неявные преобразования:

- знаковых целочисленных типов к высшим знаковым целочисленным и вещественным типам;
- беззнаковых целочисленных типов к высшим числовым типам;
- типа float к типу double.

Например, разрешено неявное преобразование типов int и uint к типу long, а типа long к типу int – нет.

Примеры неявных преобразований:

```
int x = 5;
long y = 10;
float z = x + y;
```

Явное преобразование типа задается программистом явно специальным образом. Для этого необходимо в скобках указывать тип, к которому происходит преобразование:

(тип) выражение

Рассмотрим пример, в котором явное преобразование используется для выполнения вещественного деления двух целых чисел:

```
int a = 1, b = 2;
float x;
x = (float) a / (float) b;
```

Если бы в данном примере не выполнялось явное преобразование, то деление было бы целочисленным.

При явных преобразованиях возможна потеря информации.

Типы: значения и ссылки

Типы, характеризуемые значениями, включают все числовые типы данных, символьный тип, булевский тип, перечни и структуры. Все они размещаются в стеке. При присваивании происходит создание копии – «побайтовое» копирование.

Большинство остальных типов являются ссылочными. Ссылочные типы размещаются в динамической памяти. При присваивании копируется ссылка (адрес) на объект. Создание объекта выполняется только при помощи оператора new. Исключением являются строки (тип string). Для строк можно не использовать оператор new, при присваивании происходит создании копии.

§5. Переменные и константы

Переменные используются в программе для хранения результатов вычислений. Переменные могут менять свои значения в ходе выполнения программы. Каждая переменная имеет уникальное имя и тип. Рассмотрим оператор объявления переменной:

тип имя_переменной;
тип имя_переменной = значение;

Например:

```
int a;
long b = 100;
```

Допускается множественное объявление переменных:

```
int a, c = 0;
long b = 100, d = 500;
```

Константы также используются для хранения данных, но, в отличие от переменных, они не меняют свои значения. Константы могут быть двух видов:

1) литеральные
2) символические

Тип и значение литеральной константы определяется ее внешним видом:

1) числовые, например: 17, 3.14
2) символьные, например: ’A’, ’B’, ’C’
3) строковые, например: ”Петрозаводск”

Символическим константам соответствуют идентификаторы, которым присвоены определенные значения. Эти значения не могут быть изменены в ходе выполнения программы. Для объявления символических констант необходимо указать их тип, имя и значение:

const тип имя_константы = значение;

Пример:

```
const int five = 5;
```

```
const float PI = 3.14;
```

§6. Управляющие операторы

Управляющие операторы используются в качестве средств управления порядком выполнения программы, организации ветвлений и циклов. Далее рассмотрим основные управляющие операторы языка C#, такие как:

- if
- if … else
- switch
- for
- while
- do … while
- foreach

Ветвления

Если какие-то фрагменты кода должны исполняться лишь в случае истинности некоторых логических условий, то для этого необходимо использовать ветвления. Для реализации ветвлений в языке C# используются операторы if, if … else, switch.

Условный оператор if

```
if (выражение)
        оператор
```

Если выражение истинно, то выполняется оператор, иначе – не выполняется.
Пример:

```
if (x >= 0)
        y += x;
```

Условный оператор if…else

```
if (выражение)
        оператор1
else
        оператор2
```

Если выражение истинно, то выполняется оператор1, иначе – оператор2.
Пример:

```
if (x >= 0)
        y += x;
else
        z++;
```

Оператор выбора switch

```
switch (выражение)
        {
                case константа1: оператор1
                case константа2: оператор2
```

```
        *       *       *       *       *
        case константаN: операторN
        [default: оператор]
}
```

Сначала вычисляется выражение, затем происходит переход на метку case со значением константы, равной значению выражения, и выполняется оператор, находящийся после этой метки. После каждого оператора обязательно должна стоять команда break. По команде break происходит выход из оператора switch. Если значение выражения не найдено, то происходит переход на метку default, которая является необязательной.

Пример:

```
x = 5;
a = 3;
switch (a)
        {
                case 3: y = x; break;
                case 6: y = -x; break;
                case 15: y = x * x; break;
                default: y = 0; break;
        }
```

Циклы

Если в программе возникает необходимость многократного выполнения одних и тех же действий, то для этого можно использовать циклы. Цикл – это участок кода программы, в котором одни и те же действия повторятся несколько раз. Для реализации циклов в языке C# используются операторы while, do … while, for, foreach.

Оператор цикла с предусловием while

```
while (выражение)
        оператор
```

Пока выражение истинно выполняется оператор.

Шаг 1. Если выражение истинно, то перейти на шаг 2, иначе закончить цикл.

Шаг 2. Выполнить оператор, перейти на шаг 1.

Чтобы цикл не был «вечным», внутри оператора должны быть выполнены действия, меняющие выражение. Каждое выполнение оператора называется итерацией цикла.

Пример:

```
s = 0; a = 1;
while (a <= 5)
{
        s += a;
        a++;
}
```

Оператор цикла с постусловием do…while

```
do
```

```
        оператор
    while (выражение)
```

Оператор выполняется до тех пор, пока выражение истинно. На первой итерации цикла оператор выполняется всегда.

Шаг 1. Выполнить оператор, перейти на шаг 2.

Шаг 2. Если выражение истинно, то перейти на шаг 1, иначе закончить цикл.

Пример:

```
s = 0; a = 2;
do
{
    s += a;
    a--;
}
while (a > 0)
```

Оператор цикла for

```
for (выражение1; выражение2; выражение3)
    оператор
```

Вначале вычисляется выражение1. Затем до тех пор, пока значение логического выражения2 истинно, последовательно выполняются оператор и вычисляется выражение3.

Шаг 1. Вычислить выражение1, перейти на шаг 2.

Шаг 2. Если выражение2 истинно, то перейти на шаг 3, иначе закончить цикл.

Шаг 3. Выполнить оператор, перейти на шаг 4.

Шаг 4. Вычислить выражение3, перейти на шаг 2.

Как правило, в цикле for используется специальная переменная – счетчик итераций цикла. Чаще всего, выражение1 устанавливает начальное значение счетчика цикла, выражение2 проверяет значение счетчика, а выражение3 изменяет его значение. Оператор называется телом цикла, в качестве него может быть использован простой или составной оператор.

Пример:

```
s = 0; n = 5;
for (i = 1; i <= n; i++)
    s += i;
```

Оператор цикла foreach

```
foreach (переменная in контейнер)
    оператор
```

Перебираются все значения из контейнера, и на каждой итерации цикла очередное из них присваивается переменной, и выполнятся оператор. В качестве контейнера может выступать массив. Пример печати элементов массива:

```
foreach (int a in A)
    Console.WriteLine(a);
```

Оператор выхода из цикла break

Оператор break применяется для прерывания выполнения цикла. Применяется в операторах цикла while, do…while, for, foreach и switch. Если оператор break

используется во вложенных циклах, то он прерывает выполнение только того цикла, в котором находится.

Оператор выхода из итерации цикла continue

Оператор continue применяется для прерывания выполнения текущей итерации цикла и перехода на следующую итерацию цикла. Применяется в операторах цикла while, do…while, for и foreach. В операторах цикла while и do…while при выполнение оператора continue происходит переход на проверку условия; в операторе for – на выполнение выражения3; в операторе foreach – переход к следующему элементу из контейнера.

Оператор перехода goto

```
goto метка
        *       *       *       *       *
        метка: оператор
```

Оператор goto осуществляет переход к оператору, следующему за меткой. Желательно избегать применения этого оператора.

Пример:

```
goto 7;
/***************/
7: puts(“A”);
```

§7. Массивы

Массив – набор однотипных объектов, имеющих общее имя и различающихся номером (индексом). Элементы массива располагаются последовательно друг за другом и занимают один непрерывный участок памяти. Доступ к элементу массива осуществляется указанием после имени массива в квадратных скобках индекса элемента.

При *объявлении* массива указывают тип его элементов, к типу добавляют квадратные скобки, затем указывают имя массива. Для *создания* массива (выделения памяти под его элементы) используют оператор new, после которого в квадратных скобках указывают число его элементов. *Инициализация* массива – присвоение начальных значений его элементам при объявлении.

Нумерация элементов массива начинается с нуля. Это означает, что если в массиве n элементов, то его индексы имеют значения от 0 до n-1.

Рассмотрим на примерах объявление, создание и инициализацию массивов.

Объявления целочисленного массива:

```
int[] A;
```

Создания целочисленного массива из пяти элементов:

```
A = new int [5];
```

Одновременное объявление и создание массива:

```
int[] A = new int [5];
```

Объявление, создание и инициализация массива:

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

либо:

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

либо:

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

Пример присвоения нулевому элементу массива значения 147:

```
A[0] = 147;
```

Копирование пятого элемента в целочисленную переменную n:

```
int n = A[5];
```

Рассмотрим пример программы, заполняющей с клавиатуры одномерный строковый массив из N элементов:

```
string t = Console.ReadLine();
int N = int.Parse(t);

string[] str = new string [N];

for (int i=0; i<N; i++)
    str[i] = Console.ReadLine();

foreach (string s in str)
    Console.WriteLine("{0}", s);
```

Для обращения к элементам, рассмотренных выше массивов, используется один индекс, поэтому такие массивы называются одномерными. В двумерных массивах для обращения к элементам используется два номера.

Число элементов первого измерения обычно называют числом строк. В языке C# двумерные массивы бывают двух видов: прямоугольные и ступенчатые. В прямоугольных массивах в каждой строке одинаковое количество элементов, обычно это количество называют числом столбцов. В ступенчатых – разное.

При создании двумерного прямоугольного массива необходимо указывать число строк и столбцов. При создании двумерного ступенчатого массива вначале необходимо выделить память под одномерный массив ссылок на одномерные массивы. Затем для каждой строки нужно выделить требуемый размер памяти.

Рассмотрим пример объявления и создания двумерного статического массива:

```
int[,] A = new int [2, 3];
```

В этом примере объявляется двумерный массив размером 2 на 3. В данном случае часто говорят, что объявлен массив из двух строк и трех столбцов.

При обращении к элементам двумерного прямоугольного массива необходимо через запятую перечислить все индексы:

```
A[0, 1], A[i, j]
```

Пример инициализации при объявлении двумерного прямоугольного массива:

```
int[,] A = {{5, 7, 3},{1, 4, 9}};
```

Двумерные прямоугольные массивы удобно изображать в виде таблиц. Каждый элемент таблицы имеет пару индексов – номер строки и номер столбца. Изображение объявленного выше массива в виде таблицы:

5	7	3
1	4	9

Рассмотрим пример программы, заполняющей введенными пользователем с клавиатуры числами двумерный прямоугольный массив:

```
int[,] mas = new int [5,7];
```

```
for (int i=0; i<5; i++)
        for (int j=0; j<7; j++)
                mas[i,j] = i*j;
for (int i=0; i<5; i++)
{
        for (int j=0; j<7; j++)
                Console.Write("{0}\t", mas[i,j]);
        Console.WriteLine();
}
```

При обращении к элементам двумерного ступенчатого массива необходимо каждый индекс записывать в отдельные квадратные скобки:

A[0][1], A[i][j]

Рассмотрим пример программы, заполняющей введенными пользователем с клавиатуры числами двумерный ступенчатый массив:

```
int[][] mas = new int [5][];
for (int i=0; i<5; i++)
{
        mas[i] = new int [i+1];
        for (int j=0; j<=i; j++)
                mas[i][j] = i*j;
}
for (int i=0; i<5; i++)
{
        for (int j=0; j<mas[i].Length; j++)
                Console.Write("{0}\t", mas[i][j]);
        Console.WriteLine();
}
```

§8. Функции

Функции применяются для упрощения разработки программ. Функции используются, если:

- одинаковые преобразования выполняются в различных участках кода программы;
- слишком длинный код программы для упрощения разбивается на несколько фрагментов.

Любая программа на языке C# начинается с выполнения функции Main, из нее можно вызывать другие функции программы.

Описание функции состоит из двух частей: заголовка функции и ее тела:

```
[тип_возвращаемого_значения] имя_функции ([список_параметров])
{
        [тело_функции]
}
```

Список параметров содержит описание входных параметров функции. В этом списке через запятую для каждого параметра указывается его тип и имя:

тип имя, тип имя, тип имя

Например:

int a, float b, string c

Тело функции – последовательность операторов, описывающих алгоритм функции. Возврат из функции происходит после выполнения последнего оператора тела функции либо при выполнении оператора return. Оператор return можно использовать для передачи возвращаемого значения, например:

```
float discriminant(float a, float b, float c)
{
        float D = b * b – 4 * a * c;
        return D;
}
```

Вызов функции выглядит так:

имя_функции ([список_аргументов])

Если функция возвращает некоторое значение, то оно передается в место вызова функции и является результатом ее работы. Число и типы аргументов должны совпадать с числом и типом параметров функции. При вызове функции параметры подставляются вместо аргументов.

Различают следующие виды параметров функции:

1) параметры – значения;
2) ссылочные параметры;
3) выходные параметры;
4) множественные параметры.

Передача параметра по значению

При передаче параметров по значению в параметры функции копируются значения аргументов. Если в функции значения параметров изменятся, то это не приведет к изменению аргументов.

Рассмотрим пример функции, вычисляющей сумму трех аргументов:

```
static int sum(int a, int b, int c)
{
        int s = a + b + c;
        return s;
}
        static void Main()
{
        int x = 15, y = 25, z = 55;
        int a = sum(x ,y, z);
}
```

Передача параметра по ссылке

Если параметры передаются по ссылке, то изменение параметров внутри функции приводит к изменению аргументов, поданных при вызове функции. При объявлении функции и ее вызове перед ссылочными параметрами необходимо использовать ключевое слово ref.

Рассмотрим пример функции, меняющей значения переменных:

```
static void swap(ref int a, ref int b)
{
        int c = a;
        a = b;
        b = c;
}
static void Main()
{
        int x = 1, y = 5;
        swap(ref x ,ref y);
        // x = 5, y = 1
}
```

Передача параметра по ссылке выходного параметра

Изменение внутри функции значений выходных параметров также приводит к изменению поданных при вызове функции аргументов. Отличие выходных параметров от ссылочных состоит в том, что внутри функции выходным параметрам обязательно следует присвоить некоторое значение, а при вызове функции соответствующие аргументы могут быть не инициализированы. При объявлении функции и ее вызове перед выходными параметрами необходимо использовать ключевое слово out.

Рассмотри пример функции, вычисляющей одновременно и сумму и произведение двух целых чисел:

```
static void f(int a, int b, out int sum, out int pr)
{
        sum = a + b;
        pr = a * b;
}
static void Main()
{
        int x = 15, y = 25, s, p;
        f(x ,y, out s, out p);
        // s = 40, p = 375
}
```

Передача множества аргументов в виде одного параметра

Множественная передача позволяет передавать заранее неизвестное число аргументов. Для объявления множественного параметра используется ключевое слово params.

Рассмотри пример функции, вычисляющей сумму произвольного количества целых чисел:

```
static int sum(params int[] mas)
{
        int s = 0;
        for (int i = 0; i < mas.Length; i++)
                s += mas[i];
        return s;
}
```

```
static void Main()
{
        int a = sum(5, 7, 3, 10);
        // a = 25
}
```

§9. Структуры

Структуры представляет собой производный (составной) тип, определяемый программистом, создаваемый на основе существующих типов. Структура включает в себя данные, называемые полями.

Структуры в C# — это тип переменных, который является средним между типами-значениями и ссылочными типами. Они используются тогда, когда необходимо создать простой объект небольших объемов. Структуры объявляются с помощью ключевого слова *struct*.

Описание структуры имеет следующий формат:

```
struct [имя_структуры]
{
        [тип_1] [имя_поля_1];
        …
        [тип_N] [имя_поля_N];
}
```

Структура представляет собой группу связанных между собой переменных. К примеру: комплексные числа, точки в системе координат или пары «слово-значение» в словаре. Их особенность заключается в следующем: содержат небольшое число элементов-данных; нет проверки на равенство ссылок; могут быть легко реализованы.

Синтаксис объявления структуры выглядит следующим образом (на примере комплексных чисел):

```
public struct StructComplex
{
        public double x;        // действительная часть комплексного числа
        public double y;        // мнимая часть комплексного числа
}
```

После описания структуры можно объявить один или несколько объектов (переменных) как экземпляров этой структуры. Обращение к структуре в C# осуществляется с помощью оператора *«.»*.Создание объекта структуры может быть реализовано с помощью оператора *«new».* Однако это не является строгим правилом, поскольку если мы не будем использовать данный оператор (*new*), то объект по-прежнему будет создаваться:

```
StructComplex sc = new StructComplex;
sc.x = 2,5;
sc.y = 1,5;
```

Объекты struct хранятся в памяти так же, как и переменные встроенных типов. Таким образом, переменная sc не является ссылкой на некоторый блок внешней

памяти, выделенной в отдельной области. Объект sc занимает память в той же области, что и обычная переменная. Структуры могут иметь статические и не статические поля и методы.

Кроме того, можно объявлять массивы структур. В качестве примера рассмотрим код программы, который не только обращается к массиву, но и заполняет его. Данный код заполняет массив, состоящий из 10 элементов (нумерация элементов массива – от 0 до 9); структура заполняется случайными числами от 0 до 99:

```
StructComplex[] array_sc = new StructComplex[10];
Random r = new Random();
for (int i = 0; i < 9; i++)
{
        array_sc[i].x = r.Next(100);
        array_sc[i].y = r.Next(100);
}
```

Так как структура имеет тип значение, то при передаче структуры в функцию, она получает и работает с копией структуры. Функция не может изменить исходный экземпляр структуры, только его копию.

Рассмотрим другой фрагмент кода. В данной программе у нас есть структура Structure и её поле Result. После вызова метода StructTurning поле Result не меняется, так как метод работает с копией, а не с оригиналом структуры.

```
struct Structure
{
        public string Result;
}
class TestStruct
{
        static void StructTurning(Structure s)
        {
                s.Result = "Синий";
        }
        static void Main()
        {
                Structure StructTest = new Structure();
                StructTest.Result = "Оранжевый";
                StructTurning(StructTest);
                Console.WriteLine("Цвет = {0}", StructTest.Result);
                Console.WriteLine("Press any key to exit.");
                Console.ReadKey();
                // Вывод: Цвет = Оранжевый
        }
}
```

В результате мы получим «Цвет = Оранжевый».

В следующем примере метод NewPryamougolnik получает значения сторон прямоугольника и возвращает эти значения в структуру. Перед выходом из метода нужно убедиться, что тип значения из функции совпадает с типом структуры.

```
using System;
public struct TheStructure
```

```
        {
                private double val;
                public double SetValue
                {
                        get { return val; }
                        set { val = value; }
                }
                public double GetValue()
                {
                        return double.Parse(Console.ReadLine());
                }
        }
        public struct Rectangle
        {
                TheStructure height;
                TheStructure width;
                public TheStructure SetHeight
                {
                        get { return height; }
                        set { height = value; }
                }
                public TheStructure SetWidth
                {
                        get { return width; }
                        set { width = value; }
                }
                public void NewRectangle()
                {
                        TheStructure newStr = new TheStructure();
                        Console.WriteLine("Введите стороны прямоугольника");
                        width = GetSide("Сторона A: ", newStr);
                        height = GetSide("Сторона B: ", newStr);
                }
                public TheStructure GetSide(string s, TheStructure newStr)
                {
                        Console.Write(s);
                        newStr.SetValue = newStr.GetValue();
                        return newStr;
                }
        }
        public class Program
        {
                static int Main()
                {
                        var newRect = new Rectangle();
                        newRect.NewRectangle();
                        Console.WriteLine();
                        Console.WriteLine("Площадь прямоугольника:      {0}",
newRect.SetWidth.SetValue * newRect.SetHeight.SetValue);
```

```
            Console.ReadKey();
            return 0;
        }
    }
```

Пример работы программы:
Введите стороны прямоугольника
Сторона A: 15.8
Сторона B: 36.3
Площадь прямоугольника: 573.54
Press any key to continue . . .

§10. Работа с файлами

Файл – это набор данных, который хранится на внешнем запоминающем устройстве, имеющий имя и расширение. Расширение позволяет идентифицировать какие и в каком формате данные хранятся в файле.

Под работай с файлами подразумевается создание и удаление файлов, чтение и запись данных, изменение параметров файла и прочее.

Для работы с файлами в пространстве имен System.IO реализованы все необходимые классы. Чтобы подключить данное пространство имен, необходимо в начале программы прописать строку using System.IO. Для использования кодировок - using System.Text:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

Создание файла

Для создания пустого файла в классе File есть метод *Create()*, который принимает один аргумент – путь.

Создание пустого текстового файла new_file.txt на диске D:

```
File.Create("D:\\new_file.txt");
```

Если файл с таким именем уже существует, то он будет переписан на новый пустой файл.

Метод *WriteAllText()* создает новый файл, если такого не существует, либо открывает существующий и записывает текст, заменяя всё, что было в файле:

```
File.WriteAllText("D:\\new_file.txt", "текст");
```

Метод *AppendAllText()* работает аналогичным образом, как и метод WriteAllText(), но не заменяет содержания файла, а записывает новый текст в конце файла, после уже существующих данных:

```
File.AppendAllText("D:\\new_file.txt", "тест метода AppendAllText ()");
```

Удаление файла

Метод *Delete()* удаляет файл по указанному пути:

```
File.Delete("D:\\file.txt");
```

Использование потоков

Поток – это абстрактное представление данных (в байтах), которое облегчает работу с ними. В качестве источника данных может быть файл, устройство ввода-вывода, принтер.

Класс *Stream* является абстрактным базовым классом для всех потоковых классов в C#.

Для работы с файлами необходим класс *FileStream*, представляющий поток, который позволяет выполнять операции чтения и записи в файл.

Открытие файла только для чтения:

```
FileStream file = new FileStream("d:\\file.txt", FileMode.Open, FileAccess.Read);
```

Режимы открытия FileMode:

- *Append* – открывает файл, если таковой существует, и переводит указатель в конец файла (данные будут дописываться в конец), или создает новый файл. Данный режим возможен только при режиме доступа FileAccess.Write;
- *Create* – создает новый файл; если таковой существует – заменяет данные;
- *CreateNew* – создает новый файл; если таковой существует – генерируется исключение;
- *Open* – открывает файл; если таковой не существует – генерируется исключение;
- *OpenOrCreate* – открывает файл либо создает новый, если данного файла не существует;
- *Truncate* – открывает файл, но все данные внутри файла затирает; если файла не существует – генерируется исключение.

Примеры использования режимов открытия FileMode:

```
//создание нового файла
    FileStream file1 = new FileStream("d:\\file1.txt", FileMode.CreateNew);
//открытие существующего файла
    FileStream file2 = new FileStream("d:\\file2.txt", FileMode.Open);
//открытие файла на дозапись в конец файла
    FileStream file3 = new FileStream("d:\\file3.txt", FileMode.Append);
```

Режими доступа FileAccess:

- *Read* – открытие файла только на чтение. При попытке записи генерируется исключение;
- *Write* - открытие файла только на запись. При попытке чтения генерируется исключение;
- *ReadWrite* - открытие файла на чтение и запись.

Чтение из файла

Для чтения данных из потока необходим класс *StreamReader*. В нем реализовано множество методов для удобного считывания данных.

Рассмотрим программу для вывода содержимого файла на экран:

```
//создание файлового потока
    FileStream file1 = new FileStream("d:\\file.txt", FileMode.Open);
//создание «потокового читателя», связывание его с файловым потоком
    StreamReader reader = new StreamReader(file1);
//считывание всех данных с потока, вывод на экран
```

```
        Console.WriteLine(reader.ReadToEnd());
    //закрытие потока
        reader.Close();
        Console.ReadLine();
```

Метод *ReadToEnd()* считывает все данные из файла.

ReadLine() – считывает одну строку. При этом указатель потока переходит на новую строку, и при последующем вызове метода будет считана следующая строка.

Свойство *EndOfStream* указывает, находится ли текущая позиция в потоке в конце потока, т.е. достигнут ли конец файла. Возвращает значения true или false.

Запись в файл

Для записи данных в поток используется класс *StreamWriter*.

```
    //создание файлового потока
        FileStream file1 = new FileStream("d:\\file.txt", FileMode.Create);
    //создание «потокового писателя» и связывание его с файловым потоком
        StreamWriter writer = new StreamWriter(file1);
    //запись в файл
        writer.Write("текст");
    //закрытие потока; если не закрыть поток, то в файл ничего не запишется
        writer.Close();
```

Метод *WriteLine()* записывает в файл построчно - то же самое, что и простая запись с помощью Write(), но в конце добавляется новая строка.

Необходимо всегда помнить, что после работы с потоком его необходимо закрыть, использовав метод *Close()*.

Кодировка, с которой будут считываться и записываться данные указывается при создании *StreamReader* и *StreamWriter*:

```
            FileStream file1 = new FileStream("d:\\file.txt", FileMode.Open);
            StreamReader reader = new StreamReader(file1, Encoding.Unicode);
            StreamWriter writer = new StreamWriter(file1, Encoding.UTF8);
```

Кроме того, при использовании StreamReader и StreamWriter можно не создавать отдельно файловый поток FileStream, а сделать это сразу при создании StreamReader или StreamWriter:

```
            StreamWriter writer = new StreamWriter("d:\\file.txt"); //указываем путь к файлу,
        а не поток
            writer.WriteLine("текст");
            writer.Close();
```

Создание папки

Папка создается с помощью статического метода *CreateDirectory()* класса *Directory*:

```
            Directory.CreateDirectory("d:\\new_folder");
```

Удаление папки

Для удаления пустой папки необходимо использовать метод *Delete()*:

```
            Directory.Delete("d:\\new_folder");
```

Если папка не пустая, необходимо указать параметр рекурсивного удаления - true:

```
            Directory.Delete("d:\\new_folder", true);
```

§11. Простейшие алгоритмы с массивами

Большинство алгоритмов, обрабатывающих массивы, основаны на применении оператора цикла. Например, для числовых массивов: алгоритмы подсчета суммы элементов, поиска максимального или минимального элемента, количества положительных элементов и т. п. Как правило, в таких алгоритмах счетчик итераций цикла используется в качестве индекса элементов массива. Значение счетчика цикла в этом случае совпадает с индексом элемента массива. Для определенности будем для счетчика использовать переменную i (сокращение от index). Так как нумерация элементов массива начинается с нуля, то на первой итерации цикла переменная i должна быть равна нулю. После каждой итерации цикла переменная i увеличивается на единицу до тех пор, пока не станет равна числу элементов массива N. После этого цикл заканчивается. Продемонстрируем это на примере вывода на экран всех элементов массива:

```
for (int i = 0; i < N; i++)
   Console.WriteLine("{0}", A[i]);
```

В начале работы цикла переменной i присваивается значение нуль и выводится нулевой элемент массива. Затем i увеличивается на единицу и выводится первый элемент массива. Так продолжается до тех пор, пока i меньше, чем N.

Рассмотрим примеры алгоритмов, обрабатывающих целочисленные массивы. Для каждой из приведенных ниже задач будет приведена идея алгоритма решения, а также фрагмент программы на языке C#. Во всех примерах рассматривается целочисленный массив A, состоящий из N элементов.

Алгоритм вычисления суммы элементов целочисленного массива

Вначале переменной *sum* присваиваем нуль. Затем в цикле перебираем элементы массива с индексами от 0 до *N*-1, и каждый из элементов прибавляем к переменной *sum*.

Приведем фрагмент кода программы:

```
int sum = 0;
for (int i = 0; i < N; i++)
        sum += A[i];
Console.WriteLine("Sum = {0}", sum);
```

Алгоритм поиска максимального элемента в целочисленном массиве

Вначале переменной *max* присваиваем значение нулевого элемента массива. Затем в цикле перебираем элементы массива с индексами от 1 до *N*-1. Если очередной элемент окажется больше, чем *max*, то переменой *max* присвоим значение этого элемента массива.

Приведем фрагмент кода программы:

```
int max = A[0];
for (int i = 1; i < N; i++)
        if (max < A[i])
                max = A[i];
Console.WriteLine("Max = {0}", max);
```

Алгоритм подсчета суммы квадратов элемента в целочисленного массива

Рассмотрим алгоритмы, решающие более сложные задачи. Пусть требуется найти сумму квадратов элементов числового массива. Для этого можно модифицировать алгоритм вычисления суммы элементов: на каждой итерации цикла вместо прибавления к переменной *sum* очередного элемента массива будем прибавлять его квадрат.

Приведем фрагмент кода программы:

```
int sum = 0;
for (int i = 0; i < N; i++)
        sum += A[i] * A[i];
Console.WriteLine("Sum = {0}", sum);
```

Алгоритм подсчета количества положительных элементов в целочисленном массиве

Для подсчета количества положительных чисел массива используем счетчик *count*, которому изначально присвоим нулевое значение. В цикле перебираем все элементы массива, каждый из них сравнивается с нулем. Если очередной элемент оказался больше нуля, то увеличиваем на единицу значение счетчика *count*.

```
int count = 0;
for (int i = 0; i < N; i++)
        if (A[i] > 0)
                count++;
Console.WriteLine(Количество положительных чисел {0}", count);
```

Алгоритм подсчета суммы положительных чисел в целочисленном массиве

Для подсчета суммы положительных элементов одномерного массива используется переменная *sum*, изначально равная нулю. В цикле перебираются все элементы массива, и, если значение очередного элемента больше нуля, оно прибавляется к переменной *sum*.

```
int sum = 0;
for (int i = 0; i < N; i++)
        if (A[i] > 0)
                sum += A[i];
Console.WriteLine(Суммма положительных чисел {0}", sum);
```

Алгоритм вычисления среднего арифметического значения положительных элементов целочисленного массива

Изначально переменным *sum* и *count* присваивается нуль. Затем в цикле перебираются элементы массива с индексами от 0 до *N*-1. Если очередной элемент больше нуля, то прибавляем его значение к переменной *sum* и увеличиваем счетчик положительных элементов *count* на единицу. После окончания работы цикла сравниваем *count* с нулем. Если значение *count* больше нуля, то для вычисления среднего арифметического значения делим *sum* на *count*, результат присваиваем переменной *mean*. Для того, чтобы деление было вещественным, а не целочисленным, используем явное преобразование к типу float[1].

[1] При делении целых чисел результат всегда целый. Например, 7 делим на 2 и получаем 3, а не 3.5. При делении вещественных чисел – результат вещественный. Например, 7.0 делим на 2.0 и получаем 3.5. Поэтому, если при делении целых чисел мы хотим получить арифметически правильный результат, их необходимо привести к вещественному типу.

Приведем фрагмент кода программы:

```
int sum = 0;
int count = 0;
for (int i = 0; i < N; i++)
        if (A[i] > 0)
        {
                sum += A[i];
                count++;
        }
        if (count > 0)
        {
                float mean = (float) sum / count;
                Console.WriteLine("Mean = {0}", mean);
        }
        else
                Console.WriteLine("Нет положительных элементов");
```

Алгоритм подсчета произведения отрицательных элементов массива

Подсчет произведения отрицательных элементов массива производится в цикле, в котором для произведения используется переменная *pr*, а также флаг *flag* как признак наличия отрицательных чисел. Каждый элемент массива сравнивается с нулем и, если меньше нуля, то *pr* умножается на текущий элемент:

```
int pr = 1;
bool flag = false;
for (int i = 0; i < N; i++)
        if (A[i] < 0)
        {
                pr *= A[i];
                flag = true;
        }

if (flag)
        Console.WriteLine("Произведение отр. чисел равно {0}", pr);
else
        Console.WriteLine("Нет отрицательных чисел");
Console.ReadKey();
```

Алгоритм подсчета количества элементов массива, лежащих в заданном диапазоне

Для вычисления количества элементов массива, лежащих в диапазоне от *B* до *D*, используется счетчик *k*. В цикле перебираются все элементы массива. Для каждого элемента проверяется условие попадания в диапазон. Если условие истинно, то счетчик *k* увеличивается на единицу.

```
int k = 0;
for (int i = 1; i < N; i++)
        if (A[i] >= B & A[i] <= D)
                k++;
Console.WriteLine("k = {0}", k);
```

Алгоритм поиска минимального положительного элемента целочисленного массива

Алгоритм состоит из двух частей: поиск первого положительного элемента массива и поиск минимального элемента среди всех положительных.

Изначально переменной *min* присваивается нуль. Затем в цикле перебираем элементы массива с индексами от 0 до *N*-1. Если очередной элемент больше нуля, то присвоим его значение переменной *min* и выйдем из цикла.

Далее сравним значение *min* с нулем. Если *min* равно нулю, то положительные элементы обнаружены не были и наша задача не имеет решения. Иначе, в цикле перебираем элементы массива с индексами от 0 до *N*-1. Если очередной элемент окажется больше нуля и меньше, чем *min*, то переменой *min* присвоим значение этого элемента массива.

Приведем фрагмент кода программы:

```
int min = 0;
for (int i = 0; i < N; i++)
        if (A[i] > 0)
        {
                min = A[i];
                break;
        }

if (min == 0)
        Console.WriteLine("Нет положительных элементов");
else
{
        for (int i = 0; i < N; i++)
                if (A[i] > 0 && min > A[i])
                        min = A[i];
        Console.WriteLine("Min = {0}", min);
}
```

Алгоритм вычисления суммы элементов массива, расположенных до последнего положительного элемента

Вначале найдем номер последнего положительно элемента массива. Для этого в цикле перебираем с конца в сторону начала элементы массива, пока либо не переберем все элементы массива, либо не найдем положительный элемент. Если положительный элемент найден, то запомним его номер в переменной *k* и завершим цикл. Затем находим сумму элементов массива с индексами от *0* до (*k*-1). Делаем так. Вначале переменной *sum* присвоим значение нуль. Затем в цикле по *i* от *0* до (*k*-1) элементы массива прибавляем к значению переменной *sum*:

```
int k = -1;
for (int i = N - 1; i >= 0; i--)
        if (A[i] > 0)
        {
                k = i;
                break;
        }
if (k == -1)
        Console.WriteLine("Нет положительных элементов");
```

```
else
{
    int sum = 0;
    for (int i = 0; i < k; i++)
        sum += A[i];
    Console.WriteLine("Сумма = {0}", sum);
}
```

Другой вариант программы:

```
int k = N - 1;
while (k > 0)
    if (A[k] > 0)
        break;
    else
        k--;
if (k < 0)
    Console.WriteLine("Нет положительных элементов");
else
{
    int sum = 0;
    for (int i = 0; i < k; i++)
        sum += A[i];
    Console.WriteLine("Сумма = {0}", sum);
}
```

Алгоритм вычисления суммы элементов массива, расположенных после максимального элемента

Для вычисления суммы элементов массива, расположенных после максимального элемента, вначале необходимо найти номер максимального элемента (для определенности первого, если их несколько), а затем искать саму сумму.

Определение номера максимального элемента массива делаем так. Переменной *j* присваиваем нуль. Затем в цикле перебираем все с первого элементы массива. На каждой итерации цикла текущий элемент массива сравниваем со значением *A[j]*. Если текущий элемент оказался больше, чем *A[j]*, то *j* присваиваем номер элемента. В итоге *j* будет хранить номер максимального элемента массива.

Сумму элементов массива, расположенных после максимального элемента, вычисляем так. Вначале переменной *sum* присвоим значение нуль. Затем в цикле перебираем элементы массива с индексами от *(j+1)* до *(N-1)* и каждый из них прибавляем к переменной *sum*. Если *j = N-1*, то *sum* останется равной нулю.

```
int j = 0;
for (int i = 1; i < N; i++)
    if (A[j] < A[i])
        j = i;
int sum = 0;
for (int i = j + 1; i < N; i++)
    sum += A[i];
Console.WriteLine("Сумма равна {0}", sum);
```

Второй вариант решения. Можно одновременно находить максимальный элемент и сумму элементов, расположенных после максимального элемента:

```
int sum = 0;
int max = A[0];
for (int i = 1; i < N; i++)
        if (max < A[i])
        {
                max = A[i];
                sum = 0;
        }
        else
                sum += A[i];
Console.WriteLine("Сумма равна {0}", sum);
```

Алгоритм вычисления суммы положительных элементов массива, расположенных до максимального элемента

Вначале необходимо найти индекс *k* максимального элемента (см. выше). Затем переменной *sum* присвоим нуль. Далее, в цикле перебираем все элементы массива с индексами от *0* до (*k*-1); сравниваем их с нулем и, если элемент больше нуля, прибавляем его к переменной *sum*:

```
int k = 0;
for (int i = 1; i < N; i++)
        if (A[k] < A[i])
                k = i;

int sum = 0;
for (int i = 0; i < k; i++)
        if (A[i] > 0)
                sum += A[i];
Console.WriteLine("Сумма = {0}", sum);
```

Алгоритм подсчета произведения элементов массива, расположенных между максимальным и минимальным элементами массива

Для определения произведения элементов массива, расположенных между максимальным и минимальным элементами, вначале находится номер *i_max* первого максимального элемента массива, затем номер *i_min* последнего минимального элемента массива.

Затем в цикле перебираются все элементы с номера *(i_max + 1)* до номера *(i_min – 1)* и каждый из них умножается на переменную *pr* (изначально равную единице). В случае, если *((i_min – i_max) <= 1)* выдается сообщение, что между максимумом и минимумом нет элементов.

```
int i_max = 0;
for (int i = 1; i < N; i++)
        if (A[i_max] < A[i])
                i_max = i;
int i_min = 0;
for (int i = 1; i < N; i++)
        if (A[i_min] >= A[i])
                i_min = i;
```

```
if ((i_min – i_max) <= 1)
        Console.WriteLine("Нет элементов между макс. и мин.");
else
{
        int pr = 1;
        for (int i = i_max + 1; i < i_min; i++)
                pr *= A[i];
        Console.WriteLine("Произведение равно {0}", pr);
}
```

Алгоритм вычисления суммы элементов массива, расположенных между первыми двумя отрицательными элементами массива

Сумма элементов массива, расположенных между первым и вторым отрицательными элементами находится следующим образом. Элементы массива просматриваются последовательно в цикле. При обнаружении отрицательного элемента счетчик *count* увеличивается на единицу. Если текущий элемент массива не является отрицательным и *count* равен единице, то значение элемента прибавляется к значению переменной *sum*. Как только счетчик *count* станет равен 2, происходит выход из цикла.

```
int count = 0;
int sum = 0;
bool flag = false;
for (int i = 0; i < N; i++)
{
        if (A[i] < 0)
        {
                count++;
                if (count == 2)
                break;
        }
        else
                if (count == 1)
                {
                        flag = true;
                        sum += A[i];
                }
}
 if (count < 2)
        Console.WriteLine("Отрицательных чисел меньше двух");
 else   if (flag == false)
        Console.WriteLine("Нет элементов между отрицательными значениями");
 else
        Console.WriteLine("Сумма равна {0}", sum);
```

Алгоритм подсчета суммы элементов массива, расположенных после последнего элемента равного нулю

Перебираем элементы массива с конца и прибавляем их к переменной *sum*, до тех пор, пока не встретим нулевой элемент. Если в ходе перебора будут пройдены все элементы массива, то нулей в массиве нет:

```
int i = N - 1;
int sum = 0;
while (i >= 0 && A[i] != 0)
{
        sum += A[i];
        i--;
}
if (i < 0)
        Console.WriteLine("Нулей нет");
else
        Console.WriteLine("Сумма равна {0}", sum);
```

Второй вариант решения. Для подсчета суммы элементов массива, расположенных после последнего элемента равного нулю, используем флажок *flag* (равный изначально «ложь») и переменную *sum* (равную изначально нулю). В цикле перебираем все элементы массива. Если значение очередного элемента массива оказалось равно нулю, то флажку присвоим значение «истина» (нули в массиве есть), а переменную *sum* обнулим (сумму нужно подсчитывать заново). Если значение элемента не равно нулю, то его значение прибавим к переменной *sum*. Таким образом, по окончании работы цикла, если флажок оказался равен «лжи», то нулей в массиве нет, а если равен «истине», то переменная *sum* равна сумме элементов массива, расположенных после последнего нуля.

```
bool flag = false;
int sum = 0;
for (int i = 0; i < N; i++)
        if (A[i] == 0)
        {
                flag = true;
                sum = 0;
        }
        else
                sum += A[i];
if (flag == false)
        Console.WriteLine("Нулей нет");
else
        Console.WriteLine("Сумма равна {0}", sum);
```

Алгоритм изменения порядка следования элементов на обратный

Изменение порядка следования элементов в массиве на обратный производится следующим циклом:

```
for (int i = 0; i < N / 2; i++)
{
        int t = A[i];
        A[i] = A[N - 1 - i];
```

```
        A[N - 1 - i] = t;
    }
```

Алгоритм сортировки «пузырьком» элементов массива по возрастанию

Упорядочение элементов по возрастанию происходит методом сортировки «пузырьком». В этой сортировке *(N-1)* раз (*N* – число элементов массива) сравниваются соседние элементы массивов и обмениваются местами в случае, если значение первого элемента пары превосходит значение второго элемента пары.

```
for (int i = 0; i < N - 1; i++)
    for (int j = 0; j < N - 1; j++)
        if (A[j] > A[j+1])
        {
            int t = A[j];
            A[j] = A[j + 1];
            A[j + 1] = t;
        }
```

Алгоритм упорядочения элементов массива по возрастанию абсолютных значений

Упорядочить элементы массива по возрастанию абсолютных значений (модулей) элементов можно при помощи сортировки, например, «пузырьком». Только на каждом шаге сортировки обмен значений элементов осуществлять в случае, если |*A[j]*|>|*A[j+1]*|.

```
for (int i = 0; i < N - 1; i++)
    for (int j = 0; j < N - 1; j++)
        if (Math.Abs(A[j]) > Math.Abs(A[j+1]))
        {
            int t = A[j];
            A[j] = A[j + 1];
            A[j + 1] = t;
        }
```

Алгоритм вычисления суммы элементов каждой строки двумерного целочисленного массива

Пусть задан двумерный массив A, состоящий из M строк и N столбцов. Алгоритм решения подсчета суммы элементов каждой строки состоит из двух циклов. Внешний цикл перебирает все строки массива. Внутренний цикл, который запускается на каждой итерации внешнего, вычисляет сумму элементов очередной строки.

Приведем фрагмент кода программы:

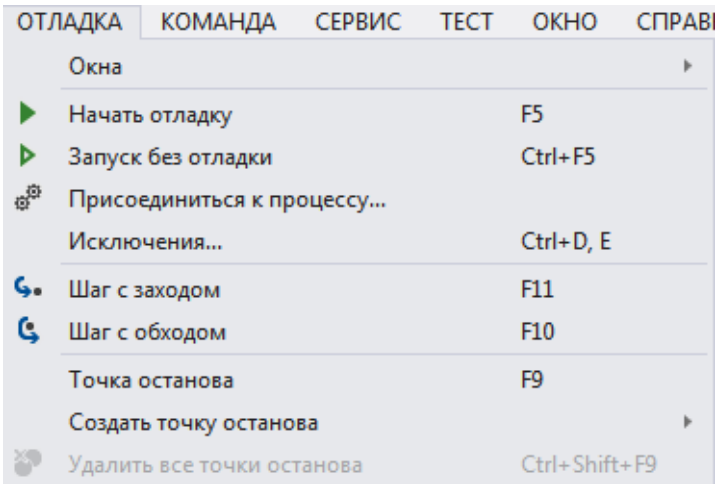
```
for (int i = 0; i < M; i++)
{
    int sum = 0;
    for (int j = 0; j < N; j++)
        sum += A[i, j];
    Console.WriteLine("Sum = {0}", sum);
}
```

Задания для самостоятельного решения:

1. Напишите программу вычисления среднего арифметического значения элементов целочисленного массива.
2. Напишите программу вычисления произведения элементов целочисленного массива.
3. Напишите программу поиска минимального элемента в целочисленном массиве.
4. Напишите программу поиска номера максимального элемента в целочисленном массиве.
5. Напишите программу поиска номера минимального элемента в целочисленном массиве.
6. Напишите программу подсчета количества отрицательных элементов в целочисленном массиве.
7. Напишите программу подсчета в целочисленном массиве количества элементов, кратных трем.
8. Напишите программу подсчета в целочисленном массиве количества элементов оканчивающихся на пять.
9. Напишите программу подсчета суммы отрицательных чисел в целочисленном массиве.
10. Напишите программу вычисления среднего арифметического значения всех элементов целочисленного массива.
11. Напишите программу вычисления среднего арифметического значения отрицательных элементов целочисленного массива.
12. Напишите программу поиска максимального отрицательного элемента целочисленного массива.
13. Напишите программу поиска минимального элемента целочисленного массива, значение которого попадает в интервал [15, 20].
14. Напишите программу, которая меняет местами максимальный и минимальные элементы (первые, если их несколько).
15. Напишите программу, которая заменяет нулями все элементы, расположенные между первым и вторым нечетными элементами массива.
16. Напишите программу, которая заменяет все нулевые элементы значением, равному максимальному элементу массива.
17. Напишите программу, которая осуществляет циклический сдвиг влево элементов целочисленного массива.
18. Напишите программу сортировка «пузырьком» элементов массива по убыванию.
19. Напишите программу сортировка элементов массива по возрастанию методом максимального элемента. В этой сортировке вначале ищется максимальный элемент, который меняется местами с последним элементом массива. Затем ищется максимальный элемент среди всех элементов, за исключением последнего. Найденный элемент меняется местами с предпоследним элементом. И так далее.
20. Напишите программу сортировка элементов массива по убыванию методом минимального элемента.
21. Напишите программу упорядочения элементов целочисленного массива по их остаткам от деления на 10.

§12. Отладка программ

Отладка программ является мощнейшим оружием разработчика не только для борьбы с ошибками, но и для глубокого понимания сути программ.



Для отладки создано отдельное меню с одноименным названием, где находятся все доступные инструменты для отладки. Но для успешной отладки просто необходимо научиться пользоваться сочетаниями клавиш. Поэтому в дальнейшем вместо указания команды всегда будет указываться только соответствующее сочетание клавиш.

Чтобы запустить отладчик следует нажать клавишу F5. Чтобы запустить программу без отладки необходимо нажать сочетание клавиш Ctrl+F5. Однако для выполнения лабораторных работ рекомендуется всегда запускать программу клавишей F5, даже если не планируется использовать отладку. Разница будет заметна, если в программе присутствуют ошибки. В случае запуска с помощью Ctrl+F5 на экране лишь появится сообщение об ошибке в программе; в случае же нажатия F5 Visual Studio явно укажет на место в коде программы, где произошел сбой. Чтобы прервать отладку существует сочетание клавиш Shift+F5.

После выполнения F5 отладчик остановит выполнение программы на первой точке останова. Чтобы установить точку останова, на текущей строке нужно нажать F9. Чтобы убрать существующую – нажать F9 еще раз.

Точка останова – это сигнал, который указывает отладчику временно остановить выполнение программы в определенной точке. Приостановка выполнения программы в точке останова называется режимом приостановки. Вход в режим приостановки выполнения не приводит к прекращению или завершению работы программы, поэтому выполнение программы может быть продолжено в любое время.

Продолжить выполнение программы можно следующими способами:

- F5 – продолжение отладки до следующей точки останова, если такая имеется. В противном случае программа выполнится до конца.
- F10 – выполнение одного следующего оператора или функции.
- F11 – выполнение следующей функции с заходом в тело функции.
- Shift+F11 – осуществление последовательной шаг за шагом отладки с завершением выполнения текущей функции и выходом с остановкой на строку, следующую за строкой, вызывавшей функцию.

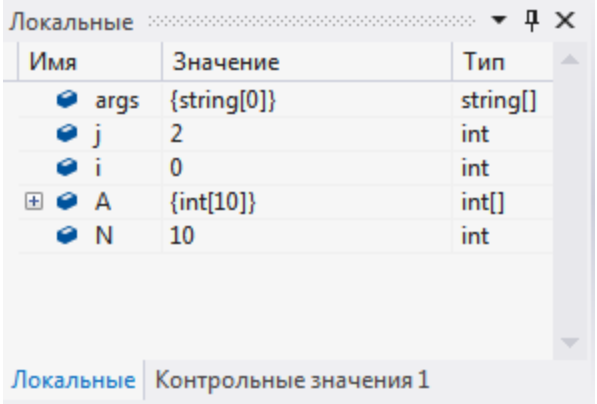
Режим приостановки выполнения – это пребывание программы в ожидании. В этом режиме все элементы, например функции, переменные и объекты, сохраняются в памяти, но их перемещения и активность приостанавливаются. Во время режима приостановки выполнения можно выполнить поиск ошибок и нарушений целостности данных, проверив положения элементов и их состояние. В режиме приостановки в программу можно вносить коррективы. Например, можно изменить значение переменной. Можно перемещать точку выполнения, изменяя оператор, который будет выполняться следующим при возобновлении выполнения программы.

Рассмотрим пример отладки на следующей программе на языке C#:

```
class BubbleSorting
{
   static void ExchangeElements(ref byte[] newArray, int firstElem, int secondElem)
   {
      byte middlingElem = newArray[firstElem];
      newArray[firstElem] = newArray[secondElem];
      newArray[secondElem] = middlingElem;
   }
   static void Main(string[] args)
   {
      Console.WriteLine("Введите размер массива:");
      int N = int.Parse(Console.ReadLine());
      byte[] newArray = new byte[N];
      Random randomArray = new Random();
      randomArray.NextBytes(newArray);
      Console.WriteLine("Вывод изначального массива:");
      foreach (byte i in newArray)
         Console.Write("{0} ", i);
      for (int i = 0; i < N - 1; i++)
         for (int j = 0; j < N - 1; j++)
            if (newArray[j] > newArray[j + 1])
               ExchangeElements(ref newArray, j, j + 1);
      Console.WriteLine("\nВывод отсортированного массива:");
      foreach (byte i in newArray)
         Console.Write("{0} ", i);
      // Не позволяет окну консоли закрыться.
      Console.WriteLine("\nНажмите любую клавишу для выхода.");
      Console.ReadKey();
   }
}
```

Установим точку останова на строке:
ExchangeElements(ref newArray, j, j + 1);
И запустим отладку F5. После ввода размера массива отладка перейдет в режим приостановки выполнения с курсором на строке с точкой останова. При нажатии F5 выполнятся все действия цикла до этой же точки останова. При нажатии F10 выполнится функция *ExchangeElements,* и курсор перейдет на следующий оператор (инкремент j). Нажав F5, снова перейдем на точку останова. При нажатии F11 курсор зайдет внутрь функции *ExchangeElements*. При нажатии Shift+F11 функция *ExchangeElements* выполнится полностью, и курсор вернется в функцию *Main*, где сразу остановится.

Для просмотра текущих значений переменных и элементов массивов существует окно «Локальные», где можно увидеть значения всех локальных переменных в данном месте программы. А если раскрыть плюс у массива, то и все



значения его элементов. Значения переменных в этом окне меняются в реальном времени в процессе отладки, что очень помогает осуществлять исправление ошибок. Кроме того можно увидеть значения переменных, наведя курсор мыши на требуемую переменную в коде программы.

Без использования этих простых возможностей отладчика было бы очень сложно, а порой просто невозможно, исправлять ошибки в программах.