1

**Алгоритм** - это система точных и понятных предписаний о содержании и последовательности выполнения конечного числа действий, необходимых для решения любой задачи данного типа.

Примеры: правила сложения, умножения, решения алгебраических уравнений и т.п.

**Свойства алгоритма:**

1.Универсальность (массовость) - применимость алгоритма к различным наборам исходных данных.

2.Дискретность - процесс решения задачи по алгоритму разбит на отдельные действия.

3.Конечность - каждое из действий и весь алгоритм в целом обязательно завершаются.

4.Результативность - по завершении выполнения алгоритма обязательно получается конечный результат.

5.Выполнимость (эффективность) - результата алгоритма достигается за конечное число шагов.

6.Детерминированность (определенность) - алгоритм не должен содержать предписаний, смысл которых может восприниматься неоднозначно. Т.е. одно и то же предписание после исполнения должно давать один и тот же результат.

7.Последовательность – порядок исполнения команд должен быть понятен исполнителю и не должен допускать неоднозначности.

**Классы алгоритмов.**

1.**вычислительные алгоритмы**, работающие со сравнительно простыми видами данных, такими как числа и матрицы, хотя сам процесс вычисления может быть долгим и сложным;

2.**информационные алгоритмы**, представляющие собой набор сравнительно простых процедур, работающих с большими объемами информации (алгоритмы баз данных);

3.**управляющие алгоритмы**, генерирующие различные управляющие воздействия на основе данных, полученных от внешних процессов, которыми алгоритмы управляют.

По типу передачи управления алгоритмы бывают: основные (главные выполняемые программы) и вспомогательные (подпрограммы).

Для задания алгоритма необходимо описать следующие его элементы:

1.набор объектов, составляющих совокупность возможных исходных данных, промежуточных и конечных результатов;

2.правило начала;

3.правило непосредственной переработки информации (описание последовательности действий);

4.правило окончания;

5.правило извлечения результатов.

**Способы описания алгоритмов.**

Символьный, когда алгоритм описывается с помощью специального набора символов (специального языка).

Словесная форма записи алгоритмов обычно используется для алгоритмов, ориентированных на исполнителя-человека. Команды такого алгоритма выполняются в естественной последовательности, если не оговорено противного.

Графическая запись с помощью блок-схем осуществляется рисованием последовательности геометрических фигур, каждая из которых подразумевает выполнение определенного действия алгоритма. Порядок выполнения действий указывается стрелками. Графическая запись алгоритма имеет ряд преимуществ: каждая операция вычислительного процесса изображается отдельной геометрической фигурой и графическое изображение алгоритма наглядно показывает разветвления путей решения задачи в зависимости от различных условий, повторение отдельных этапов вычислительного процесса и другие детали.

**Виды блоков.**



**Правила создания блок – схем:**

1.Линии, соединяющие блоки и указывающие последовательность связей между ними, должны проводится параллельно линиям рамки.

2.Стрелка в конце линии может не ставиться, если линия направлена слева направо или сверху вниз.

3.В блок может входить несколько линий, то есть блок может являться преемником любого числа блоков.

4.Из блока (кроме логического) может выходить только одна линия.

5.Логический блок может иметь в качестве продолжения один из двух блоков, и из него выходят две линии.

6.Если на схеме имеет место слияние линий, то место пересечения выделяется точкой. В случае, когда одна линия подходит к другой и слияние их явно выражено, точку можно не ставить.

7.Схему алгоритма следует выполнять как единое целое, однако в случае необходимости допускается обрывать линии, соединяющие блоки.

**Виды алгоритмов.**

В линейном алгоритме операции выполняются последовательно, в порядке их записи. Каждая операция является самостоятельной, независимой от каких-либо условий. На схеме блоки, отображающие эти операции, располагаются в линейной последовательности.

В алгоритме с ветвлением предусмотрено несколько направлений (ветвей). Каждое отдельное направление алгоритма обработки данных является отдельной ветвью вычислений. Направление ветвления выбирается логической проверкой, в результате которой возможны два ответа:

1.«да» — условие выполнено.

2.«нет» — условие не выполнено.

Циклические алгоритмы содержат цикл – это многократно повторяемый участок алгоритма.Различают циклы с предусловием и постусловием.Также циклы бывают детерминированные и итерационные.Цикл называется детерминированным, если число повторений тела цикла заранее известно или определено. Цикл называется итерационным, если число повторений тела цикла заранее неизвестно, а зависит от значений параметров (некоторых переменных), участвующих в вычислениях.

2

Компиляция!

Компиляция относится к обработке файлов исходного кода (.c, .cc, или .cpp) и создании объектных файлов проекта. На этом этапе не создается исполняемый файл. Вместо этого компилятор просто транслирует высокоуровневый код в машинный язык. Например, если вы создали (но не скомпоновали) три отдельных файла, у вас будет три объектных файла, созданные в качестве выходных данных на этапе компиляции. Расширение таких файлов будет зависеть от вашего компилятора, например \*.obj или \*.o. Каждый из этих файлов содержит машинные инструкции, которые эквивалентны исходному коду. Но вы не можете запустить эти файлы! Вы должны превратить их в исполняемые файлы операционной системы, только после этого их можно использовать. Вот тут за дело берётся компоновщик.

Компоновка!

Из нескольких объектных файлов создается единый исполняемый файл. На этом этапе полученный файл является единственным, а потому компоновщик будет жаловаться на найденные неопределенные функции. На этапе компиляции, если компилятор не мог найти определение для какой-то функции, считается, что функция была определена в другом файле. Если это не так, компилятор об этом знать не будет, так как не смотрит на содержание более чем одного файла за раз. Компоновщик, с другой стороны, может смотреть на несколько файлов и попытаться найти ссылки на функции, которые не были упомянуты.

Вы спросите, почему этапы компиляции и компоновки разделены. Во-первых, таким образом легче реализовать процесс построения программ. Компилятор делает свое дело, а компоновщик делает свое дело — посредством разделения функций, сложность программы

снижается. Другим (более очевидным) преимуществом является то, что это позволяет создавать большие программы без необходимости повторения шага компиляции каждый раз, когда некоторые файлы будут изменены. Вместо этого, используется так называемая «условная компиляция». То есть объекты составляются только для тех исходных файлов, которые были изменены, для остальных, объектные файлы не пересоздаются. Тот факт, что каждый файл компилируется отдельно от информации, содержащейся в других файлах,

существует благодаря разделению процесса построения проекта на этапы компиляции и компоновки.

[Интегрированная среда разработки (IDE)](http://cppstudio.com/dlya-nachinayushhix/vybor-kompilyatora/) эти два этапа берёт на себя и вам не стоит беспокоиться о том, какие из файлов были изменены. IDE сама решает,когда создавать объекты файлов, а когда нет.

Зная разницу между фазами компиляции и компоновки вам будет намного проще находить ошибки в своих проектах. Компилятор отлавливает, как правило, [синтаксические ошибки](http://cppstudio.com/dlya-nachinayushhix/vybor-kompilyatora/ispravlenie-oshibok-kompilyacii-process-kompilyacii/) — отсутствие точки с запятой или скобок. Если вы получаете сообщение об ошибке, множественного определения функции или переменной, знайте, вам об этом сообщает компоновщик. Эта ошибка может означать только одно, что в нескольких файлах проекта определены одна и та же функция или переменная.

3

# Парадигмы программирования

Что такое парадигма вообще? Можно сказать, что это определенный взгляд на явления окружающего мира и представление о возможных действиях с ними. В программировании под парадигмой принято понимать обобщение о том, как должна быть организована работа программы.

Среди прочего выделяют такие парадигмы программирования как **директивное (структурное)**, **объектно-ориентированное** и **декларативное (функционально-логическое)**. Многие языки поддерживают несколько парадигм программирования. С другой стороны, есть языки ориентированные исключительно на реализацию одной парадигмы.

## Структурное программирование

Некоторые представители: Fortran, Pascal, C.

Директивная программа предписывает, как достичь результата, **пошагово описывая действия**. Поэтому такое программирование является достаточно легким для понимания.

В структурном программировании от входных данных полностью зависит последовательность выполнения команд.

В директивном программировании в свое время возникла концепция локализации части кода в так называемые подпрограммы (функции, методы), с последующим их вызовом из разных мест основной программы. При вызове в подпрограмму могут передаваться какие-либо данные в виде аргументов; а подпрограмма, в свою очередь, может возвращать в главную программу результат (т.е. полученные в ходе ее выполнения данные).

## Функциональное и логическое программирование

Представители функциональных языков: List, Haskell.

Представитель логических языков: Prolog.

Декларативная программа заявляет (декларирует), что должно быть достигнуто в качестве цели. Важным является точная формулировка задачи. Программист не задает алгоритм для ее решения.

Функциональное программирование основано на математическом понятии функции, которая не изменяет свое окружение; это отличие функционального программирования от функций в структурных языках. Функциональная программа состоит из совокупности определений функций, которые в свою очередь представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Каждая функция возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается; этот процесс повторяется до тех пор, пока не будет достигнут результат.

В логическом программировании программы выражены в виде формул математической логики, и решение задачи достигается путем вывода логических следствий из них.

## Объектно-ориентированное программирование

Представители объектно-ориентированных языков: С++, Java, Python.

Особое внимание уделяется данным, которые представляются в программе в виде объектов. Объекты взаимодействуют между собой с помощью механизма передачи сообщений. Задача программиста - реализовать такие объекты, при взаимодействии которых можно будет получать желаемый результат.

ООП призвано решать более сложные и объемные задачи по сравнению с директивным программированием.

В основе ООП лежат такие понятия как **наследование**, **полиморфизм** и **инкапсуляция**.

Инкапсуляция предполагает, что малозначащие детали объекта скрыты. Объект, получая какую-либо команду, сам «знает» как ее обработать исходя из того, к какому классу он принадлежит.

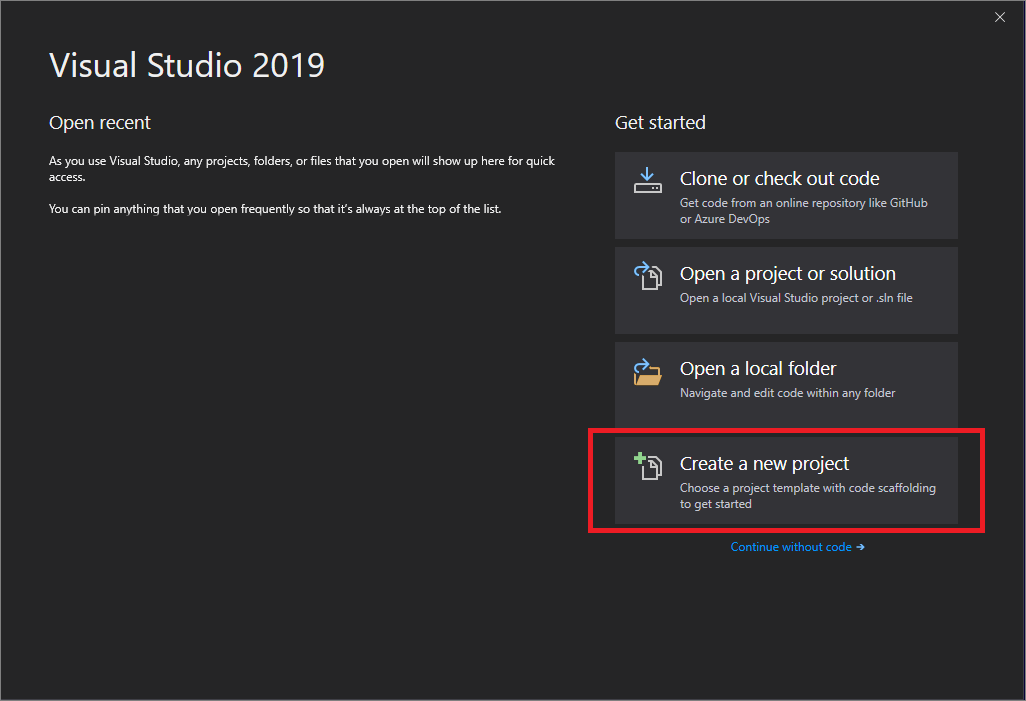
Все объекты являются экземплярами классов, которые по отношению друг к другу могут выступать в роли родитель-потомок. Дочерние классы наследуют свойства родительского. В случае, когда 100% наследование не требуется, выручает так называемый полиморфизм, который предполагает переопределение методов родительского класса в дочерних классах.

4

## Создание проекта

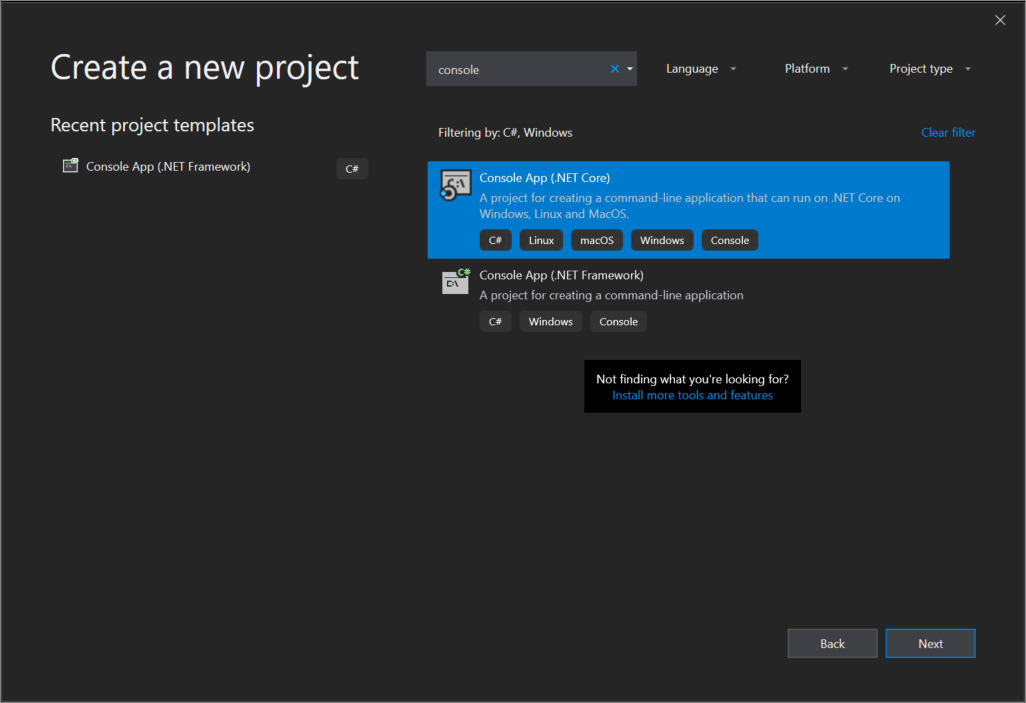
Сначала вы создадите проект приложения на C#. Для этого типа проекта уже имеются все нужные файлы шаблонов, что избавляет вас от лишней работы.

1. Запустите Visual Studio.
2. На начальном экране выберите **Создать проект**.



1. В поле поиска окна **Создание проекта** введите консоль. Затем выберите **C#** в списке языков и **Windows** в списке платформ.

Применив фильтры языка и платформы, выберите шаблон **Консольное приложение (.NET Core)** и нажмите кнопку **Далее**.

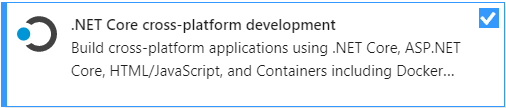


**Примечание**

Если шаблон **Консольное приложение (.NET Core)** отсутствует, его можно установить из окна **Создание проекта**. В сообщении **Не нашли то, что искали?** выберите ссылку **Установка других средств и компонентов**.

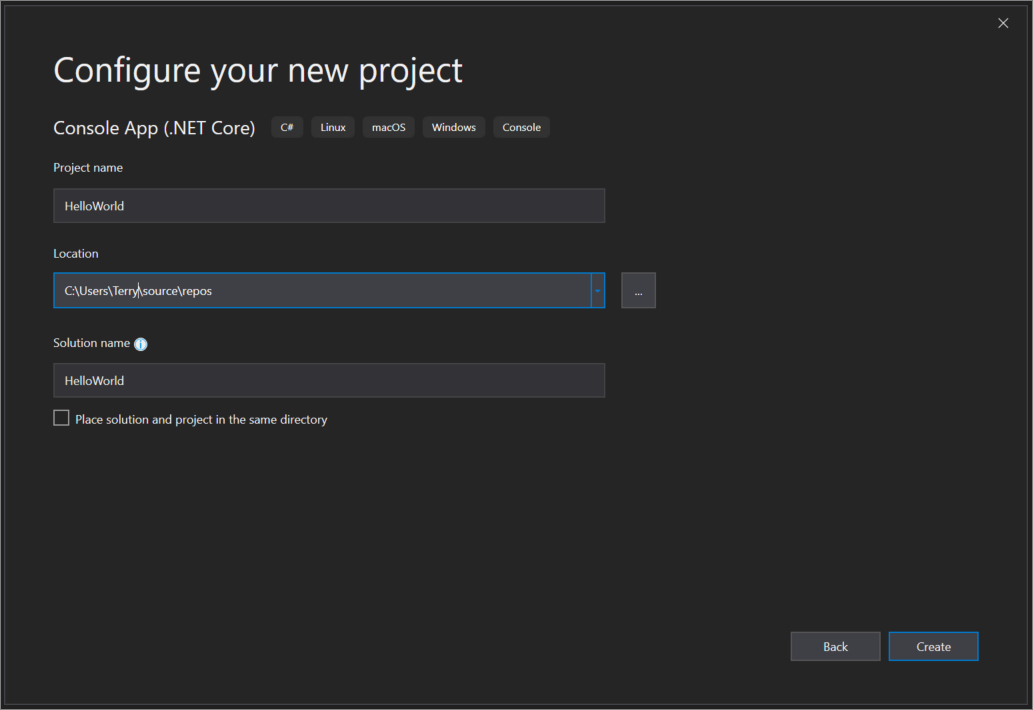
Снимок экрана: ссылка "Установка других средств и компонентов" из сообщения "Не нашли то, что искали?" в окне "Создание проекта"

После этого в Visual Studio Installer выберите рабочую нагрузку **Кроссплатформенная разработка .NET Core**.



Затем нажмите кнопку **Изменить** в Visual Studio Installer. Вам может быть предложено сохранить результаты работы; в таком случае сделайте это. Выберите **Продолжить**, чтобы установить рабочую нагрузку. После этого вернитесь к шагу 2 в процедуре [**Создание проекта**](https://docs.microsoft.com/ru-ru/visualstudio/ide/quickstart-csharp-console?view=vs-2019#create-a-project).

1. В поле **Имя проекта** окна **Настроить новый проект** введите HelloWorld. Затем нажмите **Создать**.

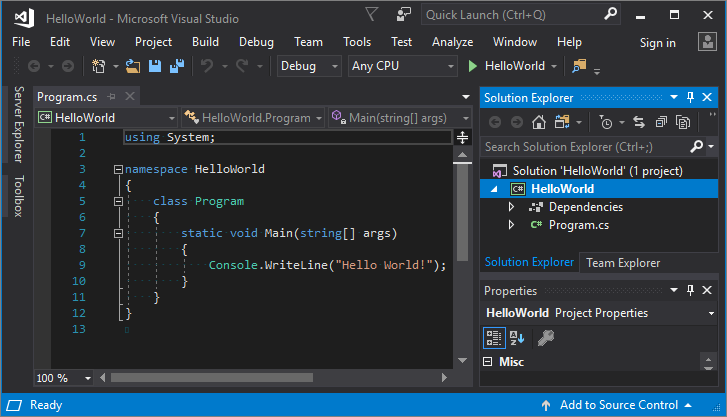


Новый проект открывается в Visual Studio.

## Создание приложения

Visual Studio включает код по умолчанию "Hello World" в проект.

(Для этого Visual Studio вызывает метод [WriteLine](https://docs.microsoft.com/ru-ru/dotnet/api/system.console.writeline) для отображения строкового литерала "Hello World!" в окне консоли.)



Если нажать клавишу **F5**, программа запустится в режиме отладки. Но окно консоли отображается на короткое время, перед тем как закрыться.

(Это происходит потому, что метод Main завершается после выполнения его единственного оператора, после чего завершается работа приложения.)

### Добавление кода

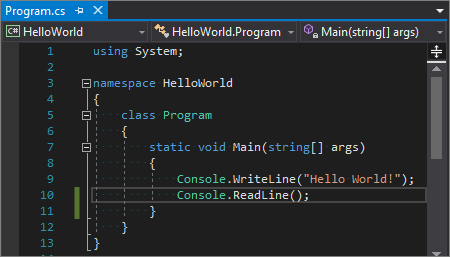
Добавим код для приостановки работы приложения, чтобы окно консоли не закрывалось, пока вы не нажмете клавишу **ВВОД**.

1. Добавьте следующий код сразу после вызова метода [WriteLine](https://docs.microsoft.com/ru-ru/dotnet/api/system.console.writeline):

C#Копировать

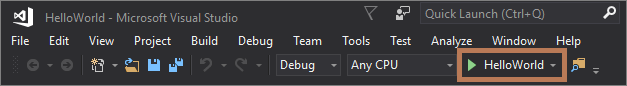
Console.ReadLine();

1. Убедитесь, что он выглядит в редакторе кода следующим образом:

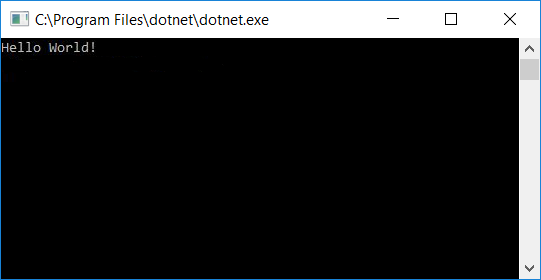


## Выполнение приложения

1. Нажмите кнопку **HelloWorld** на панели инструментов, чтобы запустить приложение в режиме отладки. Или нажмите клавишу **F5**.

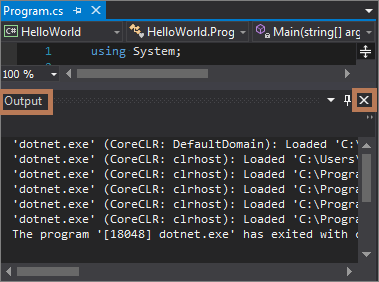


1. Просмотрите результат в окне консоли.



### Закрытие приложения

1. Нажмите клавишу **ВВОД**, чтобы закрыть окно консоли.
2. Закройте область **вывода** в Visual Studio.



1. Закройте Visual Studio

|  |  |
| --- | --- |
| Запуск без отладки | **CTRL+F5** |
| Предыдущая область разделения | **SHIFT+F6** |

5

Отладчик — важнейший инструмент для поиска и устранения ошибок в приложениях. Однако большое значение имеет контекст. Важно использовать все средства, имеющиеся в вашем распоряжении, чтобы быстро устранять ошибки. Зачастую лучшим "средством" являются правильные методики написания кода. Зная, когда лучше использовать отладчик, а когда — другие средства, вы также сможете более эффективно использовать отладчик.

6

Платформа .NET Framework — это технология, которая поддерживает создание и выполнение веб-служб и приложений Windows. При разработке платформы .NET Framework учитывались следующие цели.

* Обеспечение согласованной объектно-ориентированной среды программирования для локального сохранения и выполнения объектного кода, для локального выполнения кода, распределенного в Интернете, либо для удаленного выполнения.
* Предоставление среды выполнения кода, в которой:
  + сведена к минимуму вероятность конфликтов в процессе развертывания программного обеспечения и управления его версиями;
  + гарантируется безопасное выполнение кода, включая код, созданный неизвестным или не полностью доверенным сторонним изготовителем;
  + исключаются проблемы с производительностью сред выполнения скриптов или интерпретируемого кода;
* обеспечиваются единые принципы разработки для разных типов приложений, таких как приложения Windows и веб-приложения;
* обеспечивается взаимодействие на основе промышленных стандартов, которое гарантирует интеграцию кода платформы .NET Framework с любым другим кодом.

7

* **bool**: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

|  |  |
| --- | --- |
| 1  2 | bool alive = true;  bool isDead = false; |

* **byte**: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

|  |  |
| --- | --- |
| 1  2 | byte bit1 = 1;  byte bit2 = 102; |

* **sbyte**: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

|  |  |
| --- | --- |
| 1  2 | sbyte bit1 = -101;  sbyte bit2 = 102; |

* **short**: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

|  |  |
| --- | --- |
| 1  2 | short n1 = 1;  short n2 = 102; |

* **ushort**: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

|  |  |
| --- | --- |
| 1  2 | ushort n1 = 1;  ushort n2 = 102; |

* **int**: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

|  |  |
| --- | --- |
| 1  2  3 | int a = 10;  int b = 0b101;  // бинарная форма b =5  int c = 0xFF;   // шестнадцатеричная форма c = 255 |

* **uint**: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

|  |  |
| --- | --- |
| 1  2  3 | uint a = 10;  uint b = 0b101;  uint c = 0xFF; |

* **long**: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

|  |  |
| --- | --- |
| 1  2  3 | long a = -10;  long b = 0b101;  long c = 0xFF; |

* **ulong**: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

|  |  |
| --- | --- |
| 1  2  3 | ulong a = 10;  ulong b = 0b101;  ulong c = 0xFF; |

* **float**: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single
* **double**: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double

8

#### **1. Для чего в программе используются константы и переменные?**

Константы и переменные есть основными объектами, которые используются в программе. Константы и переменные – это именованные объекты, для которых выделяется память.

Значение константы обязательно устанавливается при ее объявлении в программе.

В отличие от переменной значение константы не может изменяться программно. Если константа уже объявлена, то при попытке присвоить константе какое-нибудь новое значение компилятор выдаст сообщение об ошибке.

Все переменные и константы должны быть объявлены до их применения. Это нужно, чтобы сообщить компилятору о типе данных, который сохраняется в переменной. Если известен тип переменной, то компилятор осуществляет соответствующую компиляцию любого оператора, в котором используется данная переменная.

### [⇑](https://www.bestprog.net/ru/2016/10/19/%d0%bf%d0%b5%d1%80%d0%b5%d0%bc%d0%b5%d0%bd%d0%bd%d1%8b%d0%b5-%d0%b8-%d0%ba%d0%be%d0%bd%d1%81%d1%82%d0%b0%d0%bd%d1%82%d1%8b/#content)

#### **2. Каким образом в C# объявляется константа? Примеры объявление констант.**

Константа объявляется с помощью оператора следующей формы:

**const тип имя\_константы = значение;**

где

***const*** – ключевое слово, которое определяет что именно это есть константа;**тип**– тип данных, которые сохраняются в константе; **имя\_константы** – имя константы; **значение**– значение константы, которое есть неизменяемым в программе.

Примеры описания констант.

**const decimal m = 333897987.3949M; // константа типа decimal**

**const double pi = 3.1415; // константа типа double**

**const bool YES = true; // константа типа bool**

**const int Max = 255; // константа типа int**

**const char AnswerYes = 'Y'; // константа типа char**

**const string Str = "Constant String"; // константа типа string**

### [⇑](https://www.bestprog.net/ru/2016/10/19/%d0%bf%d0%b5%d1%80%d0%b5%d0%bc%d0%b5%d0%bd%d0%bd%d1%8b%d0%b5-%d0%b8-%d0%ba%d0%be%d0%bd%d1%81%d1%82%d0%b0%d0%bd%d1%82%d1%8b/#content)

#### **3. Как в программе объявляется переменная? Примеры объявления переменных.**

Переменные объявляются с помощью оператора следующей формы:

**тип имя\_переменной;**

где **тип** – это тип данных, которые сохраняются в переменной; а **имя\_переменной** – это ее имя.

Объявить переменную можно любого типа, который существует в программе. Тип переменной может быть также [**типом-значением**](https://www.bestprog.net/ru/2016/08/08/02-%d0%b1%d0%b0%d0%b7%d0%be%d0%b2%d1%8b%d0%b5-%d1%81%d0%b8%d1%81%d1%82%d0%b5%d0%bc%d0%bd%d1%8b%d0%b5-%d0%b8%d0%bb%d0%b8-%d1%82%d0%b8%d0%bf%d1%8b-%d0%b7%d0%bd%d0%b0%d1%87%d0%b5%d0%bd%d0%b8/).

Пример объявления переменной, которая имеет значимый тип.

**double d, f; // 2 переменные типа double**

**int i, j; // 2 переменные типа int**

**bool answer; // переменная типа bool**

**char c; // переменная типа char**

**string s; // переменная типа string**

**d = 2.939;**

**f = -29309.39;**

**i = 50;**

**j = -300;**

**c = '=';**

**s = "This is a text";**

**answer = false;**

**9**

Ключевое **слово var** ссылается на тип неявным способом. Это псевдоним любого типа. Реальный тип определит компилятор C#. Использование **var** никак не ухудшает производительность.

10

Поскольку код C# является статически типизированным во время компиляции, после объявления переменной ее нельзя объявить повторно или назначить ей значения другого типа, если этот тип невозможно неявно преобразовать в тип переменной. Например, string невозможно неявно преобразовать в int. Поэтому после объявления i как int нельзя назначить ей строку "Hello", как показано в следующем коде:

C#Копировать

int i;

// error CS0029: Cannot implicitly convert type 'string' to 'int'

i = "Hello";

Тем не менее иногда может потребоваться скопировать значение в переменную или параметр метода другого типа. Например, может потребоваться передать целочисленную переменную в метод, параметр которого имеет тип double. Или может понадобиться присвоить переменную класса переменной типа интерфейса. Такого рода операции называются *преобразованиями типа*. В C# можно выполнять следующие виды преобразований.

* **Неявные преобразования**. Специальный синтаксис не требуется, так как преобразование всегда завершается успешно и данные не будут потеряны. Примеры включают преобразования из меньших в большие целочисленные типы и преобразования из производных классов в базовые классы.
* **Явные преобразования (приведения)** . Для явных преобразований требуется [выражение приведения](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/type-testing-and-cast#cast-expression). Приведение требуется, если в ходе преобразования данные могут быть утрачены или преобразование может завершиться сбоем по другим причинам. Типичными примерами являются числовое преобразование в тип с меньшей точностью или меньшим диапазоном и преобразование экземпляра базового класса в производный класс.
* **Пользовательские преобразования**. Такие преобразования выполняются специальными методами, которые можно определить для включения явных и неявных преобразований между пользовательскими типами без связи "базовый класс — производный класс". Дополнительные сведения см. в разделе [Операторы пользовательского преобразования](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/user-defined-conversion-operators).
* **Преобразования с использованием вспомогательных классов**. Чтобы выполнить преобразование между несовместимыми типами, например целыми числами и объектами [System.DateTime](https://docs.microsoft.com/ru-ru/dotnet/api/system.datetime) или шестнадцатеричными строками и массивами байтов, можно использовать классы [System.BitConverter](https://docs.microsoft.com/ru-ru/dotnet/api/system.bitconverter) и [System.Convert](https://docs.microsoft.com/ru-ru/dotnet/api/system.convert), а также методы Parse встроенных числовых типов, такие как [Int32.Parse](https://docs.microsoft.com/ru-ru/dotnet/api/system.int32.parse). Дополнительные сведения см. в руководствах по [преобразованию массива байтов в значение типа int](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/how-to-convert-a-byte-array-to-an-int), [преобразованию строки в число](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/how-to-convert-a-string-to-a-number) и [преобразованию из шестнадцатеричных строк в числовые типы](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/how-to-convert-between-hexadecimal-strings-and-numeric-types).

11

**Префиксный декремент** уменьшает переменную до её использования в других выражениях, а **постфиксный декремент**, наоборот, уменьшает переменную только после того, как она будет использована в выражении. **Декремент** записывается как два минуса: – – . В коде предекремент ставится перед переменной, а постдекремент – после.

### 12Форматирование строк

При выводе строк в консоли с помощью метода Console.WriteLine мы можем применять форматирование вместо конкатенации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Program  {      static void Main(string[] args)      {          Person person = new Person { Name = "Tom", Age = 23 };            Console.WriteLine("Имя: {0}  Возраст: {1}", person.Name, person.Age);          Console.Read();      }  }    class Person  {      public string Name { get; set; }      public int Age { get; set; }  } |

В строке "Имя: {0} Возраст: {1}" на место {0} и {1} затем будут вставляться в порядке следования person.Name и person.Age

То же самое мы можем сделать с помощью метода **String.Format**:

|  |  |
| --- | --- |
| 1  2 | string output = String.Format("Имя: {0}  Возраст: {1}", person.Name, person.Age);  Console.WriteLine(output); |

Метод Format принимает строку с плейсхолдерами типа {0}, {1} и т.д., а также набор аргументов, которые вставляются на место данных плейсхолдеров. В итоге генерируется новая строка.

В методе Format могут использоваться различные спецификаторы и описатели, которые позволяют настроить вывод данных. Рассмотрим основные описатели.

#### Все используемые форматы:

|  |  |
| --- | --- |
| **C / c** | Задает формат денежной единицы, указывает количество десятичных разрядов после запятой |
| **D / d** | Целочисленный формат, указывает минимальное количество цифр |
| **E / e** | Экспоненциальное представление числа, указывает количество десятичных разрядов после запятой |
| **F / f** | Формат дробных чисел с фиксированной точкой, указывает количество десятичных разрядов после запятой |
| **G / g** | Задает более короткий из двух форматов: F или E |
| **N / n** | Также задает формат дробных чисел с фиксированной точкой, определяет количество разрядов после запятой |
| **P / p** | Задает отображения знака процентов рядом с число, указывает количество десятичных разрядов после запятой |
| **X / x** | Шестнадцатеричный формат числа |

#### Форматирование валюты

Для форматирования валюты используется описатель "C":

|  |  |
| --- | --- |
| 1  2  3  4  5 | double number = 23.7;  string result = String.Format("{0:C}", number);  Console.WriteLine(result); // $ 23.7  string result2 = String.Format("{0:C2}", number);  Console.WriteLine(result2); // $ 23.70 |

Число после описателя указывает, сколько чисел будет использоваться после разделителя между целой и дробной частью. При выводе также добавляется обозначение денежного знака для текущей культуры компьютера.

#### Форматирование целых чисел

Для форматирования целочисленных значение применяется описатель "d":

|  |  |
| --- | --- |
| 1  2  3  4  5 | int number = 23;  string result = String.Format("{0:d}", number);  Console.WriteLine(result); // 23  string result2 = String.Format("{0:d4}", number);  Console.WriteLine(result2); // 0023 |

Число после описателя указывает, сколько цифр будет в числовом значении. Если в исходном числе цифр меньше, то к нему добавляются нули.

#### Форматирование дробных чисел

Для форматирования дробны чисел используется описатель F, число после которого указывает, сколько знаков будет использоваться после разделителя между целой и дробной частью. Если исходное число - целое, то к нему добавляются разделитель и нули.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int number = 23;  string result = String.Format("{0:f}", number);  Console.WriteLine(result); // 23,00    double number2 = 45.08;  string result2 = String.Format("{0:f4}", number2);  Console.WriteLine(result2); // 45,0800    double number3 = 25.07;  string result3 = String.Format("{0:f1}", number3);  Console.WriteLine(result2); // 25,1 |

#### Формат процентов

Описатель "P" задает отображение процентов. Используемый с ним числовой спецификатор указывает, сколько знаков будет после запятой:

|  |  |
| --- | --- |
| 1  2 | decimal number = 0.15345m;  Console.WriteLine("{0:P1}", number);// 15.3% |

#### Настраиваемые форматы

Используя знак #, можно настроить формат вывода. Например, нам надо вывести некоторое число в формате телефона +х (ххх)ххх-хх-хх:

|  |  |
| --- | --- |
| 1  2  3 | long number = 19876543210;  string result = String.Format("{0:+# (###) ###-##-##}", number);  Console.WriteLine(result); // +1 (987) 654-32-10 |

### Метод ToString

Метод ToString() не только получает строковое описание объекта, но и может осуществлять форматирование. Он поддерживает те же описатели, что используются в методе Format:

|  |  |
| --- | --- |
| 1  2  3  4  5 | long number = 19876543210;  Console.WriteLine(number.ToString("+# (###) ###-##-##"));// +1 (987) 654-32-10    double money = 24.8;  Console.WriteLine(money.ToString("C2")); // $ 24,80 |

### Интерполяция строк

Начиная с версии языка C# 6.0 была добавлена такая функциональность, как интерполяция строк. Эта функциональность призвана заменить форматирование строк. Так, перепишем пример с выводом значений свойств объекта Person:

|  |  |
| --- | --- |
| 1  2  3 | Person person = new Person { Name = "Tom", Age = 23 };    Console.WriteLine($"Имя: {person.Name}  Возраст: {person.Age}"); |

Знак доллара перед строкой указывает, что будет осуществляться интерполяция строк. Внутри строки опять же используются плейсхолдеры {...}, только внутри фигурных скобок уже можно напрямую писать те выражения, которые мы хотим вывести.

Интерполяция по сути представляет более лаконичное форматирование. При этом внутри фигурных скобок мы можем указывать не только свойства, но и различные выражения языка C#:

|  |  |
| --- | --- |
| 1  2  3  4 | int x = 8;  int y = 7;  string result = $"{x} + {y} = {x + y}";  Console.WriteLine(result); // 8 + 7 = 15 |

В следующем примере проверяем, не равен ли person значению null. Если не равен, то выводим его имя, иначе выводим какое-нибудь имя по умолчанию:

|  |  |
| --- | --- |
| 1  2  3 | person = null;  string output = $"{person?.Name??"Имя по умолчанию"}";  Console.WriteLine(output); |

Уже внутри строки можно применять форматирование. В этом случае мы можем применять все те же описатели, что и в методе Format. Например, выведем номер телефона в формате +x xxx-xxx-xx-xx:

|  |  |
| --- | --- |
| 1  2 | long number = 19876543210;  Console.WriteLine($"{number:+# ### ### ## ##}"); // +1 987 654 32 10 |

Добавляем пространство до и после форматируемого вывода:

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine($"Имя: {person.Name, -5} Возраст: {person.Age}"); // пробелы после  Console.WriteLine($"Имя: {person.Name, 5} Возраст: {person.Age}"); // пробелы до |