

# Ветвления и циклы

## 1 Логический тип и сравнения

`bool` — **логический тип** данных в C#. Переменная типа `bool` может принимать только два значения: `true` или `false`.  
Используется для представления логических состояний (включён/выключен, условие выполнено/не выполнено, валидно/невалидно) и для управления потоком выполнения программы (ветвления `if`, циклы `while` / `for`, тернарный оператор и т.д.).

```
// Явная инициализация
bool flag1 = true;
bool flag2 = false;

// Инициализация результатом выражения
int a = 5, b = 7;
bool less = a < b;    // true

// С использованием var (компилятор выводит bool)
var isEqual = (a == b); // false

// Операторы сравнения
int x = 10;
int y = 7;

Console.WriteLine(x == y); // False  (равенство)
Console.WriteLine(x != y); // True   (неравенство)
Console.WriteLine(x < y);   // False
Console.WriteLine(x > y);   // True
Console.WriteLine(x <= 10); // True
Console.WriteLine(y >= 7);  // True
```

## Числа с плавающей точкой — почему их нельзя сравнивать напрямую

`float` и `double` хранят приближённые значения. Операции накапливают погрешности.

К примеру, такой код выведет `false`, хотя ожидается `true`:

```
double x = 0.1 + 0.2;  
Console.WriteLine(x == 0.3); // false
```

Почему: 0.1, 0.2 не представляются точно в двоичном формате → ошибка представления → сравнение даст `false`.

Правильный способ — сравнение с эпсилон (допуском):

```
double a = 0.1 + 0.2;  
double b = 0.3;  
double eps = 1e-12;  
bool equal = Math.Abs(a - b) < eps;  
Console.WriteLine(equal); // true (при подходящем eps)
```

Рекомендация: выбирать `eps` в зависимости от контекста/масштаба чисел (для `float` можно использовать `1e-6` ...).

## 2 Логические операции

### Условные (сокращённые)

Эти операторы вычисляют правый операнд, только если это необходимо.

- `&&` — логическое И (short-circuit): если первый операнд `false`, второй не вычисляется.
- `||` — логическое ИЛИ (short-circuit): если первый операнд `true`, второй не вычисляется.
- `!` — логическое НЕ.

### Битовые (полные)

Эти операторы всегда обрабатывают оба операнда.

- `&`, `|`, `^` — применимы к целым типам (по битам) и к `bool`.
- `~` — побитовое отрицание.

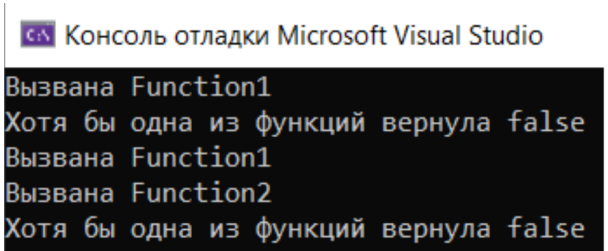
Пример:

```
Ссылка: 2
static bool Function1()
{
    Console.WriteLine("Вызвана Function1");
    return false;
}

Ссылка: 2
static bool Function2()
{
    Console.WriteLine("Вызвана Function2");
    return true;
}

Ссылка: 0
static void Main(string[] args)
{
    if (Function1() && Function2())
        Console.WriteLine("Обе функции вернули true");
    else
        Console.WriteLine("Хотя бы одна из функций вернула false");

    if (Function1() & Function2())
        Console.WriteLine("Обе функции вернули true");
    else
        Console.WriteLine("Хотя бы одна из функций вернула false");
}
```



Почему: Потому что оператор `&&` понимает, что если первый аргумент равен `false`, то и значение всего выражения — `false`, и не запрашивает вычисление второго аргумента. Оператор `&` этого не понимает.

Используйте `&&` / `||` в логических выражениях почти всегда. `&` / `|` — для битовых операций или если вам нужно всегда вычислять оба аргумента (редко).

### 3 Ветвление

**Ветвление** — это конструкция управления потоком выполнения программы, которая позволяет выполнять разные участки кода в зависимости от логических условий. По сути это способ программно отвечать на разные «сценарии» поведения: пользователь ввёл корректные данные — выполняем одну ветку, нет — другую; значение переменной соответствует варианту — обрабатываем его соответствующим образом и т.д.

Ветвления нужны для:

- принятия решений (например, доступ/запрет, валидация ввода),
- разделения логики на понятные случаи,
- предотвращения выполнения ненужного/опасного кода.



## Синтаксис `if / else if / else`

```
if (condition) {  
    // выполняется, если condition == true  
}  
else if (otherCondition) {  
    // выполняется, если condition == false и otherCondition == true  
}  
else {  
    // выполняется, если все предыдущие условия == false  
}
```

Пример:

```
int score = 78;  
  
// если выражение после if/else в одну строку, можно не писать {}  
if (score >= 90)  
    Console.WriteLine("Отлично");  
else if (score >= 75)  
    Console.WriteLine("Хорошо");  
else if (score >= 60)  
    Console.WriteLine("Удовлетворительно");  
else  
    Console.WriteLine("Неудовлетворительно");  
// Ожидаемый вывод: "Хорошо"
```

## Вложенные `if` и «плохая» вложенность

Вложенные `if` иногда необходимы, но глубоко вложенные конструкции усложняют чтение и поддержку кода.

Плохо (трудно читать):

```
if (user != null) {  
    if (user.IsActive) {  
        if (user.HasPermission("write")) {  
            // делаем что-то  
        }  
    }  
}
```

Лучше — ранние выходы:

```
if (user == null) return;  
if (!user.IsActive) return;  
if (!user.HasPermission("write")) return;  
  
// делаем что-то
```

Ранние выходы уменьшают глубину вложенности и делают логику прямолинейной.

## enum И switch

`enum` (перечисление) — это именованный набор целочисленных констант, который используется для представления фиксированного множества значений. Главная цель `enum` — сделать код более читаемым и понятным.

Например, вместо того чтобы хранить дни недели, можно использовать `enum DayOfWeek`.

```
enum DayOfWeek
{
    Monday,    // 0
    Tuesday,   // 1
    Wednesday, // 2
    Thursday,  // 3
    Friday,    // 4
    Saturday,  // 5
    Sunday     // 6
}
```

По умолчанию:

- тип данных внутри `enum` — `int`,
- нумерация начинается с `0`.

Можно явно задавать **начальное значение** и даже **тип хранения** (`byte`, `short`, `long` и др.):

```
enum Status : byte
{
    Ok = 1,
    Warning = 2,
    Error = 3
}
```



`switch` — это оператор множественного выбора. Он используется, когда нужно сравнить **одно выражение** с несколькими возможными вариантами значений и выполнить разный код в зависимости от совпадения.

Он имеет следующий синтаксис:

```
switch (выражение)
{
    case значение1:
        // код для значения1
        break;
    case значение2:
        // код для значения2
        break;
    ...
    default:
        // код, если ни одно значение не совпало
        break;
}
```

- `выражение` может быть числом, символом, строкой, `enum` или любым типом с допустимым сравнением.
- `case` — вариант сравнения.
- `break` завершает выполнение текущего блока (иначе код пойдёт дальше).
- `default` — выполняется, если ни один вариант не подошёл (аналог `else`).

Пример:

```
DayOfWeek d = DayOfWeek.Wednesday;
switch (d)
{
    case DayOfWeek.Monday:
        Console.WriteLine("Начало рабочей недели");
        break;
    case DayOfWeek.Saturday:
    case DayOfWeek.Sunday:
        Console.WriteLine("Выходной");
        break;
    default:
        Console.WriteLine("Обычный рабочий день");
        break;
}
```

В новых версиях C#, `switch`-выражение можно записать более компактно:

```
string kind = d switch
{
    DayOfWeek.Saturday or DayOfWeek.Sunday => "Weekend",
    DayOfWeek.Monday => "Start",
    _ => "Weekday"
};
```

## Тернарный оператор (?:)

`condition ? exprIfTrue : exprIfFalse` — полезен для простых выборов значений в одну строку.

Примеры:

```
int a = 5, b = 7;
string result = (a < b) ? "a меньше b" : "a не меньше b";
Console.WriteLine(result); // "a меньше b"

// Вложенный (не рекомендуется делать слишком глубоко)
int score = 85;
string grade = score >= 90 ? "A" :
               score >= 75 ? "B" :
               score >= 60 ? "C" : "F";
Console.WriteLine(grade); // "B"
```

## Когда использовать `if` vs `switch` vs тернарный оператор

- `if` — для диапазонов, сложных логических выражений, проверки `null`, диапазонов `double`, комбинированных условий.
- `switch` / `switch -expression` — для выбора по дискретным значениям (`enum`, `int`, `string`) и когда есть много альтернатив; `switch -expression` (C# 8+) даёт компактный и читаемый синтаксис.
- Тернарный оператор — для простых присваиваний/вычислений в одну строку.

# Правила математической логики

## Упрощение

`A && A == A` , `A || A == A` . Удаляйте повторения.

```
// Плохо
if (isValid && isValid) ...
// Хорошо
if (isValid) ...
```

## Законы де Моргана

```
!(A && B) == !A || !B
!(A || B) == !A && !B
```

Помогает перевести отрицание внутрь и упростить читаемость.

```
// Вместо
if (!(x > 0 && y > 0)) ...
// Лучше (эквивалентно и часто понятнее)
if (x <= 0 || y <= 0) ...
```

## Избегайте повторного вычисления одинаковых выражений

Если подвыражение дорогое или содержит побочный эффект, вычислите его один раз.

```
// Плохо: Expensive() выполняется дважды
if (Expensive() && SomeOther(Expensive())) ...
// Хорошо
var val = Expensive();
if (val && SomeOther(val)) ...
```

## Законы поглощения

```
A || (A && B) == A
```

```
A && (A || B) == A
```

```
// Плохо
if (isAdmin || (isAdmin && hasKey)) ...
// Хорошо
if (isAdmin) ...
```

## Двойное отрицание

`!!A == A`. Уменьшайте слой отрицаний для читаемости.

```
// Плохо
if (!!isEnabled) ...
// Хорошо
if (isEnabled) ...
```

## Дистрибутивность

```
(A && B) || (A && C) == A && (B || C)
```

```
(A || B) && (A || C) == A || (B && C)
```

```
// Плохо
// Неоптимальное выражение (здесь проверка age >= 18 дублируется дважды):
bool canEnter1 = (age >= 18 && hasPassport) || (age >= 18 && hasStudentCard);
// Оптимизированное выражение:
bool canEnter2 = age >= 18 && (hasPassport || hasStudentCard);
```

## 4 Массивы

**Массив** — это упорядоченная коллекция элементов одного типа, размещённая в памяти подряд. Массив позволяет хранить несколько значений и обращаться к ним по индексу (позиции). Индексация в C# начинается с 0: первый элемент — `arr[0]`, последний — `arr[arr.Length - 1]`.

Для чего нужен массив

- Хранение набора однотипных данных (например, оценки студентов, значения сенсоров).
- Быстрый доступ по индексу.
- Удобство обработки наборов данных в циклах.

Синтаксис и способы инициализации

```
// Объявление (без инициализации)
int[] a;
// Создание массива фиксированной длины (все элементы 0)
a = new int[5]; // индексы 0..4
// Одновременное объявление и инициализация
int[] b = new int[3] { 1, 2, 3 };
// Короткая форма (компилятор выведет длину)
int[] c = { 4, 5, 6 };
// Явно new без указания длины
int[] d = new int[] { 7, 8, 9 };
// Массив строк
string[] names = { "Alice", "Bob", "Eve" };
// Доступ к элементам
b[0] = 10;
int x = b[2]; // x == 3
// Длина массива
int len = b.Length;
```

# Интервалы и срезы

В C# начиная с версии 8.0 появился удобный синтаксис для работы с частями массивов — **срезы**. Они позволяют получать подмассивы, не используя вручную циклы.

- Запись `arr[start..end]` возвращает новый массив, включающий элементы с индекса `start` до `end - 1`.
- Если опустить границы:
  - `arr[..end]` — элементы с начала до `end - 1`.
  - `arr[start..]` — элементы от `start` и до конца.
  - `arr[..]` — копия всего массива.

**Пример:**

```
int[] numbers = { 10, 20, 30, 40, 50 };

int[] firstThree = numbers[..3];    // {10, 20, 30}
int[] lastTwo    = numbers[^2..];   // {40, 50}, ^2 означает "второй с конца"
int[] middle     = numbers[1..4];   // {20, 30, 40}
```

Это особенно удобно, когда нужно работать только с частью массива.

# Массивы массивов

Массив массивов — это структура, где каждый элемент "внешнего" массива сам является массивом.

Такая конструкция называется **рваный массив** (*jagged array*), потому что длина вложенных массивов может быть разной.

**Пример:**

```
int[][] jagged = new int[3][];  
jagged[0] = new int[] {1, 2, 3};  
jagged[1] = new int[] {4, 5};  
jagged[2] = new int[] {6, 7, 8, 9};
```

В итоге:

- `jagged[0]` → {1, 2, 3}
- `jagged[1]` → {4, 5}
- `jagged[2]` → {6, 7, 8, 9}

Так удобно хранить данные, у которых длины строк различаются (например, список студентов по группам, где в каждой группе разное количество людей).



# Многомерные массивы

Многомерный массив — это прямоугольная структура, где у всех измерений фиксированная длина. В C# можно объявить массивы с двумя или более измерениями.

**Пример двумерного массива:**

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

Здесь массив имеет **2 строки и 3 столбца**.

Так удобно хранить данные, которые представляют из себя таблицы фиксированной формы, например матрицы или значения пикселей на изображении и т.п.

## 5. Циклы

**Цикл** — конструкция, позволяющая повторять блок кода несколько раз, пока выполняется условие. Циклы нужны, чтобы:

- обработать все элементы массива/коллекции,
- повторить операцию N раз,
- ждать события/ввода пользователя (в контролируемом виде).

Основные циклы в C#:

- `while` — пока условие истинно;
- `for` — счетчик/индексные циклы (удобно, когда заранее известно число повторов);
- `foreach` — перебор коллекций (массивов, списков) без индекса.



## while

Цикл `while` работает следующим образом: проверяется `condition`. Если `true` — выполняется тело, затем проверка повторяется. Если `false` — цикл пропускается.

Синтаксис:

```
while (condition)
{
    // тело цикла
}
```

### Когда использовать

- Когда заранее неизвестно число итераций, и условие выхода формируется в процессе (например, чтение ввода до команды `exit`).

Пример:

```
int i = 0;
while (i < 5)
{
    Console.WriteLine(i);
    i++; // обязательно изменение состояния, чтобы цикл завершился
}
```

## for

Синтаксис:

```
for (инициализация; условие; итерация)
{
    // тело
}
```

Пример:

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

### Когда использовать

- Когда заранее известно количество итераций.
- Когда нужен индекс (i) для доступа к массиву/списку.
- Для циклов со сложной логикой итерации (шаги != 1, декремент и т.п.).

### Плюсы

- Компактно объединяет инициализацию, условие и шаг.
- Хорошо видно, откуда начинается и когда заканчивается цикл.

## foreach

Синтаксис:

```
foreach (var item in collection)
{
    // используем item (только для чтения в контексте value types)
}
```

Пример:

```
int[] arr = { 1, 2, 3 };
foreach (var v in arr)
{
    Console.WriteLine(v);
}
```

### Когда использовать

- Когда нужно обработать **все элементы** коллекции последовательно и не требуется индекс.
- Удобно, безопасно (нет риска выхода за границы).
- Поддерживается для всех `IEnumerable<T>` (массивы, списки и т.д.).

### Ограничение

- Нельзя изменять саму коллекцию (например, добавлять/удалять элементы) во время перебора; попытка изменить коллекцию обычно бросит исключение.
- Для значимых типов ( `int` ) переменная `item` — копия элемента; присваивание `item = ...` не изменит элемент массива.

## break и continue

`break` прерывает ближайший внешний цикл полностью и передаёт управление дальше по коду после цикла.

Пример:

```
for (int i = 0; i < 10; i++)
{
    if (i == 5) break; // выйдем из цикла при i == 5
    Console.WriteLine(i);
}
// Выведет 0,1,2,3,4
```

`continue` прерывает текущую итерацию и переходит к следующей.

Пример:

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) continue; // пропустить чётные
    Console.WriteLine(i);
}
// Выведет 1,3,5
```

## Выход из вложенного цикла

C# не имеет встроенного многоуровневого `break`, поэтому если нужно выйти из вложенного цикла, нужно использовать переменную-флаг:

```
bool found = false;
for (int i = 0; i < n && !found; i++)
{
    for (int j = 0; j < m; j++)
    {
        if (SomeCondition(i, j))
        {
            found = true;
            break; // выходит из внутреннего цикла
        }
    }
}
```

# Бесконечный цикл

Как сделать:

```
while (true)
{
    // код
    if (ShouldStop()) break; // условие выхода внутри цикла
}
```

Или (не рекомендуется):

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

Когда используют

- Для программ, которые работают постоянно, принимают запросы и обрабатывают их, пока их не остановят вручную (например серверы).
- Для циклов ожидания пользовательского ввода (например консольные приложения).
- В тестах/симуляциях, где цикл должен работать, пока не придёт условие остановки.

**Важно:** всегда иметь условие выхода или обработку прерывания ( `break` , сигнал завершения, исключение), чтобы избежать неуправляемого зависания программы.



# Примеры

## Найти первое чётное число в массиве (break)

```
int[] arr = { 5, 7, 8, 9, 10 };
int firstEven = -1;
for (int i = 0; i < arr.Length; i++)
{
    if (arr[i] % 2 == 0)
    {
        firstEven = arr[i];
        break; // нашли – прерываем цикл
    }
}
Console.WriteLine(firstEven); // 8
```

## Пропустить отрицательные значения (continue)

```
int[] arr = { -1, 3, -2, 4, 0 };
int sumPos = 0;
foreach (var v in arr)
{
    if (v <= 0) continue; // пропускаем нули и отрицательные
    sumPos += v;
}
Console.WriteLine(sumPos); // 7 (3 + 4)
```

## Бесконечный цикл, консольное эхо

```
while (true)
{
    Console.Write("> ");
    var line = Console.ReadLine();
    if (line == null || line == "exit") break;
    Console.WriteLine("Echo: " + line);
}
```

## Когда какой цикл использовать

- **for** — когда заранее известно количество итераций или нужен индекс; хорош для массивов и арифметики индексов.
- **while** — когда количество итераций заранее неизвестно; цикл продолжается, пока условие истинно (подходит для чтения ввода, ожидания условий).
- **foreach** — когда нужно пройти по всем элементам коллекции без индекса; безопаснее и короче для перебора.

# Практическое задание

## Что нужно сделать:

1. Продолжайте работу в проекте **ToDoList**, созданном ранее.
2. Создайте массив строк `todos`, в котором будут храниться задачи.
3. Сделайте бесконечный цикл, в котором будет проверяться введённая пользователем команда.
4. Реализуйте следующие команды:
  - `help` — выводит список всех доступных команд с кратким описанием.
  - `profile` — выводит данные пользователя в формате: `<Имя> <Фамилия>, <Год рождения>`.
  - `add` — добавляет новую задачу. Формат ввода: `add "текст задачи"`  
Чтобы извлечь текст задачи из команды, используйте метод [String.Split](#).
  - `view` — выводит все задачи из массива (только непустые элементы).
  - `exit` — завершает цикл и останавливает выполнение программы.
5. Реализуйте расширение массива `todos`:
  - Установите `todos` небольшую начальную длину, например 2 элемента.
  - При добавлении новых задач, сделайте проверку, хватит ли места в массиве для новой задачи.
  - Если места не хватает, создайте новый массив, в 2 раза длиннее текущего, через цикл перезапишите в него содержимое текущего массива.
  - Запишите в переменную `todos` новый массив.
6. Делайте коммит после **каждого** изменения, если задание будет отправлено одним коммитом — задание будет оцениваться в два раза меньше.
7. После завершения реализации — протестируйте программу, добавив несколько задач и проверив все команды.
8. Внесите изменения в **README.md** — добавьте описание новых возможностей программы.
9. Сделайте push изменений.