# Основы программирования на C#

Язык C# (произносится «Си Шарп») и платформа.NET представляют собой мощный и универсальный инструментарий для разработки широкого спектра приложений. C# — это современный, объектно-ориентированный язык программирования общего назначения, разработанный компанией Microsoft. Его создание было обусловлено стремлением предложить простой, но при этом надежный язык, способный эффективно работать в распределенных средах. Основные разработчики языка — Андерс Хейлсберг, Скотт Уилтамут и Питер Голде. Первая общедоступная версия C# появилась в июле 2000 года как часть инициативы.NET Framework.

C# разработан с учетом принципов программной инженерии, что способствует созданию надежного и долговечного программного обеспечения. Эти принципы включают строгую проверку типов, проверку границ массивов, обнаружение попыток использования неинициализированных переменных и автоматическую сборку мусора. Такой подход значительно повышает продуктивность программистов, поскольку язык берет на себя многие рутинные задачи, которые в других языках (например, C или C++) требуют ручного контроля. Благодаря встроенным механизмам безопасности, C# помогает предотвращать распространенные ошибки на ранних стадиях разработки, что приводит к более стабильному и предсказуемому поведению приложений.

Язык C# предназначен для создания программных компонентов, которые легко развертываются в распределенных системах. Важными аспектами его дизайна являются портативность исходного кода и простота освоения для программистов, уже знакомых с C и C++. Кроме того, C# обеспечивает надежную поддержку интернационализации, что позволяет разрабатывать приложения для глобального рынка. Он подходит для широкого круга систем — от крупных, использующих сложные операционные системы, до компактных встроенных устройств с ограниченными функциями. Хотя C# приложения стремятся к экономичному использованию памяти и процессорной мощности, язык не ставит своей целью прямую конкуренцию по производительности с C или языком ассемблера.

Платформа.NET, на которой работают приложения C#, предоставляет необходимую среду выполнения (Common Language Runtime, CLR) и обширную библиотеку классов (Base Class Library, BCL). Эта платформа позволяет C# приложениям работать на различных операционных системах и устройствах, обеспечивая кросс-платформенную совместимость и широкий спектр возможностей для разработчиков.

Изучение C# является крайне актуальным, поскольку это универсальный язык, используемый в различных областях разработки. Он применяется для создания веб-приложений с использованием ASP.NET, облачных решений на базе Azure,.NET Aspire и Microsoft Orleans, настольных приложений с.NET Multi-platform App UI (MAUI), Windows Presentation Foundation (WPF), Windows Forms и Universal Windows Platform (UWP), а также мобильных приложений.

Одним из значительных преимуществ изучения C# является обширная и хорошо структурированная экосистема обучения, предоставляемая Microsoft. Платформа Microsoft Learn предлагает исчерпывающие ресурсы, включая видеокурсы для начинающих 3, интерактивные учебники 6 и подробную документацию.2 Эта интегрированная обучающая среда обеспечивает доступ к высококачественному контенту непосредственно от разработчиков языка, что гарантирует точность и согласованность информации. Многомодальный подход к обучению, включающий видео, интерактивные упражнения и текстовые руководства, удовлетворяет различные стили обучения и помогает новичкам эффективно осваивать материал.

Для начала работы с C# рекомендуется настроить среду разработки, включающую последнюю версию.NET SDK, редактор Visual Studio Code и расширение C# DevKit. Microsoft Learn предоставляет подробные инструкции по установке этих компонентов как для Windows (с использованием WinGet), так и для других операционных систем.6

## Глава 1: Структура Программы C#

Программы на C# обладают четко определенной структурой, которая облегчает их организацию и понимание. Каждая программа состоит из одного или нескольких файлов, и каждый файл может содержать ноль или более пространств имен. Пространства имен, в свою очередь, служат контейнерами для различных типов, таких как классы, структуры, интерфейсы, перечисления и делегаты, а также могут содержать вложенные пространства имен.9

### Общая структура программы

Традиционно, программы на C# начинаются с метода Main, который является точкой входа в приложение. Этот метод должен быть объявлен внутри класса или структуры и быть статическим. Ниже представлен классический пример структуры программы:

using System; // Директива для импорта пространства имен System  
  
namespace MyFirstProgram // Объявление пространства имен  
{  
 class Program // Определение класса  
 {  
 static void Main(string args) // Метод Main, точка входа программы  
 {  
 Console.WriteLine("Привет, мир!"); // Оператор для вывода текста в консоль  
 }  
 }  
}

В этом примере using System; — это директива, которая позволяет использовать типы из пространства имен System (например, Console) без указания полного имени. namespace MyFirstProgram определяет пространство имен для организации кода. class Program объявляет класс, а static void Main(string args) — это сигнатура метода Main. Параметр string args позволяет программе принимать аргументы командной строки.10

Начиная с C# 9, была введена концепция операторов верхнего уровня (Top-Level Statements), которая значительно упрощает структуру программы для небольших приложений и скриптов. Используя операторы верхнего уровня, разработчикам больше не нужно явно объявлять метод Main, класс или пространство имен. Точка входа программы в этом случае определяется первой строкой кода в файле. Только один файл в проекте может содержать операторы верхнего уровня.9

Ниже приведен пример программы с операторами верхнего уровня:

// Нет необходимости в пространстве имен, классе или методе Main  
Console.WriteLine("Привет, мир из операторов верхнего уровня!");  
  
// Операторы верхнего уровня могут получать доступ к аргументам командной строки  
if (args.Length > 0)  
{  
 Console.WriteLine($"Первый аргумент: {args}");  
}

Эта эволюция языка C# в сторону операторов верхнего уровня отражает общую тенденцию к сокращению шаблонного кода и повышению доступности для начинающих программистов. Для небольших консольных приложений или для быстрого прототипирования такой подход позволяет сосредоточиться непосредственно на логике программы, минуя избыточные структурные элементы. Однако для более крупных и сложных приложений традиционная структура с явными пространствами имен и классами остается предпочтительной, так как она обеспечивает лучшую организацию и масштабируемость кода. Понимание обеих структур является ключевым для эффективной работы с C# и.NET.

### Выражения и операторы

Программы на C# строятся из выражений и операторов.9 Понимание различий между ними является фундаментальным для написания корректного и эффективного кода.

**Выражения** — это комбинации значений, переменных, операторов и вызовов методов, которые вычисляются в одно значение.2 Выражения всегда производят результат и могут использоваться везде, где ожидается значение.

Примеры выражений:

* 42 (литеральное значение)
* x + y (арифметическая операция, вычисляющая сумму)
* Math.Max(a, b) (вызов метода, возвращающий максимальное из двух значений)
* condition? trueValue : falseValue (условное выражение, возвращающее одно из двух значений в зависимости от условия)
* new Person("John") (выражение создания объекта, возвращающее новый экземпляр класса Person)

**Операторы** — это полные инструкции, которые выполняют определенное действие.9 В отличие от выражений, операторы обычно не возвращают значений (хотя есть исключения, такие как оператор

return). Они контролируют поток выполнения программы, объявляют переменные или выполняют операции.

Примеры операторов:

* int x = 42; (оператор объявления и инициализации переменной)
* Console.WriteLine("Hello"); (оператор выражения, который оборачивает вызов метода Console.WriteLine для выполнения действия вывода)
* if (condition) { /\* code \*/ } (условный оператор, управляющий потоком выполнения)
* return result; (оператор возврата, который завершает выполнение метода и, возможно, возвращает значение)

Важно отметить, что некоторые конструкции, такие как вызовы методов, могут выступать как выражения (когда их возвращаемое значение используется) или как операторы выражения (когда они вызываются исключительно ради их побочного эффекта, например, вывода на консоль). Например, Math.Max(a, b) является выражением, если его результат присваивается переменной (int result = Math.Max(a, b);), но становится оператором выражения, если вызывается самостоятельно (Math.Max(a, b);) без использования возвращаемого значения.9

Понимание этого фундаментального различия между выражениями и операторами критически важно для эффективного программирования на C#. Это позволяет точно определить, где и как можно использовать различные части кода, а также как управлять логикой выполнения программы.

## Глава 2: Переменные, Константы и Литералы

В программировании переменные, константы и литералы являются основополагающими элементами для хранения и представления данных.

### Переменные: объявление, инициализация, использование

**Переменные** в C# представляют собой именованные области хранения данных. Каждая переменная имеет определенный тип, который определяет, какие значения могут быть в ней сохранены.11 C# является строго типизированным языком, что означает, что компилятор C# гарантирует, что значения, хранящиеся в переменных, всегда соответствуют их объявленному типу.11

**Объявление переменной** включает указание ее типа данных и имени. После объявления переменная может быть использована для хранения значений.

Примеры объявления переменных:

float temperature; // Объявление переменной типа float  
string name; // Объявление переменной типа string  
MyClass myClass; // Объявление переменной пользовательского типа MyClass

**Инициализация переменной** — это процесс присвоения ей начального значения. Переменная должна быть определенно присвоена до того, как ее значение будет получено.11 Неинициализированные переменные не могут быть использованы, что предотвращает ошибки, связанные с неопределенным поведением.

Примеры объявления и инициализации переменных:

char firstLetter = 'C'; // Объявление и инициализация переменной типа char  
int count = 100; // Объявление и инициализация переменной типа int  
string message = "Hello, C#!"; // Объявление и инициализация переменной типа string

После объявления переменной ее нельзя переобъявить с новым типом, а также нельзя присвоить ей значение, несовместимое с ее объявленным типом. Например, нельзя объявить переменную типа int, а затем присвоить ей логическое значение true. Компилятор C# использует информацию о типах для обеспечения безопасности всех операций. Если попытаться выполнить несовместимые операции, например, сложение int и bool, компилятор выдаст ошибку.12

int a = 5;  
int b = a + 2; // OK  
bool test = true;  
// int c = a + test; // Ошибка компиляции: Оператор '+' не может быть применен к операндам типов 'int' и 'bool'.

Ключевое слово var:

C# позволяет использовать ключевое слово var для неявной типизации локальных переменных. Это означает, что компилятор сам определяет тип переменной на основе значения, которым она инициализируется. Несмотря на неявное объявление, переменная все равно получает статический тип во время компиляции. Ключевое слово var может использоваться только для локальных переменных, но не для членов класса.12

Примеры использования var:

var limit = 3; // Компилятор определит тип как int  
var greeting = "Hello"; // Компилятор определит тип как string  
var numbers = new int { 1, 2, 3 }; // Компилятор определит тип как int

Использование var может улучшить читаемость кода, особенно когда тип очевиден из инициализатора, но важно понимать, что это не делает C# динамически типизированным языком; типизация происходит на этапе компиляции.

#### Область видимости переменных (Variable Scope)

Область видимости переменной определяет часть программы, где эта переменная доступна.13 В C# область видимости определяется на этапе компиляции и не зависит от стека вызовов функций. Программы C# организованы в классы, и область видимости переменных делится на три основные категории: область видимости на уровне класса, на уровне метода и на уровне блока.

1. **Область видимости на уровне класса (Class-Level Scope):**
   * **Определение:** Переменные и методы, объявленные непосредственно внутри класса, вне какого-либо конкретного метода или блока, имеют область видимости на уровне класса.
   * **Доступность:** Такие переменные и методы доступны всем методам внутри этого класса.
   * **Инкапсуляция:** Область видимости на уровне класса способствует инкапсуляции, ограничивая данные и поведение классом, что повышает безопасность данных.
   * **Наследование:** Члены класса могут наследоваться производными классами, что является фундаментальным аспектом объектно-ориентированного программирования.
   * **Модульность:** Этот тип области видимости облегчает модульный дизайн, позволяя группировать связанные функциональные возможности в рамках класса, что улучшает организацию и повторное использование кода.13

Пример:C#  
using System;  
  
class DNT  
{  
 // Переменная на уровне класса  
 int a = 10;  
  
 public void Display()  
 {  
 // Доступ к переменной на уровне класса  
 Console.WriteLine(a);  
 }  
}  
  
class Program  
{  
 static void Main()  
 {  
 DNT myObject = new DNT();  
 myObject.Display(); // Выведет: 10  
 }  
}

В этом примере переменная a объявлена внутри класса DNT, что делает ее доступной из любого метода внутри DNT, например, из метода Display().

1. **Область видимости на уровне метода (Method-Level Scope):**
   * **Определение:** Переменные, объявленные внутри метода, доступны только в пределах этого конкретного метода.
   * **Ограниченный доступ:** Эти переменные имеют ограниченную видимость и не могут быть доступны извне метода, в котором они объявлены.
   * **Локальные переменные:** Область видимости на уровне метода применяется в основном к локальным переменным, определенным внутри блока метода.
   * **Инкапсуляция:** Эта область видимости способствует инкапсуляции, ограничивая доступ к переменным определенными методами и предотвращая непреднамеренные изменения.
   * **Управление памятью:** Область видимости на уровне метода способствует эффективному управлению памятью, ограничивая время жизни переменных продолжительностью выполнения метода.13

Пример:C#  
using System;  
  
class DNT  
{  
 public void display()  
 {  
 // Переменная на уровне метода  
 int m = 47;  
 Console.WriteLine(m);  
 }  
  
 public void display1()  
 {  
 // Console.WriteLine(m); // Ошибка компиляции: 'm' недоступна здесь  
 }  
}  
Здесь переменная m объявлена внутри метода display(). Попытка доступа к m из display1() приведет к ошибке компиляции, так как m находится вне области видимости display1().

1. **Область видимости на уровне блока (Block-Level Scope):**
   * **Определение:** Область видимости на уровне блока относится к видимости и доступности переменных, объектов или методов внутри конкретного блока кода, такого как циклы, операторы if или блоки try-catch.
   * **Ограниченный доступ:** Переменные, объявленные внутри блока, доступны только в пределах этого конкретного блока.
   * **Инкапсуляция:** Эта область видимости помогает инкапсулировать переменные, предотвращая непреднамеренный доступ и изменение извне блока.
   * **Управление памятью:** Переменные с областью видимости на уровне блока автоматически освобождаются из памяти после завершения выполнения их блока.
   * **Вложенные блоки:** Вложенные блоки создают иерархическую область видимости, позволяя внутренним блокам получать доступ к переменным внешних блоков, но не наоборот.13

Пример:C#  
using System;  
  
class DNT  
{  
 public void display()  
 {  
 int i = 0; // Переменная на уровне метода  
 for (i = 0; i < 4; i++)  
 {  
 Console.WriteLine($"i: {i}");  
 }  
  
 for (int j = 0; j < 5; j++) // 'j' - переменная на уровне блока  
 {  
 Console.WriteLine($"j: {j}");  
 }  
 // Console.WriteLine(j); // Ошибка компиляции: 'j' недоступна здесь  
 }  
}  
В этом примере i — переменная на уровне метода, доступная во всем методе display(). Однако j объявлена внутри второго цикла for, что ограничивает ее доступность только этим циклом. Попытка доступа к j вне ее цикла for приведет к ошибке компиляции.

Понимание этих различных типов области видимости переменных является критически важным для написания эффективного, безошибочного и хорошо организованного кода на C#.

### Константы: объявление, использование, отличие от readonly

**Константы** в C# — это поля, значения которых устанавливаются во время компиляции и никогда не могут быть изменены.14 Они используются для предоставления осмысленных имен вместо числовых литералов ("магических чисел") для специальных значений. Константы могут быть числами, логическими значениями, строками или нулевыми ссылками.15

Для объявления константы используется ключевое слово const. Инициализатор локальной константы или константного поля должен быть константным выражением, которое может быть неявно преобразовано в целевой тип. Константное выражение — это выражение, которое может быть полностью вычислено во время компиляции. Следовательно, единственно возможными значениями для констант ссылочных типов являются строки и нулевая ссылка (null).15

Примеры объявления и использования констант:

public class ConstantsExample  
{  
 public const double Pi = 3.14159; // Константное поле  
 private const int SpeedOfLight = 300000; // км/сек  
  
 public static void Main()  
 {  
 const string ProductName = "Visual C#"; // Локальная константа  
 Console.WriteLine($"Название продукта: {ProductName}");  
  
 double radius = 5.3;  
 double area = Pi \* (radius \* radius); // Использование константы Pi  
 Console.WriteLine($"Площадь круга с радиусом {radius}: {area:F2}");  
  
 int distance = 149476000; // км до Солнца  
 int secsFromSun = distance / SpeedOfLight; // Использование константы SpeedOfLight  
 Console.WriteLine($"Время от Солнца до Земли: {secsFromSun} секунд");  
  
 // Интерполированные строки также могут быть константами, если все используемые выражения являются константными строками  
 const string Language = "C#";  
 const string Platform = ".NET";  
 const string FullProductName = $"{Platform} - Язык: {Language}";  
 Console.WriteLine(FullProductName);  
 }  
}

Модификатор static не допускается в объявлении константы, поскольку константы неявно статичны.15

#### Отличие от readonly

Ключевое слово const отличается от ключевого слова readonly.

* **const:**
  + Значение устанавливается во время компиляции.
  + Может быть инициализировано только при объявлении поля.
  + Является константой времени компиляции.
  + При использовании константы в другом коде (например, в другой сборке), ее значение "встраивается" в этот код. Если значение константы изменится в исходной сборке, другие сборки, использующие эту константу, должны быть перекомпилированы, чтобы увидеть изменения.15
* **readonly:**
  + Значение может быть установлено либо при объявлении поля, либо в конструкторе класса.
  + Является константой времени выполнения. Это означает, что его значение может быть вычислено во время выполнения программы, а не обязательно во время компиляции.
  + Поля readonly могут иметь разные значения в зависимости от используемого конструктора.
  + Пример: public static readonly uint l1 = (uint)DateTime.Now.Ticks;.15 Это позволяет инициализировать поле значением, которое неизвестно до запуска программы.

Выбор между const и readonly зависит от того, когда значение должно быть определено и требуется ли его изменение во время выполнения. Для значений, которые никогда не меняются и известны на этапе компиляции (например, математические константы), используется const. Для значений, которые устанавливаются один раз при создании объекта или загрузке приложения и не меняются впоследствии, но могут быть неизвестны во время компиляции, используется readonly.

### Литералы: типы и примеры

**Литерал** в C# — это фиксированное значение, используемое непосредственно в программе. Эти значения записываются непосредственно в код и могут быть использованы переменными. Литерал может представлять собой целое число, число с плавающей запятой, строку, символ, логическое значение или даже null.16

Основные типы литералов в C# включают:

Целочисленные литералы (Integer Literals):  
Представляют собой целые числа. Могут быть представлены в десятичной (по умолчанию), шестнадцатеричной (с префиксом 0x) или двоичной (с префиксом 0b) форме.16 Могут иметь суффиксы для указания типа:  
U или u для беззнаковых чисел (uint, ulong), L или l для длинных чисел (long, ulong). По умолчанию целочисленный литерал имеет тип int.16  
Примеры:  
int decimalInt = 101; // Десятичный литерал (int)  
int hexInt = 0xFace; // Шестнадцатеричный литерал (int, значение 64206)  
int binaryInt = 0b101; // Двоичный литерал (int, значение 5)  
uint unsignedInt = 304U; // Беззнаковый int  
long longValue = 1234567890123L; // long  
ulong unsignedLong = 9876543210UL; // ulong

Литералы с плавающей запятой (Floating-point Literals):  
Представляют числа с десятичной точкой. Могут быть в десятичной или экспоненциальной форме. По умолчанию имеют тип double. Для указания float используется суффикс F или f, для decimal — M или m.16  
Примеры:  
double dValue = 3.14159; // double (по умолчанию)  
float fValue = 1.23f; // float  
decimal mValue = 23.45m; // decimal  
double expValue = 3.12569E-5; // Экспоненциальная форма (double)

Символьные литералы (Character Literals):  
Представляют одиночный символ, заключенный в одинарные кавычки (' '). Могут быть представлены как обычные символы, Unicode-представления (\uxxxx) или управляющие последовательности (escape sequences).16  
Примеры:  
char charA = 'a'; // Обычный символ  
char unicodeChar = '\u0061'; // Unicode-представление символа 'a'  
char newLineChar = '\n'; // Управляющая последовательность (новая строка)  
char singleQuote = '\''; // Экранированная одинарная кавычка  
  
Таблица управляющих последовательностей:

|  |  |
| --- | --- |
| Управляющая последовательность | Значение |
| \' | Одинарная кавычка |
| \" | Двойная кавычка |
| \\ | Обратная косая черта |
| \0 | Нулевой символ |
| \a | Звонок/Сигнал |
| \b | Возврат на шаг |
| \f | Прогон страницы |
| \n | Новая строка |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |

1. Строковые литералы (String Literals):  
   Представляют последовательность символов, заключенную в двойные кавычки (" "). Строки являются ссылочным типом и неизменяемы.17

**Обычные строковые литералы:** Используют управляющие последовательности для специальных символов.  
string s1 = "Привет, мир!"; // Обычная строка  
string s2 = "Он сказал, \"Это последний шанс!\""; // Экранирование двойной кавычки  
string s3 = "Путь к файлу: C:\\Program Files\\"; // Экранирование обратной косой черты

**Дословные строковые литералы (Verbatim String Literals):** Начинаются с символа @ перед открывающей кавычкой. В таких строках управляющие последовательности (кроме "" для двойной кавычки) интерпретируются буквально. Часто используются для путей к файлам или регулярных выражений.18

string filename = @"c:\documents\files\u0066.txt"; // \u0066 интерпретируется буквально  
string quote = @"Её имя было ""Сара."""; // Две двойные кавычки для одной

**"Сырые" строковые литералы (Raw String Literals) (C# 11+):** Начинаются и заканчиваются как минимум тремя двойными кавычками ("""). Позволяют включать любой текст без экранирования, включая кавычки и новые строки. Отступ, заданный на последней строке, автоматически удаляется.17  
string xml = """  
 <element attr="content">  
 <body>  
 </body>  
 </element>  
 """;  
// Внутренние кавычки не требуют экранирования  
string json = """{ "name": "Alice", "age": 30 }""";

**Интерполированные "сырые" строковые литералы (C# 11+):** Объединяют возможности интерполяции строк ($) и "сырых" строковых литералов. Используют несколько символов $ и """. Количество открывающих/закрывающих фигурных скобок для интерполяции должно соответствовать количеству $.17

int X = 2;  
int Y = 3;  
string pointMessage = $$"""Точка {{{X}}, {{Y}}} находится на расстоянии {{Math.Sqrt(X \* X + Y \* Y)}} от начала координат.""";  
Console.WriteLine(pointMessage); // Выведет: Точка {2, 3} находится на расстоянии 3.605551275463989 от начала координат.

Логические литералы (Boolean Literals):  
Представляют логические значения true (истина) и false (ложь).  
Примеры:  
  
bool isTrue = true;  
bool isFalse = false;  
// bool invalid = 0; // Ошибка компиляции: неявное преобразование int в bool невозможно

Литерал null (Null Literal):  
Представляет нулевую ссылку, указывающую на отсутствие экземпляра объекта. Совместим со всеми ссылочными типами и обнуляемыми типами значений.20  
Пример:  
string emptyString = null;  
MyClass myObject = null;

Литералы являются фундаментальными строительными блоками в C#, позволяя напрямую представлять фиксированные значения в коде.

## Глава 3: Типы Данных

Система типов C# является унифицированной и строго типизированной. Это означает, что каждое значение любого типа может быть обработано как объект, и каждый тип в C# прямо или косвенно наследуется от базового класса object.12 Строгая типизация гарантирует, что переменные всегда содержат значения соответствующего типа, что повышает безопасность и надежность кода.

### Обзор системы типов C#

В C# существует две основные категории типов: **типы значений (value types)** и **ссылочные типы (reference types)**.21 Эти категории существенно различаются по способу хранения данных и поведению при присваивании.

### Типы значений

Переменные типов значений напрямую содержат свои данные.21 Когда переменная типа значения присваивается другой переменной, создается

**копия данных**. Это означает, что операции, выполняемые над одной переменной, не влияют на данные другой переменной, поскольку каждая из них имеет свою независимую копию.12 Все типы значений неявно наследуются от

System.ValueType, который, в свою очередь, наследуется от object. Типы значений неявно запечатаны (sealed), что означает, что от них нельзя наследовать другие типы.21

C# предоставляет набор предопределенных типов структур, известных как **простые типы**, которые идентифицируются ключевыми словами, являющимися псевдонимами для предопределенных типов struct в пространстве имен System.21

Основные виды типов значений:

* **Числовые типы (Numeric Types):**
  + **Целочисленные типы:** Представляют целые числа.
    - sbyte: знаковое 8-битное целое число (от -128 до 127)
    - byte: беззнаковое 8-битное целое число (от 0 до 255)
    - short: знаковое 16-битное целое число (от -32768 до 32767)
    - ushort: беззнаковое 16-битное целое число (от 0 до 65535)
    - int: знаковое 32-битное целое число (от -2,147,483,648 до 2,147,483,647)
    - uint: беззнаковое 32-битное целое число (от 0 до 4,294,967,295)
    - long: знаковое 64-битное целое число (от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807)
    - ulong: беззнаковое 64-битное целое число (от 0 до 18,446,744,073,709,551,615)
  + **Типы с плавающей запятой:** Представляют числа с десятичной точкой.
    - float: 32-битное число с одинарной точностью (приблизительно от 1.5 × 10⁻⁴⁵ до 3.4 × 10³⁸ с 7 знаками точности)
    - double: 64-битное число с двойной точностью (приблизительно от 5.0 × 10⁻³²⁴ до 1.7 × 10³⁰⁸ с 15-16 знаками точности)
  + **Тип decimal:** 128-битный тип данных, подходящий для финансовых расчетов, представляющий значения от -7.9 × 10⁻²⁸ до 7.9 × 10²⁸ с точностью не менее 28 знаков. В отличие от float и double, decimal может точно представлять дробные числа, такие как 0.1.21

Примеры:C#  
int age = 30;  
double price = 19.99;  
decimal salary = 50000.75m;

**Логический тип (bool):** Представляет логические величины со значениями true (истина) и false (ложь). Между bool и другими типами значений нет стандартных преобразований.21  
Пример:  
bool isActive = true;

**Символьный тип (char):** Представляет 16-битные беззнаковые целые числа со значениями от 0 до 65535, соответствующие набору символов Unicode. Хотя char имеет то же представление, что и ushort, не все операции, разрешенные для одного типа, разрешены для другого.21  
Пример:

char initial = 'A';

* **Типы структур (struct):** Тип значения, который может объявлять константы, поля, методы, свойства, события, индексаторы, операторы, конструкторы экземпляров, статические конструкторы и вложенные типы.21  
  Пример:

public struct Point  
{  
 public int X, Y;  
 public Point(int x, int y) { X = x; Y = y; }  
}  
Point p1 = new Point(10, 20);

* **Типы перечислений (enum):** Отдельный тип с именованными константами. Каждый тип перечисления имеет базовый целочисленный тип (например, byte, int), и его набор значений совпадает с набором значений базового типа.12  
  Пример:

public enum DayOfWeek { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }  
DayOfWeek today = DayOfWeek.Monday;

**Кортежные типы (Tuple Types):** Представляют собой упорядоченную, фиксированной длины последовательность значений с необязательными именами и индивидуальными типами. Например, (int number, string word) является кортежным типом. Этот синтаксис является сокращением для универсальных типов структур System.ValueTuple<...>.21  
Пример:

(string name, int age) person = ("Alice", 30);  
Console.WriteLine(person.name); // Выведет: Alice

### Ссылочные типы

Переменные ссылочных типов хранят **ссылки** на свои данные, которые известны как **объекты**.21 Это означает, что несколько переменных могут ссылаться на один и тот же объект, и операции, выполняемые над одной переменной, могут влиять на объект, на который ссылаются другие переменные.12 Специальное значение

null совместимо со всеми ссылочными типами и указывает на отсутствие экземпляра.21

Основные виды ссылочных типов:

* **Типы классов (class):** Определяют структуру данных, содержащую члены данных (константы, поля) и члены функций (методы, свойства, события, индексаторы, операторы, конструкторы, финализаторы). Типы классов поддерживают наследование, позволяя производным классам расширять и специализировать базовые классы. Экземпляры создаются с помощью выражений new.21
  + object: Конечный базовый класс для всех других типов в C#. Каждый тип прямо или косвенно наследуется от object. Ключевое слово object является псевдонимом для System.Object.21
  + string: Запечатанный тип класса, который наследуется непосредственно от object. Экземпляры представляют строки символов Unicode. Ключевое слово string является псевдонимом для System.String. Строки являются неизменяемыми, то есть после создания строкового объекта его значение не может быть изменено. Любая операция, которая, казалось бы, изменяет строку, на самом деле создает новый строковый объект.12

Примеры:C#  
class MyClass  
{  
 public int MyProperty { get; set; }  
}  
MyClass obj1 = new MyClass();  
obj1.MyProperty = 10;  
MyClass obj2 = obj1; // obj1 и obj2 ссылаются на один и тот же объект  
obj2.MyProperty = 20;  
Console.WriteLine(obj1.MyProperty); // Выведет: 20  
string greeting = "Hello";  
string anotherGreeting = greeting;  
greeting += " World"; // Создает новый строковый объект для greeting  
Console.WriteLine(anotherGreeting); // Выведет: Hello

* **Типы интерфейсов (interface):** Определяют контракт, который могут реализовать классы или структуры. Интерфейс может наследовать от нескольких базовых интерфейсов, а класс или структура могут реализовать несколько интерфейсов. Интерфейсы не содержат деталей реализации.22  
  Пример:

interface ILogger { void Log(string message); }  
class ConsoleLogger : ILogger { public void Log(string message) { Console.WriteLine(message); } }  
ILogger logger = new ConsoleLogger();  
logger.Log("Сообщение");

* **Типы массивов (Array Types):** Массив — это структура данных, содержащая ноль или более переменных (элементов) одного и того же типа, доступ к которым осуществляется по вычисляемым индексам.12  
  Пример:

int numbers = { 1, 2, 3 };

* **Типы делегатов (delegate):** Структура данных, которая ссылается на один или несколько методов. Для методов экземпляра она также ссылается на соответствующие им экземпляры объектов. Делегаты похожи на указатели на функции в C/C++, но могут ссылаться как на статические, так и на методы экземпляра.21

Пример:

public delegate void MyDelegate(string message);  
class Greeter { public void SayHello(string name) { Console.WriteLine($"Привет, {name}!"); } }  
Greeter greeter = new Greeter();  
MyDelegate del = new MyDelegate(greeter.SayHello);  
del("Мир"); // Вызовет метод SayHello

* **Динамический тип (dynamic):** Как и object, может ссылаться на любой объект. Разрешение операций над выражениями типа dynamic откладывается до выполнения программы (динамическое связывание). Таким образом, если операция не может быть законно применена к ссылаемому объекту, ошибка не выдается во время компиляции.21  
  Пример:

dynamic myDynamicVar = "Hello";  
Console.WriteLine(myDynamicVar.GetType()); // Выведет: System.String  
myDynamicVar = 123;  
Console.WriteLine(myDynamicVar.GetType()); // Выведет: System.Int32

* **Типы записей (record) (C# 9+):** Ссылочные типы, предоставляющие встроенные функции для инкапсуляции данных. Они особенно полезны для неизменяемых моделей данных и предлагают такие функции, как равенство на основе значений.22  
  Пример:

public record Person(string FirstName, string LastName);  
Person person1 = new Person("Джон", "Доу");  
Person person2 = new Person("Джон", "Доу");  
Console.WriteLine(person1 == person2); // Выведет: True (сравнение по значению)

### Различия в выделении памяти (Stack vs. Heap)

Понимание того, как C# управляет памятью, является ключевым для оптимизации производительности и предотвращения ошибок. Память в C# в основном управляется с использованием двух основных механизмов: стека (Stack) и кучи (Heap).24

Стек (Stack):

Стек — это область памяти, которая работает по принципу "последним пришел — первым вышел" (LIFO - Last-In, First-Out).24 Он используется для статического выделения памяти, что включает хранение локальных переменных, параметров методов и адресов возврата.24

* **Характеристики стека:**
  + **Быстрый доступ:** Выделение и освобождение памяти на стеке происходят очень быстро благодаря его строгому LIFO-порядку.
  + **Фиксированный размер:** Стек имеет ограниченный размер, и чрезмерное использование памяти может привести к переполнению стека (StackOverflowException).
  + **Автоматическое управление:** Память на стеке управляется средой выполнения автоматически. Когда вызов функции завершается, ее локальные переменные автоматически освобождаются.24
* **Выделение и освобождение памяти на стеке:** При вызове функции создается фрейм стека. Этот фрейм содержит все локальные переменные и параметры функции. По завершении работы функции ее фрейм стека уничтожается, и память освобождается.24

**Пример работы со стеком:**

using System;  
public class StackExample  
{  
 public static void Main()  
 {  
 int x = 10; // x выделяется на стеке  
 int y = 20; // y выделяется на стеке  
 int result = Add(x, y); // result выделяется на стеке  
 Console.WriteLine(result);  
 }  
 public static int Add(int a, int b) // a и b - параметры, выделяются на стеке  
 {  
 int sum = a + b; // sum - локальная переменная, выделяется на стеке  
 return sum;  
 }  
}  
  
В этом примере переменные x, y и result в методе Main выделяются на стеке. При вызове метода Add создается новый фрейм стека для его параметров a и b, а также для локальной переменной sum. После возврата из Add его фрейм стека уничтожается, и память освобождается.24

Куча (Heap):

Куча — это область памяти, используемая для динамического выделения памяти.24 В отличие от стека, память на куче не освобождается автоматически при возврате функции; вместо этого она требует явного управления со стороны сборщика мусора C#.24

* **Характеристики кучи:**
  + **Гибкий размер:** Куча может расти и уменьшаться по мере необходимости, что делает ее подходящей для объектов, время жизни которых не привязано к одному вызову функции.
  + **Более медленный доступ:** Доступ к памяти на куче обычно медленнее, чем к памяти на стеке, из-за накладных расходов на динамическое выделение и сборку мусора.
  + **Явное управление:** Объекты на куче управляются сборщиком мусора, который автоматически освобождает память, на которую больше нет ссылок.24
* **Выделение и освобождение памяти на куче:** Когда объект создается с использованием ключевого слова new, память для этого объекта выделяется на куче. Объект остается в памяти до тех пор, пока на него не перестанут ссылаться, после чего он будет собран сборщиком мусора.24

**Пример работы с кучей:**

using System;  
public class HeapExample  
{  
 public static void Main()  
 {  
 Person person1 = new Person("Alice"); // Экземпляр Person выделяется на куче  
 Person person2 = new Person("Bob"); // Экземпляр Person выделяется на куче  
 Console.WriteLine(person1.Name);  
 Console.WriteLine(person2.Name);  
 }  
}  
public class Person  
{  
 public string Name { get; private set; } // Свойство Name (string) выделяется на куче  
 public Person(string name)  
 {  
 Name = name;  
 }  
}  
  
В этом примере person1 и person2 являются ссылочными типами, и их экземпляры выделяются на куче. Свойство Name также выделяется на куче, поскольку оно является частью объекта Person. Сами ссылки person1 и person2 хранятся на стеке.24

В C# тип переменной определяет, будет ли ее память выделена на стеке или на куче 24:

* **Типы значений:** Хранятся на стеке. Примеры: int, float, bool и структуры (struct).
* **Ссылочные типы:** Хранятся на куче. Примеры: классы (class), массивы (array) и строки (string).

Понимание этих различий имеет решающее значение для эффективного управления памятью и оптимизации производительности в C# приложениях.

|  |  |  |
| --- | --- | --- |
| Аспект | Стек (Stack) | Куча (Heap) |
| **Выделение** | Статическое (порядок LIFO) | Динамическое |
| **Управление** | Автоматическое (управляется средой выполнения) | Автоматическое (управляется сборщиком мусора) |
| **Время жизни** | Привязано к вызовам функций | Привязано к ссылкам на объекты |
| **Скорость** | Быстрее | Медленнее |
| **Ограничение размера** | Фиксированное | Гибкое |
| **Примеры использования** | Примитивные типы данных, локальные переменные | Объекты, сложные структуры данных |

### Boxing и Unboxing

**Boxing (упаковка)** — это процесс неявного преобразования типа значения в ссылочный тип object или в любой тип интерфейса, который он реализует.20 При этом создается новый объект на куче, и значение типа значения копируется в этот новый объект.20

Пример упаковки:

int num = 123; // Тип значения, хранится на стеке  
object obj = num; // Упаковка: значение num копируется в новый объект на куче  
 // obj теперь ссылается на этот упакованный объект

Процесс упаковки включает в себя создание копии упаковываемого значения. Это отличается от преобразования ссылочного типа в тип object, при котором значение продолжает ссылаться на тот же экземпляр и просто рассматривается как менее производный тип object.20

**Unboxing (распаковка)** — это процесс явного преобразования ссылочного типа (который был создан в результате упаковки) обратно в тип значения.20 Распаковка включает два этапа:

1. Проверка того, что объект является упакованным значением целевого типа значения.
2. Копирование значения из объекта обратно в переменную типа значения.20

Пример распаковки:

object obj = 123; // Упаковка (как в предыдущем примере)  
int num = (int)obj; // Распаковка: значение из упакованного объекта копируется обратно в num

Если во время распаковки исходный операнд равен null или объект несовместим с целевым типом значения, будет выброшено исключение System.NullReferenceException или System.InvalidCastException соответственно.20

Пример, демонстрирующий копирование при упаковке:

struct Point { public int x, y; public Point(int x, int y) { this.x = x; this.y = y; } }  
class Program  
{  
 static void Main()  
 {  
 Point p = new Point(10, 10); // p - тип значения  
 object box = p; // Упаковка: значение p копируется в box  
 p.x = 20; // Изменяем оригинальную структуру p  
  
 // Распаковываем box и получаем значение x  
 Console.WriteLine(((Point)box).x); // Выведет: 10  
 }  
}

В этом примере box содержит копию p в момент упаковки. Изменение p.x после упаковки не влияет на значение, хранящееся в box. Если бы Point был классом, p и box ссылались бы на один и тот же экземпляр, и изменение p.x отразилось бы и на box.20

Производительность и использование:

Операции упаковки и распаковки могут влиять на производительность из-за накладных расходов, связанных с выделением памяти на куче и копированием данных. В современных версиях C# и.NET использование обобщений (generics) часто позволяет избежать упаковки и распаковки, обеспечивая типобезопасность и производительность. Однако упаковка все еще необходима в сценариях, где требуется обработка типов значений как объектов (например, при использовании старых коллекций, таких как ArrayList, или при работе с некоторыми API, которые ожидают object 25).

## Глава 4: Операции и Выражения

Операторы в C# позволяют выполнять различные действия над данными. Они являются фундаментальными строительными блоками выражений.

### Арифметические операции

Арифметические операторы в C# выполняют математические операции над числовыми операндами. Эти операторы поддерживаются всеми целочисленными типами и типами с плавающей запятой.26

1. **Унарные операторы:**
   * **++ (Оператор инкремента):** Увеличивает операнд на 1. Операнд должен быть переменной, доступом к свойству или индексатору.
     + **Постфиксный инкремент (x++):** Результатом является значение x *до* операции.

int i = 3;  
Console.WriteLine(i); // Выведет: 3  
Console.WriteLine(i++); // Выведет: 3 (i становится 4)  
Console.WriteLine(i); // Выведет: 4

* + - **Префиксный инкремент (++x):** Результатом является значение x *после* операции.

double a = 1.5;  
Console.WriteLine(a); // Выведет: 1.5  
Console.WriteLine(++a); // Выведет: 2.5 (a становится 2.5)  
Console.WriteLine(a); // Выведет: 2.5

* + **-- (Оператор декремента):** Уменьшает операнд на 1. Подобно инкременту, имеет постфиксную (x--) и префиксную (--x) формы.26
    - **Постфиксный декремент (x--):**

int i = 3;  
Console.WriteLine(i--); // Выведет: 3 (i становится 2)

* + - **Префиксный декремент (--x):**

double a = 1.5;  
Console.WriteLine(--a); // Выведет: 0.5 (a становится 0.5)

* + **+ (Унарный плюс):** Возвращает значение своего операнда.26

Console.WriteLine(+4); // Выведет: 4

* + **- (Унарный минус):** Вычисляет числовое отрицание своего операнда.26

Console.WriteLine(-4); // Выведет: -4  
Console.WriteLine(-(-4)); // Выведет: 4

1. **Бинарные операторы:**
   * **\* (Умножение):** Вычисляет произведение операндов.26

Console.WriteLine(5 \* 2); // Выведет: 10  
Console.WriteLine(0.5 \* 2.5); // Выведет: 1.25

* + **/ (Деление):** Делит левый операнд на правый.
    - **Целочисленное деление:** Для целочисленных типов результат является целым числом и равен частному, округленному в сторону нуля.26

Console.WriteLine(13 / 5); // Выведет: 2  
Console.WriteLine(-13 / 5); // Выведет: -2

* + - **Деление с плавающей запятой:** Для типов float, double и decimal результат является точным частным.26

Console.WriteLine(13 / 5.0); // Выведет: 2.6

* + **% (Остаток от деления):** Вычисляет остаток после деления левого операнда на правый.26

Console.WriteLine(5 % 4); // Выведет: 1  
Console.WriteLine(-5 % 4); // Выведет: -1

* + **+ (Сложение):** Вычисляет сумму операндов. Также используется для конкатенации строк и объединения делегатов.26  
      
    Console.WriteLine(5 + 4); // Выведет: 9  
    Console.WriteLine("Hello " + "World"); // Выведет: Hello World
  + **- (Вычитание):** Вычитает правый операнд из левого. Также используется для удаления делегатов.26

Console.WriteLine(47 - 3); // Выведет: 44

#### Приоритет и ассоциативность

**Приоритет операторов** определяет порядок выполнения операций в выражении. Операторы с более высоким приоритетом выполняются раньше, чем операторы с более низким приоритетом.2 Например, умножение и деление имеют более высокий приоритет, чем сложение и вычитание.

**Ассоциативность операторов** определяет порядок выполнения операций, когда несколько операторов имеют одинаковый приоритет. Большинство бинарных арифметических операторов в C# являются левоассоциативными, что означает, что они вычисляются слева направо. Операторы присваивания и условный оператор ?: являются правоассоциативными.2

**Порядок приоритета (от высшего к низшему):**

1. Постфиксные инкремент (x++) и декремент (x--) 2
2. Префиксные инкремент (++x) и декремент (--x), унарные + и - 2
3. Мультипликативные \*, /, и % 2
4. Аддитивные + и - 2

Пример, демонстрирующий приоритет и ассоциативность:

Console.WriteLine(2 + 2 \* 2); // Выведет: 6 (сначала 2 \* 2, затем +2)  
Console.WriteLine((2 + 2) \* 2); // Выведет: 8 (скобки меняют порядок)  
Console.WriteLine(9 / 5 / 2); // Выведет: 0 (левоассоциативность: (9/5) = 1, затем 1/2 = 0)  
Console.WriteLine(9 / (5 / 2)); // Выведет: 4 (скобки меняют порядок: 5/2 = 2, затем 9/2 = 4)

#### Переполнение целых чисел (checked/unchecked)

При выполнении целочисленных арифметических операций может произойти **переполнение**, если результат операции выходит за пределы диапазона значений, которые может хранить целевой тип данных. Поведение в случае переполнения контролируется контекстом проверки переполнения, который может быть checked (проверяемый) или unchecked (непроверяемый).26

* **Контекст checked:**
  + Если переполнение происходит в константном выражении, возникает ошибка компиляции.
  + В противном случае, когда операция выполняется во время выполнения, выбрасывается исключение OverflowException.26
* **Контекст unchecked:**
  + Результат усекается путем отбрасывания любых старших битов, которые не помещаются в целевой тип. Исключение не выбрасывается.26

По умолчанию арифметические операции в C# выполняются в unchecked контексте. Вы можете явно указать контекст с помощью операторов checked и unchecked или с помощью блоков checked и unchecked.

Пример:

int a = int.MaxValue; // 2147483647  
int b = 3;  
  
// В unchecked контексте (по умолчанию)  
Console.WriteLine(unchecked(a + b)); // Выведет: -2147483646 (переполнение, усечение)  
  
try  
{  
 // В checked контексте  
 int d = checked(a + b); // Выбросит OverflowException  
}  
catch (OverflowException)  
{  
 Console.WriteLine($"Произошло переполнение при сложении {a} и {b}.");  
}

Деление целого числа на ноль всегда приводит к выбрасыванию исключения DivideByZeroException.26

|  |  |  |
| --- | --- | --- |
| Оператор | Описание | Пример |
| + (унарный) | Возвращает значение операнда | +5 (результат: 5) |
| - (унарный) | Вычисляет числовое отрицание операнда | -5 (результат: -5) |
| ++ (инкремент) | Увеличивает операнд на 1 | i++ (постфикс), ++i (префикс) |
| -- (декремент) | Уменьшает операнд на 1 | i-- (постфикс), --i (префикс) |
| \* | Умножение | a \* b |
| / | Деление | a / b |
| % | Остаток от деления | a % b |
| + | Сложение | a + b |
| - | Вычитание | a - b |

### Поразрядные операции

Поразрядные операторы и операторы сдвига в C# выполняют операции над отдельными битами целочисленных типов. Эти операторы принимают операнды целочисленных числовых типов (таких как int, uint, long, ulong, nint, nuint) или типа char.28 Поразрядные операции и операции сдвига никогда не вызывают переполнения и дают одинаковые результаты как в

checked, так и в unchecked контекстах.28

1. **Поразрядный оператор дополнения (~):**
   * **Назначение:** Оператор ~ инвертирует каждый бит своего операнда (1 становится 0, а 0 становится 1).28
   * **Пример:**

uint a = 0b\_0000\_1111\_0000\_1111\_0000\_1111\_0000\_1100;  
uint b = ~a; // Инвертирует все биты  
Console.WriteLine(Convert.ToString(b, toBase: 2));  
// Выведет: 11110000111100001111000011110011

1. **Оператор левого сдвига (<<):**
   * **Назначение:** Оператор << сдвигает левый операнд влево на указанное количество битов. Старшие биты, выходящие за пределы диапазона результирующего типа, отбрасываются, а младшие освободившиеся битовые позиции заполняются нулями.28

**Пример:**  
uint x = 0b\_1100\_1001\_0000\_0000\_0000\_0000\_0001\_0001;  
Console.WriteLine($"До: {Convert.ToString(x, toBase: 2)}");  
uint y = x << 4; // Сдвиг влево на 4 бита  
Console.WriteLine($"После: {Convert.ToString(y, toBase: 2)}");  
// Выведет:  
// До: 11001001000000000000000000010001  
// После: 10010000000000000000000100010000

1. **Оператор правого сдвига (>>):**
   * **Назначение:** Оператор >> сдвигает левый операнд вправо на указанное количество битов. Младшие биты отбрасываются. Старшие освободившиеся битовые позиции заполняются в зависимости от типа левого операнда.28
   * **Поведение:**
     + **Арифметический сдвиг (для int или long):** Значение наиболее значимого бита (знакового бита) распространяется. Если операнд неотрицательный, заполняются нули; если отрицательный, заполняются единицы.

int a = -8; // Двоичное: 11111111111111111111111111111000  
Console.WriteLine($"До: {Convert.ToString(a, toBase: 2)}");  
int b = a >> 2; // Сдвиг вправо на 2 бита (арифметический сдвиг)  
Console.WriteLine($"После: {Convert.ToString(b, toBase: 2)}");  
// Выведет:  
// До: 11111111111111111111111111111000  
// После: 11111111111111111111111111111110 (-2)

* + - **Логический сдвиг (для uint или ulong):** Старшие освободившиеся битовые позиции всегда устанавливаются в ноль.28

uint c = 0b\_1000\_0000\_0000\_0000\_0000\_0000\_0000\_0000;  
Console.WriteLine($"До: {Convert.ToString(c, toBase: 2)}");  
uint d = c >> 3; // Сдвиг вправо на 3 бита (логический сдвиг)  
Console.WriteLine($"После: {Convert.ToString(d, toBase: 2).PadLeft(32, '0')}");  
// Выведет:  
// До: 10000000000000000000000000000000  
// После: 00010000000000000000000000000000

1. **Оператор беззнакового правого сдвига (>>>) (C# 11+):**
   * **Назначение:** Оператор >>> сдвигает левый операнд вправо на указанное количество битов.
   * **Поведение:** Он *всегда* выполняет логический сдвиг, что означает, что старшие освободившиеся битовые позиции всегда устанавливаются в ноль, независимо от типа левого операнда. Это отличает его от оператора >> для знаковых типов.28
   * **Пример:**

int x = -8; // Двоичное: 11111111111111111111111111111000  
Console.WriteLine($"До: {Convert.ToString(x, toBase: 2).PadLeft(32, '1')}");  
int z = x >>> 2; // Беззнаковый сдвиг вправо на 2 бита  
Console.WriteLine($"После >>>: {Convert.ToString(z, toBase: 2).PadLeft(32, '0')}");  
// Выведет:  
// До: 11111111111111111111111111111000  
// После >>>: 00111111111111111111111111111110 (1073741822)

1. **Побитовый логический оператор И (&):**
   * **Назначение:** Вычисляет побитовое логическое И своих целочисленных операндов.
   * **Поведение:** Для каждой битовой позиции, если оба соответствующих бита в операндах равны 1, результирующий бит равен 1; в противном случае — 0.28
   * **Пример:**

uint a = 0b\_1111\_1000; // 248  
uint b = 0b\_1001\_1101; // 157  
uint c = a & b; // 0b\_1001\_1000 (152)  
Console.WriteLine(Convert.ToString(c, toBase: 2)); // Выведет: 10011000

1. **Побитовый логический оператор исключающее ИЛИ (^):**
   * **Назначение:** Вычисляет побитовое логическое исключающее ИЛИ (XOR) своих целочисленных операндов.
   * **Поведение:** Для каждой битовой позиции, если соответствующие биты в операндах различны (один 0, другой 1), результирующий бит равен 1; в противном случае — 0.28
   * **Пример:**

uint a = 0b\_1111\_1000; // 248  
uint b = 0b\_0001\_1100; // 28  
uint c = a ^ b; // 0b\_1110\_0100 (228)  
Console.WriteLine(Convert.ToString(c, toBase: 2)); // Выведет: 11100100

1. **Побитовый логический оператор ИЛИ (|):**
   * **Назначение:** Вычисляет побитовое логическое ИЛИ своих целочисленных операндов.
   * **Поведение:** Для каждой битовой позиции, если хотя бы один из соответствующих битов в операндах равен 1, результирующий бит равен 1; в противном случае — 0.28

**Пример:**  
uint a = 0b\_1010\_0000; // 160  
uint b = 0b\_1001\_0001; // 145  
uint c = a | b; // 0b\_1011\_0001 (177)  
Console.WriteLine(Convert.ToString(c, toBase: 2)); // Выведет: 10110001

|  |  |  |
| --- | --- | --- |
| Оператор | Описание | Пример |
| ~ | Побитовое дополнение (NOT) | ~a |
| << | Левый сдвиг | a << 2 |
| >> | Правый сдвиг (арифметический) | a >> 2 |
| >>> | Беззнаковый правый сдвиг (логический, C# 11+) | a >>> 2 |
| & | Побитовое И (AND) | a & b |
| ^ | Побитовое исключающее ИЛИ (XOR) | a ^ b |
| ` | ` | Побитовое ИЛИ (OR) |

### Операции присваивания

Операторы присваивания используются для присвоения значения правого операнда переменной, свойству или элементу индексатора, заданному левым операндом. Результатом выражения присваивания является значение, присвоенное левому операнду.29

1. **Оператор простого присваивания (=):**
   * **Назначение:** Присваивает значение правого операнда левому. Тип правого операнда должен быть таким же, как тип левого, или неявно преобразуемым в него.29
   * **Ассоциативность:** Оператор = является правоассоциативным, то есть выражение вида a = b = c вычисляется как a = (b = c).29
   * **Поведение:** При присваивании операндов типов значений копируется содержимое правого операнда. При присваивании операндов ссылочных типов копируется ссылка на объект.29
   * **Пример:**

int x = 10; // Присваивает 10 переменной x  
int y = x; // Присваивает значение x переменной y  
string name = "Alice"; // Присваивает строковый литерал переменной name

1. **Присваивание по ссылке (= ref):**
   * **Назначение:** Делает левый операнд псевдонимом для правого операнда. Это означает, что обе переменные теперь ссылаются на одну и ту же область памяти.
   * **Пример:**

double arr = { 0.0, 0.0, 0.0 };  
Console.WriteLine(string.Join(" ", arr)); // Выведет: 0 0 0  
  
ref double arrayElement = ref arr; // arrayElement становится псевдонимом для arr  
arrayElement = 3.0; // Изменяет arr через псевдоним  
Console.WriteLine(string.Join(" ", arr)); // Выведет: 3 0 0  
  
arrayElement = ref arr[arr.Length - 1]; // arrayElement переназначается на arr  
arrayElement = 5.0; // Изменяет arr через новый псевдоним  
Console.WriteLine(string.Join(" ", arr)); // Выведет: 3 0 5

Левый операнд присваивания ref может быть локальной ссылочной переменной, полем ref или параметром метода ref, out или in. Оба операнда должны быть одного типа.29

1. **Составные операторы присваивания (op=):**
   * **Назначение:** Для бинарного оператора op выражение составного присваивания вида x op= y эквивалентно x = x op y, за исключением того, что x вычисляется только один раз.26
   * **Поддерживаемые операторы:** Арифметические, логические и побитовые операторы, а также операторы сдвига поддерживают составное присваивание.26

**Примеры:**  
int val = 7;  
  
val += 3; // Эквивалентно val = val + 3;  
Console.WriteLine("Addition Assignment: " + val); // Выведет: 10  
  
val -= 2; // Эквивалентно val = val - 2;  
Console.WriteLine("Subtraction Assignment: " + val); // Выведет: 8  
  
val \*= 7; // Эквивалентно val = val \* 7;  
Console.WriteLine("Multiplication Assignment: " + val); // Выведет: 56  
  
val /= 7; // Эквивалентно val = val / 7;  
Console.WriteLine("Division Assignment: " + val); // Выведет: 8  
  
val %= 3; // Эквивалентно val = val % 3;  
Console.WriteLine("Modulo Assignment: " + val); // Выведет: 2  
  
int flags = 0b\_0000\_0101; // 5  
flags |= 0b\_0000\_1010; // Эквивалентно flags = flags | 0b\_0000\_1010; (5 | 10 = 15)  
Console.WriteLine("Bitwise OR Assignment: " + Convert.ToString(flags, 2)); // Выведет: 1111  
  
int num = 1;  
num <<= 3; // Эквивалентно num = num << 3; (1 \* 2^3 = 8)  
Console.WriteLine("Left Shift Assignment: " + num); // Выведет: 8

Начиная с C# 14, пользовательские типы могут явно перегружать составные операторы присваивания для более эффективных реализаций, обычно когда значение может быть обновлено на месте, а не путем выделения нового экземпляра.30

1. **Присваивание с нулевым слиянием (??=):**
   * **Назначение:** Присваивает значение правого операнда левому операнду только в том случае, если левый операнд равен null.29
   * **Пример:**

string message = null;  
message??= "Default message"; // Присвоит "Default message", так как message == null  
Console.WriteLine(message); // Выведет: Default message  
  
message??= "Another message"; // Не изменит message, так как оно уже не null  
Console.WriteLine(message); // Выведет: Default message

|  |  |  |
| --- | --- | --- |
| Оператор | Описание | Пример |
| = | Простое присваивание | x = 10 |
| += | Присваивание сложением | x += 5 (эквивалентно x = x + 5) |
| -= | Присваивание вычитанием | x -= 3 (эквивалентно x = x - 3) |
| \*= | Присваивание умножением | x \*= 2 (эквивалентно x = x \* 2) |
| /= | Присваивание делением | x /= 4 (эквивалентно x = x / 4) |
| %= | Присваивание по модулю | x %= 3 (эквивалентно x = x % 3) |
| &= | Присваивание побитовым И | x &= y (эквивалентно x = x & y) |
| ` | =` | Присваивание побитовым ИЛИ |
| ^= | Присваивание побитовым XOR | x ^= y (эквивалентно x = x ^ y) |
| <<= | Присваивание левым сдвигом | x <<= 1 (эквивалентно x = x << 1) |
| >>= | Присваивание правым сдвигом | x >>= 1 (эквивалентно x = x >> 1) |
| >>>= | Присваивание беззнаковым правым сдвигом (C# 11+) | x >>>= 1 (эквивалентно x = x >>> 1) |
| ??= | Присваивание с нулевым слиянием | x??= y (присвоит y, если x равно null) |
| = ref | Присваивание по ссылке | ref int a = ref b |

### Преобразования базовых типов данных

Преобразования типов в C# позволяют изменять или рассматривать выражение как принадлежащее определенному типу, что может включать изменение его представления. Преобразования могут быть неявными или явными, что определяет, требуется ли явное приведение типов.20

#### Неявные и явные преобразования

1. **Неявные преобразования (Implicit Conversions):**
   * Происходят автоматически без необходимости явного приведения типов.
   * Всегда успешны и никогда не выбрасывают исключений.20
   * Примеры включают преобразования от меньших целочисленных типов к большим (например, int к long), от производных классов к базовым, а также упаковка (boxing) типов значений в object.20

Пример:C#  
int a = 123;  
long b = a; // Неявное преобразование из int в long  
Console.WriteLine(b); // Выведет: 123

1. **Явные преобразования (Explicit Conversions):**
   * Требуют явного приведения типов с использованием оператора ().
   * Могут не всегда быть успешными, могут приводить к потере информации или выбрасывать исключения.20
   * Примеры включают преобразования от больших целочисленных типов к меньшим (например, long к int), от базовых классов к производным, а также распаковка (unboxing) object в тип значения.20

Пример:C#  
long b = 1234567890123L;  
int c = (int)b; // Явное преобразование из long в int (может привести к потере данных)  
Console.WriteLine(c); // Выведет: -1061109533 (потеря данных)  
  
double d = 3.14;  
int i = (int)d; // Явное преобразование из double в int (потеря дробной части)  
Console.WriteLine(i); // Выведет: 3

|  |  |  |
| --- | --- | --- |
| Характеристика | Неявные преобразования | Явные преобразования |
| **Синтаксис** | Автоматически | Требуется оператор приведения () |
| **Безопасность** | Всегда безопасны, без потери данных | Могут привести к потере данных или исключениям |
| **Возможные ошибки** | Нет исключений | InvalidCastException, OverflowException, FormatException |
| **Примеры** | int -> long, byte -> int, DerivedClass -> BaseClass, int -> object (boxing) | long -> int, double -> int, object -> int (unboxing), BaseClass -> DerivedClass |

#### Класс System.Convert

Класс System.Convert предоставляет статические методы для различных преобразований между базовыми типами данных.NET. Он особенно полезен для преобразований, которые не являются неявными или требуют обработки потенциальных ошибок.31

Методы System.Convert могут приводить к следующим результатам:

* **Отсутствие преобразования:** Если тип источника и назначения совпадают, метод просто возвращает исходное значение.
* **InvalidCastException:** Выбрасывается, если конкретное преобразование не поддерживается (например, char в bool или DateTime в int).
* **FormatException:** Выбрасывается, если строковое значение не может быть преобразовано в целевой тип, потому что оно имеет неправильный формат (например, "abc" в int).
* **Успешное преобразование:** Для поддерживаемых преобразований, которые не приводят к потере данных.
* **OverflowException:** Выбрасывается, если сужающее преобразование приводит к потере данных из-за выхода значения за пределы диапазона целевого типа (например, 10000 в byte).31

Примеры использования System.Convert:

using System;  
  
public class ConvertExample  
{  
 public static void Main()  
 {  
 string stringNumber = "123";  
 try  
 {  
 int intNumber = Convert.ToInt32(stringNumber); // Преобразование строки в int  
 Console.WriteLine($"Преобразовано '{stringNumber}' в int: {intNumber}");  
 }  
 catch (FormatException)  
 {  
 Console.WriteLine($"Ошибка формата при преобразовании '{stringNumber}' в int.");  
 }  
 catch (OverflowException)  
 {  
 Console.WriteLine($"Переполнение при преобразовании '{stringNumber}' в int.");  
 }  
  
 double dValue = 23.15;  
 bool bValue = Convert.ToBoolean(dValue); // Преобразование double в bool  
 Console.WriteLine($"Преобразовано double в bool: {bValue}"); // Выведет: True (любое ненулевое число)  
  
 int largeInt = 200;  
 try  
 {  
 byte bByte = Convert.ToByte(largeInt); // Преобразование int в byte  
 Console.WriteLine($"Преобразовано int в byte: {bByte}");  
 }  
 catch (OverflowException)  
 {  
 Console.WriteLine($"Переполнение при преобразовании {largeInt} в byte.");  
 }  
  
 // Преобразование в недесятичные системы счисления  
 int value = 32767;  
 string binary = Convert.ToString(value, 2); // В двоичную систему  
 string hex = Convert.ToString(value, 16); // В шестнадцатеричную систему  
 Console.WriteLine($"{value} в двоичной: {binary}");  
 Console.WriteLine($"{value} в шестнадцатеричной: {hex}");  
 }  
}

Класс Convert также поддерживает преобразования между пользовательскими объектами и базовыми типами, если пользовательский тип реализует интерфейс IConvertible. Кроме того, он предоставляет методы для кодирования Base64.31

## Глава 5: Управление Потоком Выполнения

Управление потоком выполнения позволяет программе принимать решения и выполнять различные действия в зависимости от определенных условий. Это фундаментальный аспект программирования, имитирующий процесс принятия решений в реальной жизни.

### Условные выражения

Условные выражения, или условные логические конструкции, позволяют программе изменять путь выполнения кода на основе проверки переменных или условий. Они являются основой для создания адаптивного и интерактивного программного обеспечения.32

### Конструкция if..else и тернарная операция

1. Конструкция if:  
   Оператор if позволяет выполнить блок кода, если заданное логическое выражение истинно. Если условие истинно, код внутри блока if выполняется; в противном случае он пропускается.32  
   Пример:

int score = 85;  
if (score >= 60)  
{  
 Console.WriteLine("Поздравляем! Вы прошли.");  
}

1. Конструкция if..else:  
   Для выполнения различных блоков кода в зависимости от того, истинно или ложно условие, используется конструкция if..else. Блок else выполняется только тогда, когда условие в if ложно.34  
   Пример:

int age = 17;  
if (age >= 18)  
{  
 Console.WriteLine("Вы совершеннолетний.");  
}  
else  
{  
 Console.WriteLine("Вы несовершеннолетний.");  
}

1. Конструкция if..else if..else: Для проверки нескольких условий последовательно используется цепочка else if. Каждое else if проверяется только в том случае, если предыдущие условия были ложными. Если ни одно из условий if или else if не истинно, выполняется блок else (если он присутствует).34  
   Пример:

int temperature = 25;  
if (temperature < 0)  
{  
 Console.WriteLine("Очень холодно!");  
}  
else if (temperature >= 0 && temperature < 20)  
{  
 Console.WriteLine("Прохладно.");  
}  
else if (temperature >= 20 && temperature < 30)  
{  
 Console.WriteLine("Приятная температура.");  
}  
else  
{  
 Console.WriteLine("Жарко!");  
}

1. Вложенные if-else:  
   Операторы if..else могут быть вложены друг в друга, создавая более сложные структуры принятия решений. Это позволяет проверять условия внутри других условий.34  
   Пример:

bool hasLicense = true;  
int driverAge = 20;  
  
if (hasLicense)  
{  
 if (driverAge >= 18)  
 {  
 Console.WriteLine("Водитель может управлять автомобилем.");  
 }  
 else  
 {  
 Console.WriteLine("Водитель имеет права, но слишком молод.");  
 }  
}  
else  
{  
 Console.WriteLine("У водителя нет прав.");  
}

1. Тернарная операция (?:):  
   Тернарный оператор ?: является сокращенной формой оператора if..else для простых условных выражений, которые возвращают одно из двух значений. Он состоит из трех операндов: условия, выражения, выполняемого при истинном условии, и выражения, выполняемого при ложном условии.34  
   Синтаксис: результат = (условие)? выражение\_если\_истина : выражение\_если\_ложь;  
   Пример:

int age = 20;  
string status = (age >= 18)? "Совершеннолетний" : "Несовершеннолетний";  
Console.WriteLine($"Статус: {status}"); // Выведет: Статус: Совершеннолетний  
  
Тернарный оператор обеспечивает более компактный и читаемый код для простых условных присваиваний.34

### Циклы

Циклы в C# используются для повторного выполнения одного или нескольких операторов. Они являются неотъемлемой частью любого программирования, позволяя автоматизировать повторяющиеся задачи.35

1. **Цикл while:**
   * **Назначение:** Оператор while проверяет условие *перед* каждым выполнением блока кода. Он продолжает выполнять блок до тех пор, пока условие остается истинным. Если условие изначально ложно, блок кода не будет выполнен ни разу.35
   * **Синтаксис:**

int counter = 0;  
while (counter < 5)  
{  
 Console.WriteLine($"Счетчик: {counter}");  
 counter++; // Инкремент счетчика, чтобы избежать бесконечного цикла  
}

* + **Важное замечание:** Необходимо убедиться, что условие while цикла в конечном итоге станет ложным, иначе программа попадет в бесконечный цикл.35

1. **Цикл do-while:**
   * **Назначение:** Цикл do-while выполняет блок кода *один раз*, а затем проверяет условие. Это гарантирует, что блок кода будет выполнен как минимум один раз, независимо от того, истинно ли условие изначально.35
   * **Синтаксис:**

int counter = 0;  
do  
{  
 Console.WriteLine($"Счетчик: {counter}");  
 counter++;  
} while (counter < 5);

1. **Цикл for:**
   * **Назначение:** Цикл for часто используется, когда известно количество итераций. Он состоит из трех частей: инициализатора (объявление и инициализация переменной цикла), условия (условие продолжения цикла) и итератора (изменение переменной цикла после каждой итерации).35
   * **Синтаксис:**

for (int i = 0; i < 5; i++)  
{  
 Console.WriteLine($"Итерация for: {i}");  
}

* + **Части оператора for:**
    - **Инициализатор:** int i = 0; — объявляет и инициализирует переменную i.
    - **Условие:** i < 5; — цикл выполняется, пока i меньше 5.
    - **Итератор:** i++; — i увеличивается на 1 после каждой итерации.

1. **Цикл foreach:**
   * **Назначение:** Цикл foreach предназначен для итерации по элементам коллекций, таких как массивы, списки или другие объекты, реализующие интерфейс IEnumerable.35 Он автоматически перебирает каждый элемент в коллекции по очереди, не требуя использования индекса или счетчика.37
   * **Синтаксис:**

string names = { "Alice", "Bob", "Charlie" };  
foreach (string name in names)  
{  
 Console.WriteLine($"Привет, {name}!");  
}

* + **Использование с различными коллекциями:**
    - **Массивы:**

int numbers = { 1, 2, 3, 4, 5 };  
foreach (int num in numbers)  
{  
 Console.WriteLine(num);  
}

* + - **Списки:**

List<string> fruits = new List<string> { "Яблоко", "Банан", "Апельсин" };  
foreach (string fruit in fruits)  
{  
 Console.WriteLine(fruit);  
}

* + - **Словари:** Для словарей foreach итерирует по парам "ключ-значение" с использованием типа KeyValuePair.37

Dictionary<string, int> ages = new Dictionary<string, int>  
{  
 { "Алекс", 25 },  
 { "Ханна", 20 }  
};  
foreach (KeyValuePair<string, int> pair in ages)  
{  
 Console.WriteLine($"{pair.Key} возраст: {pair.Value}");  
}

* + **Ограничения foreach:**
    - Не отслеживает индекс элемента.37
    - Не может итерировать в обратном порядке.37
    - Не подходит для изменения коллекции во время итерации (может привести к ошибкам).37

#### Вложенные циклы

Циклы могут быть вложены друг в друга, что позволяет обрабатывать данные в многомерных структурах, таких как матрицы или таблицы. Вложенные циклы часто используются для перебора всех комбинаций элементов из нескольких наборов.35

Пример вложенных циклов:

for (int row = 1; row <= 3; row++) // Внешний цикл для строк  
{  
 for (int col = 1; col <= 3; col++) // Внутренний цикл для столбцов  
 {  
 Console.Write($"({row},{col}) ");  
 }  
 Console.WriteLine(); // Переход на новую строку после каждой строки  
}  
/\* Выведет:  
(1,1) (1,2) (1,3)  
(2,1) (2,2) (2,3)  
(3,1) (3,2) (3,3)  
\*/

В этом примере внешний цикл выполняется три раза (для каждой строки), и для каждой итерации внешнего цикла внутренний цикл также выполняется три раза (для каждого столбца).

|  |  |  |
| --- | --- | --- |
| Тип цикла | Когда использовать | Особенности |
| while | Когда количество итераций неизвестно, и цикл должен продолжаться, пока условие истинно. | Условие проверяется перед выполнением. Может не выполниться ни разу. |
| do-while | Когда количество итераций неизвестно, но блок кода должен быть выполнен как минимум один раз. | Блок кода выполняется один раз, затем проверяется условие. |
| for | Когда количество итераций известно или может быть легко определено. | Инициализатор, условие и итератор в одной строке. |
| foreach | Для итерации по всем элементам коллекции (массивы, списки) без необходимости работы с индексами. | Простой синтаксис, не требует счетчика. Не отслеживает индекс, не позволяет изменять коллекцию во время итерации. |

## Глава 6: Массивы

Массивы в C# — это структуры данных, предназначенные для хранения нескольких переменных одного и того же типа. Они представляют собой фиксированную по длине упорядоченную коллекцию значений или объектов одного типа.39 Массивы являются ссылочными типами, что означает, что переменная массива хранит ссылку на данные, которые фактически находятся в памяти.39

### Объявление, инициализация и доступ к элементам

**Объявление массива** включает указание типа его элементов, за которым следуют квадратные скобки (``). Если требуется, чтобы массив хранил элементы любого типа, можно указать object в качестве его типа, поскольку все типы C# наследуются от Object.39

Примеры объявления:

int numbers; // Объявление одномерного массива целых чисел  
string[,] names; // Объявление двумерного массива строк  
double jaggedArray; // Объявление зубчатого массива чисел с плавающей запятой

Массивы являются ссылочными типами, поэтому переменная массива может быть обнуляемым ссылочным типом (type? arrayName;), а элементы массива могут быть обнуляемыми типами (type? arrayName;).39

**Инициализация массива** происходит с использованием оператора new, за которым следуют тип элемента и размерности. Элементы, которые не инициализированы явно, устанавливаются в значение по умолчанию для их типа (например, 0 для целых чисел, null для строк).39 Начиная с C# 12, все типы коллекций могут быть инициализированы с помощью выражений коллекций (Collection expressions), используя квадратные скобки `` для более лаконичного синтаксиса.39

Примеры инициализации:

int array1 = new int; // Объявляет неинициализированный массив из 5 целых чисел (все элементы равны 0)  
string weekDays = ["Пн", "Вт", "Ср", "Чт", "Пт", "Сб", "Вс"]; // Объявляет и инициализирует 7 строк

**Доступ к элементам массива** осуществляется с использованием их индекса в квадратных скобках. Массивы в C# являются **нулевыми индексами**, то есть первый элемент массива имеет индекс 0, второй — 1 и так далее, до n-1 для массива из n элементов.39

Примеры доступа к элементам:

string firstDay = weekDays; // Доступ к первому элементу (Пн)  
Console.WriteLine(firstDay);  
  
weekDays = "Воскресенье"; // Изменение значения первого элемента  
Console.WriteLine(weekDays);

### Одномерные, многомерные и зубчатые массивы

C# поддерживает три основных типа массивов:

1. **Одномерные массивы (Single-dimensional arrays):**
   * Простейший тип массива, представляющий собой последовательность однотипных элементов.39
   * **Пример:**

int scores = new int { 90, 85, 92, 78, 95 };  
Console.WriteLine($"Второй балл: {scores}"); // Выведет: 85

1. **Многомерные массивы (Multidimensional arrays):**
   * Массивы, имеющие более одной размерности, например, двумерные (матрицы) или трехмерные.
   * **Объявление и инициализация:** Размеры каждой размерности указываются при создании экземпляра массива и не могут быть изменены в течение его жизни.39

**Пример двумерного массива:**  
int[,] matrix = new int; // Матрица 2 строки, 3 столбца  
matrix = 1;  
matrix = 2;  
matrix = 3;  
matrix = 4;  
matrix = 5;  
matrix = 6;  
  
// Инициализация при объявлении:  
int[,] matrix2 = { { 1, 2, 3 }, { 4, 5, 6 } };  
Console.WriteLine($"Элемент : {matrix2}"); // Выведет: 5

1. **Зубчатые массивы (Jagged arrays):**
   * Массив массивов, где каждый элемент является массивом, и эти внутренние массивы могут иметь разную длину.39
   * **Объявление и инициализация:** Элементы зубчатого массива являются ссылочными типами и по умолчанию инициализируются как null. Каждый внутренний массив должен быть инициализирован отдельно перед использованием.39
   * **Пример:**

int jaggedArray = new int; // Объявляем зубчатый массив из 3 массивов  
jaggedArray = new int { 1, 2, 3 }; // Первый внутренний массив  
jaggedArray = new int { 4, 5 }; // Второй внутренний массив  
jaggedArray = new int { 6, 7, 8, 9 }; // Третий внутренний массив  
  
Console.WriteLine($"Элемент : {jaggedArray}"); // Выведет: 3  
Console.WriteLine($"Длина второго внутреннего массива: {jaggedArray.Length}"); // Выведет: 2

### Свойства массивов (Length, Rank)

Все массивы в C# наследуются от абстрактного базового типа System.Array, который предоставляет ряд полезных свойств и методов.39

* **Length:** Возвращает общее количество элементов во всех измерениях массива.39

int singleDim = { 1, 2, 3, 4, 5 };  
Console.WriteLine($"Длина одномерного массива: {singleDim.Length}"); // Выведет: 5  
  
int[,] multiDim = { { 1, 2 }, { 3, 4 }, { 5, 6 } };  
Console.WriteLine($"Длина многомерного массива: {multiDim.Length}"); // Выведет: 6 (2\*3)

**Rank:** Возвращает количество измерений в массиве.39  
C#  
int singleDim = { 1, 2, 3 };  
Console.WriteLine($"Ранг одномерного массива: {singleDim.Rank}"); // Выведет: 1  
  
int[,] multiDim = { { 1, 2 }, { 3, 4 } };  
Console.WriteLine($"Ранг двумерного массива: {multiDim.Rank}"); // Выведет: 2

### Задачи с массивами (Common Array Operations)

Класс System.Array предоставляет статические методы для выполнения общих операций с массивами.

1. **Sort():** Сортирует элементы в одномерном массиве.

int numbers = { 34, 72, 13, 44, 25 };  
Console.Write("Исходный массив: ");  
foreach (int num in numbers) { Console.Write(num + " "); }  
Console.WriteLine();  
  
Array.Sort(numbers); // Сортировка массива  
Console.Write("Отсортированный массив: ");  
foreach (int num in numbers) { Console.Write(num + " "); }  
Console.WriteLine();  
// Выведет: 13 25 34 44 72

1. **Reverse():** Изменяет порядок следования элементов на противоположный в массиве или его части.43

int numbers = { 10, 20, 30, 40, 50 };  
Console.Write("Исходный массив: ");  
foreach (int num in numbers) { Console.Write(num + " "); }  
Console.WriteLine();  
  
Array.Reverse(numbers); // Изменение порядка на обратный  
Console.Write("Перевернутый массив: ");  
foreach (int num in numbers) { Console.Write(num + " "); }  
Console.WriteLine();  
// Выведет: 50 40 30 20 10

1. **Copy() и CopyTo():**
   * **Array.Copy(sourceArray, sourceIndex, destinationArray, destinationIndex, length):** Копирует диапазон элементов из одного массива в другой. Может выполнять приведение типов и упаковку.43
   * **sourceArray.CopyTo(destinationArray, index):** Копирует все элементы из текущего одномерного массива в указанный одномерный массив, начиная с указанного индекса.

int source = { 1, 2, 3, 4, 5 };  
int destination1 = new int;  
int destination2 = new int;  
  
source.CopyTo(destination1, 0); // Копирует все элементы  
Console.Write("CopyTo(): ");  
foreach (int num in destination1) { Console.Write(num + " "); }  
Console.WriteLine(); // Выведет: 1 2 3 4 5  
  
Array.Copy(source, 1, destination2, 2, 3); // Копирует 3 элемента, начиная с индекса 1, в destination2 начиная с индекса 2  
Console.Write("Array.Copy(): ");  
foreach (int num in destination2) { Console.Write(num + " "); }  
Console.WriteLine(); // Выведет: 0 0 2 3 4

**Clear():** Устанавливает диапазон элементов в массиве в ноль, false или null в зависимости от типа элемента.43

int data = { 10, 20, 30, 40, 50 };  
Console.Write("Исходный массив: ");  
foreach (int num in data) { Console.Write(num + " "); }  
Console.WriteLine();  
  
Array.Clear(data, 1, 3); // Очищает 3 элемента, начиная с индекса 1  
Console.Write("Массив после Clear(): ");  
foreach (int num in data) { Console.Write(num + " "); }  
Console.WriteLine();  
// Выведет: 10 0 0 0 50

Эти методы предоставляют базовые, но мощные средства для манипулирования данными в массивах C#.

## Глава 7: Методы

Методы в C# представляют собой именованные блоки кода, которые выполняют определенную задачу.44 Они являются ключевым элементом модульного программирования, позволяя организовывать код, повышать его читаемость, повторное использование и упрощать отладку. Каждая инструкция, выполняемая в C#, происходит в контексте какого-либо метода.45

### Определение, синтаксис и вызов методов

**Определение метода** включает несколько компонентов, которые вместе образуют сигнатуру метода:

* **Уровень доступа (Access Level):** Необязательный модификатор, определяющий видимость метода (например, public, private, protected, internal). По умолчанию — private.45
* **Модификаторы (Modifiers):** Необязательные ключевые слова, такие как abstract, sealed, virtual, static, async.45
* **Возвращаемое значение (Return Value):** Тип значения, которое метод возвращает. Если метод не возвращает никаких значений, используется ключевое слово void.45
* **Имя метода (Method Name):** Идентификатор метода.
* **Параметры метода (Method Parameters):** Список параметров, заключенный в круглые скобки и разделенный запятыми. Пустые скобки () означают, что метод не принимает параметров.45

Важно отметить, что возвращаемый тип метода не является частью его сигнатуры для целей перегрузки методов. Однако он является частью сигнатуры при определении совместимости между делегатом и методом, на который он указывает.45

Пример определения класса с методами:

namespace CarExample  
{  
 class Car  
 {  
 // Public метод, доступный из любого места  
 public void StartEngine()  
 {  
 Console.WriteLine("Двигатель запущен.");  
 }  
  
 // Protected метод, доступный только из производных классов  
 protected void AddFuel(int liters)  
 {  
 Console.WriteLine($"Добавлено {liters} литров топлива.");  
 }  
  
 // Virtual метод, который может быть переопределен в производных классах  
 public virtual int Drive(int miles, int speed)  
 {  
 Console.WriteLine($"Машина проехала {miles} миль со скоростью {speed} миль/ч.");  
 return miles;  
 }  
  
 // Перегруженный метод Drive с разными параметрами  
 public virtual int Drive(TimeSpan time, int speed)  
 {  
 Console.WriteLine($"Машина проехала {time.TotalHours} часов со скоростью {speed} миль/ч.");  
 return (int)(time.TotalHours \* speed);  
 }  
 }  
}

#### Статические vs. Методы экземпляра

Методы в C# могут быть либо **методами экземпляра**, либо **статическими методами**.

* **Методы экземпляра:**
  + Работают с данными конкретного экземпляра объекта.
  + Для вызова метода экземпляра необходимо сначала создать объект (инстанцировать класс).45

**Пример вызова:**

class TestCar : Car  
{  
 static void Main()  
 {  
 var myCar = new TestCar(); // Создание экземпляра объекта  
 myCar.StartEngine(); // Вызов метода экземпляра  
 myCar.AddFuel(20); // Вызов метода экземпляра  
 int distance = myCar.Drive(100, 60); // Вызов метода экземпляра  
 Console.WriteLine($"Пройденное расстояние: {distance} миль.");  
 }  
}

* **Статические методы:**
  + Не работают с данными экземпляра объекта. Они принадлежат классу, а не конкретному объекту.
  + Вызываются путем обращения к имени типа (класса), которому принадлежит метод. Попытка вызвать статический метод через экземпляр объекта приведет к ошибке компиляции.45
  + **Пример вызова:**

public static class Calculator  
{  
 public static int Square(int number) // Статический метод  
 {  
 return number \* number;  
 }  
}  
  
public class Program  
{  
 public static void Main()  
 {  
 int result = Calculator.Square(5); // Вызов статического метода через имя класса  
 Console.WriteLine($"Квадрат 5: {result}"); // Выведет: 25  
 }  
}

Базовый синтаксис вызова методов:

Вызов метода аналогичен доступу к полю. Используется имя объекта (для методов экземпляра) или имя типа (для статических методов), за которым следует точка (.), имя метода и круглые скобки ().45 Любые аргументы, требуемые методом, перечисляются внутри этих скобок, разделенные запятыми.

#### Типы аргументов при вызове методов

* **Позиционные аргументы:** Наиболее распространенный способ, когда аргументы передаются в том же порядке, в каком параметры определены в сигнатуре метода.44

myCar.Drive(150, 70); // 150 - мили, 70 - скорость

* **Именованные аргументы:** Позволяют указывать аргумент для параметра по его имени, а не по его позиции. Это повышает читаемость кода, особенно для методов с большим количеством параметров или с параметрами одного типа. Именованные аргументы указываются как имя\_параметра: значение.44 Порядок именованных аргументов не имеет значения, если все обязательные аргументы присутствуют.

myCar.Drive(speed: 70, miles: 150); // Порядок изменен, но понятно

* **Смешанные позиционные и именованные аргументы:** Можно использовать оба типа, но позиционные аргументы должны предшествовать всем именованным аргументам.44

myCar.Drive(150, speed: 70); // 150 - позиционный, speed: 70 - именованный

### Параметры методов

В C# параметры методов определяют, как аргументы передаются в метод. По умолчанию аргументы передаются **по значению**, что означает, что в метод передается копия переменной. Однако модификаторы параметров позволяют передавать аргументы **по ссылке**.46

#### Передача параметров по значению и по ссылке

1. **Передача по значению (Pass by Value):**
   * **Типы значений (struct):** При передаче типа значения по значению в метод передается *копия значения*. Метод оперирует этой копией и не может изменить исходную структуру в вызывающем методе. Любые изменения, внесенные в метод, касаются только этой копии.46
   * **Ссылочные типы (class):** При передаче ссылочного типа по значению в метод передается *копия ссылки* на экземпляр объекта. Исходная переменная и параметр метода будут ссылаться на один и тот же объект в памяти. Хотя метод не может переназначить экземпляр в вызывающем методе (то есть, не может заставить исходную переменную ссылаться на новый объект), он может использовать копию ссылки для доступа и изменения членов этого объекта. Изменения, внесенные в члены объекта, будут видны в вызывающем методе, так как обе ссылки указывают на один и тот же экземпляр.46

Пример:C#  
public record struct Point(int X, int Y); // Тип значения  
public class MyClass { public int Value { get; set; } } // Ссылочный тип  
  
public static void ModifyValueType(Point p)  
{  
 p.X = 100; // Изменяет копию, не оригинал  
 Console.WriteLine($"В методе (Point): {p}");  
}  
  
public static void ModifyReferenceType(MyClass mc)  
{  
 mc.Value = 200; // Изменяет оригинальный объект  
 Console.WriteLine($"В методе (MyClass): {mc.Value}");  
}  
  
public static void TestPassByValue()  
{  
 Point pt = new Point(10, 20);  
 Console.WriteLine($"До вызова (Point): {pt}"); // X=10, Y=20  
 ModifyValueType(pt);  
 Console.WriteLine($"После вызова (Point): {pt}"); // X=10, Y=20 (без изменений)  
  
 MyClass obj = new MyClass { Value = 50 };  
 Console.WriteLine($"До вызова (MyClass): {obj.Value}"); // Value=50  
 ModifyReferenceType(obj);  
 Console.WriteLine($"После вызова (MyClass): {obj.Value}"); // Value=200 (изменено)  
}

1. Передача по ссылке (Pass by Reference):  
   Для передачи аргументов по ссылке используются модификаторы ref, out и in. Это означает, что параметр метода становится псевдонимом для переменной, переданной в качестве аргумента. Любые изменения, внесенные в параметр внутри метода, будут отражены в исходной переменной в вызывающем методе.46
   * **ref:**
     + **Требование:** Аргумент *должен быть инициализирован* до вызова метода.46
     + **Поведение:** Метод может присвоить новое значение параметру, но не обязан это делать. Изменения, внесенные в параметр внутри метода, видны в вызывающем методе.46

**Пример:**

public static void ModifyWithRef(ref int number)  
{  
 number = 100; // Изменяет оригинальную переменную  
 Console.WriteLine($"В методе (ref): {number}");  
}  
  
public static void TestRef()  
{  
 int x = 10;  
 Console.WriteLine($"До вызова (ref): {x}"); // 10  
 ModifyWithRef(ref x);  
 Console.WriteLine($"После вызова (ref): {x}"); // 100  
}

* + **out:**
    - **Требование:** Вызывающий метод *не обязан инициализировать* аргумент до вызова метода. Однако вызванный метод *должен присвоить значение* параметру перед возвратом.46
    - **Назначение:** Используется для возврата нескольких значений из метода.
    - **Пример:**

public static void GetCoordinates(out int x, out int y)  
{  
 x = 5; // Обязательное присваивание  
 y = 10; // Обязательное присваивание  
 Console.WriteLine($"В методе (out): x={x}, y={y}");  
}  
  
public static void TestOut()  
{  
 int coordX, coordY; // Не обязательно инициализировать  
 GetCoordinates(out coordX, out coordY);  
 Console.WriteLine($"После вызова (out): x={coordX}, y={coordY}"); // x=5, y=10  
}

* + **in:**
    - **Назначение:** Передает аргументы по ссылке для *только для чтения* доступа, часто для повышения производительности при работе с большими структурами.
    - **Требование:** Аргумент *должен быть инициализирован* до вызова метода.46
    - **Поведение:** Метод может читать значение аргумента, но *не может его изменять* (не может присвоить новое значение параметру).46
    - **Пример:**

public struct LargeData { public int Id; public string Name; }  
  
public static void ProcessData(in LargeData data)  
{  
 Console.WriteLine($"Обрабатываем: Id={data.Id}, Name={data.Name}");  
 // data.Id = 99; // Ошибка компиляции: нельзя присвоить 'in' параметру  
}  
  
public static void TestIn()  
{  
 LargeData myData = new LargeData { Id = 1, Name = "Тест" };  
 ProcessData(in myData); // Передача по ссылке (только для чтения)  
 Console.WriteLine($"Исходные данные неизменны: Id={myData.Id}"); // Id=1  
}

#### Массив параметров и ключевое слово params

Ключевое слово params позволяет методу принимать переменное количество аргументов указанного типа.46 Параметр с модификатором

params должен быть последним в списке параметров метода, и в сигнатуре метода может быть только один такой параметр.46

Тип параметра params должен быть типом коллекции, например, одномерным массивом (T), System.Span<T>, System.ReadOnlySpan<T> или типом, реализующим System.Collections.Generic.IEnumerable<T> (с параметризованным конструктором и методом Add). До C# 13 он должен был быть только одномерным массивом.46

При вызове метода с параметром params можно передать:

* Список аргументов, разделенных запятыми, типа элементов массива. Компилятор автоматически преобразует их в массив.46
* Уже существующую коллекцию (например, массив) соответствующего типа.46
* Никаких аргументов. В этом случае длина массива params будет равна нулю.46

Пример:

public class ParamsExample  
{  
 public static void PrintNumbers(params int numbers)  
 {  
 Console.Write("Переданные числа: ");  
 foreach (int num in numbers)  
 {  
 Console.Write($"{num} ");  
 }  
 Console.WriteLine();  
 }  
  
 public static void Main()  
 {  
 PrintNumbers(1, 2, 3, 4); // Передача списка аргументов  
 PrintNumbers(10, 20); // Другой список  
 PrintNumbers(); // Без аргументов (пустой массив)  
  
 int myNumbers = { 100, 200, 300 };  
 PrintNumbers(myNumbers); // Передача массива  
 }  
}

#### Необязательные и именованные параметры

* Необязательные параметры (Optional Parameters):  
  Позволяют задать значение по умолчанию для параметра непосредственно в сигнатуре метода. Если вызывающий код не предоставляет аргумент для такого параметра, используется его значение по умолчанию. Необязательные параметры должны быть объявлены после всех обязательных параметров.44  
  Пример:

public static void Greet(string name, string greeting = "Привет")  
{  
 Console.WriteLine($"{greeting}, {name}!");  
}  
  
public static void TestOptional()  
{  
 Greet("Алиса"); // Выведет: Привет, Алиса! (используется значение по умолчанию)  
 Greet("Боб", "Здравствуй"); // Выведет: Здравствуй, Боб! (передано явное значение)  
}

* Именованные аргументы (Named Arguments):  
  Позволяют передавать аргументы методу, указывая их по имени параметра, а не по позиции. Это может значительно улучшить читаемость кода, особенно для методов с большим количеством параметров. Именованные аргументы указываются в формате имяПараметра: значение.44  
  Пример:

public static void DisplayPersonInfo(string firstName, string lastName, int age, string city)  
{  
 Console.WriteLine($"Имя: {firstName} {lastName}, Возраст: {age}, Город: {city}");  
}  
  
public static void TestNamedArguments()  
{  
 // Использование позиционных аргументов  
 DisplayPersonInfo("Джон", "Доу", 30, "Нью-Йорк");  
  
 // Использование именованных аргументов (порядок не важен)  
 DisplayPersonInfo(lastName: "Смит", firstName: "Джейн", city: "Лондон", age: 25);  
  
 // Смешивание позиционных и именованных аргументов (позиционные должны быть первыми)  
 DisplayPersonInfo("Алиса", "Уандерленд", city: "Фэнтезиленд", age: 10);  
}

|  |  |  |  |
| --- | --- | --- | --- |
| Модификатор/Тип | Описание | Требования к аргументу | Поведение в методе |
| **По значению** | Передается копия значения (для типов значений) или копия ссылки (для ссылочных типов). | Не требуется инициализация. | Изменения типа значения не влияют на оригинал. Изменения объекта ссылочного типа влияют на оригинал. |
| ref | Передается ссылка на переменную. | Должен быть инициализирован до вызова. | Метод может читать и записывать новое значение в параметр; изменения видны в вызывающем коде. |
| out | Передается ссылка на переменную. | Не требуется инициализация до вызова. | Метод *должен* присвоить значение параметру перед возвратом; изменения видны в вызывающем коде. |
| in | Передается ссылка на переменную для чтения. | Должен быть инициализирован до вызова. | Метод может читать значение, но *не может* его изменять. |
| params | Позволяет передавать переменное количество аргументов в виде массива. | Может быть список аргументов, массив или ничего. | Аргументы доступны как массив внутри метода. Должен быть последним параметром. |
| **Необязательный** | Параметр имеет значение по умолчанию, если аргумент не передан. | Необязательный аргумент может быть опущен. | Используется значение по умолчанию, если аргумент опущен. |
| **Именованный** | Аргумент идентифицируется по имени параметра, а не по позиции. | Имя параметра указывается при вызове. | Порядок аргументов может быть произвольным (если не смешивается с позиционными). |

### Возвращение значения и оператор return

Оператор return в C# используется для завершения выполнения функции, в которой он появляется, и возврата управления вызывающему коду.52

1. **Возвращение значения:**
   * Если метод вычисляет значение (т.е. имеет не void возвращаемый тип), оператор return используется с выражением. Это выражение должно быть неявно преобразуемо в возвращаемый тип метода.52
   * **Пример:**

public static double CalculateArea(double radius)  
{  
 double area = Math.PI \* radius \* radius;  
 return area; // Возвращает вычисленное значение  
}  
  
public static void Main()  
{  
 double circleArea = CalculateArea(5.0);  
 Console.WriteLine($"Площадь круга: {circleArea:F2}"); // Выведет: 78.54  
}

1. **Методы void:**
   * Если метод не вычисляет значение (т.е. является void методом), оператор return используется без выражения. Это обычно делается для досрочного выхода из метода.52
   * Если void метод не содержит оператора return, он завершается после выполнения своей последней инструкции.52
   * **Пример:**

public static void DisplayMessage(int number)  
{  
 if (number % 2 == 0)  
 {  
 Console.WriteLine("Число четное. Завершаем.");  
 return; // Досрочный выход из метода  
 }  
 Console.WriteLine($"Число нечетное: {number}");  
}  
  
public static void Main()  
{  
 DisplayMessage(6); // Выведет: Число четное. Завершаем.  
 DisplayMessage(7); // Выведет: Число нечетное: 7  
}

1. **Возвращение по ссылке (ref returns):**
   * По умолчанию оператор return возвращает *значение* выражения. Однако C# позволяет возвращать *ссылку* на переменную, что называется **возвратом по ссылке** или ref возвратом.52
   * Это означает, что вызывающий код может изменить возвращенное значение, и это изменение будет отражено в состоянии объекта в вызванном методе, поскольку обе переменные (возвращенная и оригинальная) являются псевдонимами одной и той же области памяти.52
   * Методы, возвращающие по ссылке, не могут иметь возвращаемый тип void. Возвращаемое значение должно иметь время жизни, которое выходит за пределы выполнения метода (т.е. не может быть локальной переменной метода).52
   * **Пример:**

public class ArrayModifier  
{  
 private int data = { 10, 20, 30, 40, 50 };  
  
 public ref int FindValue(int valueToFind)  
 {  
 for (int i = 0; i < data.Length; i++)  
 {  
 if (data[i] == valueToFind)  
 {  
 return ref data[i]; // Возвращаем ссылку на элемент массива  
 }  
 }  
 // В реальном коде здесь нужно обработать случай, когда значение не найдено,  
 // например, выбросить исключение или вернуть ref null (C# 7.2+)  
 throw new InvalidOperationException("Значение не найдено.");  
 }  
  
 public void PrintData()  
 {  
 Console.WriteLine(string.Join(", ", data));  
 }  
}  
  
public static void Main()  
{  
 ArrayModifier modifier = new ArrayModifier();  
 modifier.PrintData(); // Выведет: 10, 20, 30, 40, 50  
  
 ref int element = ref modifier.FindValue(30); // Получаем ссылку на элемент со значением 30  
 element = 300; // Изменяем элемент массива через полученную ссылку  
  
 modifier.PrintData(); // Выведет: 10, 20, 300, 40, 50  
}

Возвращаемое значение ref также может быть ref readonly, что позволяет возвращать ссылку, но запрещает вызывающему коду модифицировать ее.52

### Рекурсивные функции

**Рекурсия** — это процесс, при котором функция вызывает сама себя, прямо или косвенно, для решения задачи. Рекурсивные функции разбивают большую задачу на более мелкие, идентичные подзадачи, пока не будет достигнут базовый случай, который может быть решен напрямую.54

1. **Базовый случай и рекурсивный случай:**
   * **Базовый случай (Base Case):** Это условие, которое определяет, когда рекурсивная функция должна прекратить вызывать сама себя. Без базового случая функция будет вызывать себя бесконечно, что приведет к переполнению стека (StackOverflowException).54
   * **Рекурсивный случай (Recursive Case):** Это часть функции, которая выполняет рекурсивный вызов себя с измененными (обычно уменьшенными) входными данными, приближая задачу к базовому случаю.54
2. Пример: Вычисление факториала:  
   Факториал числа n (обозначается n!) — это произведение всех положительных целых чисел от 1 до n.
   * Базовый случай: 0! = 1 и 1! = 1.
   * Рекурсивный случай: n! = n \* (n-1)!

public class FactorialCalculator  
{  
 public static long Factorial(int n)  
 {  
 // Базовый случай: условие завершения рекурсии  
 if (n <= 1)  
 {  
 return 1;  
 }  
 // Рекурсивный случай: функция вызывает сама себя  
 else  
 {  
 return n \* Factorial(n - 1);  
 }  
 }  
  
 public static void Main()  
 {  
 Console.WriteLine($"Факториал 5: {Factorial(5)}"); // Выведет: 120 (5 \* 4 \* 3 \* 2 \* 1)  
 Console.WriteLine($"Факториал 0: {Factorial(0)}"); // Выведет: 1  
 }  
}

1. **Ограничения рекурсии:**
   * **Переполнение стека:** Каждый рекурсивный вызов добавляет новый фрейм в стек вызовов. Если глубина рекурсии слишком велика, это может привести к исчерпанию доступной памяти стека и выбросу StackOverflowException.54 Размер стека ограничен (например, 1 МБ по умолчанию для потока в CLR, но может быть меньше в некоторых хостах, таких как ASP.NET 54).
   * **Эффективность:** Рекурсивные решения часто менее эффективны, чем итеративные (с использованием циклов), из-за накладных расходов на создание фреймов стека для каждого вызова метода (передача аргументов, инициализация локальных переменных и возврат значений).59
   * **Взаимная рекурсия:** Когда две или более функций вызывают друг друга, это может привести к еще более сложным проблемам с производительностью и отладкой.59

Несмотря на эти ограничения, рекурсия часто предлагает более элегантные и интуитивно понятные решения для проблем, которые по своей природе являются рекурсивными (например, обход деревьев, некоторые алгоритмы сортировки).

### Локальные функции

**Локальные функции** в C# — это методы, которые вложены в другой член (например, метод, конструктор, свойство) и могут быть вызваны только из этого содержащего члена.60 Они были введены для улучшения читаемости и организации кода, явно показывая, что вспомогательный метод используется только в одном конкретном контексте.

1. **Синтаксис и область видимости:**
   * Локальная функция определяется как вложенный метод. Она не может иметь модификаторов доступа (например, public или private), поскольку по своей природе она всегда является приватной для содержащего члена.60
   * Все локальные переменные, определенные в содержащем члене (включая его параметры), доступны в нестатической локальной функции. Это называется "захватом переменных" (variable capture).60
   * Локальная функция может быть объявлена с модификатором static. **Статическая локальная функция** не может захватывать локальные переменные или состояние экземпляра из содержащего члена, что гарантирует отсутствие выделения памяти на куче для замыканий.60

Пример:C#

public class LocalFunctionExample  
{  
 public static string ProcessText(string input, string separator)  
 {  
 // Локальная функция, доступная только внутри ProcessText  
 string AppendSeparator(string text)  
 {  
 return text.EndsWith(separator)? text : text + separator;  
 }  
  
 // Статическая локальная функция, не может захватывать input или separator  
 static string ToUpperInvariant(string text)  
 {  
 return text.ToUpperInvariant();  
 }  
  
 string processedInput = AppendSeparator(input);  
 return ToUpperInvariant(processedInput);  
 }  
  
 public static void Main()  
 {  
 Console.WriteLine(ProcessText("hello", "!")); // Выведет: HELLO!  
 }  
}

1. Отличия от лямбда-выражений:  
   Локальные функции и лямбда-выражения имеют схожие возможности по захвату переменных и созданию анонимных функций, но между ними есть важные различия 60:
   * **Именование:** Локальные функции имеют явные имена, как обычные методы. Лямбда-выражения являются анонимными и должны быть присвоены переменным делегатного типа (Action или Func).60
   * **Определение типов:** Для локальных функций типы аргументов и возвращаемый тип являются частью их объявления. Лямбда-выражения выводят эти типы из делегатной переменной, которой они присваиваются.60
   * **Определенное присваивание:** Локальные функции определяются во время компиляции и могут быть вызваны из любой точки в их области видимости, даже до их определения. Лямбда-выражения объявляются и присваиваются во время выполнения, требуя определенного присваивания перед использованием, что усложняет реализацию рекурсивных алгоритмов с лямбдами.60
   * **Реализация как делегата:** Локальные функции преобразуются в делегаты только тогда, когда они явно используются как делегаты. Если локальная функция вызывается как обычный метод, она не преобразуется в делегат. Лямбда-выражения всегда преобразуются в делегаты при их объявлении.60
   * **Выделение памяти на куче:** Когда локальная функция захватывает переменные из внешней области видимости, она реализуется с использованием замыкания, как и делегаты. Однако локальные функции могут иногда избегать выделения памяти на куче, которое всегда необходимо для лямбда-выражений. Это происходит, если локальная функция никогда не преобразуется в делегат и ее захваченные переменные не захватываются другими лямбдами или локальными функциями, которые *преобразуются* в делегаты. Объявление локальной функции как static гарантирует отсутствие выделения памяти на куче, так как она не может захватывать локальные переменные или состояние экземпляра.60
   * **Ключевое слово yield:** Локальные функции могут быть реализованы как итераторы с использованием синтаксиса yield return для создания последовательности значений. Оператор yield return не разрешен в лямбда-выражениях.60

Таким образом, локальные функции являются более эффективным выбором, когда вспомогательная функция требуется только внутри одного метода, обеспечивая лучшую инкапсуляцию и потенциально избегая накладных расходов на выделение памяти.

## Глава 8: Дополнительные Основы

### Конструкция switch

Конструкция switch в C# позволяет сопоставлять входное выражение с различными характеристиками и выполнять соответствующий блок кода. Она является мощной альтернативой длинным цепочкам if-else if для обработки множества возможных значений или условий.32

1. Базовый синтаксис, метки case и default: Оператор switch вычисляет выражение и пытается сопоставить его результат с рядом меток case. Если совпадение найдено, выполняется блок кода, связанный с этой меткой case. Каждая ветвь case должна завершаться оператором перехода, таким как break, return, goto или throw.32

int day = 3;  
string dayName;  
  
switch (day)  
{  
 case 1:  
 dayName = "Понедельник";  
 break;  
 case 2:  
 dayName = "Вторник";  
 break;  
 case 3:  
 dayName = "Среда";  
 break;  
 default: // Блок default выполняется, если ни одна метка case не совпала  
 dayName = "Неизвестный день";  
 break;  
}  
Console.WriteLine($"День: {dayName}"); // Выведет: День: Среда  
  
Блок default является необязательным и обрабатывает все входные значения, которые не соответствуют ни одной из меток case. Если default отсутствует и ни один case не совпал, выполнение программы продолжается после блока switch (если это оператор switch), или выбрасывается исключение (если это выражение switch).61

1. Возможности сопоставления с образцом (Pattern Matching) в switch:  
   Начиная с C# 7.0, оператор switch был значительно расширен за счет возможностей сопоставления с образцом, что позволяет использовать более сложные условия, чем простое равенство с константами.61
   * **Константный образец (Constant Pattern):** Проверяет, равно ли выражение указанной константе. Это базовый тип образца, используемый в приведенном выше примере switch.61
   * **Образец объявления (Declaration Pattern):** Проверяет тип выражения во время выполнения и, если совпадение успешно, присваивает результат выражения новой объявленной переменной.61

object item = "Hello World";  
switch (item)  
{  
 case int i:  
 Console.WriteLine($"Это целое число: {i}");  
 break;  
 case string s:  
 Console.WriteLine($"Это строка: {s.Length} символов"); // s - переменная, захватывающая значение  
 break;  
 case null:  
 Console.WriteLine("Это null");  
 break;  
 default:  
 Console.WriteLine("Неизвестный тип");  
 break;  
}  
// Выведет: Это строка: 11 символов

**Образец типа (Type Pattern):** Подобен образцу объявления, но используется только для проверки типа без объявления новой переменной.61

object obj = 123;  
switch (obj)  
{  
 case int:  
 Console.WriteLine("Объект является int.");  
 break;  
 case string:  
 Console.WriteLine("Объект является string.");  
 break;  
 default:  
 Console.WriteLine("Объект другого типа.");  
 break;  
}  
// Выведет: Объект является int.

**Реляционные образцы (Relational Patterns):** Сравнивают результат выражения с константой, используя операторы сравнения (<, >, <=, >=).61

int score = 75;  
string grade = score switch  
{  
 < 60 => "F",  
 < 70 => "D",  
 < 80 => "C",  
 < 90 => "B",  
 \_ => "A" // Дискард-образец как default  
};  
Console.WriteLine($"Оценка: {grade}"); // Выведет: Оценка: C

* + **Логические образцы (Logical Patterns):** Объединяют другие образцы с помощью операторов not, and, or.61
    - and: Соответствует, если оба образца совпадают.
    - or: Соответствует, если хотя бы один образец совпадает.
    - not: Соответствует, если отрицаемый образец не совпадает.

int month = 4; // Апрель  
string season = month switch  
{  
 3 or 4 or 5 => "Весна",  
 6 or 7 or 8 => "Лето",  
 9 or 10 or 11 => "Осень",  
 12 or 1 or 2 => "Зима",  
 \_ => "Неизвестно"  
};  
Console.WriteLine($"Месяц {month} - это {season}"); // Выведет: Месяц 4 - это Весна  
  
// Комбинация с 'and'  
int value = 15;  
string range = value switch  
{  
 >= 0 and < 10 => "Маленькое",  
 >= 10 and < 20 => "Среднее",  
 \_ => "Большое"  
};  
Console.WriteLine($"Значение {value} - {range}"); // Выведет: Значение 15 - Среднее

**Образцы свойств (Property Patterns):** Проверяют, соответствуют ли свойства или поля выражения вложенным образцам.61

public class Product { public string Name { get; set; } public decimal Price { get; set; } }  
Product p = new Product { Name = "Книга", Price = 15.0m };  
string category = p switch  
{  
 { Price: < 10.0m } => "Бюджетный",  
 { Price: >= 10.0m and < 50.0m } => "Средний сегмент",  
 \_ => "Премиум"  
};  
Console.WriteLine($"Продукт '{p.Name}' относится к категории: {category}"); // Выведет: Средний сегмент

**Образцы списков (List Patterns) (C# 11+):** Сопоставляют массив или список с последовательностью образцов.61

int data = { 1, 2, 3 };  
bool matches = data is ; // Соответствует последовательности элементов  
Console.WriteLine(matches); // Выведет: True  
  
bool partialMatch = new { 1, 2, 3, 4, 5 } is [1,.., 5]; // Срез.. соответствует нулю или более элементов  
Console.WriteLine(partialMatch); // Выведет: True

**Дискард-образец (\_):** Соответствует любому выражению (включая null). Часто используется как default случай в выражениях switch.61

Сопоставление с образцом в switch значительно повышает выразительность и читаемость кода, позволяя обрабатывать сложные условия элегантным и структурированным образом.

### Перечисления enum

**Перечисления (Enums)** в C# — это отдельные типы значений, которые объявляют набор именованных констант.62 Они предоставляют способ определения набора логически связанных целочисленных констант, делая код более читаемым и менее подверженным ошибкам, чем использование "магических чисел".

1. Объявление перечислений: Объявление перечисления начинается с ключевого слова enum, за которым следует имя перечисления и тело, заключенное в фигурные скобки {}. Тело содержит члены перечисления — именованные константы.62  
   Пример:

public enum Season  
{  
 Spring,  
 Summer,  
 Autumn,  
 Winter  
}

1. Базовый тип (Underlying Type):  
   Каждый тип перечисления имеет соответствующий базовый целочисленный тип, который по умолчанию является int, если не указан явно. Базовый тип должен быть способен представлять все значения перечислителя. Разрешены следующие целочисленные типы: byte, sbyte, short, ushort, int, uint, long, ulong.62  
   Пример с явным базовым типом:

public enum ErrorCode : short  
{  
 None = 0,  
 FileNotFound = 1,  
 AccessDenied = 2  
}

1. Члены перечисления: Члены перечисления — это именованные константы. Каждый член имеет связанное с ним постоянное значение, тип которого совпадает с базовым типом перечисления. Значения могут быть присвоены явно или неявно.62
   * **Неявное присваивание:** Если член перечисления не имеет инициализатора:
     + Первый член получает значение 0.62
     + Последующие члены получают значение, на единицу большее, чем у предыдущего члена.62
   * **Явное присваивание:** Значение может быть явно присвоено с помощью константного выражения.62 Несколько членов могут иметь одно и то же значение.

Пример:C#

public enum Status  
{  
 Pending, // 0 (по умолчанию)  
 Approved = 10, // 10 (явно)  
 Rejected, // 11 (10 + 1)  
 Cancelled = 10 // 10 (явно, дублирует Approved)  
}

|  |  |
| --- | --- |
| Член Status | Значение |
| Pending | 0 |
| Approved | 10 |
| Rejected | 11 |
| Cancelled | 10 |

1. **Использование перечислений:**
   * **Объявление и присваивание:**

Season currentSeason = Season.Summer;  
Console.WriteLine(currentSeason); // Выведет: Summer

* + **Преобразование в/из целочисленного типа:** Требуется явное приведение типов.62

int seasonValue = (int)Season.Winter; // Преобразование enum в int  
Console.WriteLine(seasonValue); // Выведет: 3 (если Winter = 3)  
  
Season newSeason = (Season)1; // Преобразование int в enum  
Console.WriteLine(newSeason); // Выведет: Summer (если Summer = 1)

* + **Класс System.Enum:** Все типы перечислений неявно наследуются от класса System.Enum, который предоставляет статические методы для работы с перечислениями.62
    - Enum.Parse() / Enum.TryParse(): Преобразует строковое представление имени или значения перечисления в соответствующий член перечисления.63
    - Enum.ToObject(): Преобразует целочисленное значение в член перечисления.63
    - Enum.GetNames(): Возвращает массив строк с именами членов перечисления.63
    - Enum.GetValues(): Возвращает массив значений членов перечисления.63
    - Enum.IsDefined(): Проверяет, определено ли указанное значение или имя в перечислении.63
    - ToString(format): Форматирует значение перечисления в строку (например, "G" для имени, "D" для базового значения).63
  + **Атрибут [Flags]:**
    - Используется для перечислений, где члены могут быть объединены с помощью побитовых операций (например, для представления набора флагов или опций). Значения членов должны быть степенями двойки.63
    - **Пример:**

[Flags]  
public enum Permissions  
{  
 None = 0,  
 Read = 1,  
 Write = 2,  
 Execute = 4,  
 All = Read | Write | Execute // Комбинация флагов  
}  
  
public static void Main()  
{  
 Permissions userPerms = Permissions.Read | Permissions.Write;  
 Console.WriteLine(userPerms); // Выведет: Read, Write  
 if (userPerms.HasFlag(Permissions.Write)) // Проверка флага  
 {  
 Console.WriteLine("Пользователь имеет право на запись.");  
 }  
}

Рекомендуется использовать HasFlag для проверки наличия флага, а не прямые побитовые операции, для улучшения читаемости.63

Перечисления значительно улучшают читаемость и поддерживаемость кода, заменяя "магические числа" на осмысленные имена.

## Заключение

Данный подробный курс по основам программирования на C# охватил ключевые концепции, необходимые для начинающих разработчиков, и предоставил прочную основу для дальнейшего изучения языка и платформы.NET. Были рассмотрены фундаментальные аспекты, начиная от общей структуры программы и заканчивая сложными механизмами управления потоком выполнения и работы с данными.

В ходе изучения были детально разобраны:

* **Структура программы C#**, включая традиционные подходы с методом Main и современные операторы верхнего уровня, что демонстрирует эволюцию языка в сторону упрощения и повышения доступности.
* **Переменные, константы и литералы**, с акцентом на их объявление, инициализацию, область видимости и различия между const и readonly.
* **Типы данных**, где было проведено четкое разграничение между типами значений и ссылочными типами, объяснены их поведение при присваивании и различия в выделении памяти (стек против кучи), а также рассмотрены механизмы упаковки и распаковки.
* **Операции и выражения**, включая арифметические, поразрядные операторы и операторы присваивания, с подробным объяснением их приоритета, ассоциативности и обработки переполнения целых чисел. Были также изучены различные виды преобразований типов, включая неявные, явные и использование класса System.Convert.
* **Управление потоком выполнения**, где были представлены условные выражения, конструкции if..else и тернарный оператор, а также основные типы циклов (for, while, do-while, foreach) и их вложенные варианты.
* **Массивы** как структуры данных для хранения коллекций однотипных элементов, включая одномерные, многомерные и зубчатые массивы, а также методы для их манипуляции.
* **Методы** как основные строительные блоки программы, с подробным описанием их определения, синтаксиса, вызова, различных типов параметров (по значению, по ссылке, params, именованные и необязательные), а также механизмов возврата значений, включая ref возвраты.
* **Рекурсивные и локальные функции**, представляющие собой более продвинутые концепции организации кода, с объяснением их принципов работы, преимуществ и ограничений.
* **Конструкция switch** с ее мощными возможностями сопоставления с образцом, позволяющими элегантно обрабатывать сложные условия.
* **Перечисления enum**, которые повышают читаемость и типобезопасность кода за счет использования именованных констант.

C# выделяется своей строгой типизацией, встроенными механизмами безопасности (такими как автоматическая сборка мусора и проверка границ массивов) и обширной экосистемой Microsoft, которая предоставляет высококачественные инструменты и обучающие ресурсы. Эти аспекты делают C# надежным и продуктивным выбором для разработки, а также относительно легким для освоения новичками, поскольку язык помогает предотвращать многие распространенные ошибки на ранних этапах.

Постоянная практика и изучение официальной документации Microsoft Learn помогут закрепить полученные знания и освоить более сложные аспекты C# и.NET.

#### Источники

1. Introduction - C# language specification | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction>
2. C# language reference | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/>
3. C# Fundamentals for Absolute Beginners | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/shows/csharp-fundamentals-for-absolute-beginners/>
4. C# Fundamentals for Absolute Beginners | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/shows/c-fundamentals-for-absolute-beginners/>
5. C# for Beginners | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/shows/csharp-for-beginners/>
6. Interactive tutorials - A tour of C# | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/tutorials/>
7. Learn C# - free tutorials, courses, videos, and more | .NET - Microsoft, дата последнего обращения: июля 20, 2025, <https://dotnet.microsoft.com/en-us/learn/csharp>
8. Classes and objects tutorial - C# - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/classes>
9. General Structure of a Program - C# | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/el-gr/dotnet/csharp/fundamentals/program-structure/>
10. Main() and command-line arguments - C# | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/main-command-line>
11. Variables - C# language specification - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/variables>
12. Learn the fundamentals of the C# type system - C# | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/>
13. Scope of Variables in C# : Class Level, Method Level and Block ..., дата последнего обращения: июля 20, 2025, <https://www.scholarhat.com/tutorial/csharp/scope-of-variables-in-csharp>
14. How to define constants - C# | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/bs-latn-ba/dotnet/csharp/programming-guide/classes-and-structs/how-to-define-constants>
15. The const keyword - C# reference | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/const>
16. C# Literals - GeeksforGeeks, дата последнего обращения: июля 20, 2025, <https://www.geeksforgeeks.org/c-sharp/c-sharp-literals/>
17. Strings - C# | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/>
18. Verbatim text and strings - @ - C# reference - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/verbatim>
19. Raw string literal - C# feature specifications - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-11.0/raw-string-literal>
20. Conversions - C# language specification - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/conversions>
21. Types - C# language specification - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types>
22. Reference types - C# reference | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types>
23. Structure types - C# reference - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct>
24. Stack Vs. Heap In C#: What Every Developer Should Know | Nile Bits, дата последнего обращения: июля 20, 2025, <https://www.nilebits.com/blog/2024/06/stack-vs-heap-in-csharp/>
25. .net - Why do we need boxing and unboxing in C#? - Stack Overflow, дата последнего обращения: июля 20, 2025, <https://stackoverflow.com/questions/2111857/why-do-we-need-boxing-and-unboxing-in-c>
26. Arithmetic operators - C# reference - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators>
27. Addition operators - + and += - C# reference - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/addition-operator>
28. Bitwise and shift operators - perform boolean (AND, NOT, OR, XOR ..., дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators>
29. Assignment operators - assign an expression to a variable - C# ..., дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/assignment-operator>
30. User Defined Compound Assignment Operators - C# - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/user-defined-compound-assignment>
31. System.Convert class - .NET | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-convert>
32. Learn C#: Conditionals Cheatsheet | Codecademy, дата последнего обращения: июля 20, 2025, <https://www.codecademy.com/learn/learn-c-sharp/modules/learn-c-conditionals/cheatsheet>
33. ConditionalExpression Class (System.Linq.Expressions) - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.linq.expressions.conditionalexpression?view=net-9.0>
34. Understand all about C# If else Statement with Examples | Simplilearn, дата последнего обращения: июля 20, 2025, <https://www.simplilearn.com/tutorials/c-sharp-tutorial/c-sharp-if-else>
35. Branches and loops - Introductory tutorial - A tour of C# | Microsoft ..., дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/tutorials/branches-and-loops>
36. C# Ternary Operator (?:) with Examples - Tutlane, дата последнего обращения: июля 20, 2025, <https://www.tutlane.com/tutorial/csharp/csharp-ternary-operator-with-examples>
37. C# Foreach Loop: Mastering foreach in C# - Simplilearn.com, дата последнего обращения: июля 20, 2025, <https://www.simplilearn.com/tutorials/c-sharp-tutorial/c-sharp-foreach>
38. Foreach loop - C# for .NET in 101 Shorts - 16 /101 - YouTube, дата последнего обращения: июля 20, 2025, <https://www.youtube.com/watch?v=Pk12_6PAfMc>
39. The array reference type - C# reference | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/arrays>
40. Learn C#: Arrays Cheatsheet - Codecademy, дата последнего обращения: июля 20, 2025, <https://www.codecademy.com/learn/learn-c-sharp/modules/learn-csharp-arrays/cheatsheet>
41. Arrays - C# language specification - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/arrays>
42. NET: Array Types in .NET - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2002/february/net-array-types-in-net>
43. C# Array Class, дата последнего обращения: июля 20, 2025, <https://www.tutorialspoint.com/csharp/csharp_array_class.htm>
44. Learn C#: Methods Cheatsheet - Codecademy, дата последнего обращения: июля 20, 2025, <https://www.codecademy.com/learn/learn-c-sharp/modules/csharp-methods/cheatsheet>
45. Overview of methods - C# | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/methods>
46. Method parameters and modifiers - C# reference | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/method-parameters>
47. Passing a class as a ref parameter in C# does not always work as expected. Can anyone explain? - Stack Overflow, дата последнего обращения: июля 20, 2025, <https://stackoverflow.com/questions/9996359/passing-a-class-as-a-ref-parameter-in-c-sharp-does-not-always-work-as-expected>
48. C# Pass By Reference (Ref) with Examples - Tutlane, дата последнего обращения: июля 20, 2025, <https://www.tutlane.com/tutorial/csharp/csharp-pass-by-reference-ref-with-examples>
49. C# | Method Parameters - GeeksforGeeks, дата последнего обращения: июля 20, 2025, <https://www.geeksforgeeks.org/c-sharp/c-sharp-method-parameters/>
50. C#: params keyword vs. list - Stack Overflow, дата последнего обращения: июля 20, 2025, <https://stackoverflow.com/questions/2114297/c-params-keyword-vs-list>
51. Params collections - C# feature specifications - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-13.0/params-collections>
52. Jump statements - break, continue, return, and goto - C# reference ..., дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements>
53. Statements - C# language specification - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/statements>
54. Using Recursion in C# - Stack Overflow, дата последнего обращения: июля 20, 2025, <https://stackoverflow.com/questions/610743/using-recursion-in-c-sharp>
55. Recursion in C# - GeeksforGeeks, дата последнего обращения: июля 20, 2025, <https://www.geeksforgeeks.org/c-sharp/recursion-in-c-sharp/>
56. Understanding Recursion in C# | CodeSignal Learn, дата последнего обращения: июля 20, 2025, <https://codesignal.com/learn/courses/sorting-and-searching-algorithms-in-csharp/lessons/understanding-recursion-in-csharp>
57. Compound assignment operators in C# - Tutorials Point, дата последнего обращения: июля 20, 2025, <https://www.tutorialspoint.com/compound-assignment-operators-in-chash>
58. How To Use Recursion in C# - YouTube, дата последнего обращения: июля 20, 2025, <https://www.youtube.com/watch?v=fCoQP7fFqow>
59. Recursive Procedures - Visual Basic - Learn Microsoft, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/procedures/recursive-procedures>
60. Local functions - C# | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions>
61. Patterns - Pattern matching using the is and switch expressions. - C# ..., дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns>
62. Enums - C# language specification | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/enums>
63. System.Enum class - .NET | Microsoft Learn, дата последнего обращения: июля 20, 2025, <https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-enum>