

# Лекция 1. Основа языка C#. Создание приложений WPF

## 1.1 Типы данных в C#

Под типом данных понимается описание данных определенного вида, для которых известен их способ представления в памяти (формат), следующие из него размерность и диапазон значений, а также определен набор операций.

Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект.

В языке C# есть следующие базовые типы данных:

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

```
bool needShow = true;  
bool isFinish = false;
```

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

```
byte bit1 = 241;  
byte bit2 = 3;
```

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

```
sbyte bit1 = 92;  
sbyte bit2 = -75;
```

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

```
short sh1 = -8;  
short sh2 = 6102;
```

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

```
ushort sh1 = 3671;  
ushort sh2 = 7;
```

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

```
int count = 10;
```

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

```
uint row = 10;
```

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

```
long n = -1983850;
```

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

```
ulong b = 746421710;
```

- **float**: хранит число с плавающей точкой от  $-3.4 \cdot 10^{38}$  до  $3.4 \cdot 10^{38}$  и занимает 4 байта. Представлен системным типом System.Single

```
float position = 4.56;
```

- **double**: хранит число с плавающей точкой от  $\pm 5.0 \cdot 10^{324}$  до  $\pm 1.7 \cdot 10^{308}$  и занимает 8 байта. Представлен системным типом System.Double

```
Double pos = 75.2;
```

- **char**: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:

```
char a = 'A';  
char b = '\x5A';  
char c = '\u0420';
```

- **string**: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.

```
string hello = "Hello";  
string word = "world";
```

- **object**: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе.

Представлен системным типом `System.Object`, который является базовым для всех других типов и классов `.NET`.

```
object a = 78;  
object b = 3.14;  
object c = "name";
```

При объявлении переменных мы явно указывали ее тип, например, `int a = 10`; Компилятор при запуске уже знает, что `a` хранит целочисленное значение. Однако мы можем использовать и неявную типизацию:

```
var name = "Mike";  
var age = 18;
```

Для неявной типизации вместо названия типа данных используется ключевое слово `var`. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. Так как по умолчанию все целочисленные значения рассматриваются как значения типа `int`, то поэтому в итоге переменная `age` будет иметь тип `int`. Аналогично переменной `name` присваивается строка, поэтому эта переменная будет иметь тип `string`. Эти переменные подобны обычным, однако они имеют некоторые ограничения.

Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:

```
// этот код работает  
int a;  
a = 34;  
  
// этот код не работает  
var c;  
c = 68;
```

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной `null`:

```
// этот код не работает  
var c = null;
```

Так как значение `null`, то компилятор не сможет вывести тип данных.

Все типы данных можно разделить на типы значений, еще называемые **значимыми** типами, (`value types`) и **ссылочные** типы (`reference types`). Важно понимать между ними различия.

Типы значений:

- Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong)
- Типы с плавающей запятой (float, double)
- Тип decimal
- Тип bool
- Тип char
- Перечисления enum
- Структуры (struct)

Ссылочные типы:

- Тип object
- Тип string
- Классы (class)
- Интерфейсы (interface)
- Делегаты (delegate)

Чтобы понять их различия, необходимо знать организацию памяти в .NET.

## 1.2 Организация памяти

Память делится на два типа: стек и куча (heap). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

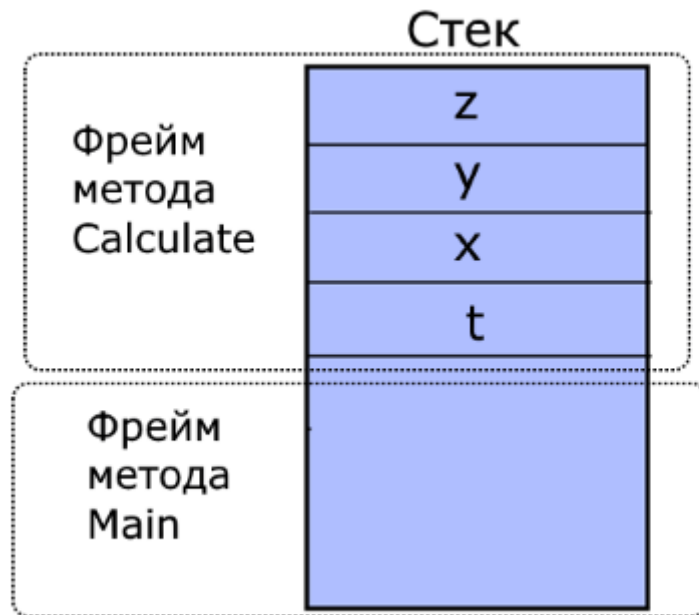
Например:

```
class Program
{
    static void Main(string[] args)
    {
        Calculate(5);
    }

    static void Calculate(int t)
    {
        int x = 6;
        int y = 7;
        int z = y + t;
    }
}
```

```
}
```

При запуске такой программы в стеке будут определяться два фрейма - для метода Main (так как он вызывается при запуске программы) и для метода Calculate:



При вызове этого метода Calculate в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами. Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредственное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода Calculate представляют значимый тип - тип int, поэтому в стеке будут храниться их числовые значения.

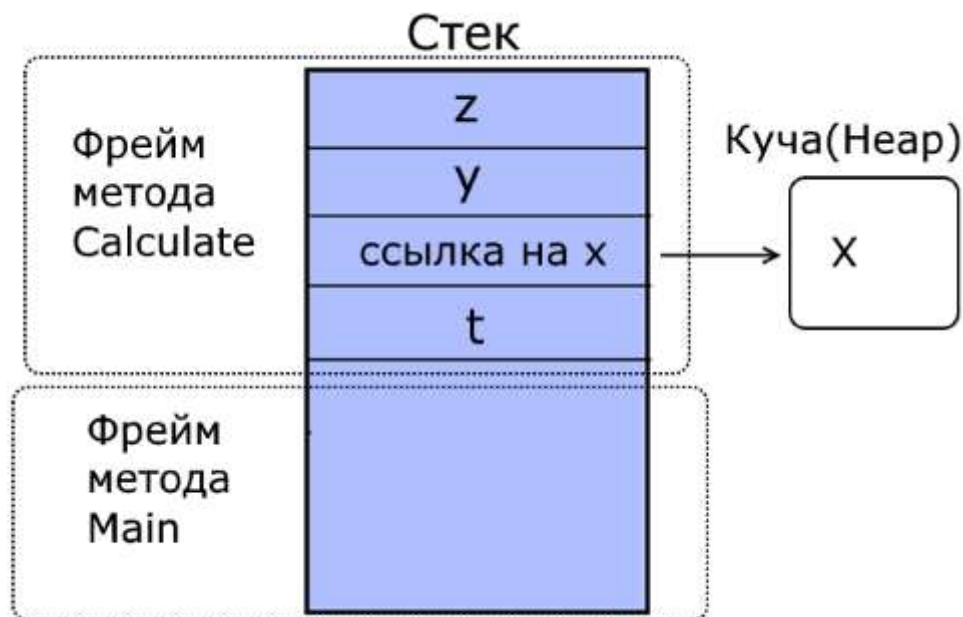
Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это оставшаяся часть памяти, которая доступна процессу.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.

Изменим метод Calculate следующим образом:

```
static void Calculate(int t)
{
    object x = 9;
    int y = 5;
```

```
int z = y + t;  
}
```

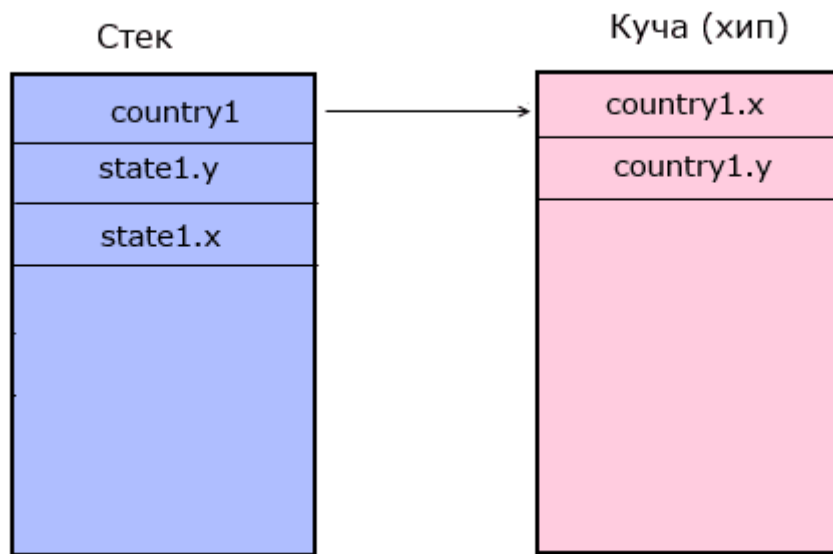


### 1.3 Составные типы данных

Теперь рассмотрим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:

```
// State - структура, ее данные размещены в стеке  
State state1 = new State();  
  
// Country - класс, в стек помещается ссылка на адрес в хипе  
// а в хипе располагаются все данные объекта country1  
Country country1 = new Country();  
struct State  
{  
    public int x;  
    public int y;  
}  
class Country  
{  
    public int x;  
    public int y;  
}
```

Здесь в методе Main в стеке выделяется память для объекта state1. Далее в стеке создается ссылка для объекта country1 (Country country1), а с помощью вызова конструктора с ключевым словом new выделяется место в хипе (new Country()). Ссылка в стеке для объекта country1 будет представлять адрес на место в хипе, по которому размещен данный объект.



Таким образом, в стеке окажутся все поля структуры `state1` и ссылка на объект `country1` в хипе.

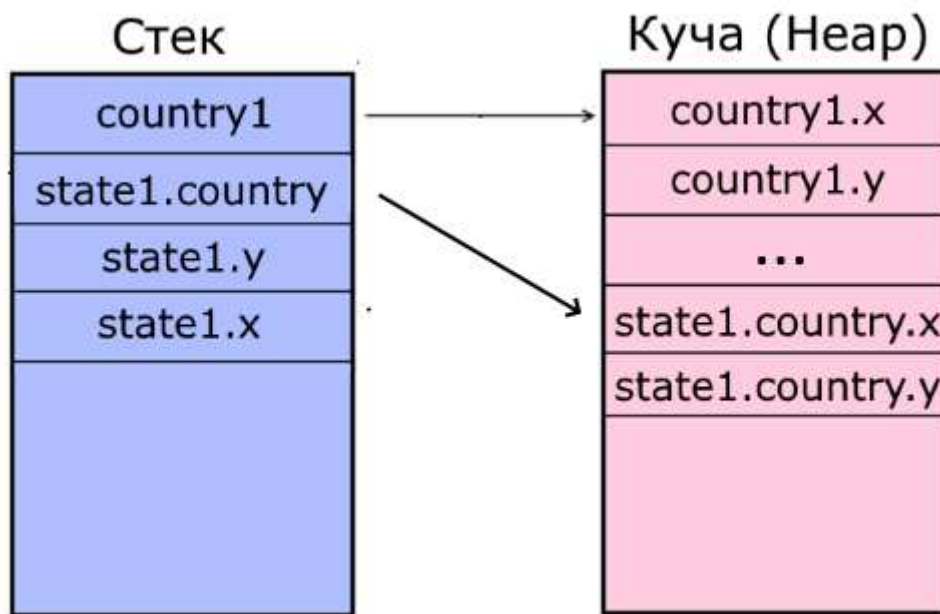
Но, допустим, в структуре `State` также определена переменная ссылочного типа `Country`. Где она будет хранить свое значение, если она определена в типе значений?

```
State state1 = new State();
Country country1 = new Country();

struct State
{
    public int x;
    public int y;
    public Country country;
    public State()
    {
        x = 0;
        y = 0;
        country = new Country();
    }
}

class Country
{
    public int x;
    public int y;
}
```

Значение переменной `state1.country` также будет храниться в куче, так как эта переменная представляет ссылочный тип:



### 1.4 Копирование значений

Тип данных надо учитывать при копировании значений. При присвоении данных объекту значимого типа он получает копию данных. При присвоении данных объекту ссылочного типа он получает не копию объекта, а ссылку на этот объект в хипе. Например:

```
State state1 = new State(); // Структура State
State state2 = new State();
state2.x = 1;
state2.y = 2;
state1 = state2;
state2.x = 5; // state1.x=1 по-прежнему
Console.WriteLine(state1.x); // 1
Console.WriteLine(state2.x); // 5

Country country1 = new Country(); // Класс Country
Country country2 = new Country();
country2.x = 1;
country2.y = 4;
country1 = country2;
country2.x = 7; // теперь и country1.x = 7, так как обе ссылки и
//country1 и country2 указывают на один объект в хипе
Console.WriteLine(country1.x); // 7
Console.WriteLine(country2.x); // 7
```

Так как `state1` - структура, то при присвоении `state1 = state2` она получает копию структуры `state2`. А объект класса `country1` при присвоении `country1 =`



country2; получает ссылку на тот же объект, на который указывает country2. Поэтому с изменением country2, так же будет меняться и country1.

### 1.5 Ссылочные типы внутри типов значений

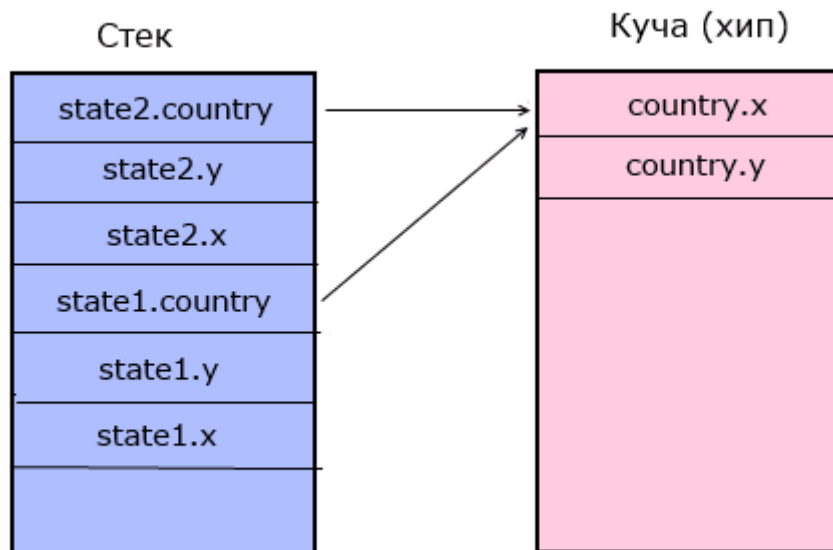
Теперь рассмотрим более изощренный пример, когда внутри структуры у нас может быть переменная ссылочного типа, например, какого-нибудь класса:

```
State state1 = new State();
State state2 = new State();

state2.country = new Country();
state2.country.x = 5;
state1 = state2;
state2.country.x = 8; // теперь и state1.country.x=8, так как
                      //state1.country и state2.country указывают на
                      //один объект в хипе
Console.WriteLine(state1.country.x); // 8
Console.WriteLine(state2.country.x); // 8

struct State
{
    public int x;
    public int y;
    public Country country;
    public State()
    {
        x = 0;
        y = 0;
        country = new Country(); // выделение памяти для объекта
                                //Country
    }
}
class Country
{
    public int x;
    public int y;
}
```

Переменные ссылочных типов в структурах также сохраняют в стеке ссылку на объект в хипе. И при присвоении двух структур state1 = state2; структура state1 также получит ссылку на объект country в хипе. Поэтому изменение state2.country повлечет за собой также изменение state1.country.



## 1.6 Опасность ссылочных типов

- В переменных ссылочных типов хранятся ссылки на их данные (объекты). Две переменные ссылочного типа могут ссылаться на один и тот же объект, поэтому операции над одной переменной могут затрагивать объект, на который ссылается другая переменная.
- Присваивание переменных ссылочного типа влечет копирование не самого объекта, а его ссылки.

## 1.7 Методы

Если переменные хранят некоторые значения, то методы содержат собой набор инструкций, которые выполняют определенные действия. По сути метод - это именованный блок кода, который выполняет некоторые действия. Общее определение методов выглядит следующим образом:

```
[модификаторы] тип_возвращаемого_значения название_метода  
([параметры])  
{  
    // тело метода  
}
```

Модификаторы и параметры необязательны.

```
void SayHello()  
{  
    Console.WriteLine("Hello");  
}
```

Здесь определен метод SayHello, который выводит некоторое сообщение. К названиям методов предъявляются в принципе те же требования, что и к

названиям переменных. Однако, как правило, названия методов начинаются с большой буквы. Перед названием метода идет возвращаемый тип данных. Здесь это тип `void`, который указывает, что фактически ничего не возвращает, он просто производит некоторые действия. После названия метода в скобках идет перечисление параметров. Но в данном случае скобки пустые, что означает, что метод не принимает никаких параметров. После списка параметров в круглых скобках идет блок кода, который представляет набор выполняемых методом инструкций. В данном случае блок метода `SayHello` содержит только одну инструкцию, которая выводит строку в консоль. Вызов метода осуществляется следующим образом:

```
class Program
{
    static void Main(string[] args)
    {
        SayHello(); // вызов метода
    }

    void SayHello()
    {
        Console.WriteLine("Hello");
    }
}
```

### 1.7.1 Параметры метода

Параметры позволяют передать в метод некоторые входные данные. Параметры определяются через запятую в скобках после названия метода в виде:

```
тип_метода имя_метода (тип_параметра1 параметр1, тип_параметра2
параметр2, ...)
{
    // действия метода
}
```

Определение параметра состоит из двух частей: сначала идет тип параметра и затем его имя.

```
void PrintMessage(string text)
{
    Console.WriteLine(text);
}

string str = "Hello my friend!";

PrintMessage("This is a Message");
```

```
PrintMessage(str);
```

## 1.7.2 Возвращаемое значение метода

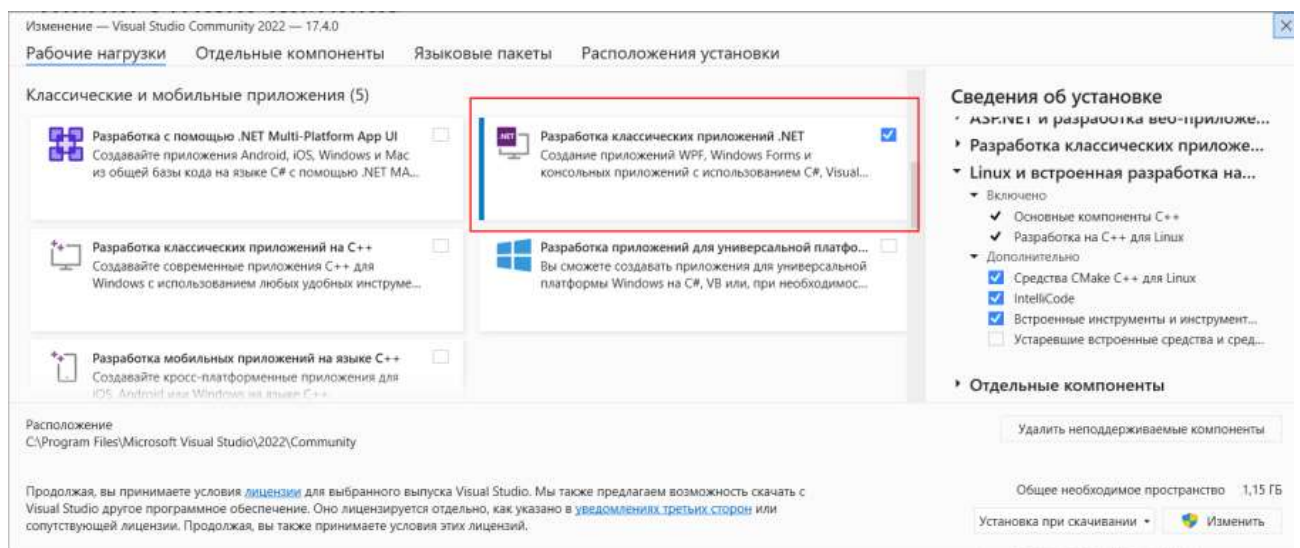
Метод может возвращать значение, какой-либо результат. Если перед названием метода стоит тип `void`, то такой метод не возвращает никакого значения. Он просто выполняет некоторые действия.

Для того чтобы метод вернул какое-то значение используется оператор **return**, после которого идет возвращаемое значение:

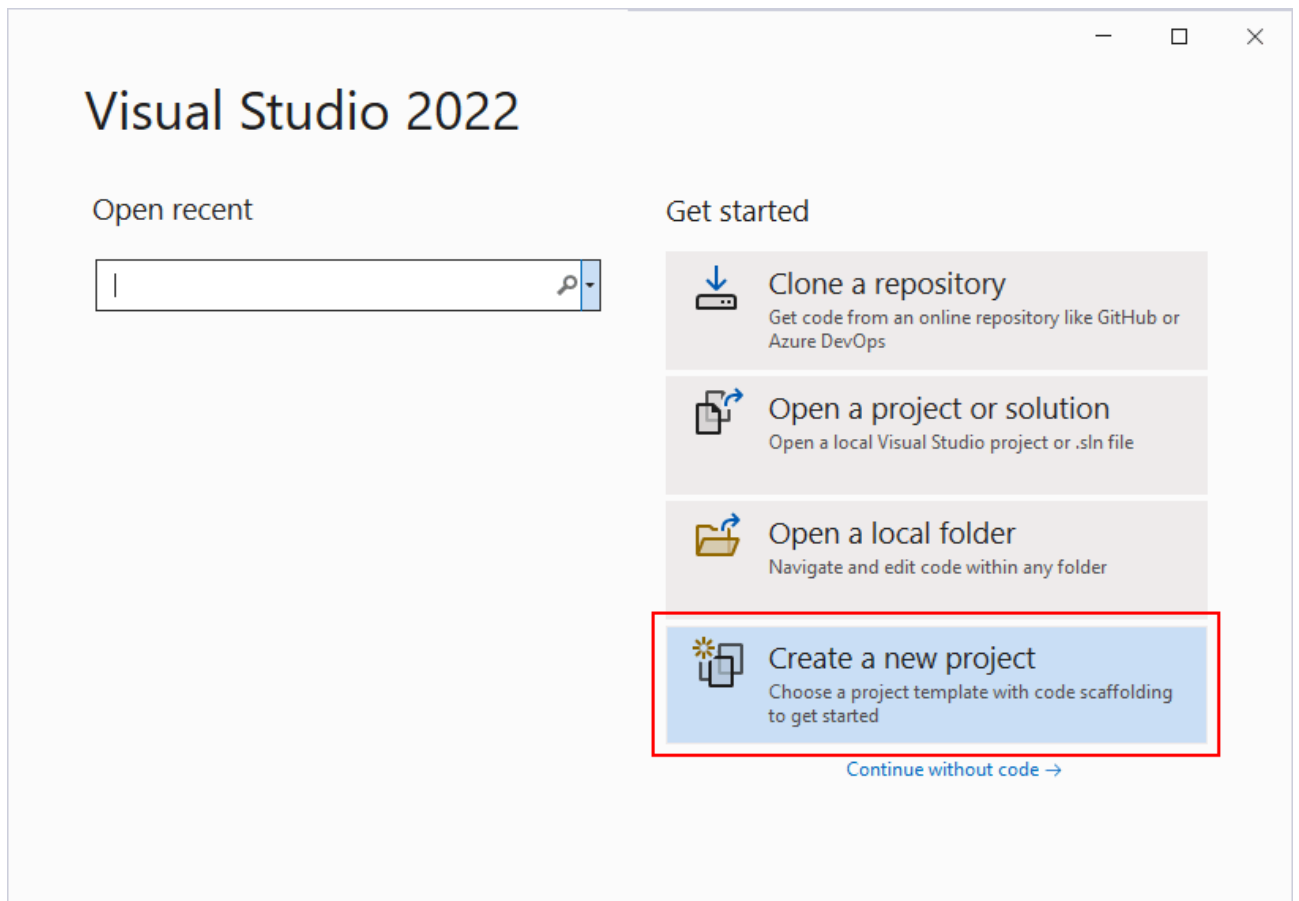
```
// метод возвращает сумму двух чисел
int GetSum()
{
    int a = 5;
    int b = 7;
    return a + b;
}
```

## 1.8 Создание приложений WPF

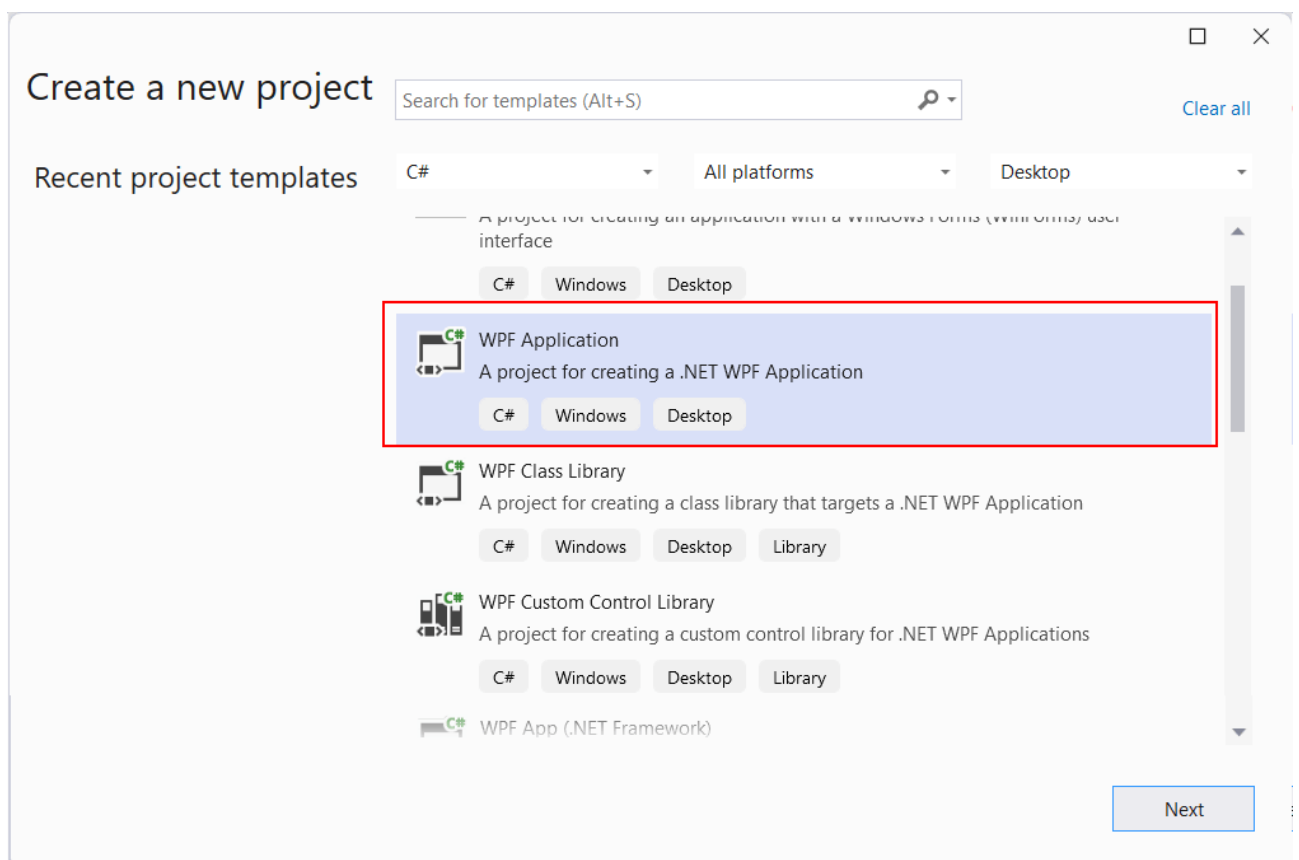
Для создания приложений с помощью технологии WPF можно использовать среду разработки Visual Studio. Чтобы добавить в Visual Studio поддержку проектов для WPF и C# в программе установки среди рабочих нагрузок нужно выбрать только пункт Разработка классических приложений .NET. Можно выбрать и больше опций или вообще все опции, однако стоит учитывать свободный размер на жестком диске - чем больше опций будет выбрано, соответственно тем больше места на диске будет занято.



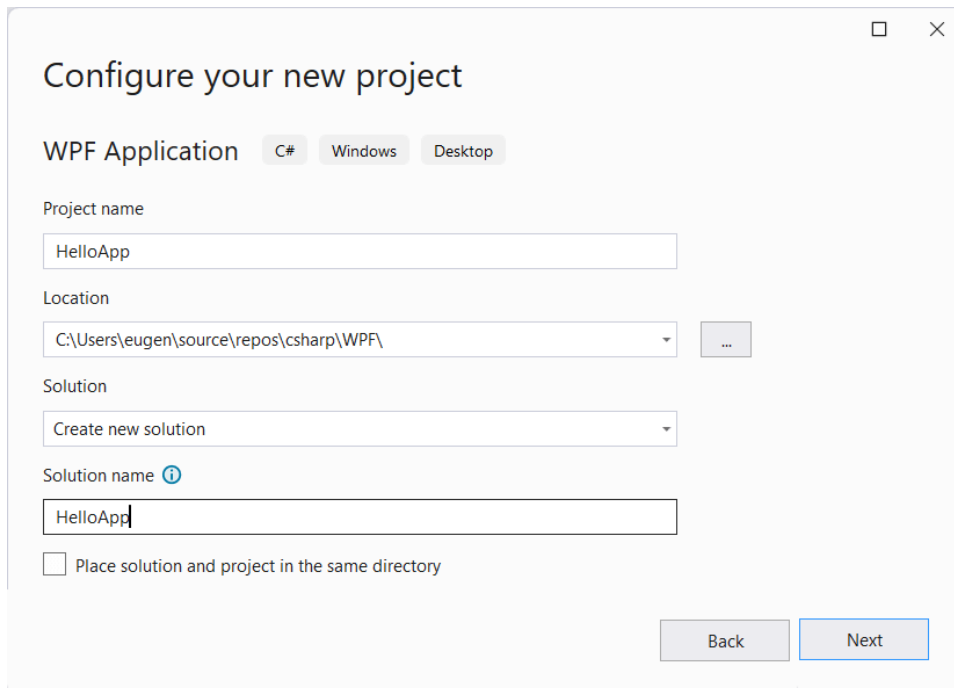
После установки среды и всех ее компонентов, запустим Visual Studio и создадим проект графического приложения. На стартовом экране выберем **Create a new project** (Создать новый проект).



На следующем окне в качестве типа проекта выберем WPF Application:



Далее на следующем этапе нам будет предложено указать имя проекта и каталог, где будет располагаться проект.



Configure your new project

WPF Application C# Windows Desktop

Project name  
HelloApp

Location  
C:\Users\eugen\source\repos\csharp\WPF\

Solution  
Create new solution

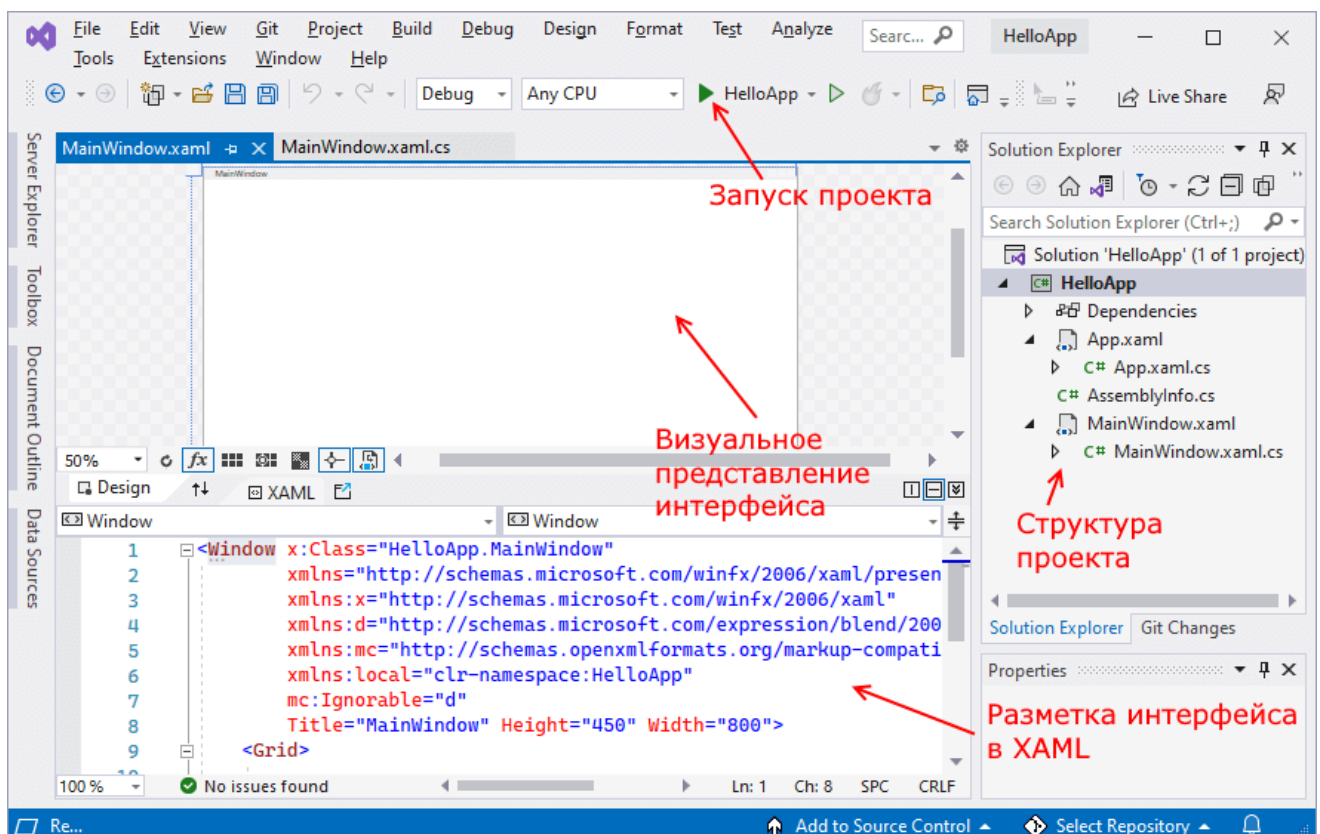
Solution name ⓘ  
HelloApp

☐ Place solution and project in the same directory

Back Next

В поле Project Name дадим проекту какое-либо название. В данном случае это HelloApp.

После этого Visual Studio откроет наш проект с созданными по умолчанию файлами:



### 1.8.1 Структура проекта

Справа находится окно Solution Explorer, в котором можно увидеть структуру нашего проекта. В данном случае у нас сгенерированная по умолчанию структура, которая аналогична той, что создается с помощью .NET CLI:

- **Dependencies** - это узел содержит сборки dll, которые добавлены в проект по умолчанию. Эти сборки как раз содержат классы библиотеки .NET, которые будет использовать C#
- **App.xaml** задает ресурсы приложения и ряд конфигурационных настроек в виде кода XAML. В частности, в файле App.xaml задается файл окна программы, которое будет открываться при запуске приложения. Если вы откроете этот файл, то можете найти в нем строку `StartupUri="MainWindow.xaml"` - то есть в данном случае, когда мы запустим приложение, будет создаваться интерфейс из файла `MainWindow.xaml`.
- **App.xaml.cs** - это файл кода на C#, связанный с файлом App.xaml, который также позволяет задать ряд общих ресурсов и общую логику для приложения, но в виде кода C#.
- **AssemblyInfo.cs** содержит информацию о создаваемой в процессе компиляции сборке
- **MainWindow.xaml** представляет визуальный интерфейс окна приложения в виде кода XAML.
- **MainWindow.xaml.cs** - это файл логики кода на C#, связанный с окном `MainWindow.xaml`.

По умолчанию эти файлы открыты в текстовом редакторе Visual Studio. Причем файл `MainWindow.xaml` имеет два представления: визуальное - в режиме WYSIWIG отображает весь графический интерфейс данного окна приложения, и под ним декларативное объявление интерфейса в XAML. Если мы изменим декларативную разметку, например, определим там кнопку, то эти изменения отображаться в визуальном представлении. Таким образом, мы сможем сразу же получить представление об интерфейсе окна приложения.

### 1.8.2 Запуск проекта

Чтобы запустить приложение в режиме отладки, нажмем на клавишу F5 или на зеленую стрелочку на панели Visual Studio. И после этого запустится пустое окно по умолчанию. После запуска приложения студия компилирует

его в файл с расширением exe. Найти данный файл можно, зайдя в папку проекта и далее в каталог `\bin\Debug\net7.0-windows`