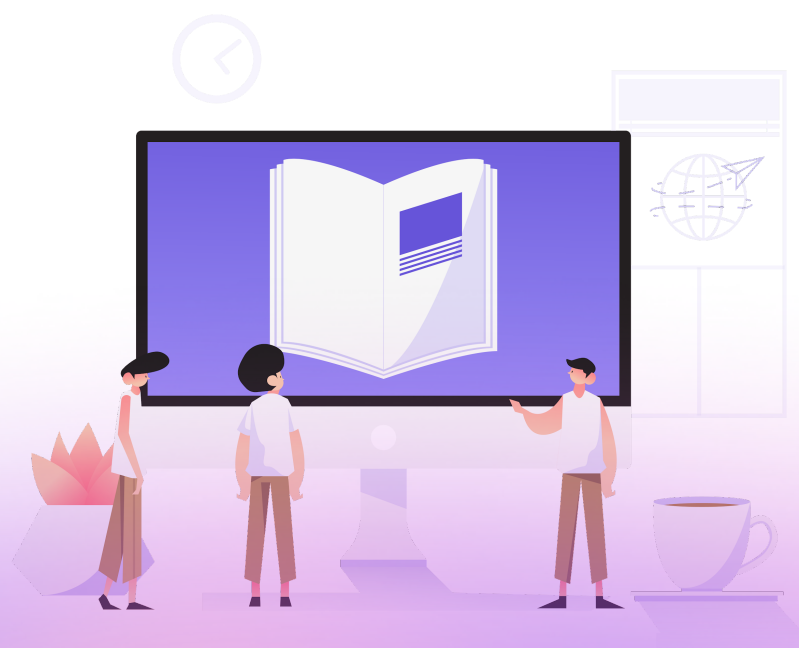


Введение в C#

Обзор инфраструктуры .NET



На этом уроке

1. Рассмотрим виды компиляции.
2. Проанализируем содержимое IL-кода.
3. Познакомимся с инструментами сборки и декомпиляции IL-кода.
4. Рассмотрим способы эффективного поиска информации.

Оглавление

[Виды компиляции. Как работает компиляция в C#](#)

[Декомпиляция .NET-приложений](#)

[ILDasm](#)

[dotPeek](#)

[Защита программного обеспечения](#)

[Сборка мусора](#)

[Работа с большими строками](#)

[Альтернативные способы сборки приложений](#)

[Компилятор csc.exe](#)

[MSBuild](#)

[Поиск источников информации](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Виды компиляции. Как работает компиляция в C#

Компилируемые языки программирования можно разделить на две группы по типу компиляции.

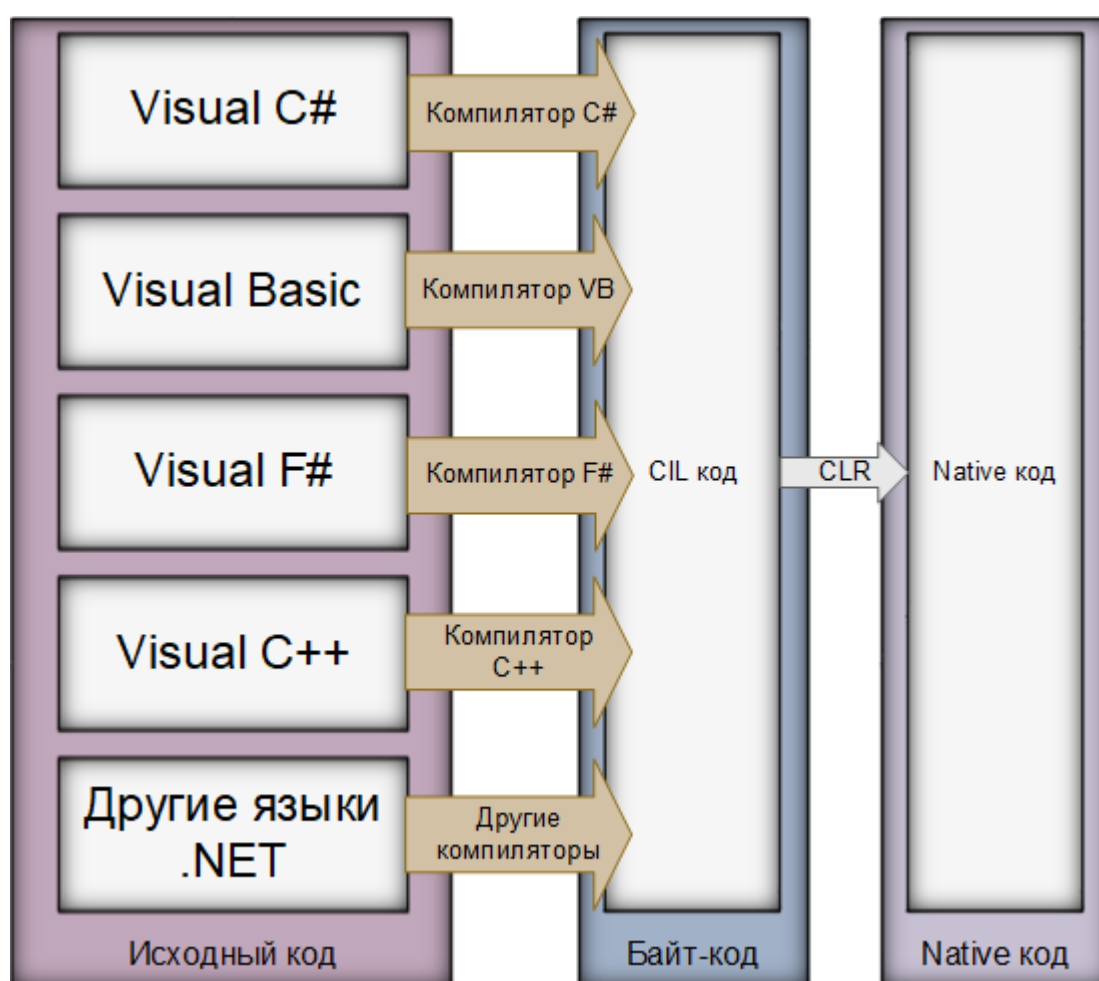
Одни языки компилируют исходный код напрямую в бинарный файл для конкретной операционной системы. Он состоит из машинных команд для определённой архитектуры процессора. К таким языкам можно отнести Си, C++, Delphi. Иногда мы можем запустить такой исполняемый файл на разных архитектурах в рамках одной ОС.

Например, программа, скомпилированная для 32-битной версии ОС Windows успешно выполняется под 64-битной версией ОС, но не наоборот. Компиляцию такого рода называют АОТ-компиляцией (ahead-of-time compilation).

Другие языки, такие как Java и C#, транслируют исходные коды в байт-код — машинный код на промежуточном языке. Такой исполняемый файл не может быть выполнен напрямую, так как, с точки зрения процессора, это некорректная программа.

Перед непосредственным выполнением исполняемый файл переводится в машинные коды «на лету» с помощью динамической компиляции. Динамическую компиляцию также называют JIT-компиляцией.

Когда мы собираем приложение на языке C#, результирующий бинарный файл содержит байт-код — инструкции на языке CIL (Common Intermediate Language — «общий промежуточный язык»). Его чаще называют просто IL. Когда такой бинарный файл запускается на выполнение, он вначале передаёт компиляцию в специальный JIT-компилятор для .NET — CLR (Common Language Runtime — «общезыковая исполняющая среда»). Общий принцип работы компиляции .NET:



Как видно на схеме, .NET поддерживает разработку на множестве языков. При этом везде предоставляется стандартный набор библиотек, а компиляция производится в один и тот же байт-код.

Декомпиляция .NET-приложений

Декомпиляция — процесс, обратный компиляции, при котором получают исходный код приложения из его исполняемого файла. Полученный таким образом код можно модифицировать, а затем пересобрать заново в исполняемый файл.

К декомпиляции зачастую прибегают в том случае, когда нужно внести изменения в приложение, исходный код которого недоступен. Цели могут быть разными: от решения проблем совместимости старых программ с новыми версиями ОС до получения расширенной функциональности.

Среди инструментов для модификации исполняемых файлов стоит выделить декомпиляторы, компиляторы, дизассемблеры и ассемблеры. Дадим определение каждому из этих инструментов.

ILDasm

ILDasm — наиболее простой инструмент для декомпиляции .NET-приложений. Как следует из названия, это дизассемблер IL-кода. Дизассемблерами называют инструменты, которые преобразуют двоичный файл в исходные коды на низкоуровневом языке программирования.

В результате дизассемблирования мы увидим код не на языке C#, а на промежуточном низкоуровневом языке IL. О нём мы говорили ранее. ILDasm можно использовать как консольную утилиту. Есть и графический интерфейс.

Далее в этом разделе мы будем рассматривать примеры исходного кода на языке IL. В конце урока приведены ссылки на документацию и описание используемых инструкций.

Попробуем посмотреть с помощью дизассемблера код простой программы на языке .NET:

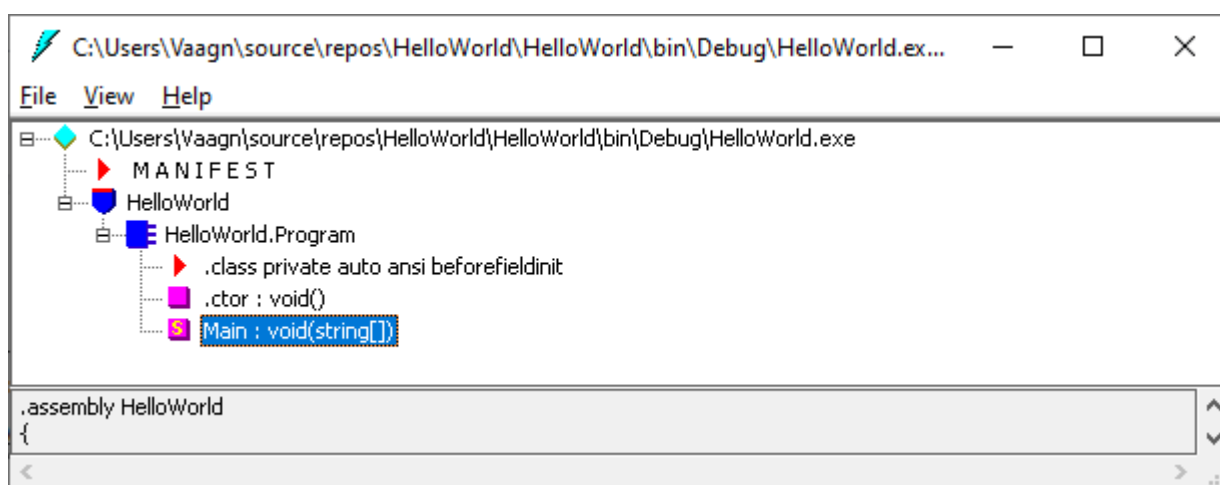
```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            string secret = "some secret password";
            Console.WriteLine("Enter password:");
            string input = Console.ReadLine();
            if (input == secret)
            {
                Console.WriteLine("Welcome!");
            }
        }
    }
}
```

Осуществим сборку и перейдём в папку с исполняемым файлом. Далее запустим ILDasm. Для этого в меню «Пуск» наберём develop и выберем Developer Command Prompt for VS 2019. Откроется терминал командной строки Windows, в котором будут заранее объявлены пути к инструментам разработчика. Запустим ILDasm:

```
ildasm
```

Запустится графический интерфейс программы. Теперь мы можем просто перетащить мышкой исполняемый файл нашей программы в окно ILDasm. В результате содержимое окна изменится:



Если дважды кликнем на методе Main, то увидим его содержимое, представленное на языке IL:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size          49 (0x31)
    .maxstack 2
    .locals init ([0] string secret,
                  [1] string input,
                  [2] bool V_2)
    IL_0000: nop
    IL_0001: ldstr      "some secret password"
    IL_0006: stloc.0
    IL_0007: ldstr      "Enter password:"
    IL_000c: call       void [mscorlib]System.Console::WriteLine(string)
    IL_0011: nop
    IL_0012: call       string [mscorlib]System.Console::ReadLine()
    IL_0017: stloc.1
    IL_0018: ldloc.1
    IL_0019: ldloc.0
    IL_001a: call       bool [mscorlib]System.String::op_Equality(string,
                                                         string)
```

```
IL_001f: stloc.2
IL_0020: ldloc.2
IL_0021: brfalse.s IL_0030
IL_0023: nop
IL_0024: ldstr      "Welcome!"
IL_0029: call      void [mscorlib]System.Console::WriteLine(string)
IL_002e: nop
IL_002f: nop
IL_0030: ret
} // end of method Program::Main
```

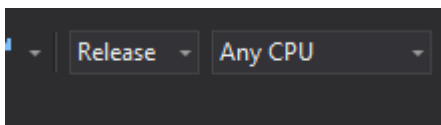
Код на языке C# выглядит намного лаконичнее и понятнее. Некоторые его операции объединяют в себе несколько операций на языке IL. Поэтому C# — это язык высокого уровня, а IL — низкоуровневый.

Разберёмся с листингом IL-кода, чтобы понять, как работает наша программа. Но перед этим познакомимся с часто встречающейся в коде инструкцией `nop`. Инструкция `nop` ничего не делает. Она так и расшифровывается — No Operation. Её присутствие в коде обусловлено тем, что мы собрали приложение в конфигурации для отладки (Debug).

Visual Studio, точнее, встроенный в неё отладчик, использует зарезервированные инструкцией `nop`-места в программном коде, когда мы расставляем точки останова. В таком случае оператор `nop` в памяти заменяется на специальную инструкцию, которая позволяет отладчику встроиться в процесс выполнения. Она даёт приостановить программу и отобразить сведения, которые мы можем наблюдать в окне IDE. Например, значения локальных переменных и многое другое.



Если мы соберём приложение в конфигурации Release, исполняемый файл не будет содержать отладочной информации. Будет отсутствовать и `nop`-инструкции. Это уменьшит размер приложения, но сделает отладку неудобной. На практике во время разработки приложение компилируется в Debug-конфигурации. Перед публикацией или передачей клиентам производится сборка в Release.

Попробуем собрать нашу программу в конфигурации Release и продолжим разбор её работы на более чистом IL-коде. Выберем соответствующую конфигурацию и запустим сборку:



Теперь выходная папка проекта содержит два файла:

» HelloWorld » HelloWorld » bin »

Name	^	Date modified	Type	?
 Debug		06.12.2020 11:51	File folder	
 Release		06.12.2020 12:34	File folder	

Перейдём в папку Release, перетащим HelloWorld.exe в окно ILDasm и вновь взглянем на исходный код метода Main:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size      40 (0x28)
    .maxstack 2
    .locals init ([0] string secret)
    IL_0000: ldstr      "some secret password"
    IL_0005: stloc.0
    IL_0006: ldstr      "Enter password:"
    IL_000b: call       void [mscorlib]System.Console::WriteLine(string)
    IL_0010: call       string [mscorlib]System.Console::ReadLine()
    IL_0015: ldloc.0
    IL_0016: call       bool [mscorlib]System.String::op_Equality(string,
                                                                string)
    IL_001b: brfalse.s  IL_0027
    IL_001d: ldstr      "Welcome!"
    IL_0022: call       void [mscorlib]System.Console::WriteLine(string)
    IL_0027: ret
} // end of method Program::Main
```

Конфигурации сборки и их назначение рассмотрим более подробно на следующем уроке.

Как видно из листинга, инструкции `nop` были удалены. Мы видим объявление метода Main:

```
.method private hidebysig static void Main(string[] args) cil managed
```

Этот метод — точка входа в приложение, поэтому в его теле указывается директива (специальная инструкция) `.entrypoint`.

Далее идёт комментарий с указанием размера программного кода в байтах. Он был подсчитан ILDasm — это только размер инструкций, а не всего приложения.

Следующая директива `.maxstack` указывает на количество инструкций в стеке, место, которое необходимо заранее подготовить перед выполнением метода. Это лишь стартовое значение, и при выполнении приложения стек может содержать гораздо больше операций.

Далее идёт блок объявления всех локальных переменных метода:

```
.locals init ([0] string secret)
```


Каждая локальная переменная имеет свой индекс. У переменной `secret` это значение 0. Индекс будет использоваться для обращения к переменным далее в коде программы.

```
IL_0000: ldstr      "some secret password"
IL_0005: stloc.0
```

Инструкция `ldstr` помещает на вершину стека ссылку на строку `"some secret password"`. Следующая инструкция `stloc` достаёт эту ссылку из стека и сохраняет в переменную с индексом 0, то есть `secret`. Эти две инструкции выполняют строку C#-кода:

```
secret = "some secret password";
```

Далее происходит вывод строки в консоль. Сначала в стеке размещаются аргументы метода `Console.WriteLine`, а затем происходит его вызов:

```
IL_0006: ldstr      "Enter password:"
IL_000b: call        void [mscorlib]System.Console::WriteLine(string)
```

Далее следует интересный момент. Нужно сравнить введённое значение с локальной переменной `secret`, имеющей индекс 0. Для этого вызываем метод `ReadLine`. Ссылка на введённую строку остаётся лежать в стеке. Следом загружаем в стек локальную переменную с индексом 0 и инструкцией `ldloc` и, наконец, вызываем операцию сравнения (`==`). Операторы скрывают за собой обычный вызов метода, который мы видим в третьей строке:

```
IL_0010: call        string [mscorlib]System.Console::ReadLine()
IL_0015: ldloc.0
IL_0016: call        bool [mscorlib]System.String::op_Equality(string,
                                                         string)
```

Оператор извлекает из стека два значения — ссылки на две строки. Сравнивает значения строк.

Результат выполнения оператора сравнения далее переходит к инструкции `brfalse`:

```
IL_001b: brfalse.s  IL_0027
```

Если результат сравнения будет равен `False`, то произойдёт переход на эту строку. Это приведёт к завершению программы:

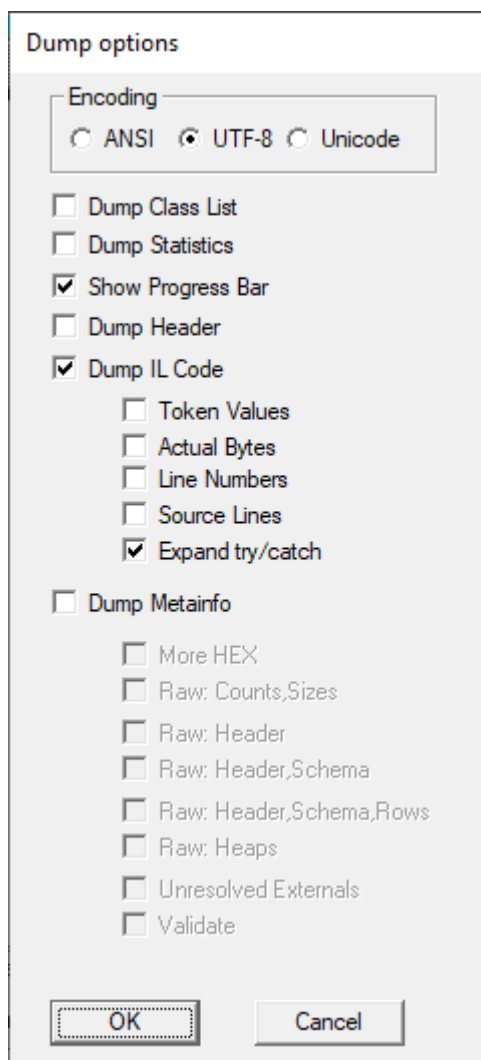
```
IL_0027: ret
```

В противном случае будет осуществлён вывод строки "Welcome!":

```
IL_001d: ldstr      "Welcome!"
IL_0022: call       void [mscorlib]System.Console::WriteLine(string)
```

Попробуем изменить поведение нашей программы, чтобы она принимала неправильные пароли. Для этого нужно просто инвертировать логику сравнения двух строк. Конечно, в таком случае правильный пароль будет считаться неверным, но мы не будем рассматривать этот сценарий для простоты примера. Иначе пришлось бы внести несколько значительных изменений в IL-код.

Чтобы отредактировать IL-код, нужно сначала выгрузить его в текстовый файл. В меню ILDasm выберем File → Dump, выставим следующие настройки и нажмём OK:



Укажем имя HelloPatched и сохраним результат в файл. Откроем HelloPatched.il в любом текстовом редакторе и найдём операцию сравнения:

```
IL_001b: brfalse.s IL_0027
```

Заменяем инструкцию `brfalse.s` на `brtrue.s`:

```
IL_001b: brtrue.s IL_0027
```

Теперь логика завершения работы изменена на обратную: при вводе неправильного пароля мы увидим строку `Welcome!`

Осталось собрать исполняемый файл. Для сборки программ из IL-кода есть инструмент `ILasm`. В терминале командной строки `Developer Command Prompt for VS 2019` выполним такую команду:

```
cd <путь к папке с файлом HelloPatched.il>  
ilasm HelloPatched.il
```

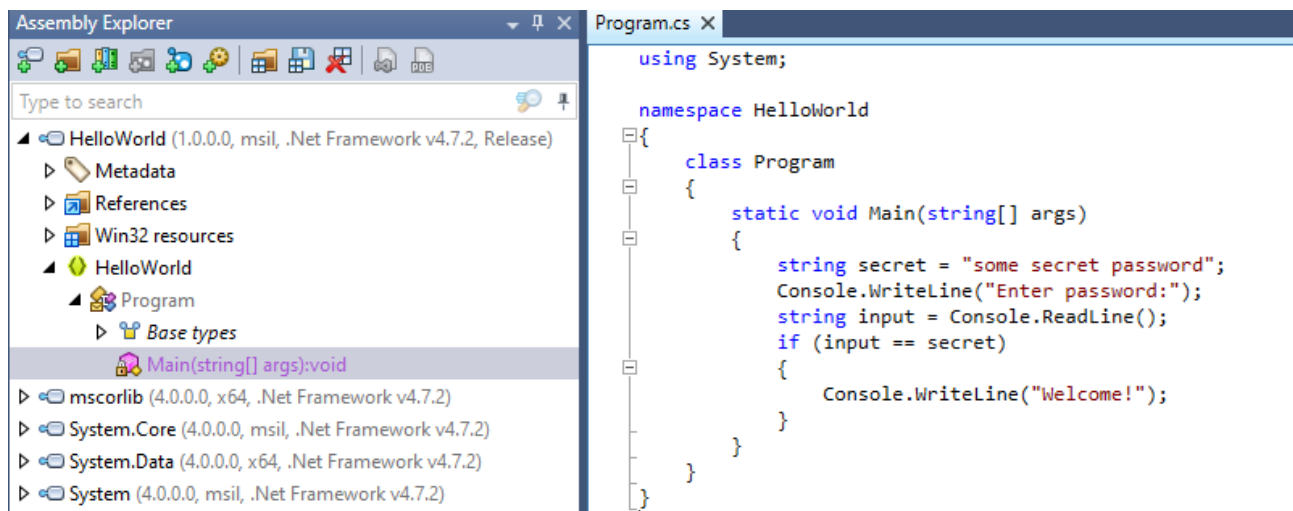
В результате рядом с файлом `HelloPatched.il` создастся файл `HelloPatched.exe`. Запустим его в терминале:

```
...HelloWorld\bin\Release>HelloPatched.exe  
Enter password:  
2345  
Welcome!
```

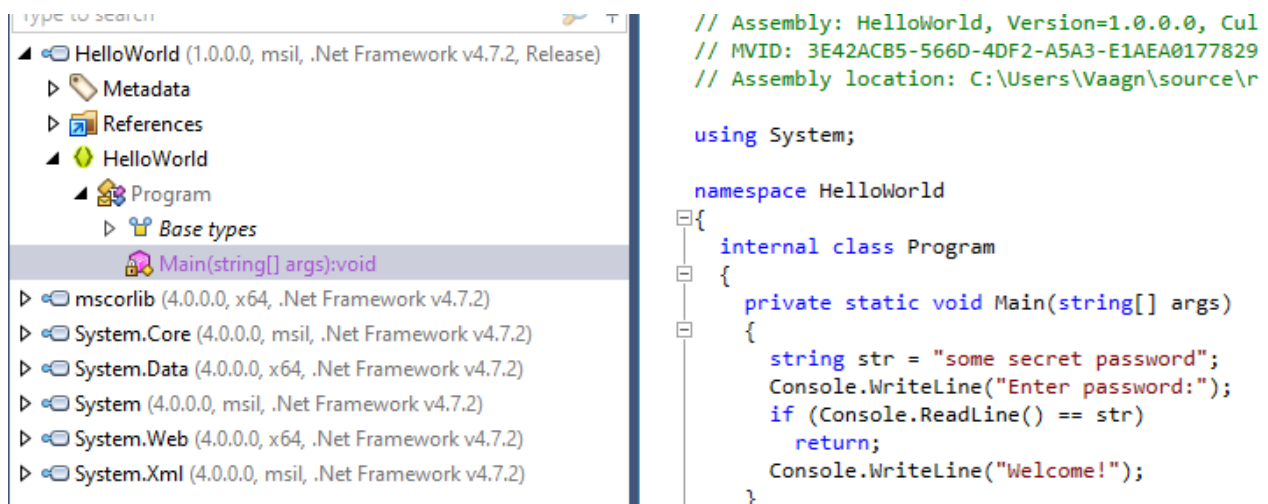
dotPeek

Другой, более удобный инструмент анализа исполняемых файлов — `dotPeek`. В отличие от `ILDasm`, `dotPeek` — это полноценный декомпилятор .NET-приложений, отображающий исходный код на языке C# или на другом языке семейства .NET. `dotPeek` позволяет генерировать полноценный проект Visual Studio на основе данных из двоичного файла программы.

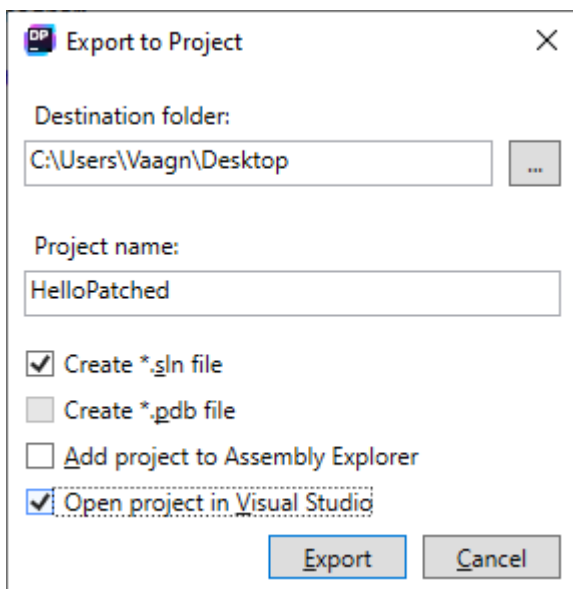
Перетащим `HelloWorld.exe` в окно `dotPeek` и посмотрим на содержимое `Program`:



Теперь добавим в dotPeek HelloPatched.exe, который мы собрали с помощью ilasm.exe в предыдущем разделе. Посмотрим, какой C#-код получился после наших правок в IL-коде:



Экспортируем сборку в проект Visual Studio. Для этого кликнем правой кнопкой мыши по сборке HelloWorld и выберем Export to Project... Затем выставим следующие настройки и нажмём на Export:



В результате сгенерированный проект откроется в окне Visual Studio.

Защита программного обеспечения

В предыдущем разделе мы выяснили, как довольно простым путём можно получить исходный код приложения из имеющейся сборки. Независимо от того, на каком языке разработана программа, она может быть декомпилирована с целью внесения нежелательных правок. Это порождает множество проблем, которые в целом сводятся к одной из следующих:

1. Платное программное обеспечение может быть взломано с целью получения всей функциональности без покупки лицензии.

Если алгоритм проверки лицензии размещён в самом приложении, а не на сервере, то недобросовестные пользователи могут обойти механизм проверки лицензирования. Они могут использовать полную версию ПО, не имея подлинного ключа продукта.

2. На основе нашего исходного кода конкуренты могут разработать альтернативное программное обеспечение. Некоторые форматы файлов не имеют публичной спецификации и могут быть обработаны только в специальных редакторах. Декомпиляция такого редактора может привести к серьёзным убыткам.

Чтобы усложнить анализ декомпилированного исходного кода, применяются специальные инструменты — обфускаторы (от англ. obfuscate — «запутывать»). Они нужны, чтобы сделать исходный код нечитаемым после декомпиляции.

Например, обфускаторы могут заменить понятные названия методов на однобуквенные названия a(), b(), c() или разделить строковые значения на несколько составляющих, чтобы их было сложнее найти поиском по файлу ("pa" "sswo" "rd" вместо "password") и многое другое. Наиболее распространённый

обфускатор для платформы .NET — Dotfuscator. Процесс установки обфускатора и примеры его использования вы найдёте в дополнительных материалах.

Сборка мусора

При разработке на языках, компилируемых напрямую в машинный код, зачастую приходится вручную управлять выделением и очищением памяти под ресурсы. Например, в языке C недостаточно просто объявить строку и присвоить ей значение.

Необходимо сначала выделить участок памяти из кучи, получить указатель на этот участок. Затем внести соответствующие данные в память и только потом сохранить эту ссылку в переменную. А после завершения работы со строкой или массивом его необходимо удалить из памяти, чтобы освободить её.

При разработке на языке C# не приходится вручную управлять выделением и очисткой памяти. Эти функции выполняет среда выполнения .NET. За освобождение памяти отвечает механизм под названием Garbage Collector или «Сборщик мусора».

При каждом создании объекта среда CLR выделяет память для него из управляемой кучи. Пока в управляемой куче есть доступное место, среда выполнения продолжает выделять пространство для новых объектов.

Тем не менее ресурсы памяти не безграничны. В конечном счёте сборщику мусора необходимо выполнить сбор, чтобы освободить память.

Механизм оптимизации сборщика мусора определяет наилучшее время для выполнения сбора, основываясь на выполненных операциях выделения памяти. Когда сборщик мусора выполняет сборку, он проверяет наличие в управляемой куче объектов, которые больше не используются приложением. Затем выполняет необходимые операции, чтобы освободить память.

Работа с большими строками

Как мы упоминали ранее, строковые данные — неизменяемые. Они хранятся в специальном пуле интернирования.

Каждый раз при изменении значения строки мы на самом деле создаём новый экземпляр объекта в памяти. Для него требуется выделение нового пространства. В случаях, когда необходимо выполнять повторяющиеся изменения строки, издержки, связанные с созданием объекта, могут оказаться значительными.

Чтобы избежать создания нового объекта, можно использовать класс `StringBuilder`, который может повысить производительность при приращении строк в цикле. Рассмотрим пример, в котором сравним время выполнения 30 тысяч операций приращения строки.

Сначала сделаем это обычным оператором прибавления `+=` , а затем с помощью класса `StringBuilder`. Для измерения времени будем использовать класс `StopWatch`. Он работает как обычный секундомер с кнопкой: его можно запустить, остановить и сбросить. Обратите внимание на необходимость подключения пространств имён:

```
using System;
using System.Diagnostics;
using System.Text;

using System;
using System.Diagnostics;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            string str = "";
            Stopwatch sw = new Stopwatch();
            sw.Restart();
            for (int i = 0; i < 30000; i++)
            {
                str += Guid.NewGuid();
            }
            sw.Stop();
            Console.WriteLine(sw.ElapsedMilliseconds);

            StringBuilder stringBuilder = new StringBuilder();
            sw.Restart();
            for (int i = 0; i < 30_000; i++)
            {
                stringBuilder.Append(Guid.NewGuid());
            }
            sw.Stop();
            Console.WriteLine(sw.ElapsedMilliseconds);
        }
    }
}
```

В качестве прибавляемого значения используем метод `Guid.NewGuid()`, который создаёт уникальный строковый идентификатор.

При запуске программы на ноутбуке с процессором Core i3 1.7 ГГц программа выдала следующий результат:

```
31223
35
```

Первый тест занял чуть больше 30 секунд, а второй — всего 35 миллисекунд. Если запустить тест несколько раз, значения будут немного меняться, но порядок останется тем же. Такие результаты говорят о высокой эффективности класса `StringBuilder`.

Но использование этого класса на небольших данных практически лишено смысла. В некоторых случаях это только усложнит ваш код. Поэтому стоит использовать `StringBuilder` по назначению.

Альтернативные способы сборки приложений

Сборку исходного кода необязательно производить в IDE. Хотя это и наиболее удобный способ ведения разработки, есть и другие способы преобразовать исходный код в исполняемую программу. Они могут быть полезны в основном для автоматизации сборок.

Например, мы можем добавить на сервер скрипт, который будет автоматически собирать приложение из исходного кода и загружать новую версию на наш сайт.

Компилятор `csc.exe`

Наиболее примитивный способ сборки заключается в вызове компилятора `csc` с передачей ему файлов с исходным кодом. Как и ранее, вызов утилит разработчика необходимо производить в специальном терминале Developer Command Prompt for VS 2019:

```
cd C:\...\HelloWorld\HelloWorld # переход в папку с проектом
csc Program.cs
```

В результате мы получим одноимённый исполняемый файл в папке с проектом.

Если программа состоит из нескольких файлов, необходимо передать их все в любом порядке. Допустим, наше приложение состоит из двух файлов:

```
// Greeting.cs
namespace HelloWorld
{
    class Greeting
    {
        public static void SayHello()
        {
            System.Console.WriteLine("Hello there!");
        }
    }
}
```



```
}
```

```
// Program.cs
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Greeting.SayHello();
        }
    }
}
```

Команда сборки этой программы:

```
csc Greeting.cs Program.cs
```

Независимо от порядка файлов выходной файл будет назван по имени файла, содержащего точку входа — `Program.exe`

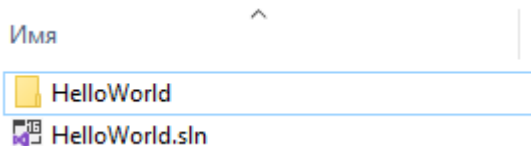
MSBuild

Сборка больших проектов с помощью компилятора может быть утомительной. Нам придётся каждый раз обновлять команду сборки, добавляя в аргументы компилятора новые файлы с исходным кодом.

Для удобной и гибкой сборки проектов .NET есть специальная платформа MSBuild. На самом деле при сборке проектов в Visual Studio также используется MSBuild. Мы можем использовать его напрямую.

В отличие от `csc.exe`, MSBuild собирает не отдельные файлы исходного кода, а полноценные проекты или даже решения. Решение — это совокупность нескольких проектов.

Когда мы создаём новый проект в Visual Studio, у нас автоматически создаётся файл проекта с расширением `.csproj`. По умолчанию Visual Studio дополнительно создаёт файл решения (`.sln`). В итоге получается следующая структура файлов: в корне расположен файл решения, а сам проект находится в одноимённой папке.



Структура папки с проектом:

Имя	Дата изменения	Тип	Размер
bin	06.12.2020 13:10	Папка с файлами	
obj	06.12.2020 13:10	Папка с файлами	
Properties	24.10.2020 14:38	Папка с файлами	
App.config	24.10.2020 14:38	XML Configuratio...	1 КБ
Greeting.cs	06.12.2020 17:00	Visual C# Source F...	1 КБ
HelloWorld.csproj	06.12.2020 16:22	Visual C# Project f...	4 КБ
HelloWorld.csproj.user	06.12.2020 16:21	Per-User Project O...	1 КБ
Program.cs	07.12.2020 20:35	Visual C# Source F...	1 КБ

Запустим сборку решения с помощью MSBuild. Для этого откроем уже знакомую нам консоль с утилитами разработчика Developer Command Prompt for VS 2019:

```
cd C:\...\HelloWorld# переход в папку с решением
msbuild HelloWorld.sln
```

В результате запустится сборка решения в конфигурации Debug (конфигурация по умолчанию). Для сборки в конфигурации Release нужно передать дополнительный аргумент:

```
msbuild HelloWorld.sln -p:Configuration=Release
```

Поиск источников информации

Умение найти необходимую информацию — один из ключевых навыков успешного программиста. Разработка качественного программного обеспечения требует от разработчика обширных знаний в разных областях: программирование, тестирование, отладка, архитектура, алгоритмы, стандартные библиотеки и фреймворки.

Разумеется, никто не ожидает от разработчика знания наизусть всех особенностей языка или стандартных классов. Это было бы неэффективно.

Информация из мира технологий имеет свойство устаревать, и разработчикам приходится постоянно поддерживать актуальность своих знаний. Именно поэтому важно уметь оперативно находить

информацию об ошибке, с которой вы столкнулись. Или отслеживать состав изменений в последней версии используемой библиотеки.

При этом полезно быть осведомлённым в фундаментальных знаниях, которые вряд ли будут претерпевать изменения в ближайшем будущем. Например, сетевые протоколы, синтаксис языка, основы реляционных баз данных сохраняют свою актуальность уже десятки лет. Конечно, эти сферы прогрессируют и пополняются множеством новых технологий, но все они строятся вокруг основ.

Наиболее удобные инструменты для поиска информации — как ни странно, поисковые сервисы. Можно выбрать любой подходящий под ваши критерии поисковик, будь то Google, Яндекс или DuckDuckGo. Главной рекомендацией будет построение поисковых фраз на английском языке.

Во-первых, большинство форумов и руководств написаны на английском. Во-вторых, вы сможете увеличить свой словарный запас.

Для большего удобства стоит также использовать англоязычную IDE и инструменты. Это поможет мыслить в рамках одного языка и эффективнее искать решения. Согласитесь, намного проще скопировать код ошибки из англоязычной IDE и использовать его в качестве поисковой фразы, чем угадывать оригинальные значения переведённых терминов.

Ранее мы уже упоминали наиболее популярный форум по технологиям — StackOverflow. Скорее всего, если вы будете искать решение какой-нибудь проблемы, первые результаты поиска приведут вас напрямую на этот форум. В противном случае можно перейти на [StackOverflow](#) и использовать поиск по сайту.

Google предоставляет дополнительный инструмент для расширенного поиска информации [Google Advanced Search](#). Его можно использовать, если обычный поиск выдаёт неточные результаты.

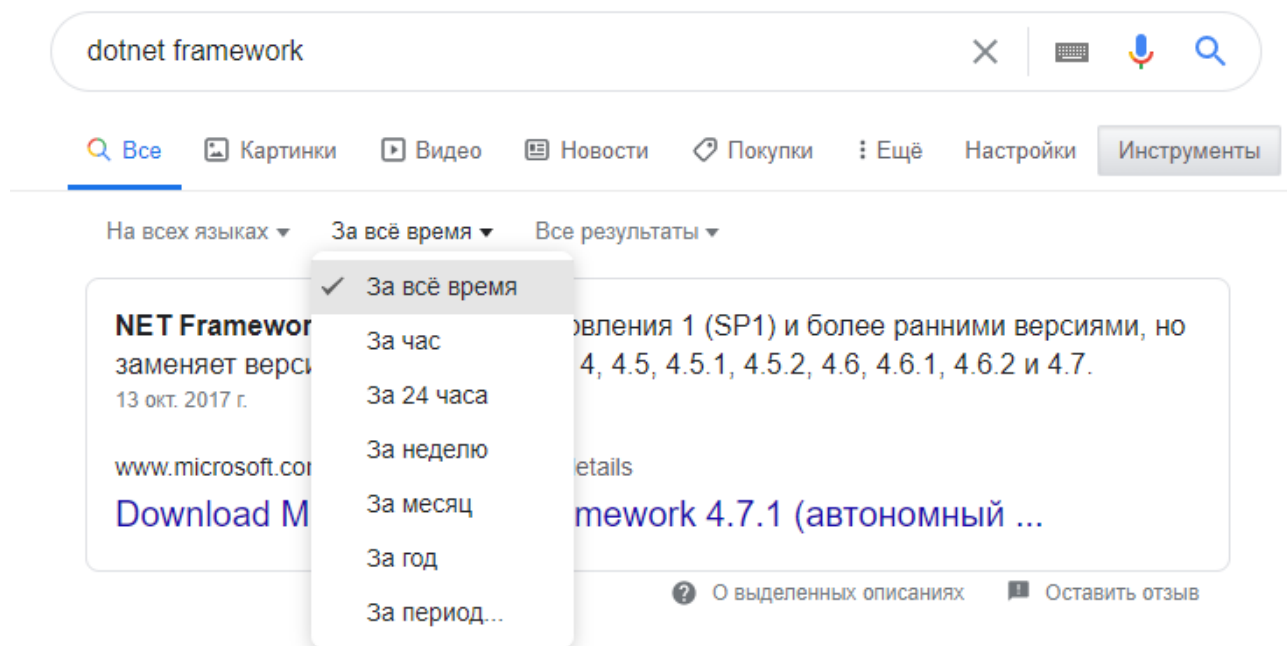
Можно включать фильтры напрямую в поисковом запросе. Например, чтобы указать обязательное вхождение слова, его нужно заключить в кавычки. А чтобы исключить результаты, содержащие какое-либо слово, необходимо добавить дефис перед этим словом.

Этот запрос в Google выведет обладателей фамилии Толстой, исключив при этом Л. Н. Толстого:

ТОЛСТОЙ -лев

По умолчанию поиск не зависит от регистра.

При поиске технической информации полезно задавать временный интервал. Мы можем указать, что хотим получать результаты за определённый период:



Важно помнить, что разработка — довольно сложная, но интересная профессия. Вы непременно будете сталкиваться со множеством ошибок. И с увеличением опыта сложность ошибок также будет возрастать. Этот неизбежный процесс не должен вас пугать или мешать прогрессу. Помните, что, скорее всего, кто-то когда-то уже столкнулся с такой же ошибкой или с подобной задачей. Он наверняка спросил об этом на каком-либо форуме или написал статью в блоге.

В дополнительных материалах к этому уроку приведены популярные ресурсы, связанные с миром IT.

Практическое задание

1. Написать любое приложение, произвести его сборку с помощью MSBuild, осуществить декомпиляцию с помощью dotPeek, внести правки в программный код и пересобрать приложение.
2. (*) выполнить задание 1, используя вместо dotPeek инструменты ildasm/ilasm.

Дополнительные материалы

1. [Dotfuscator Community — Visual Studio](#).
2. [Хабр](#).
3. [itProger — сообщество программистов](#).
4. [Techrocks.ru | программирование, стартапы, новости IT](#).
5. [Technical documentation](#).
6. [Техническая документация](#).

7. [Повышаем продуктивность работы программиста: советы по поиску в Google.](#)

Используемые источники

1. [Описание IL-инструкций.](#)
2. [List of CIL instructions.](#)
3. [Ilasm.exe \(ассемблер IL\).](#)
4. [Ildasm.exe \(дизассемблер IL\).](#)