# Platforma .NET

## 2. seminář

Radek Janoštík

KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

# Reakce na úkoly

- Vesměs v pořádku
- Ošetření výjimečných stavů
- Datové struktury

# Pointery v C#

- Velmi podobné jako v C a C++
- Prakticky se moc nepoužívají (je dobré o tom vědět)
- Je potřeba explicitně říci, že budeme používat pointery – klíčové slovo `unsafe`
- Je potřeba kompilátoru předat `/unsafe`
  - VisualStudio – Project → Properties → Build → Allow unsafe code
  - Rider – Project → Properties → Configurations → Allow unsafe code
- Použití pří optimalizaci kódu (příklad – grafika - bitmapa)

# Operátory s pointery

| Operátor | Popis |
|---|---|
| `*` | Vytvoření pointeru, operátor dereference |
| `&` | Získání adresy proměnné v paměti |
| `->` | Přístup k fieldům typu reprezentovaného pointerem (unsafe verze . ) |
| `[]` | Indexování v pointeru |
| `++, --` | Inkrementace, dekrementace |
| `+, -` | Sčítání, odečítání pointerů |
| `==, !=, <, >, <=, >=` | Porovnávání pointerů |
| `stackalloc` | Přímé alokování Pole na zásobníku |
| `fixed` | Dočasně zafixuje adresu pointeru (GC ji nezmění) |

# Klíčové slovo `unsafe`

- Slouží k vymezení bloku, kde se bude pracovat s pointery

```
1  unsafe {
2     int cislo;
3     int* pointerNaCislo = &cislo;
4     *pointerNaCislo = 987;
5  }
6
7  unsafe struct Node {
8     public int Value;
9     public Node* Left;
10    public Node* Right;
11 }
```

```
struct Node2 {
   public int Value;
   public unsafe Node2* Left;
   public unsafe Node2* Right;
}
```

- `Value` Node2 bude přístupná i mimo `unsafe` blok

# Fixed a moveable proměnné

- *Fixed* (pevné, nemovité) proměnné – jejich adresu nezmění *garbage collector (GC)*
- *Moveable* (přesunutelné, movité) – *GC* je může přesunout či odmazat
- Operátor `&` umí získat adresu pouze *fixed* proměnné
- Adresu *moveable* můžeme dočasně zafixovat pomocí klíčového slova: `fixed`
- *Fixed proměnné* jsou (zjednodušeně):
  - Lokální proměnné, hodnotové parametry metod
  - Proměnné vzniklé z *member access* – `V.I`, kde `V` je fixed struct
  - Hodnoty z dereference (`*P`), *pointer member access* (`P->I`) a *pointer element access* (`P[I]`)
- Ukázka

# Alokování paměti na zásobníku

- Alokuje se přímo na callstacku, nezasahuje do nich GC
- Klíčové slovo `stackalloc`

```
1 char* p = stackalloc char[256];
2 for (int i=0; i<256; i++) {
3    p[i] = (char)i;
4 }
5
6 for (int i = 0; i < 256; i++) {
7    Console.WriteLine(p[i]);
8 }
```

# Klíčové slovo `sizeof`

- `sizeof()` vrátí velikost *interního* typu v paměti (v bajtech)
- Mimo `unsafe` nefunguje pro uživatelsky definované typy

```
1 Console.WriteLine($"Velikost shortu: {sizeof(short)}");
2 Console.WriteLine($"Velikost intu: {sizeof(int)}");
3 Console.WriteLine($"Velikost longu: {sizeof(long)}");
4 Console.WriteLine($"Velikost floatu: {sizeof(float)}");
5 Console.WriteLine($"Velikost doublu: {sizeof(double)}");
6 unsafe {
7   Console.WriteLine($"Velikost Node: {sizeof(Node)}");
8 }
```

# Common Intermediate Language (CIL)

- Veškeré ukázky budou pro jednoduchost provedeny ve starém .NET Frameworku
- Pokud někdo nemáte Windows, můžete použít vzdálenou plochu na `ts.inf.upol.cz`
- V .NET možné také, horší podpora (3rd party knihovny)
  - CIL kód již není ve `.exe` souboru ale v `.dll`
  - V `.exe` souboru je pouze platformově specifická „obálka“, která načte kód z `.dll`
- V Rideru k dispozici ILViewer

# Common Intermediate Language (CIL)

- *Common Intermediate Language (CIL)* – interní jazyk platformy .NET
- Všechny ostatní jazyky se do něj kompilují
- Komplexní jazyk umožňující veškerou funkcionalitu (plná syntax, sémantika, kompilátor)
- Dobré znát pro optimalizace
- Zásobníkový jazyk – PAPR2, slidy doc. Krupky: `https://apollo.inf.upol.cz/~janostik/slides/papr2-11.pdf`
- Zkráceně: Operátory své argumenty hledají vždy na vrcholu zásobníku, odkud je odeberou a zpracují
- Na první pohled podobný x86 assembleru
- Operátory mají své (interní) číselné kódy, ale i lidsky čitelné zkratky

# ildasm.exe – disassembler

- Ve VisualStudiu si otevřeme Developer PowerShell (View→Terminal)
- Spustíme `ildasm.exe cesta\k\exe`
- Objeví se nám dekompilované `.exe` s naším projektem
- Můžeme si prohlížet zdrojový kód v CIL

```
1 static void Main(string[] args) {
2   int a = 6;
3   int b = 36;
4   int c = a + b;
5 }
```

- Se přeloží na:

# ildasm.exe – přeložený kód

```
.method private hidebysig static void  Main(string[] args) cil managed {
  .entrypoint
  // Code size       11 (0xb)
  .maxstack  2
  .locals init ([0] int32 a,
           [1] int32 b,
           [2] int32 c)
  IL_0000:  nop
  IL_0001:  ldc.i4.6
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.s   36
  IL_0005:  stloc.1
  IL_0006:  ldloc.0
  IL_0007:  ldloc.1
  IL_0008:  add
  IL_0009:  stloc.2
  IL_000a:  ret
} // end of method Program::Main
```

# Co vidíme v kódu

- *CIL direktivy* – tokeny popisující celkovou strukturu .NET sestavení
  - Začínají tečkou (např. `.namespace`, `.class`, ...
  - Často vymezují blok kódu (`.namespace MujNamespace {}`
- *CIL atributy* – rozšiřují jednotlivé direktivy
  - např.: `public` atribut u `.class` (viditelnost)
  - `extends` (dědičnost)
- *CIL Opcodes* – Operační kódy, kódy operátorů, ale spíše zkratky jmen
  - `ldc.i4.6`
  - `stloc.0`
- *Labels* – nepovinné návěstí, na které se dá odkazovat ve skoku
  - Dekompilátor automaticky doplňuje hexa číslo
  - `IL_0000:`, `IL_000a:`
  - Můžou být libovolné

# Co se děje v kódu (1/2)

- `.method private hidebysig static void Main(string[] args) cil managed {`
  Direktiva označující metodu, s atributy, jménem a seznamem argumentů
- `.entrypoint` – vstupní bod sestavení – `main`
- `.maxstack 2` – říkáme, kolik polí na zásobníku použijeme (výchozí: 8)
- `.locals init ([0] int32 a, [1] int32 b, [2] int32 c)` – Deklarace lokálních proměnných, musí být na začátku, číslováno od nuly.
  - V nestatické třídě je vždy nultým argumentem reference na aktuální objekt – `this`
- `IL_0000: nop` – definice návěstí `IL_0000` a operace „Do Nothing“, „No Operation“
  - Použita v Debug režimu, podpora breakpointů
- `IL_0001: ldc.i4.6` – Operace „Push 6 onto the stack as int32“
- `IL_0002: stloc.0` – Operace „Pop a value from stack into local variable 0“ – do lokální proměnné `a` se vložil vrchol zásobníku

# Co se děje v kódu (2/2)

- `IL_0003: ldc.i4.s 36` – Operace „Push num onto the stack as int32, short form“
- `IL_0005: stloc.1` – Operace „Pop a value from stack into local variable 1“ – do lokální proměnné `b` se vložil vrchol zásobníku
- `IL_0006: ldloc.0` – Operace „Load local variable 0 onto stack“
- `IL_0007: ldloc.1` – Operace „Load local variable 1 onto stack“ – načtení operandů pro součet
- `IL_0008: add` – Operace „Add two values, returning a new value“ – na vrcholu bude hodnota součtu
- `IL_0009: stloc.2` – Operace „Pop a value from stack into local variable 2“ – uložení výsledku do `c`
- `IL_000a: ret` – Operace „Return from method, possibly with a value“

# Základní operace

| Opcodes | Význam |
|---|---|
| `add, sub, mul, div, rem` | Základní aritmetické operace |
| `and, or, not, xor` | Bitové operace |
| `ceq, cgt, clt` | Porovnání na rovnost, větší než, menší než |
| `ret` | Ukončení metody, s návratovou hodnotou |
| `beq, bgt, ble, blt` | operace pro větvení programu: skoč na návěstí případě rovnosti, v případě větší než, v případě menší rovno, v případě menší než |
| `call` | Zavolání metody |
| `pop` | Odebere prvek ze zásobníku |
| `ldarg, ldstr` | Vloží argument metody na zásobník, řetězec na zásobník |
| `starg, stloc` | Vloží vrchol zásobníku do argumentu na indexu, lokální proměnné |

- Seznam operací např. na `https://en.wikipedia.org/wiki/List_of_CIL_instructions`

# Programování v CIL (1/3)

- Otevřeme svůj oblíbený textový editor
- Vytvoříme soubor `test.il` a můžeme začít programovat
- Kostra:

```
.assembly extern mscorlib {
.publickeytoken = (B7 7A 5C 56 19 34 E0 89)
.ver 4:0:0:0
}

.assembly MujAssebly {
.ver 1:0:0:0
}

.namespace MujCilSpace {
 .class public MyClass {
  .method private hidebysig static void Main(string[] args) cil managed {
   .entrypoint
   .maxstack 8
   // Tady můžeme programovat
  }
```

# Programování v CIL (2/3)

- Zjištění public key knihovny:

```
> sn.exe /T "C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NE
Microsoft (R) .NET Framework Strong Name Utility  Version 4.0.30319.0
Copyright (c) Microsoft Corporation.  All rights reserved.
Public key token is b77a5c561934e089
```

- Kompilace:

```
> ilasm.exe /exe .\test.il /output=Program.exe
Microsoft (R) .NET Framework IL Assembler.  Version 4.7.3190.0
Copyright (c) Microsoft Corporation.  All rights reserved.
Assembling '.\test.il'  to EXE --> 'Program.exe'
Source file is UTF-8
Assembled method MujCilSpace.MyClass::Main
Creating PE file
Emitting classes:
Class 1:        MujCilSpace.MyClass
Emitting fields and methods:
Global
Class 1 Methods: 1;
Emitting events and properties:
Global
Class 1
Writing PE file
Operation completed successfully
```

# Programování v CIL (3/3)

- `ilasm.exe` je pouze kompilátor, nekontroluje význam operací
- Nekontroluje, zda je na konci volání funkce prázdný zásobník
- Naštěstí máme nástroj `peverify.exe`, který toto umí zkontrolovat

```
> ilasm.exe /exe .\test.il /output=Program1.exe
...
Global
Writing PE file
Operation completed successfully

> PEVerify.exe .\Program1.exe

Microsoft (R) .NET Framework PE Verifier.  Version  4.0.30319.0
Copyright (c) Microsoft Corporation.  All rights reserved.

[IL]: Error: [C:\Users\janora01\source\repos\CIL\CIL\Program1.exe :
          MujCilSpace.MyClass::Main][offset 0x0000000B] Unable to resolve token.
[IL]: Error: [C:\Users\janora01\source\repos\CIL\CIL\Program1.exe :
          MujCilSpace.MyClass::JumpTest][offset 0x0000001A] Stack underflow.
2 Error(s) Verifying .\Program1.exe
```

# CIL – volání metod

- Argumenty nejdříve uložíme na zásobník
- Poté můžeme využít operaci `call`
- Musíme uvést celou cestu k metodě + assembly
- Ukázka

# CIL – větvení

- Větvení umožněno pomocí operací se skokem. Příklad:
- `blt IL_0009` – porovná dvě hodnoty na zásobníku a pokud je první menší než druhá, skočí na návěstí `blt IL_0009`
- Takto můžeme implementovat cykly (vzpomeňte na x86 assembler)
- Operace `br` – „Branch to target“ skáče bez podmínky

# Závěr

- Na tomto semináři CIL probrán pouze „letmo“
- Doporučuji si k tomu přečíst kapitoly v doporučené literatuře
- Neprobrána spousta věcí
  - Datové typy
  - Práce s třídami, Interface, dědičnost, properties
  - ...
- Pouze abychom měli trochu tušení, co se děje „pod kapotou“

# Úkol

1. V C# naprogramujte Euklidův algoritmus pro zjištění největšího společného dělitele
   - Zjistěte jeho CIL kód
   - Řádek po řádku okomentujte, co se děje
2. Bez spuštění zjistěte co dělá kód v souboru
   `https://apollo.inf.upol.cz/~janostik/code/MysteriousMethod.il`
3. V CIL naprogramujte metodu, která vypočítá diskriminant kvadratické rovnice
4. V CIL naprogramujte metodu `FizzBuzz(int limit)`, která:
   - Vypíše čísla od 1 do `limit`
   - Je-li číslo dělitelné 3 bude nahrazeno řetězcem „Fizz“ a
   - Je-li číslo dělitelné 5 bude nahrazeno řetězcem „Buzz“
   - Více na: `https://en.wikipedia.org/wiki/Fizz_buzz`
   - Příklad: `1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz Buzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, Fizz, Buzz, 26, Fizz, 28, 29, Fizz Buzz, 31, 32, Fizz, 34, Buzz, Fizz, ...`