Variable scope/reusage in the MSIL code

Asked 13 years, 5 months ago Modified 13 years, 5 months ago Viewed 371 times

3

While debugging some C# code during a sort of peer-review, I noticed an odd behavior that seemed at first to be some sort of scoping violation, but in retrospect looks like perhaps the compiler attempting to save on memory by reusing references. The code is:

```
for(int i = 0; i < 10; i++)
{
    // Yadda yadda, something happens here
}

// At this point, i is out of scope and is not
// accessible. This is verified by intellisense
// and by attempting to look at the variable
// during debug
string whatever = "";

// At this point if I put a break on the following
// for line, I can look at the variable I before
// it is initialized and see that it already holds
// the value of 10. If a different variable name
// is used, I get a value of 0 (not initialized).
for(int i = 0; i < 10; i++)
{
    // Inside the loop, i has been re-initialized
    // so it performs its function as expected
}
```

Is the compiler simply reusing an existing reference? In C/C++ where variables/references need to be managed more closely, this would be behavior I would sort of expect. With C# I was under the impression that each time a variable was declared within the scope of a loop that it would partition out a new separate section of memory, but obviously that's not the case. Is this a memory saving feature, potentially a hold-over from C/C++ behaviors or is this case simply ignored since the compiler forces you to reinitialize anyway?

~~**Edit:**~~
~~Some things I've noticed in just doing some other checks is that this behavior is not exhibited across methods within a class. It does appear across multiple~~ `using` ~~statements, but only does this if the type and name are the same.~~

~~Upon further investigation, I'm beginning to believe this is less about the MISL code than it is about the IDE retaining these references in its own memory. I've seen nothing to indicate that this behavior would actually exist at the code level, and so now I'm leaning towards the idea that this is simply a quirk of the IDE.~~

Edit 2:
It looks like the answer from @Vijay Gill has disproved the IDE quirk.

c# scope

Share
Improve this question
Follow

edited Jul 27, 2011 at 15:41

asked Jul 26, 2011 at 12:34

Joel Etherton
37.5k ● 10 ● 91 ● 105

2 Answers

Sorted by: Highest score (default)

3

It totally depends upon the compiler and what configuration you are using for compilation. In the following text dump, you can see that in Release mode, two int variables are declared where-as in dubug mode, only one.

Why it does so it totally beyond me (for the time being, I will investigate more when I go home)

Edit: See more findings near end of this answer

```
    private static void f1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("Loop 1");
        }

        Console.WriteLine("Interval");

        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("Loop 2");
        }
    }
```

Release mode: (note the local variables i & V_1)

```
.method private hidebysig static void  f1() cil managed
{
  // Code size       57 (0x39)
  .maxstack  2
  .locals init ([0] int32 i,
           [1] int32 V_1)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  br.s       IL_0012
  IL_0004:  ldstr      "Loop 1"
```

```
  IL_0009:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000e:  ldloc.0
  IL_000f:  ldc.i4.1
  IL_0010:  add
  IL_0011:  stloc.0
  IL_0012:  ldloc.0
  IL_0013:  ldc.i4.s   10
  IL_0015:  blt.s      IL_0004
  IL_0017:  ldstr      "Interval"
  IL_001c:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0021:  ldc.i4.0
  IL_0022:  stloc.1
  IL_0023:  br.s       IL_0033
  IL_0025:  ldstr      "Loop 2"
  IL_002a:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_002f:  ldloc.1
  IL_0030:  ldc.i4.1
  IL_0031:  add
  IL_0032:  stloc.1
  IL_0033:  ldloc.1
  IL_0034:  ldc.i4.s   10
  IL_0036:  blt.s      IL_0025
  IL_0038:  ret
} // end of method Program::f1
```

Debug Mode: (note the local variable i)

```
.method private hidebysig static void  f1() cil managed
{
  // Code size       73 (0x49)
  .maxstack  2
  .locals init ([0] int32 i,
           [1] bool CS$4$0000)
  IL_0000:  nop
  IL_0001:  ldc.i4.0
  IL_0002:  stloc.0
  IL_0003:  br.s       IL_0016
  IL_0005:  nop
  IL_0006:  ldstr      "Loop 1"
  IL_000b:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0010:  nop
  IL_0011:  nop
  IL_0012:  ldloc.0
  IL_0013:  ldc.i4.1
  IL_0014:  add
  IL_0015:  stloc.0
  IL_0016:  ldloc.0
  IL_0017:  ldc.i4.s   10
  IL_0019:  clt
  IL_001b:  stloc.1
  IL_001c:  ldloc.1
  IL_001d:  brtrue.s   IL_0005
  IL_001f:  ldstr      "Interval"
  IL_0024:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0029:  nop
  IL_002a:  ldc.i4.0
  IL_002b:  stloc.0
  IL_002c:  br.s       IL_003f
  IL_002e:  nop
```

```
  IL_002f:  ldstr      "Loop 2"
  IL_0034:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0039:  nop
  IL_003a:  nop
  IL_003b:  ldloc.0
  IL_003c:  ldc.i4.1
  IL_003d:  add
  IL_003e:  stloc.0
  IL_003f:  ldloc.0
  IL_0040:  ldc.i4.s   10
  IL_0042:  clt
  IL_0044:  stloc.1
  IL_0045:  ldloc.1
  IL_0046:  brtrue.s   IL_002e
  IL_0048:  ret
} // end of method Program::f1
```

Assembly code generated is given below. This is for the IL compiled in Release mode only. Now even in the machine language (disassembled here) I see that two local variables are created. I could nto find any answer to that. Only MS guys can tell us. But this behaviour is very important to remember when we write recursive methods, in relation to the stack usage.

```
00000000  push        ebp
00000001  mov         ebp,esp
00000003  sub         esp,0Ch
00000006  mov         dword ptr [ebp-4],ecx
00000009  cmp         dword ptr ds:[04471B50h],0
00000010  je          00000017
00000012  call        763A4647

-- initialisation of local variables
-- this is why we get all ints set to zero initially (will see similar
behavioir for other types too)
00000017  xor         edx,edx
00000019  mov         dword ptr [ebp-8],edx
0000001c  xor         edx,edx
0000001e  mov         dword ptr [ebp-0Ch],edx

00000021  xor         edx,edx -- zero out register edx which will be saved to
memory where i (first one) is located
00000023  mov         dword ptr [ebp-8],edx -- initialise variable i (first
one) with 0
00000026  nop
00000027  jmp         00000037 -- jump to the loop condition

00000029  mov         ecx,dword ptr ds:[01B32088h]
0000002f  call        76A84E7C -- calls method to print the message "Loop 1"

00000034  inc         dword ptr [ebp-8] -- increment i (first one) by 1
00000037  cmp         dword ptr [ebp-8],0Ah  -- compare with 10
0000003b  jl          00000029 -- if still less, go to address 00000029

0000003d  mov         ecx,dword ptr ds:[01B3208Ch]
00000043  call        76A84E7C -- prints the message "Half way there"

00000048  xor         edx,edx  -- zero out register edx which will be saved to
memory where i (second one) is located
```

```
0000004a  mov         dword ptr [ebp-0Ch],edx -- initialise i (second one) with
0
0000004d  nop
0000004e  jmp         0000005E -- jump to the loop condition

00000050  mov         ecx,dword ptr ds:[01B32090h]
00000056  call        76A84E7C -- calls method to print the message "Loop 1"

0000005b  inc         dword ptr [ebp-0Ch]  -- increment i (second one) by 1
0000005e  cmp         dword ptr [ebp-0Ch],0Ah -- compare with 10
00000062  jl          00000050  -- if still less, go to address 00000050


00000064  nop
00000065  mov         esp,ebp
00000067  pop         ebp
00000068  ret
```

Share

Improve this answer

Follow

edited Jul 27, 2011 at 9:31

answered Jul 26, 2011 at 13:18

Vijay Gill

1,518 ● 1 ● 14 ● 16

it depends on the switch "Optimize Code". If the Optimized code switch is disabled in the Release Configuration it reacts as in Debug more. – fixagon Jul 26, 2011 at 13:49

That I saw too :) my main reason of "more investigation" is to know "why using two variables is more optimum than one?". – Vijay Gill Jul 26, 2011 at 13:52

This is really interesting. I was leaning towards this being an IDE quirk, but looking at your posts here this might not be at all. – Joel Etherton Jul 26, 2011 at 14:20

Not it seems that this behavior extends right upto generation of machine language. I cannot put formatted code here in comments so I am putting it in the answer itself. Look at the end of my answer above. – Vijay Gill Jul 27, 2011 at 9:26

3

it has to be like that, that the compiler reuses the same variable: (it was already most probable with your example, but just to show that truly the same address is used...)

proof: (both variables share the same memory address)

```
public unsafe void test()
{
    for (int i = 0; i < 10; i++)
    {
        // Yadda yadda, something happens here
        int* ptr = &i;
        IntPtr addr = (IntPtr)ptr;
        if (i == 9)
        {
            Console.WriteLine(addr.ToString("x"));
            MessageBox.Show(addr.ToString("x"));
        }
    }
```

```
    for (int i = 0; i < 10; i++)
    {
        int* ptr = &i;
        IntPtr addr = (IntPtr)ptr;
        if (i == 9)
        {
            Console.WriteLine(addr.ToString("x"));
            MessageBox.Show(addr.ToString("x"));
        }
    }
}
```

it would be interresting to see the decompiled version.

Share

Improve this answer

Follow

edited Jul 26, 2011 at 13:08

answered Jul 26, 2011 at 13:01

fixagon

5,566 ● 23 ● 26

could garbage collection happen somewhere in between? would this make any difference? – Matt Jul 26, 2011 at 13:04

2 Variable "i" is value type and is created on stack, so GC will never touch that and its memory will be reclaimed when the method finishes. – Vijay Gill Jul 26, 2011 at 13:41