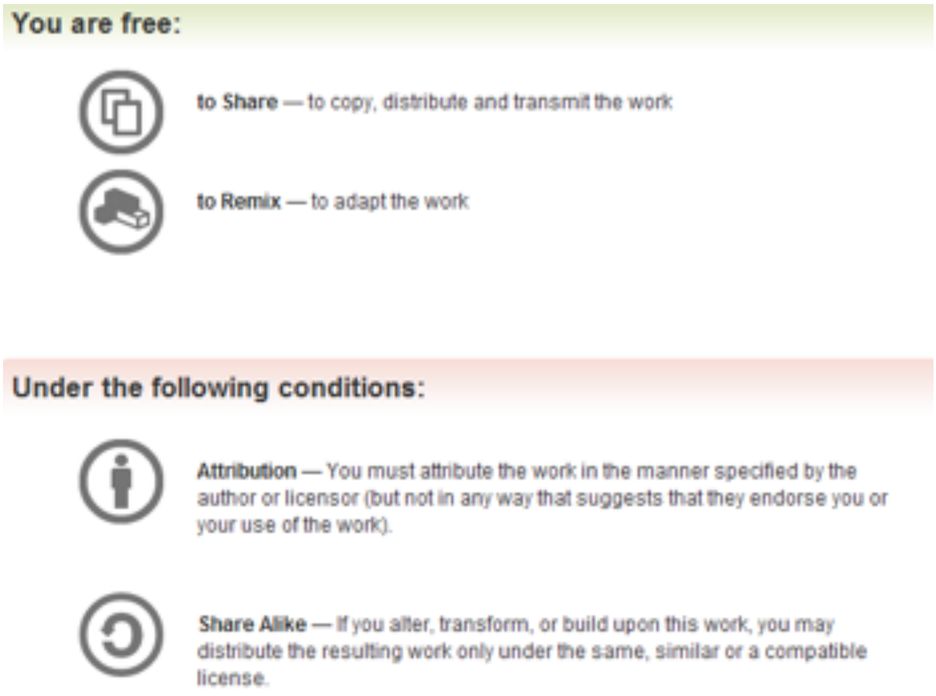
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014-2015

xeno@legbacore.com

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html”

Attribution condition: You must indicate that derivative work

"Is derived from Xeno Kovah's ‘Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html"

ExampleSubroutine1.c

The stack frames in this example will be very simple.
Only saved return addresses (RIP).

```
//ExampleSubroutine1:
// using the stack & subroutine
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int func(){
    return 0xbeef;
}
int main(){
    func();
    return 0xf00d;
}
```

```
func:
0000000140001000  mov         eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub         rsp,28h
0000000140001014  call        func (0140001000h)
0000000140001019  mov         eax,0F00Dh
000000014000101E  add         rsp,28h
0000000140001022  ret
```

4

CALL - Call Procedure

- CALL’s job is to transfer control to a different function, in a way that control can later be resumed where it left off
- First it pushes the address of the next instruction onto the stack
  - For use by RET for when the procedure is done
- Then it changes RIP to the address given in the instruction
- Destination address can be specified in multiple ways
  - Absolute address
  - Relative address (relative to the end of the instruction, or some other register)

Book page 108

RET - Return from Procedure

- Two forms
  - Pop the top of the stack into RIP (remember, *pop* increments stack pointer, RSP)
    - In this form, the instruction is just written as “ret”
  - Pop the top of the stack into RIP and also add a constant number of bytes to RSP
    - In this form, the instruction is written as “ret 0x8”, or “ret 0x20”, etc

Book page 109

Intel vs. AT&T Syntax

(we'll come back to this again much later)

- Intel: Destination <- Source(s)
  - Windows. Think algebra or C: y = 2x + 1;
  - mov rbp, rsp
  - add rsp, 0x14 ; (rsp = rsp + 0x14)
- AT&T: Source(s) -> Destination
  - *nix/GNU. Think elementary school: 1 + 2 = 3
  - mov %rsp, %rbp
  - add $0x14,%rsp
  - So registers get a % prefix and immediates get a $
- My classes will use Intel syntax except in this section
- But it's important to know both, so you can read documents in either format.

6

MOV - Move

- Can move:
  - register to register
  - memory to register, register to memory
  - immediate to register, immediate to memory
- Never memory to memory!
- Memory addresses are given in r/mX form talked about next

Book page 48

“r/mX” Addressing Forms

- Anywhere you see an r/mX it means it could be taking a value either from a register, or a memory address.
- I’m just calling these “r/mX forms” because anywhere you see “r/m16”, “r/m32”, or “r/m64” in the manual, the instruction can be a variation of the below forms.
- In Intel syntax, most of the time square brackets [] means to treat the value within as a memory address, and fetch the value at that address (like dereferencing a pointer)
  - mov rax, rbx
  - mov rax, [rbx]
  - mov rax, [rbx+rcx*X] (X=1, 2, 4, 8)
  - mov rax, [rbx+rcx*X+Y] (Y= one byte, 0-255 or 4 bytes, 0-2^32-1)
- Most complicated form is: `[base + index*scale + disp]`

Book page 123

More info: Intel v2a, Section 2.1.5 page 2-4
in particular Tables 2-2 and 2-3

7 8

ADD and SUB

- Adds or Subtracts, just as expected
- Destination operand can be r/mX or register
- Source operand can be r/mX or register or immediate
- No source ***and*** destination as r/mXs, because that could allow for memory to memory transfer, which isn’t allowed on x86
- Evaluates the operation *as if it were on signed AND unsigned data*, and sets flags as appropriate. Instructions modify OF, SF, ZF, AF, PF, and CF flags for what it’s worth
- add rsp, 8 == (rsp = rsp + 8)
- sub rax, [rbx*2] == (rax = rax - memorypointedtoby(rbx*2))

Book page 59 = Add, page 62 = Sub

ExampleSubroutine1.c 1:

EIP = 00000001`40001010, but no instruction yet executed

rax	000007fe`f239c3a8 ⌘
rsp	00000000`0012feb8 ⌘

Key:

- ⌧ **executed instruction,**
- ♍ **modified value**
- ⌘ **start value**

```
func:
0000000140001000  mov   eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret
```

00000000`0012FEB8	00000001`400012ed
…	…
00000000`0012FE90	undef
00000000`0012FE88	undef
00000000`0012F[illegible]80	undef
00000000`0012[illegible]E78	undef

Belongs to the frame *before* main() is called

Colon notation means the full value is represented by the concatenation of the two values

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c 2:

rax	000007fe`f239c3a8 ⌘
rsp	00000000`0012fe90 ♍

Key:

⌧ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
func:
0000000140001000  mov   eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub   rsp,28h ⌧
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret
```

00000000`0012FEB8	00000001`400012ed
…	…
00000000`0012FE90	undef
00000000`0012FE88	undef
00000000`0012FE80	undef
00000000`0012FE78	undef

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c 3:

rax 000007fe`f239c3a8 ⌘
rsp 00000000`0012fe88 ♍

Key:
⌧ **executed instruction,**
♍ **modified value**
⌘ **start value**

```
func:
0000000140001000  mov   eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func(0140001000h) ⌧
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret
```

00000000`0012FEB8	00000001`400012ed
…	…
00000000`0012FE90	undef
00000000`0012FE88	00000001`40001019 ♍
00000000`0012FE80	undef
00000000`0012FE78	undef

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c 4:

rax	00000000`0000beef ♍
rsp	00000000`0012fe88

Note that it “zero extended” the reg (meaning it filled in the upper 32 bits of RAX with zeros)

```
func:
0000000140001000  mov   eax,0BEEFh ⌧
0000000140001005  ret
main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret
```

Key:

⌧ **executed instruction,**

♍ **modified value**

⌘ **start value**

00000000`0012FEB8	00000001`400012ed
…	…
00000000`0012FE90	undef
00000000`0012FE88	00000001`40001019
00000000`0012FE80	undef
00000000`0012FE78	undef

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c 4:

rax	00000000`0000beef ♍
rsp	00000000`0012fe88

Note that it “zero extended” the reg (meaning it filled in the upper 32 bits of RAX with zeros)

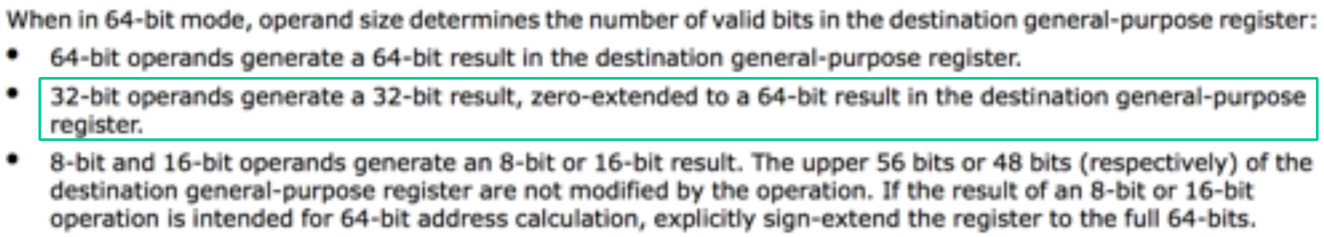
Key:

⌧ **executed instruction,**

♍ **modified value**

⌘ **start value**

From section 3.4.1.1 in the June 2014 Manual included with class materials:



When in 64-bit mode, operand size determines the number of valid bits in the destination general-purpose register:

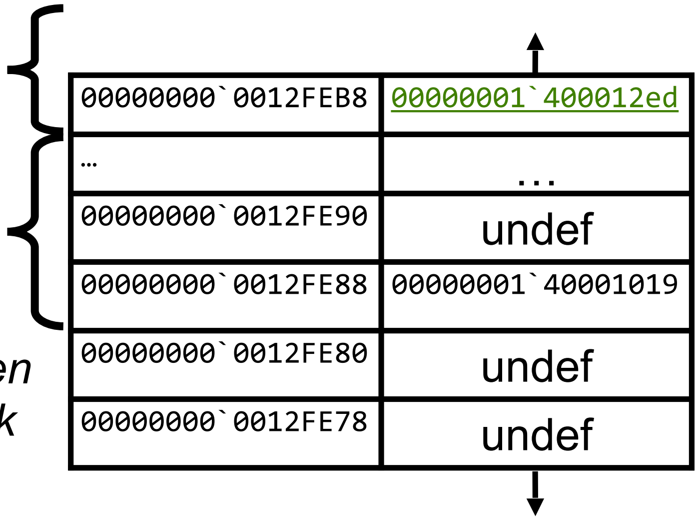
- 64-bit operands generate a 64-bit result in the destination general-purpose register.
- 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.
- 8-bit and 16-bit operands generate an 8-bit or 16-bit result. The upper 56 bits or 48 bits (respectively) of the destination general-purpose register are not modified by the operation. If the result of an 8-bit or 16-bit operation is intended for 64-bit address calculation, explicitly sign-extend the register to the full 64-bits.

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c:
STACK FRAME TIME OUT

```
func:
mov   eax,0BEEFh
ret
main:
sub   rsp,28h
call  func (0140001000h)
mov   eax,0F00Dh
add   rsp,28h
ret
```



“Function-before-main”’s frame

main’s frame

func never even makes a stack frame

00000000`0012FEB8	00000001`400012ed
…	…
00000000`0012FE90	undef
00000000`0012FE88	00000001`40001019
00000000`0012FE80	undef
00000000`0012FE78	undef

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c 5:

rax	00000000`0000beef
rsp	00000000`0012fe90 ♍

Key:

- ⌧ **executed instruction,**
- ♍ **modified value**
- ⌘ **start value**

```
func:
0000000140001000  mov   eax,0BEEFh
0000000140001005  ret   ⌧
main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret
```

00000000`0012FEB8	00000001`400012ed
…	…
00000000`0012FE90	undef
00000000`0012FE88	undef ♍
00000000`0012FE80	undef
00000000`0012FE78	undef

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c 6:

rax 00000000`0000f00d ♍
rsp 00000000`0012fe90

Key:
⌧ **executed instruction,**
♍ **modified value**
⌘ **start value**

```
func:
0000000140001000  mov   eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh ⌧
000000014000101E  add   rsp,28h
0000000140001022  ret
```

00000000`0012FEB8	00000001`400012ed
…	…
00000000`0012FE90	undef
00000000`0012FE88	undef
00000000`0012FE80	undef
00000000`0012FE78	undef

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c 7:

rax	00000000`0000f00d
rsp	00000000`0012feb8 ♍

Key:

⌧ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
func:
0000000140001000  mov   eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h ⌧
0000000140001022  ret
```

00000000`0012FEB8	00000001`400012ed
…	…
00000000`0012FE90	undef
00000000`0012FE88	undef
00000000`0012FE80	undef
00000000`0012FE78	undef

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

ExampleSubroutine1.c 8:

rax	00000000`0000f00d
rsp	00000000`0012fec0 ♍

Key:

⌧ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
func:
0000000140001000  mov   eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret ⌧
```

00000000`0012FEB8	undef ♍
…	…
00000000`0012FE90	undef
00000000`0012FE88	undef
00000000`0012FE80	undef
00000000`0012FE78	undef

Execution would continue at the value ret
removed from the stack: 00000001`400012ed

Colon notation means the full value is represented by the concatenation of the two values

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

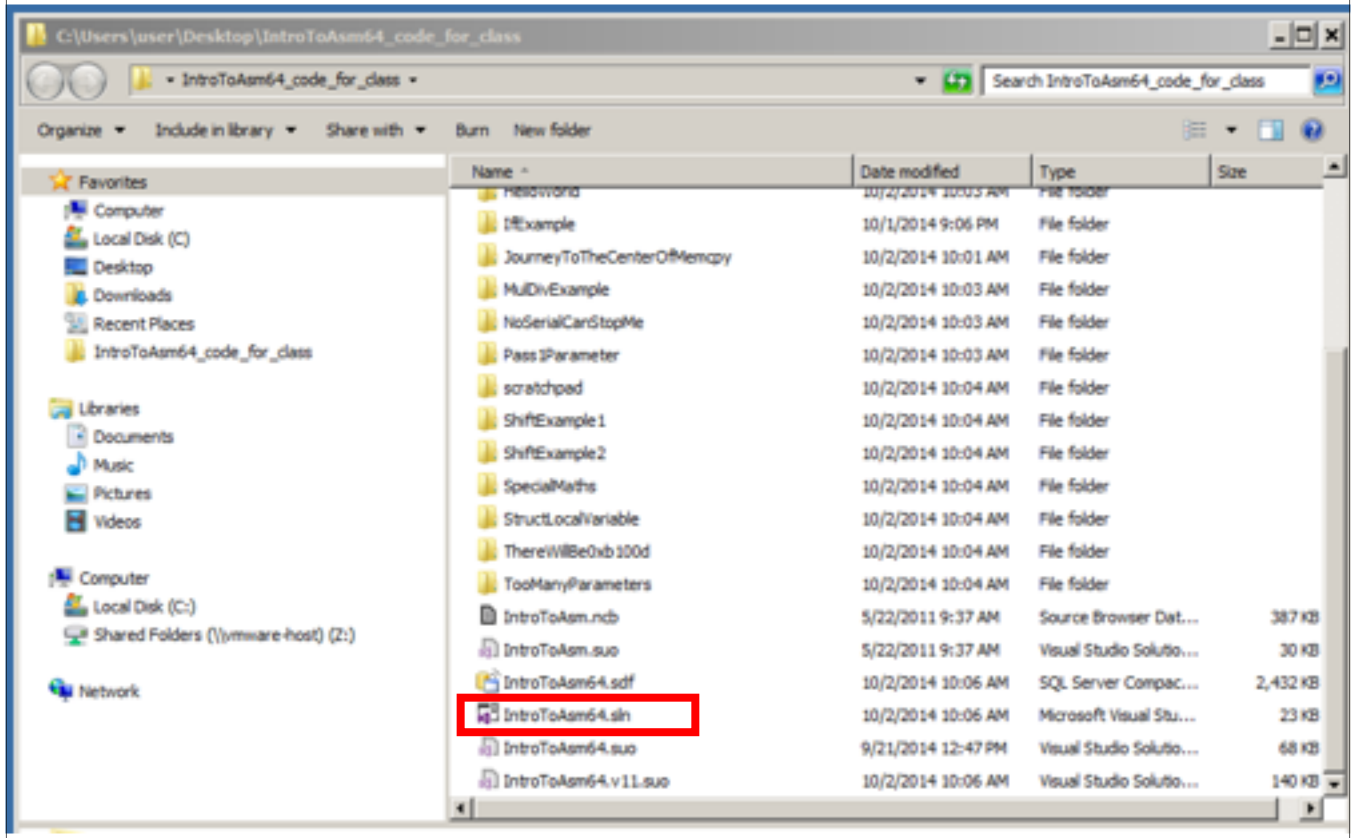
ExampleSubroutine1 Notes

- func() is dead code - its return value is not used for anything, and main() always returns 0xF00D. If optimizations were turned on in the compiler, it would remove func()
- We don't yet understand why main() does "`sub rsp,28h`" & "`add rsp,28h`"...We will figure that out later.

Let’s do that in a tool

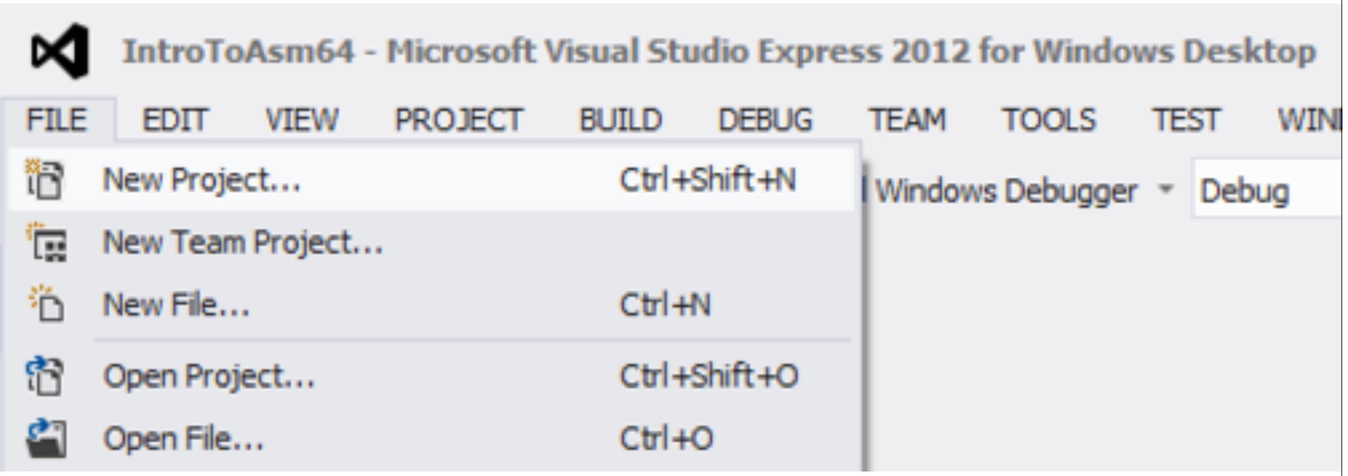
- Visual C++ 2012 Express edition (which I will shorthand as “VisualStudio” or VS)
- Standard Windows development environment
- Available for free, but missing some features that pro developers might want
- Keep in mind you can’t move express-edition-compiled applications to other systems and get them to run without first installing the “redistributable libraries”

Open IntroToAsm64.sln

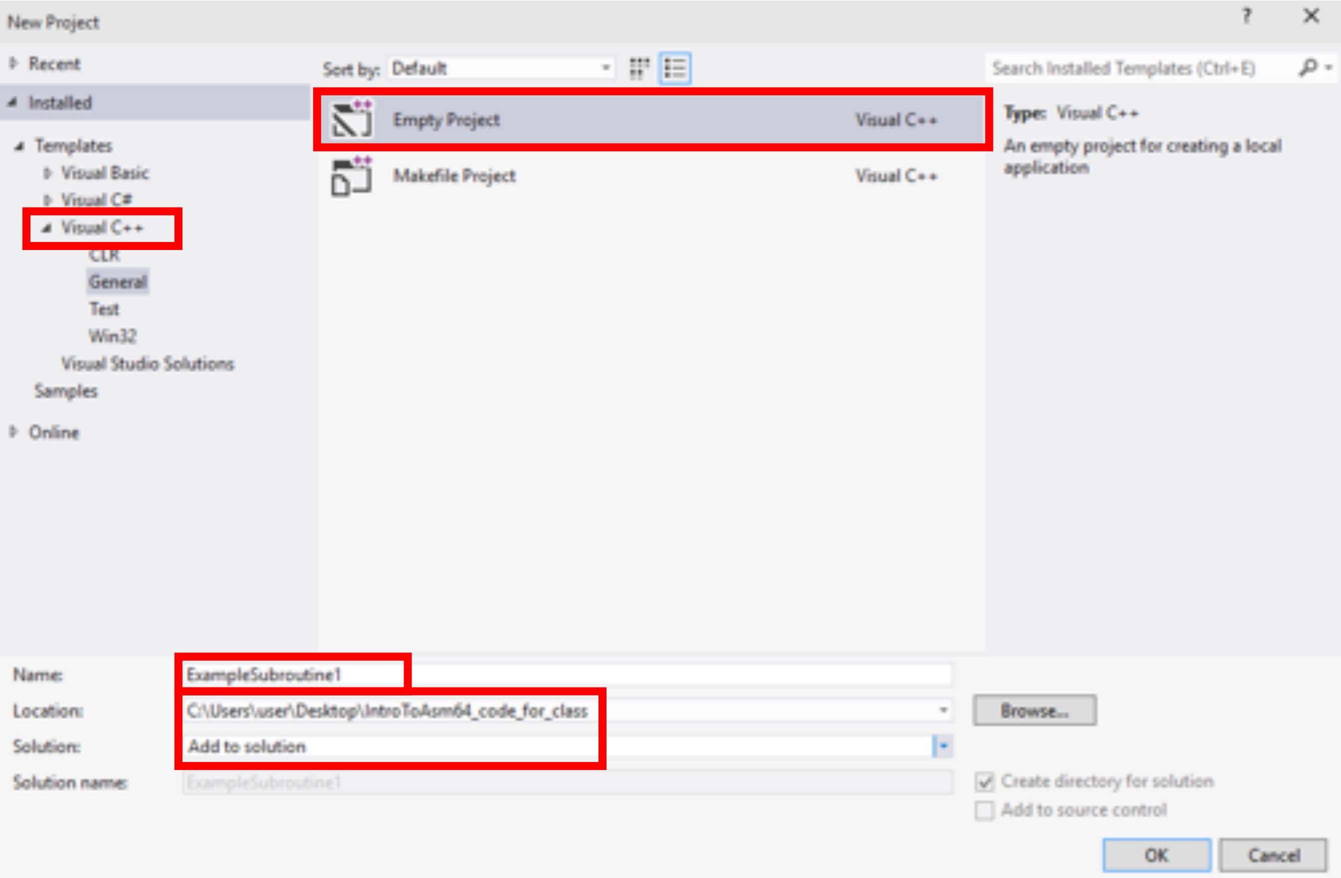


Name	Date modified	Type	Size
IfExample	10/1/2014 9:06 PM	File folder	
JourneyToTheCenterOfMemcpy	10/2/2014 10:01 AM	File folder	
MulDivExample	10/2/2014 10:03 AM	File folder	
NoSerialCanStopMe	10/2/2014 10:03 AM	File folder	
Pass1Parameter	10/2/2014 10:03 AM	File folder	
scratchpad	10/2/2014 10:04 AM	File folder	
ShiftExample1	10/2/2014 10:04 AM	File folder	
ShiftExample2	10/2/2014 10:04 AM	File folder	
SpecialMaths	10/2/2014 10:04 AM	File folder	
StructLocalVariable	10/2/2014 10:04 AM	File folder	
ThereWillBe0xb100d	10/2/2014 10:04 AM	File folder	
TooManyParameters	10/2/2014 10:04 AM	File folder	
IntroToAsm.ncb	5/22/2011 9:37 AM	Source Browser Dat...	387 KB
IntroToAsm.suo	5/22/2011 9:37 AM	Visual Studio Solutio...	30 KB
IntroToAsm64.sdf	10/2/2014 10:06 AM	SQL Server Compac...	2,432 KB
IntroToAsm64.sln	10/2/2014 10:06 AM	Microsoft Visual Stu...	23 KB
IntroToAsm64.suo	9/21/2014 12:47 PM	Visual Studio Solutio...	68 KB
IntroToAsm64.v11.suo	10/2/2014 10:06 AM	Visual Studio Solutio...	140 KB

Creating a new project - 1



Creating a new project - 2



Sort by: Default

Empty Project Visual C++

Type: Visual C++

An empty project for creating a local application

Name: ExampleSubroutine1

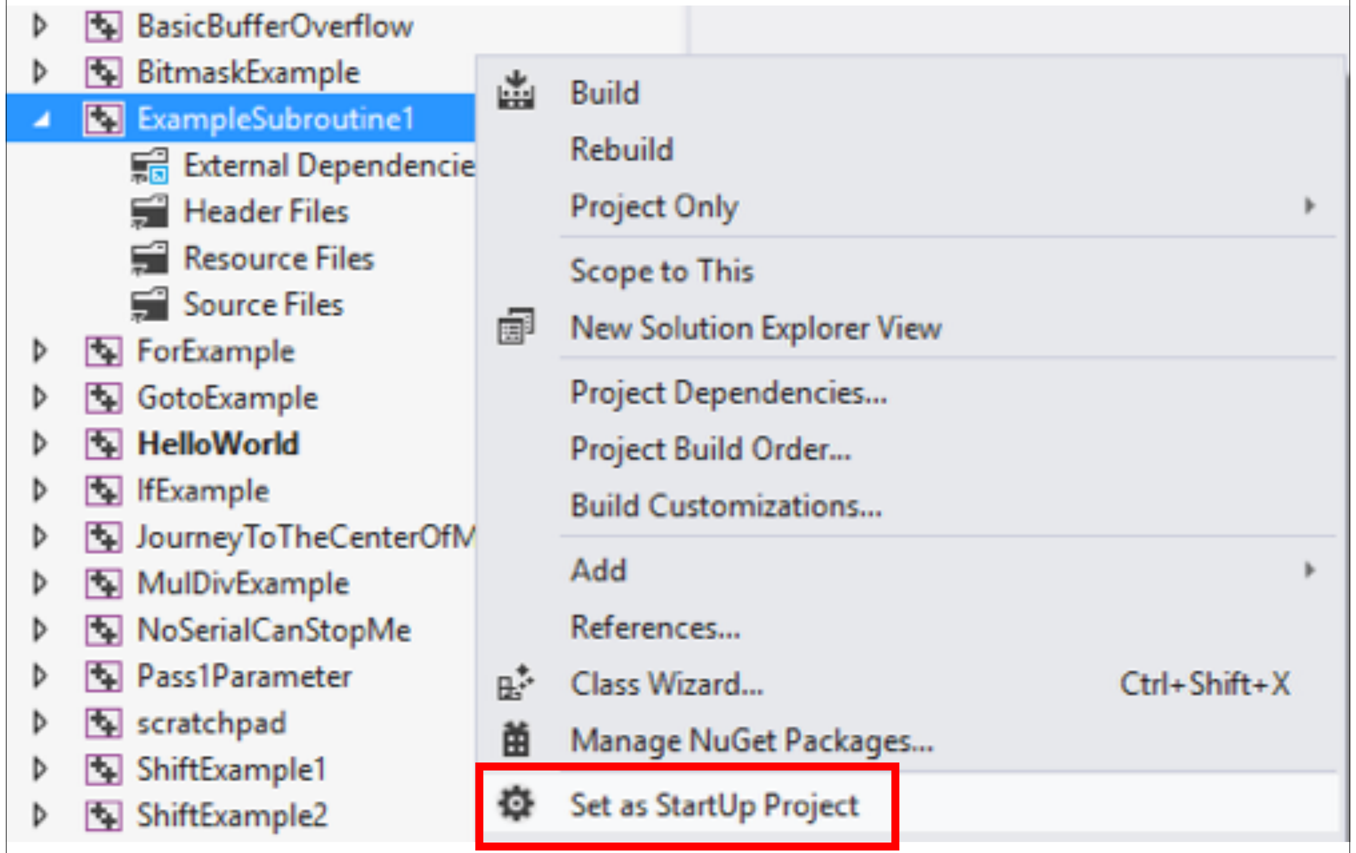
Location: C:\Users\user\Desktop\IntroToAsm64_code_for_class

Solution: Add to solution

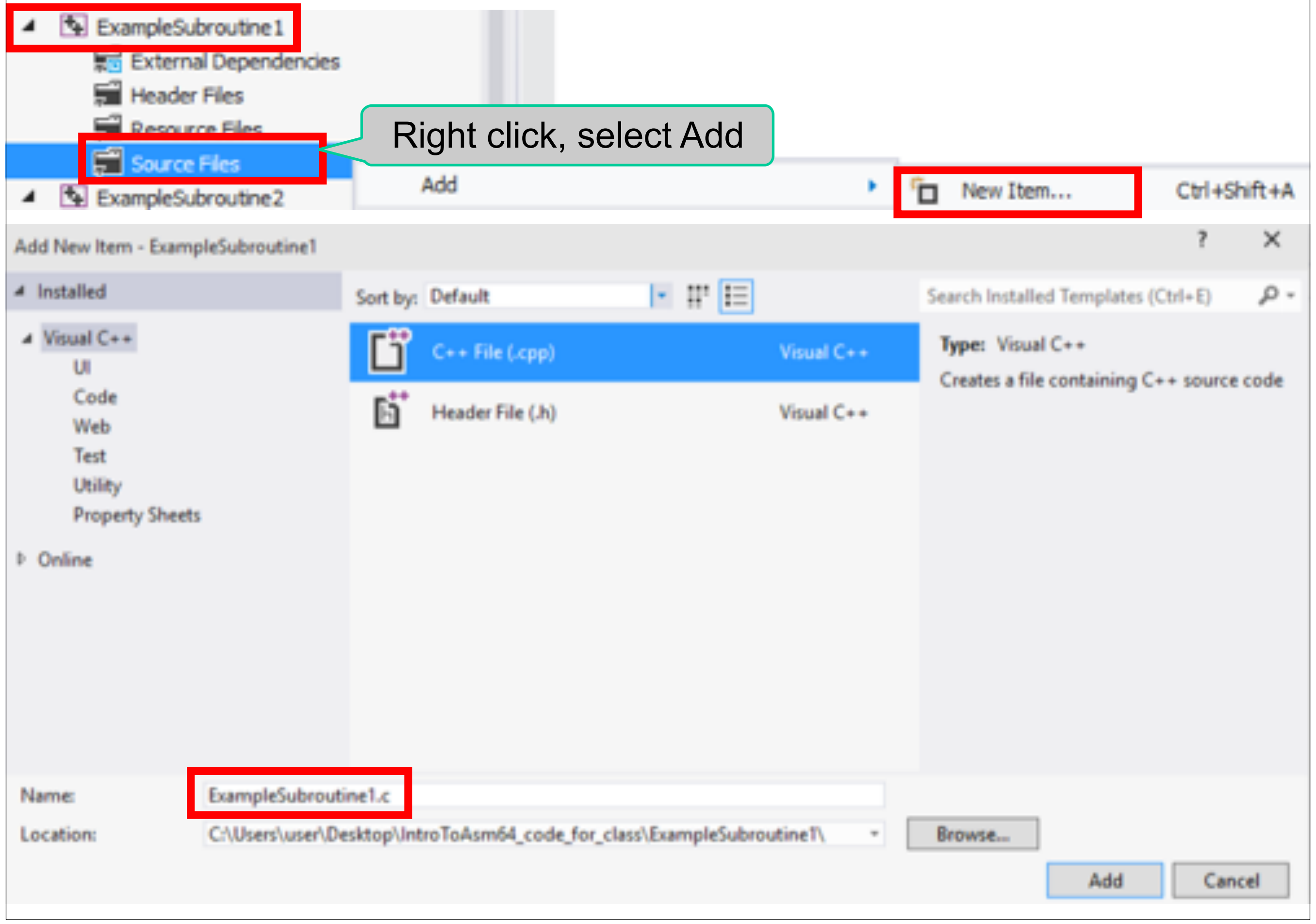
Solution name: ExampleSubroutine1

☑ Create directory for solution

☐ Add to source control


BasicBufferOverflow
BitmaskExample
ExampleSubroutine1
External Dependencie
Header Files
Resource Files
Source Files
ForExample
GotoExample
HelloWorld
IfExample
JourneyToTheCenterOfM
MulDivExample
NoSerialCanStopMe
Pass1Parameter
scratchpad
ShiftExample1
ShiftExample2
Build
Rebuild
Project Only
Scope to This
New Solution Explorer View
Project Dependencies...
Project Build Order...
Build Customizations...
Add
References...
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Set as StartUp Project

Creating a new project - 4



Right click, select Add

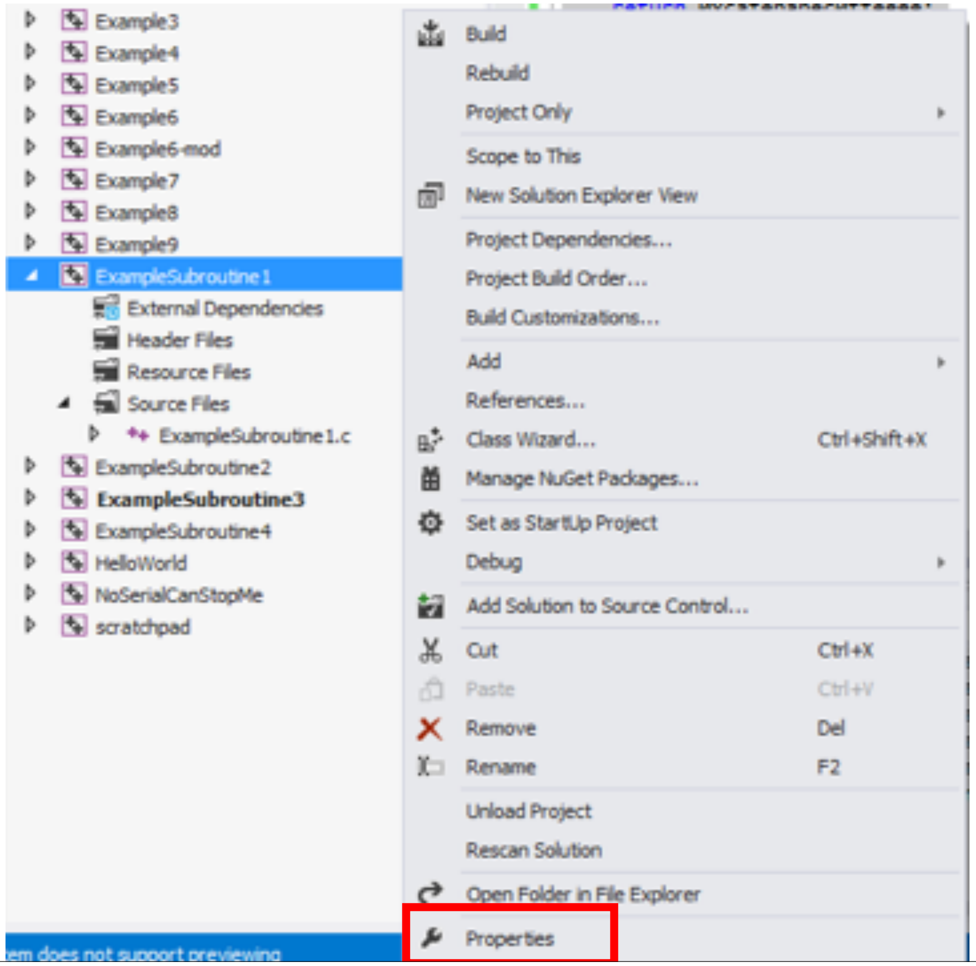
Type: Visual C++

Creates a file containing C++ source code

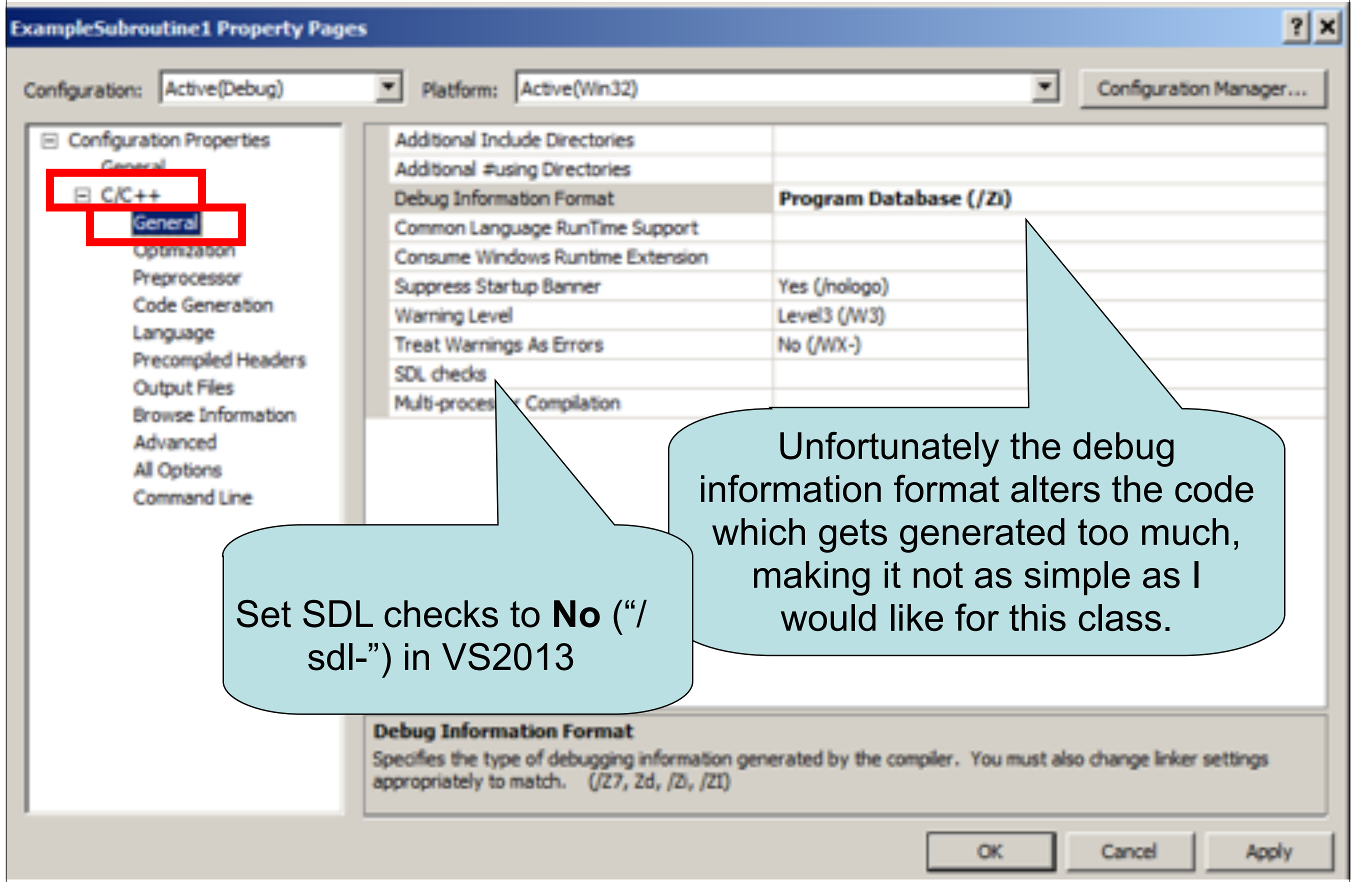
Name: ExampleSubroutine1.c

Location: C:\Users\user\Desktop\IntroToAsm64_code_for_class\ExampleSubroutine1\

Setting project properties - 1



Setting project properties - 2



Configuration: Active(Debug)

Platform: Active(Win32)

- Configuration Properties
  - General
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - All Options
    - Command Line

Additional Include Directories	
Additional #using Directories	
Debug Information Format	**Program Database (/Zi)**
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level3 (/W3)
Treat Warnings As Errors	No (/WX-)
SDL checks	
Multi-processor Compilation	

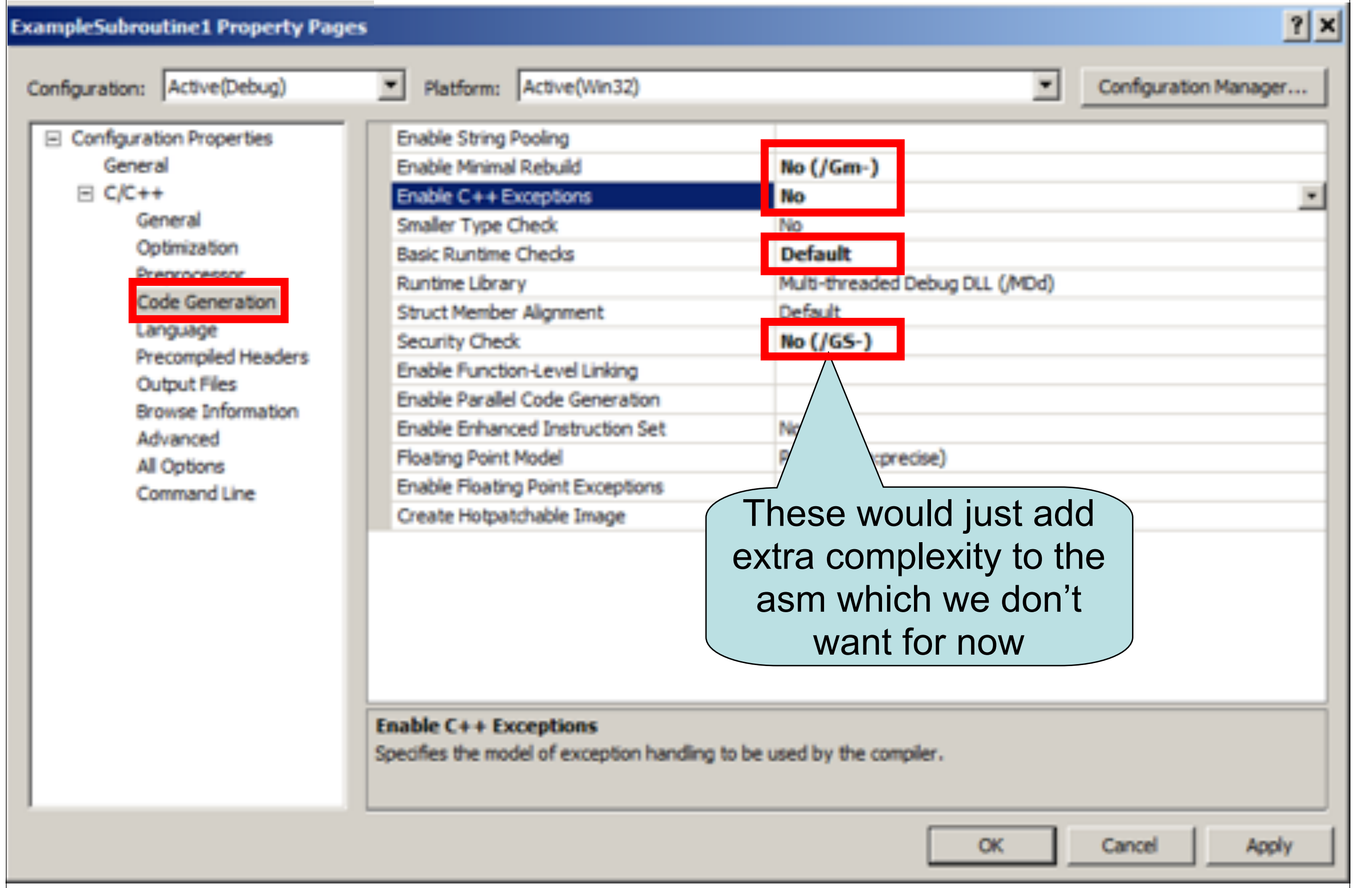
Debug Information Format

Specifies the type of debugging information generated by the compiler. You must also change linker settings appropriately to match. (/Z7, Zd, /Zi, /ZI)

Set SDL checks to **No** ("/sdl-") in VS2013

Unfortunately the debug information format alters the code which gets generated too much, making it not as simple as I would like for this class.

Setting project properties - 3



Configuration: Active(Debug)

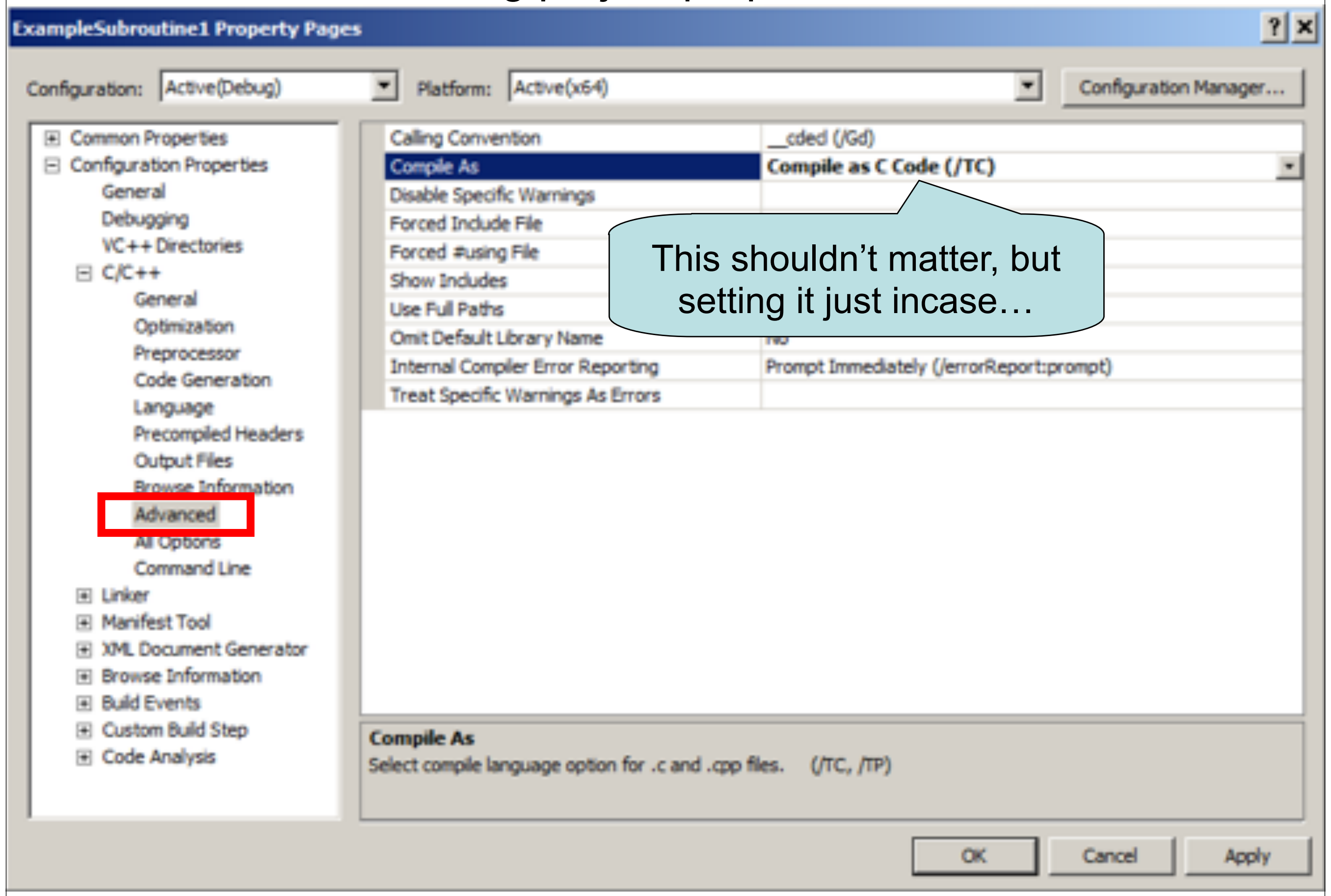
Platform: Active(Win32)

Enable String Pooling	
Enable Minimal Rebuild	**No (/Gm-)**
Enable C++ Exceptions	**No**
Smaller Type Check	No
Basic Runtime Checks	**Default**
Runtime Library	Multi-threaded Debug DLL (/MDd)
Struct Member Alignment	Default
Security Check	**No (/GS-)**
Enable Function-Level Linking	
Enable Parallel Code Generation	
Enable Enhanced Instruction Set	No
Floating Point Model	P precise)
Enable Floating Point Exceptions	
Create Hotpatchable Image	

Enable C++ Exceptions
Specifies the model of exception handling to be used by the compiler.

These would just add extra complexity to the asm which we don't want for now

Setting project properties - 4



ExampleSubroutine1 Property Pages

Configuration: Active(Debug)

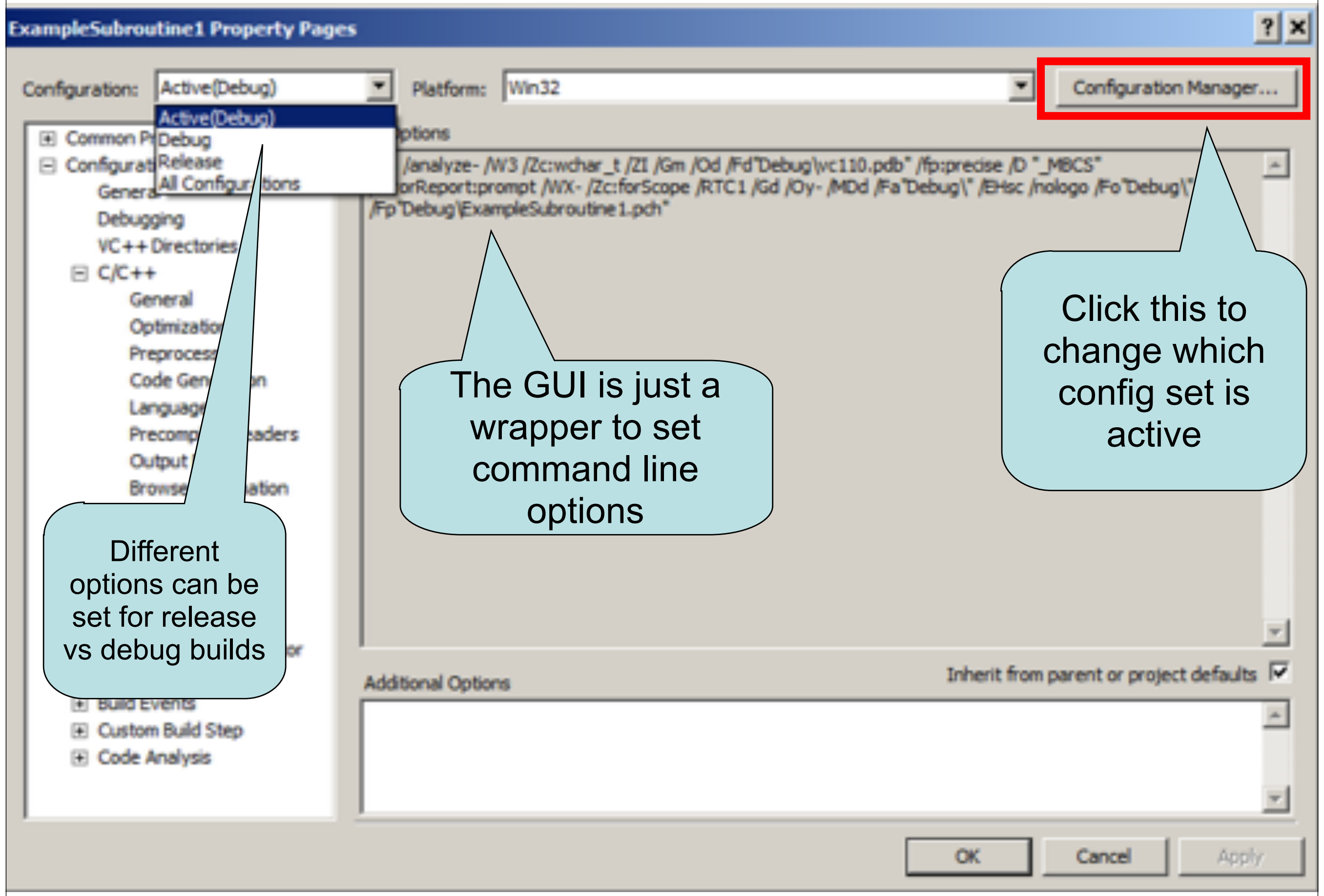
Platform: Active(x64)

Calling Convention	__cdecl (/Gd)
Compile As	Compile as C Code (/TC)
Disable Specific Warnings	
Forced Include File	
Forced #using File	
Show Includes	
Use Full Paths	
Omit Default Library Name	No
Internal Compiler Error Reporting	Prompt Immediately (/errorReport:prompt)
Treat Specific Warnings As Errors	

Compile As
Select compile language option for .c and .cpp files. (/TC, /TP)

This shouldn't matter, but setting it just incase…

Setting project properties - 5



```
 /analyze- /W3 /Zc:wchar_t /ZI /Gm /Od /Fd"Debug\vc110.pdb" /fp:precise /D "_MBCS"
 orReport:prompt /WX- /Zc:forScope /RTC1 /Gd /Oy- /MDd /Fa"Debug\" /EHsc /nologo /Fo"Debug\"
/Fp"Debug\ExampleSubroutine1.pch"
```

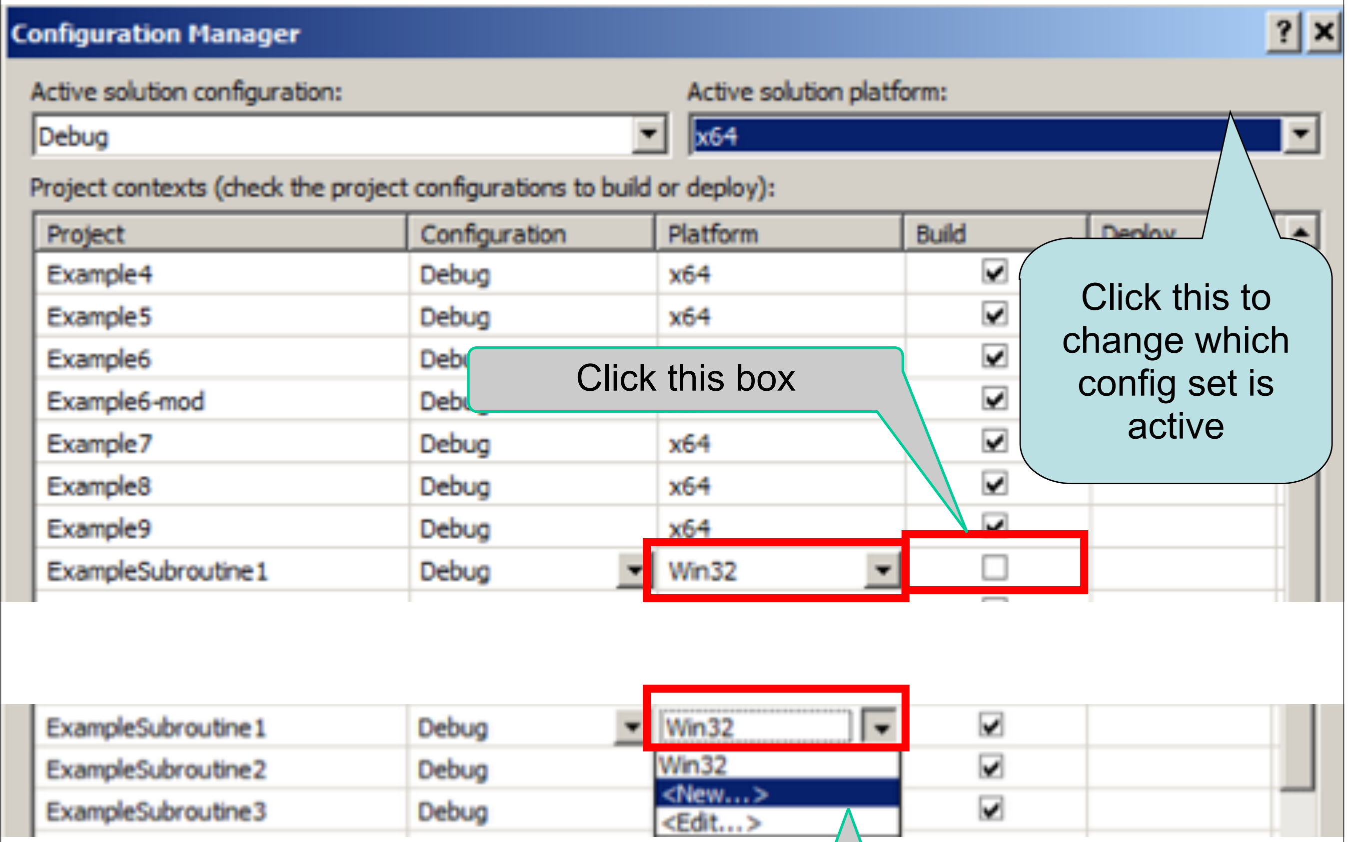
Inherit from parent or project defaults ☑

Different options can be set for release vs debug builds

The GUI is just a wrapper to set command line options

Click this to change which config set is active

Setting project properties - 5.1



Active solution configuration: Debug

Active solution platform: x64

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
Example4	Debug	x64	☑	
Example5	Debug	x64	☑	
Example6	Deb		☑	
Example6-mod	Deb		☑	
Example7	Debug	x64	☑	
Example8	Debug	x64	☑	
Example9	Debug	x64	☑	
ExampleSubroutine1	Debug	Win32	☐	

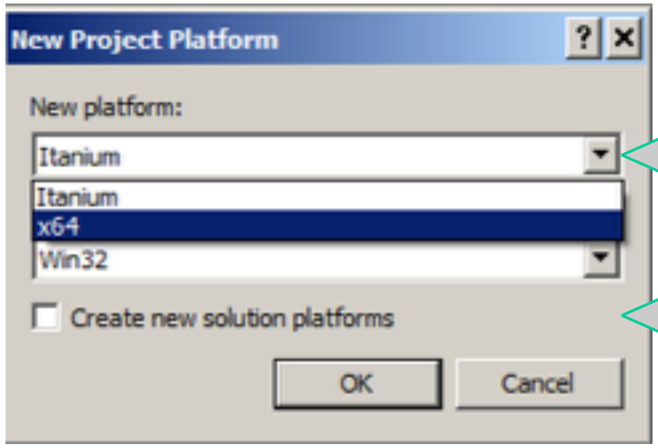
ExampleSubroutine1	Debug	Win32	☑
ExampleSubroutine2	Debug		☑
ExampleSubroutine3	Debug		☑

Click this to change which config set is active

Click this box

And then select “New” here

Setting project properties - 5.2

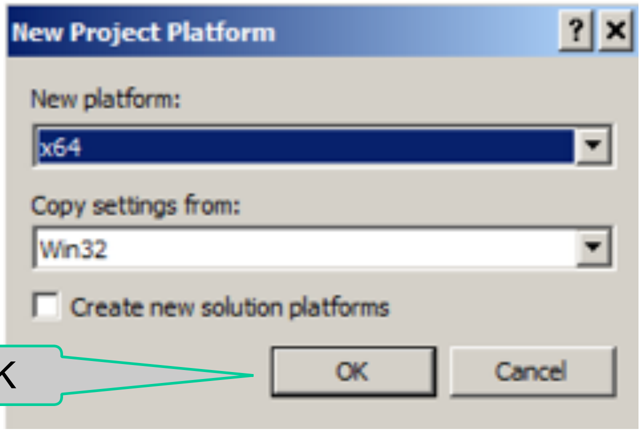


New platform: Itanium

☐ Create new solution platforms

Pull down to select x64 build platform option

Very important!
This box should be un-checked



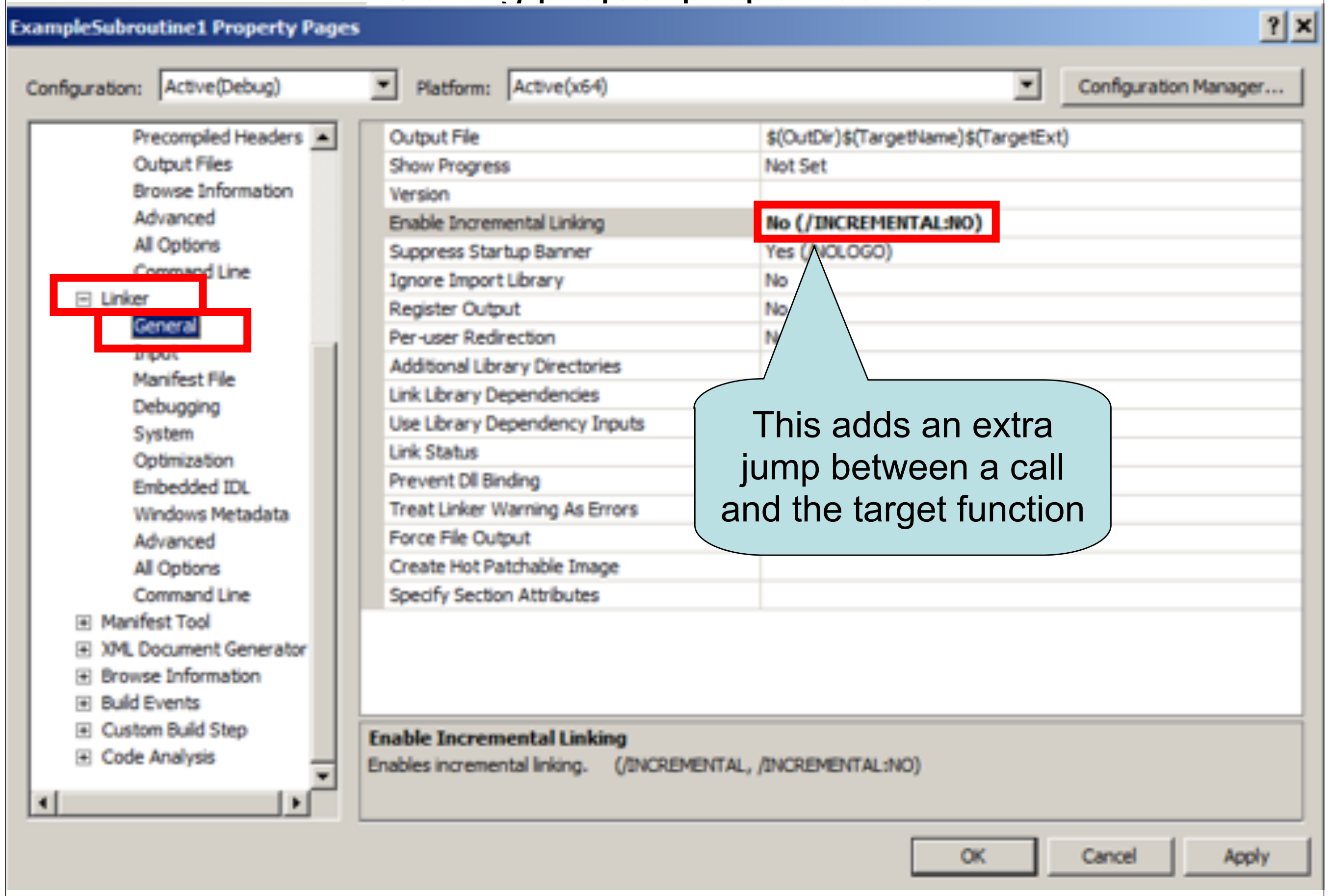
New platform: x64

Copy settings from: Win32

☐ Create new solution platforms

Once it’s selected, hit OK

Setting project properties - 6

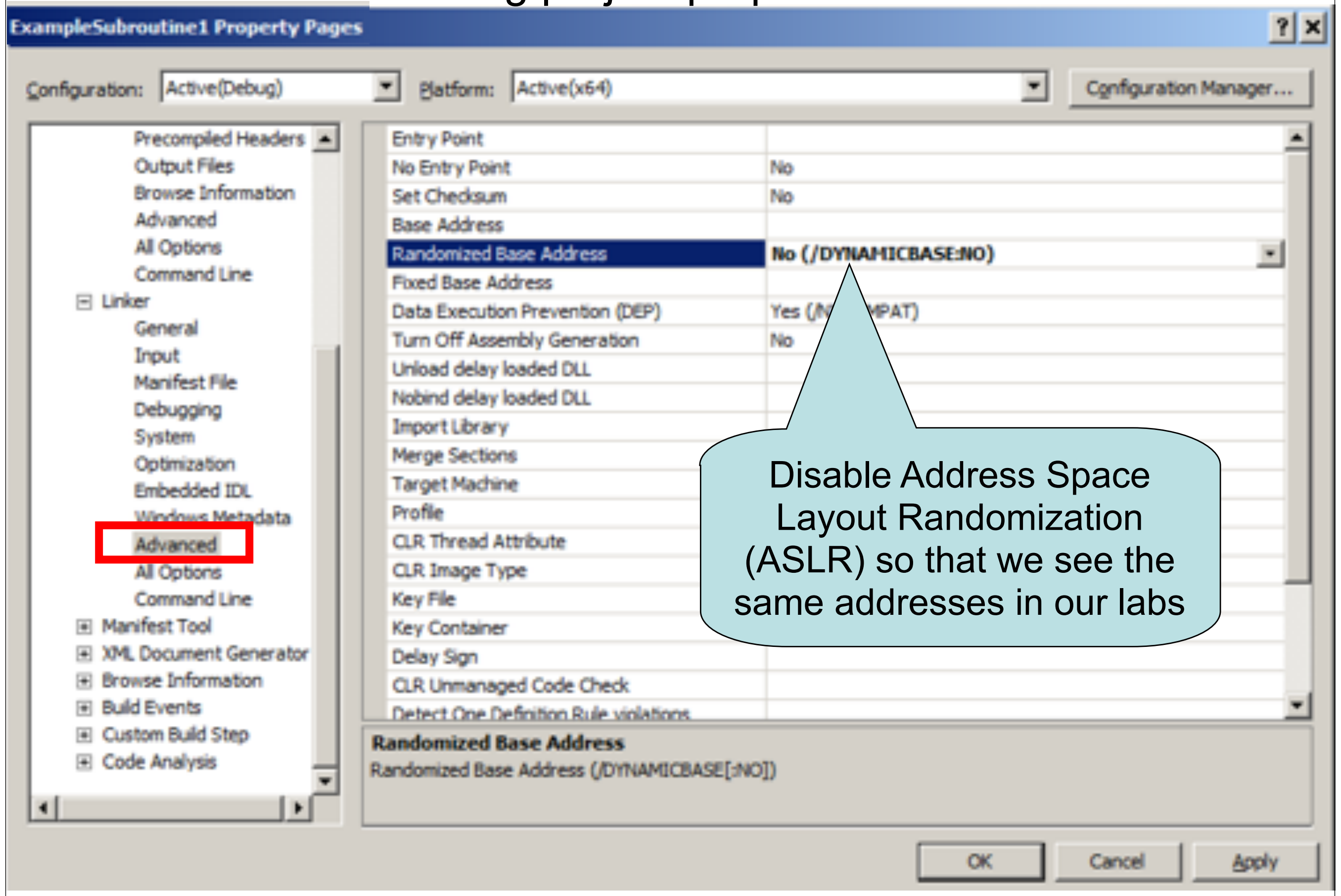


Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	**No (/INCREMENTAL:NO)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	N
Additional Library Directories	
Link Library Dependencies	
Use Library Dependency Inputs	
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	
Force File Output	
Create Hot Patchable Image	
Specify Section Attributes	

Enable Incremental Linking
Enables incremental linking. (/INCREMENTAL, /INCREMENTAL:NO)

This adds an extra jump between a call and the target function

Setting project properties - 7



Configuration: Active(Debug)

Platform: Active(x64)

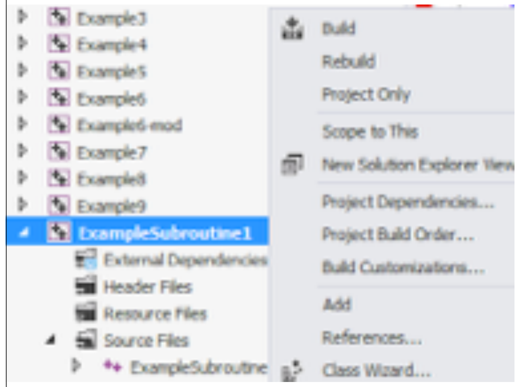
Entry Point	
No Entry Point	No
Set Checksum	No
Base Address	
Randomized Base Address	**No (/DYNAMICBASE:NO)**
Fixed Base Address	
Data Execution Prevention (DEP)	Yes (/N MPAT)
Turn Off Assembly Generation	No
Unload delay loaded DLL	
Nobind delay loaded DLL	
Import Library	
Merge Sections	
Target Machine	
Profile	
CLR Thread Attribute	
CLR Image Type	
Key File	
Key Container	
Delay Sign	
CLR Unmanaged Code Check	
Detect One Definition Rule violations	

Randomized Base Address

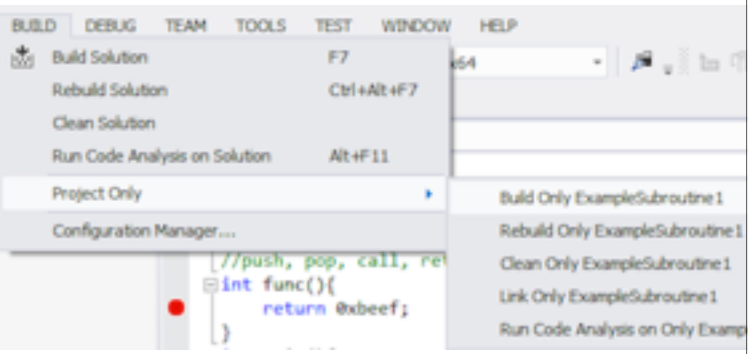
Randomized Base Address (/DYNAMICBASE[:NO])

Disable Address Space Layout Randomization (ASLR) so that we see the same addresses in our labs

2 ways to build the project



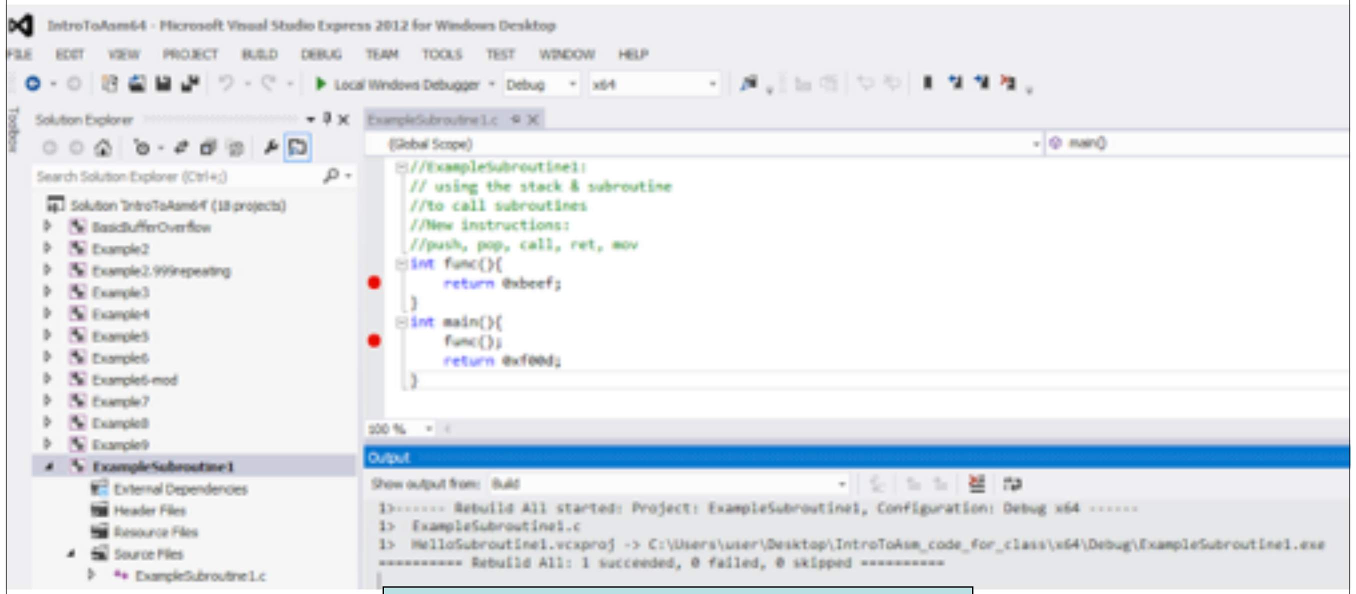
Right click on project
and select build



```
//push, pop, call, re
int func(){
    return 0xbeef;
}
```

Or select Build Only … from the
menu bar

Building the project - 2

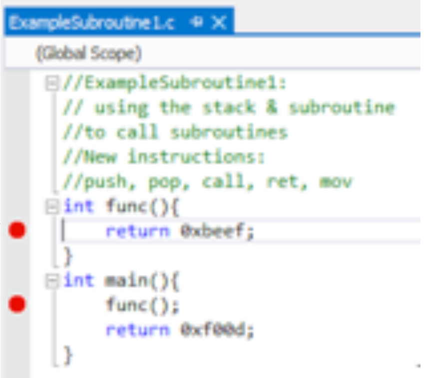


```
//ExampleSubroutine1:
// using the stack & subroutine
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int func(){
	return 0xbeef;
}
int main(){
	func();
	return 0xf00d;
}
```

```
1>------ Rebuild All started: Project: ExampleSubroutine1, Configuration: Debug x64 ------
1>  ExampleSubroutine1.c
1>  HelloSubroutine1.vcxproj -> C:\Users\user\Desktop\IntroToAsm_code_for_class\x64\Debug\ExampleSubroutine1.exe
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

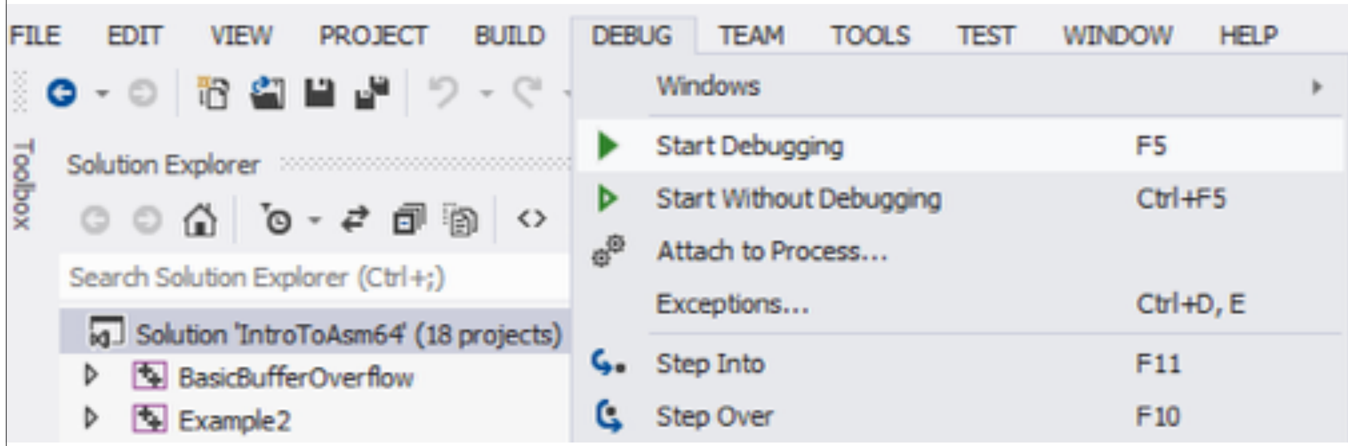
Information about whether the build succeeded will be in the Output window. If it fails, a separate Error tab will open up

Setting breakpoints & start debugger

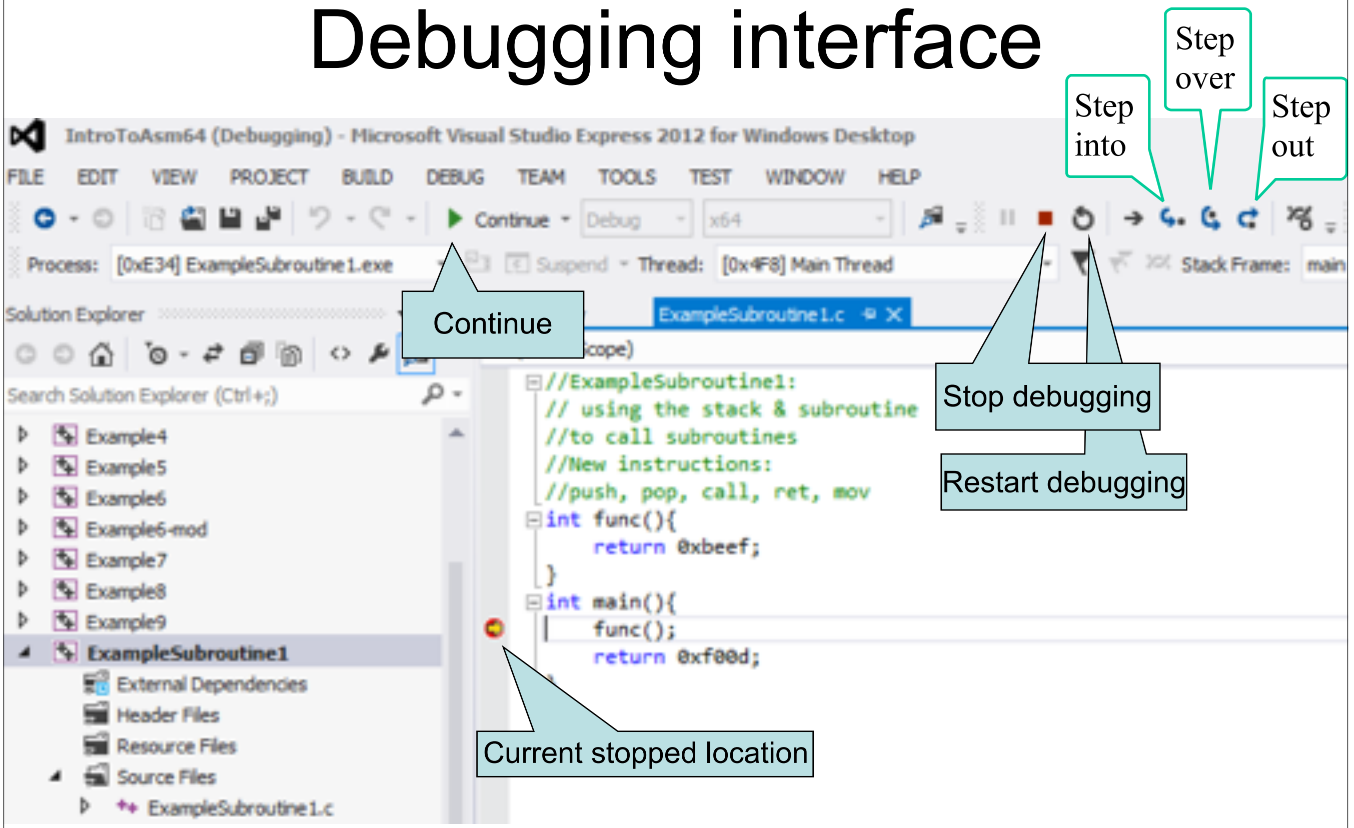


```
//ExampleSubroutine1:
// using the stack & subroutine
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int func(){
    return 0xbeef;
}
int main(){
    func();
    return 0xf00d;
}
```

Click to the left of the line to break at.
A red circle will appear



Debugging interface



```
//ExampleSubroutine1:
// using the stack & subroutine
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int func(){
	return 0xbeef;
}
int main(){
	func();
	return 0xf00d;
```

Step into

Step over

Step out

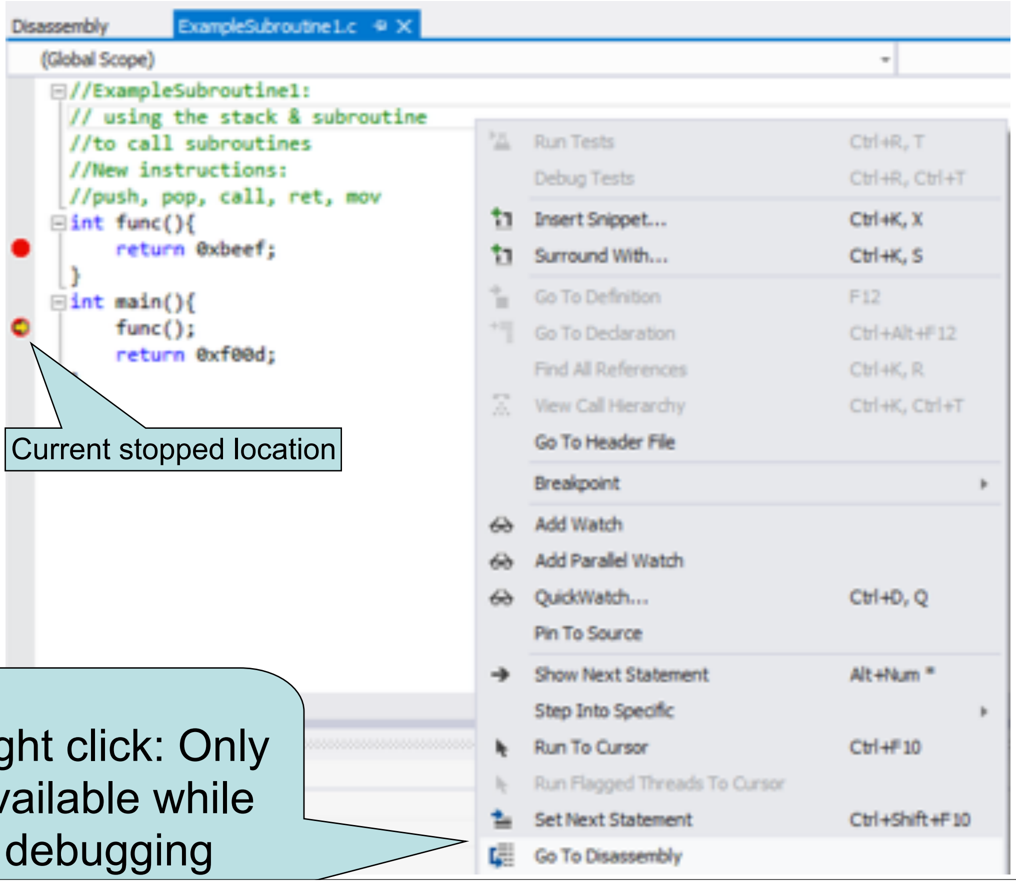
Continue

Stop debugging

Restart debugging

Current stopped location

Showing assembly

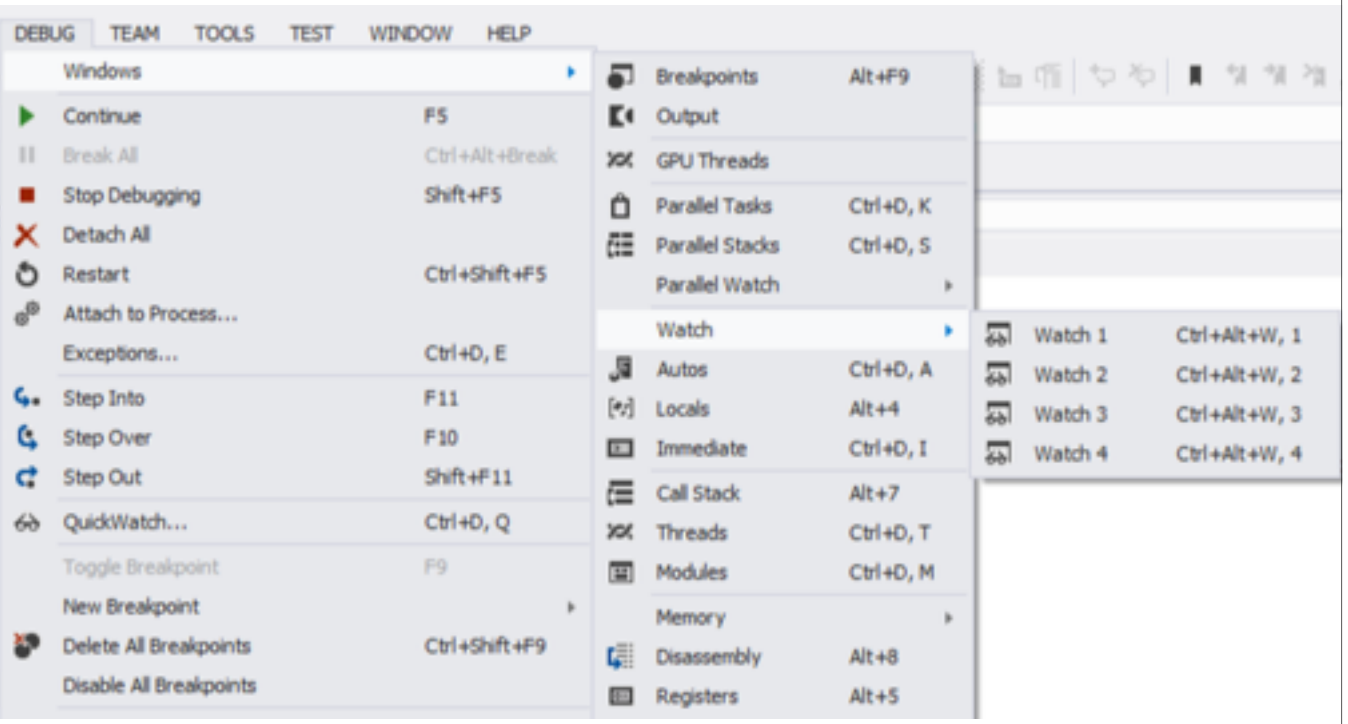


```
//ExampleSubroutine1:
// using the stack & subroutine
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int func(){
    return 0xbeef;
}
int main(){
    func();
    return 0xf00d;
```

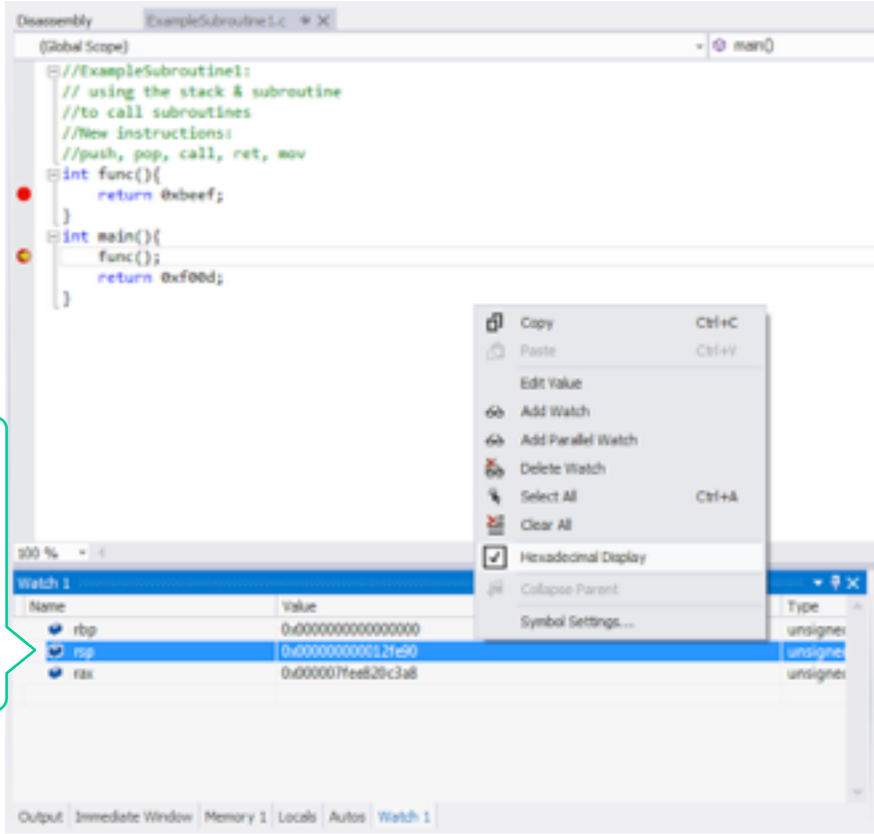
Current stopped location

Right click: Only available while debugging

Debugging window options



Watching registers (“watch”)

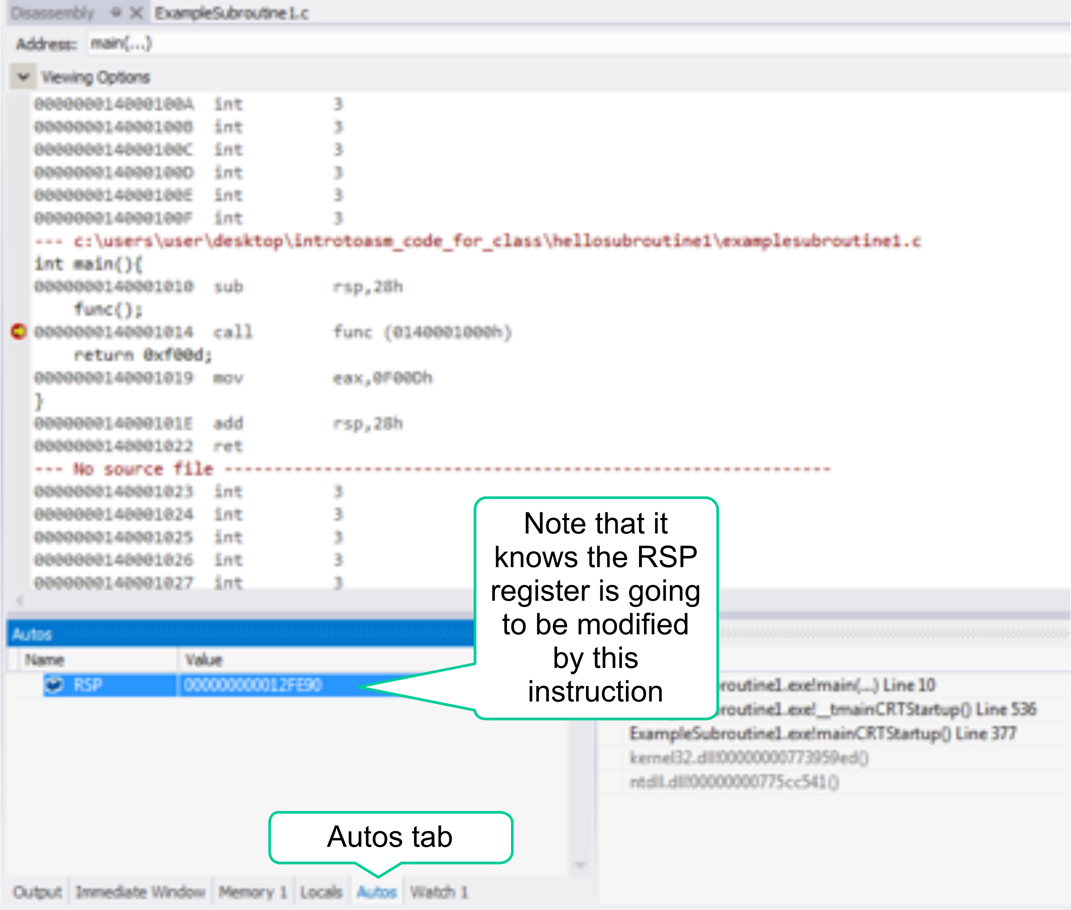


```
//ExampleSubroutine1:
// using the stack & subroutine
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int func(){
    return 0xbeef;
}
int main(){
    func();
    return 0xf00d;
}
```

Name	Value	Type
rbp	0x0000000000000000	unsigne
rsp	0x000000000012fe90	unsigne
rax	0x000007fee820c3a8	unsigne

In the “Watch” tab you can enter register names or variable names

Watching registers ("autos")



```
000000014000100A  int         3
000000014000100B  int         3
000000014000100C  int         3
000000014000100D  int         3
000000014000100E  int         3
000000014000100F  int         3
--- c:\users\user\desktop\introtoasm_code_for_class\hellosubroutine1\examplesubroutine1.c
int main(){
0000000140001010  sub         rsp,28h
    func();
0000000140001014  call        func (0140001000h)
    return 0xf00d;
0000000140001019  mov         eax,0F00Dh
}
000000014000101E  add         rsp,28h
0000000140001022  ret
--- No source file -------------------------------------------------------------
0000000140001023  int         3
0000000140001024  int         3
0000000140001025  int         3
0000000140001026  int         3
0000000140001027  int         3
```

Autos

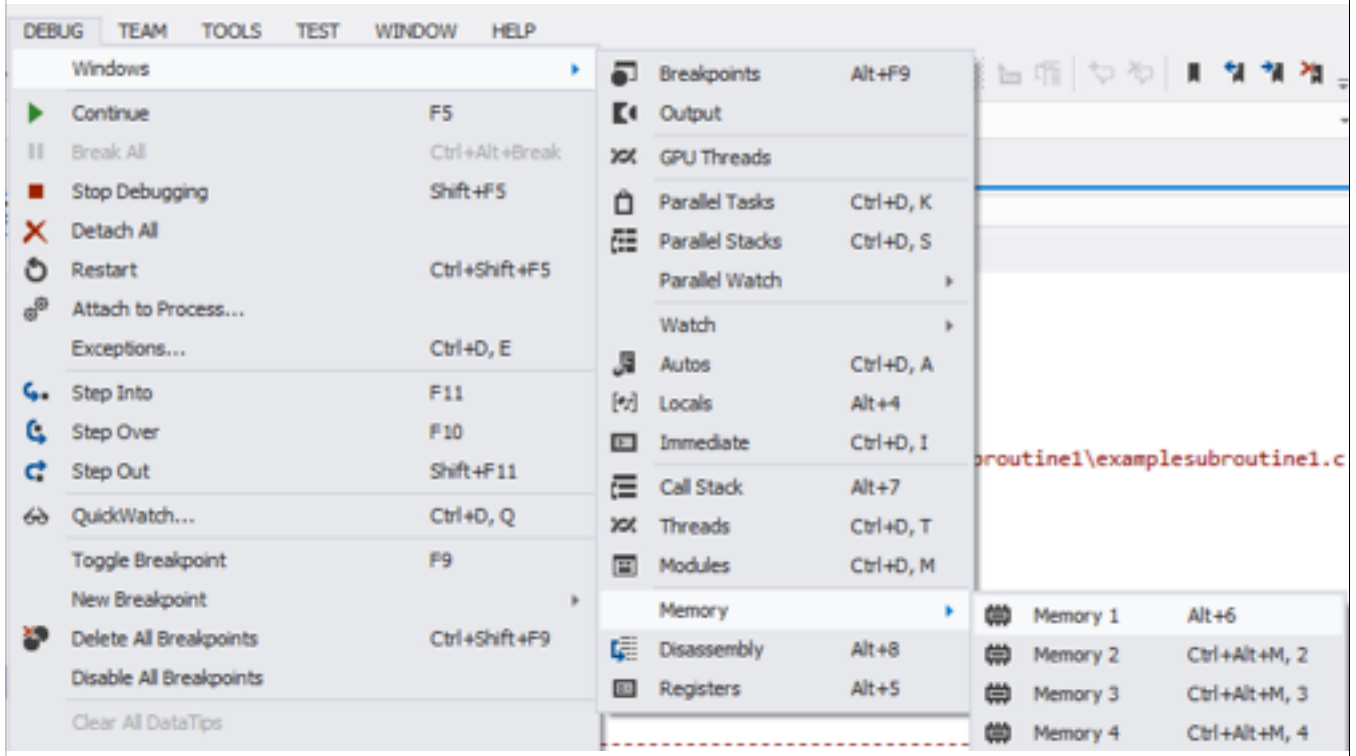
Name	Value
RSP	000000000012FE90

```
routine1.exe!main(...) Line 10
routine1.exe!__tmainCRTStartup() Line 536
ExampleSubroutine1.exe!mainCRTStartup() Line 377
kernel32.dll!00000000773959ed()
ntdll.dll!00000000775cc541()
```

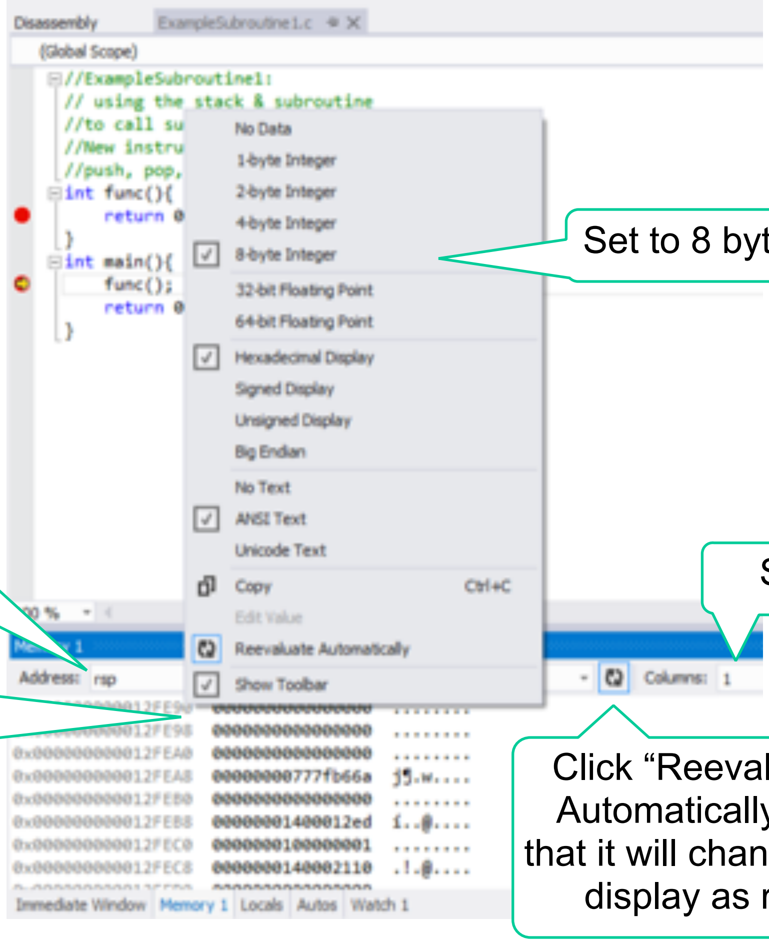
Note that it knows the RSP register is going to be modified by this instruction

Autos tab

Watching the stack change - 1



Watching the stack change - 2



```
//ExampleSubroutine1:
// using the stack & subroutine
//to call su
//New instru
//push, pop,
int func(){
    return 0
}
int main(){
    func();
    return 0
}
```

Address: rsp

Columns: 1

```
0x000000000012FE98  0000000000000000  ........
0x000000000012FEA0  0000000000000000  ........
0x000000000012FEA8  00000000777fb66a  j¶.w....
0x000000000012FEB0  0000000000000000  ........
0x000000000012FEB8  00000001400012ed  í..@....
0x000000000012FEC0  0000000100000001  ........
0x000000000012FEC8  0000000140002110  .!.@....
```

Set to 8 byte

Set address to rsp (will always be the top of the stack)

Set to 1

Right click on the body of the data in the window and make sure everything's set like this

Click “Reevaluate Automatically” so that it will change the display as rsp

ExampleSubroutine1.c takeaways

- In VS (when optimization is turned off), there is an over-allocation of stack space as a result of calling a function
  - 0x28 reserved with no apparent storage of data on the stack
  - More about this later once we start passing function parameters

```
int func(){
    return 0xbeef;
}
int main(){
    func();
    return 0xf00d;
}
```

```
func:
0000000140001000  mov         eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub         rsp,28h
0000000140001014  call        func (0140001000h)
0000000140001019  mov         eax,0F00Dh
000000014000101E  add         rsp,28h
0000000140001022  ret
```

Instructions we now know (8)

- NOP
- PUSH/POP
- CALL/RET
- MOV
- ADD/SUB