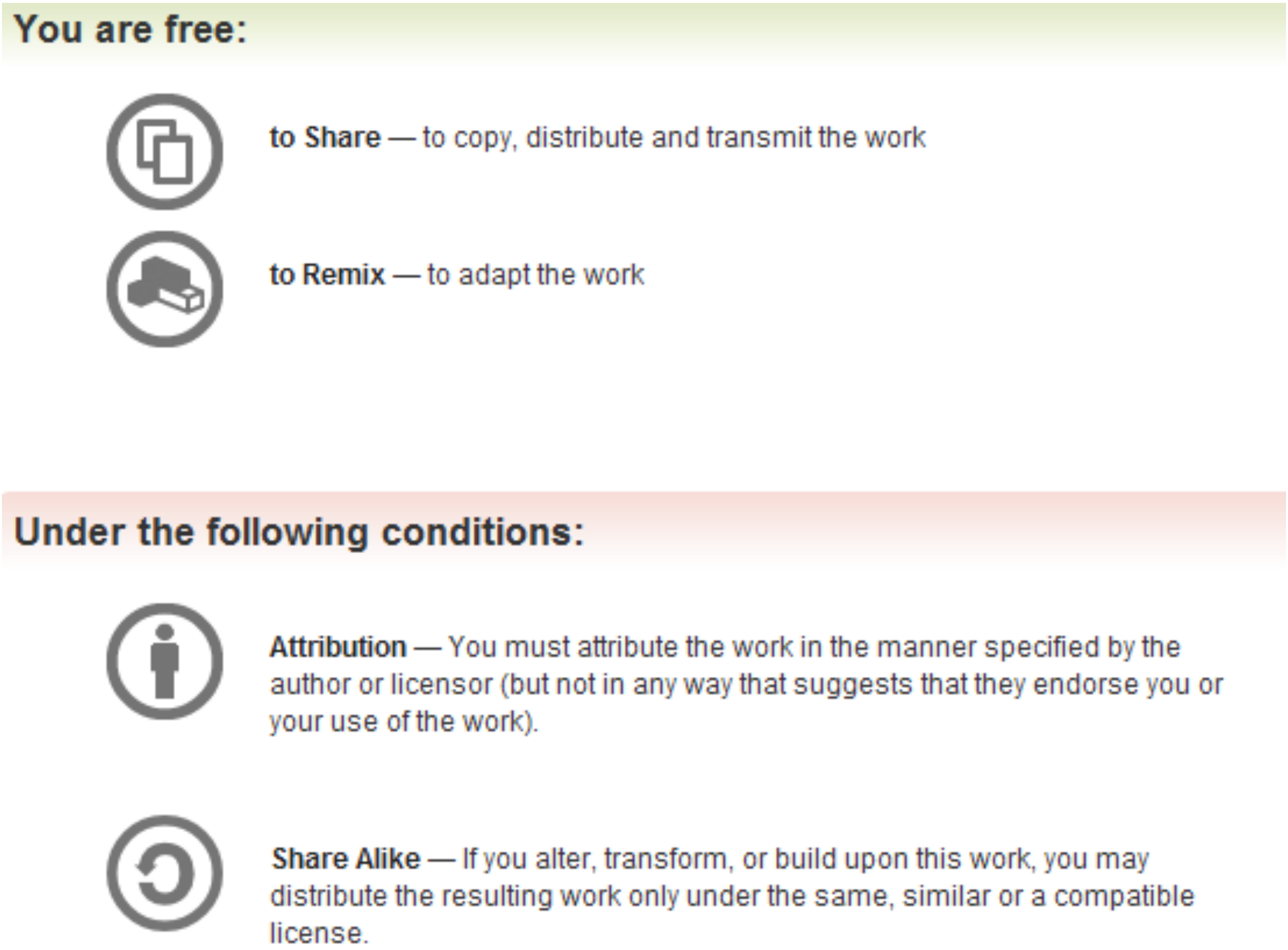
Introduction to Intel x86 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2009/2010
xkovah at gmail

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.

Additional Content/Ideas/Info Provided By:

- Jon A. Erickson, Christian Arllen, Dave Keppler
- Who suggested what, is inline with the material

About Me

- Security nerd - generalist, not specialist
- Realmz ~1996, Mac OS 8, BEQ->BNE FTW!
- x86 ~2002
- Know or have known ~5 assembly languages(x86, SPARC, ARM, PPC, 68HC12). x86 is by far the most complex.
- Routinely read assembly when debugging my own code, reading exploit code, and reverse engineering things

About You?

- Name & Department
- Why did you want to take the class?
- If you know you will be applying this knowledge, to which OS and/or development environment?

About the Class

- The intent of this class is to expose you to the most commonly generated assembly instructions, and the most frequently dealt with architecture hardware.
  - 32 bit instructions/hardware
  - Implementation of a Stack
  - Common tools
- Many things will therefore be left out or deferred to later classes.
  - Floating point instructions/hardware
  - 16/64 bit instructions/hardware
  - Complicated or rare 32 bit instructions
  - Instruction pipeline, caching hierarchy, alternate modes of operation, hw virtualization, etc

About the Class 2

- The hope is that the material covered will be provide the required background to delve deeper into areas which may have seemed daunting previously.
- Because I can't anticipate the needs of all job classes, if there are specific areas which you think would be useful to certain job types, let me know. The focus areas are currently primarily influenced by my security background, but I would like to make the class as widely applicable as possible.

Agenda

- Day 1 - Part 1 - Architecture Introduction, Windows tools
- Day 1 - Part 2 - Windows Tools & Analysis, Learning New Instructions
- Day 2 - Part 1 - Linux Tools & Analysis
- Day 2 - Part 2 - Inline Assembly, Read The Fun Manual, Choose Your Own Adventure

Miss Alaineous

- Questions: Ask 'em if you got 'em
  - If you fall behind and get lost and try to tough it out until you understand, it's more likely that you will stay lost, so ask questions ASAP.
- Browsing the web and/or checking email during class is a good way to get lost ;)
- Vote on class schedule.
- Benevolent dictator clause.
- It's called x86 because of the progression of Intel chips from 8086, 80186, 80286, etc. I just had to get that out of the way. :)

What you're going to learn

```
#include <stdio.h>
int main(){
   printf(“Hello World!\n”);
   return 0x1234;
}
```

Is the same as…

```
.text:00401730 main
.text:00401730                push    ebp
.text:00401731                mov     ebp, esp
.text:00401733                push    offset aHelloWorld ; "Hello world\n"
.text:00401738                call    ds:__imp__printf
.text:0040173E                add     esp, 4
.text:00401741                mov     eax, 1234h
.text:00401746                pop     ebp
.text:00401747                retn
```

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Disassembled with IDA Pro 4.9 Free Version

Is the same as…

```
08048374 <main>:
 8048374:       8d 4c 24 04                     lea    0x4(%esp),%ecx
 8048378:       83 e4 f0                        and    $0xfffffff0,%esp
 804837b:       ff 71 fc                        pushl  -0x4(%ecx)
 804837e:       55                              push   %ebp
 804837f:       89 e5                           mov    %esp,%ebp
 8048381:       51                              push   %ecx
 8048382:       83 ec 04                        sub    $0x4,%esp
 8048385:       c7 04 24 60 84 04 08            movl   $0x8048460,(%esp)
 804838c:       e8 43 ff ff ff                  call   80482d4 <puts@plt>
 8048391:       b8 2a 00 00 00                  mov    $0x1234,%eax
 8048396:       83 c4 04                        add    $0x4,%esp
 8048399:       59                              pop    %ecx
 804839a:       5d                              pop    %ebp
 804839b:       8d 61 fc                        lea    -0x4(%ecx),%esp
 804839e:       c3                              ret
 804839f:       90                              nop
```

Ubuntu 8.04, GCC 4.2.4
Disassembled with “objdump -d”

Is the same as…

```
_main:
00001fca        pushl   %ebp
00001fcb        movl    %esp,%ebp
00001fcd        pushl   %ebx
00001fce        subl    $0x14,%esp
00001fd1        calll   0x00001fd6
00001fd6        popl    %ebx
00001fd7        leal    0x0000001a(%ebx),%eax
00001fdd        movl    %eax,(%esp)
00001fe0        calll   0x00003005      ; symbol stub for: _puts
00001fe5        movl    $0x00001234,%eax
00001fea        addl    $0x14,%esp
00001fed        popl    %ebx
00001fee        leave
00001fef        ret
```

Mac OS 10.5.6, GCC 4.0.1
Disassembled from command line with “otool -tV”

But it all boils down to…

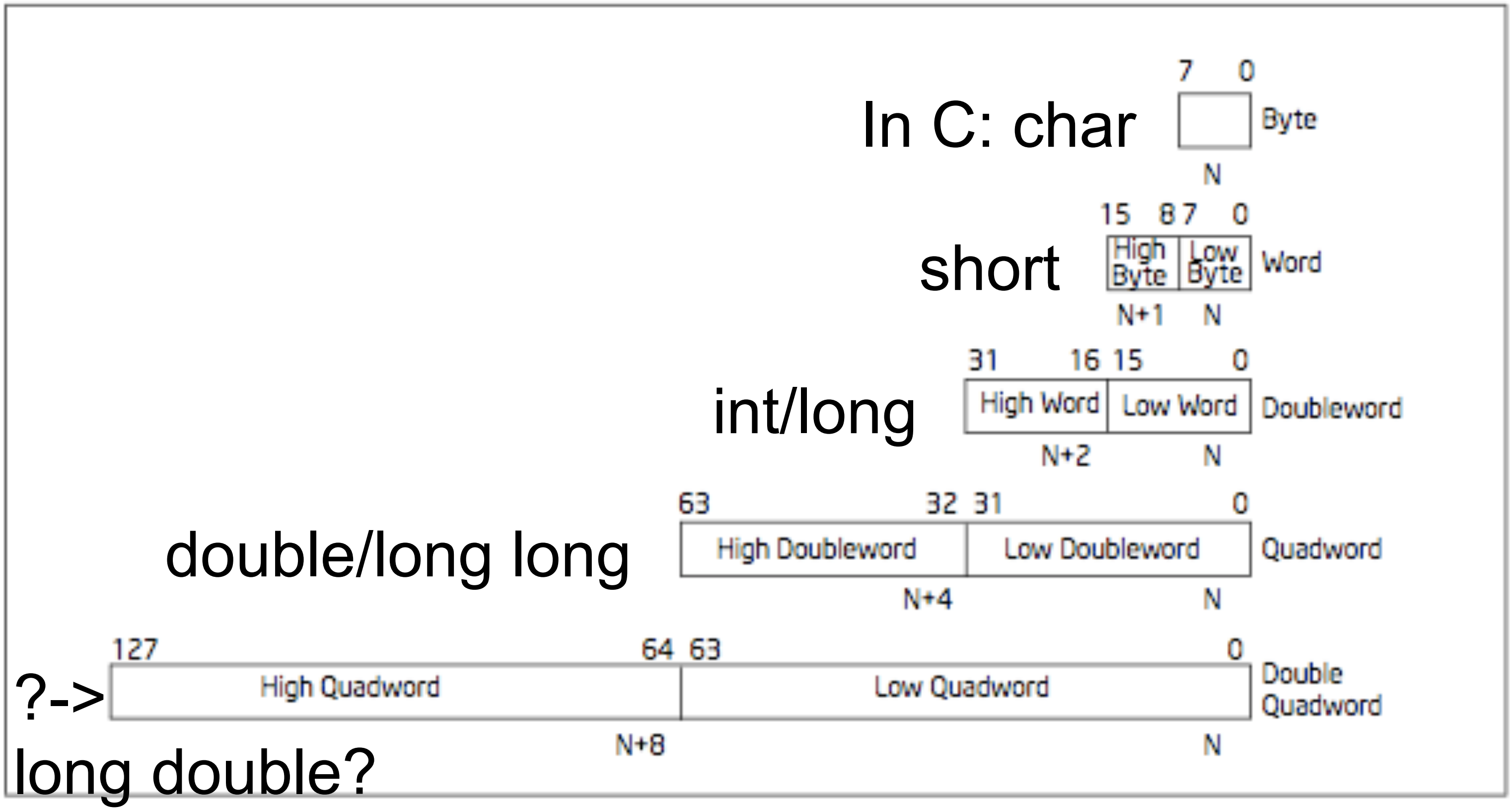
```
.text:00401000 main
.text:00401000       push    offset aHelloWorld ; "Hello world\n"
.text:00401005       call    ds:__imp__printf
.text:0040100B       pop     ecx
.text:0040100C       mov     eax, 1234h
.text:00401011       retn
```

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Optimize for minimum size (/O1) turned on
Disassembled with IDA Pro 4.9 Free Version

Take Heart!

- By one measure, only 14 assembly instructions account for 90% of code!
    - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bilar.pdf
- I think that knowing about 20-30 (not counting variations) is good enough that you will have the check the manual very infrequently
- You've already seen 11 instructions, just in the hello world variations!

Refresher - Data Types



In C: char

short

int/long

double/long long

?->
long double?

Figure 4-1. Fundamental Data Types

Refresher - Alt. Radices
Decimal, Binary, Hexidecimal

If you don't know this, you must memorize tonight

Decimal (base 10)	Binary (base 2)	Hex (base 16)
00	0000b	0x00
01	0001b	0x01
02	0010b	0x02
03	0011b	0x03
04	0100b	0x04
05	0101b	0x05
06	0110b	0x06
07	0111b	0x07
08	1000b	0x08
09	1001b	0x09
10	1010b	0x0A
11	1011b	0x0B
12	1100b	0x0C
13	1101b	0x0D
14	1110b	0x0E
15	1111b	0x0F

Refresher - Negative Numbers

- “one's complement” = flip all bits. 0->1, 1->0
- “two's complement” = one's complement + 1
- Negative numbers are defined as the “two's complement” of the positive number

Number	One's Comp.	Two's Comp. (negative)
00000001b : 0x01	11111110b : 0xFE	11111111b : 0xFF : -1
00000100b : 0x04	11111011b : 0xFB	11111100b : 0xFC : -4
00011010b : 0x1A	11100101b : 0xE5	11100110b : 0xE6 : -26
?	?	10110000b : 0xB0 : -?

- 0x01 to 0x7F positive byte, 0x80 to 0xFF negative byte
- 0x00000001 to 0x7FFFFFFF positive dword
- 0x80000000 to 0xFFFFFFFF negative dword

Architecture - CISC vs. RISC

- Intel is CISC - Complex Instruction Set Computer
  - Many very special purpose instructions that you will never see, and a given compiler may never use - just need to know how to use the manual
  - Variable-length instructions, between 1 and 16(?) bytes long.
    - 16 is max len in theory, I don't know if it can happen in practice
- Other major architectures are typically RISC - Reduced Instruction Set Computer
  - Typically more registers, less and fixed-size instructions
  - Examples: PowerPC, ARM, SPARC, MIPS

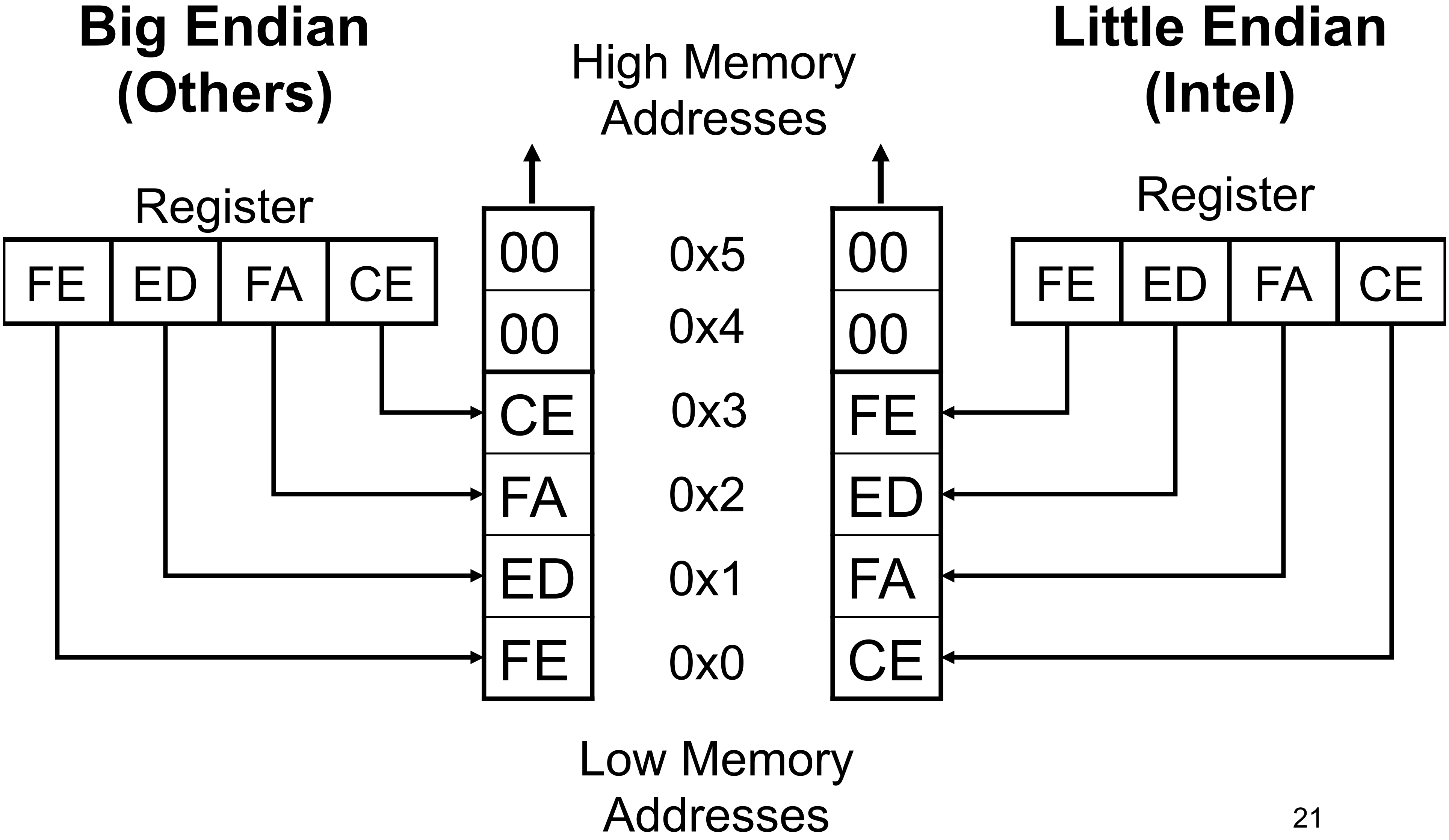
Architecture - Endian

- Endianness comes from Jonathan Swift's *Gulliver's Travels*. It doesn't matter which way you eat your eggs :)
- Little Endian - 0x12345678 stored in RAM “little end” first. The least significant byte of a word or larger is stored in the lowest address. E.g. 0x78563412
  - Intel is Little Endian
- Big Endian - 0x12345678 stored as is.
  - Network traffic is Big Endian
  - Most everyone else you've heard of (PowerPC, ARM, SPARC, MIPS) is either Big Endian by default or can be configured as either (Bi-Endian)

Book p. 163

Book for this class is “Professional Assembly Language” by Blum

Endianess pictures



Big Endian (Others)

Little Endian (Intel)

High Memory Addresses

Low Memory Addresses

Architecture - Registers

- Registers are small memory storage areas built into the processor (still volatile memory)
- 8 “general purpose” registers + the instruction pointer which points at the next instruction to execute
  - But two of the 8 are not that general
- On x86-32, registers are 32 bits long
- On x86-64, they're 64 bits

Book p. 25

Architecture - Register Conventions 1

- These are Intel's suggestions to compiler developers (and assembly handcoders). Registers don't have to be used these ways, but if you see them being used like this, you'll know why. But I simplified some descriptions. I also color coded as **GREEN** for the ones which we will actually see in *this* class (as opposed to future ones), and **RED** for not.
- **EAX** – Stores function return values
- **EBX** – Base pointer to the data section
- **ECX** – Counter for string and loop operations
- **EDX** – I/O pointer

Intel Arch v1 Section 3.4.1 - General-Purpose Registers

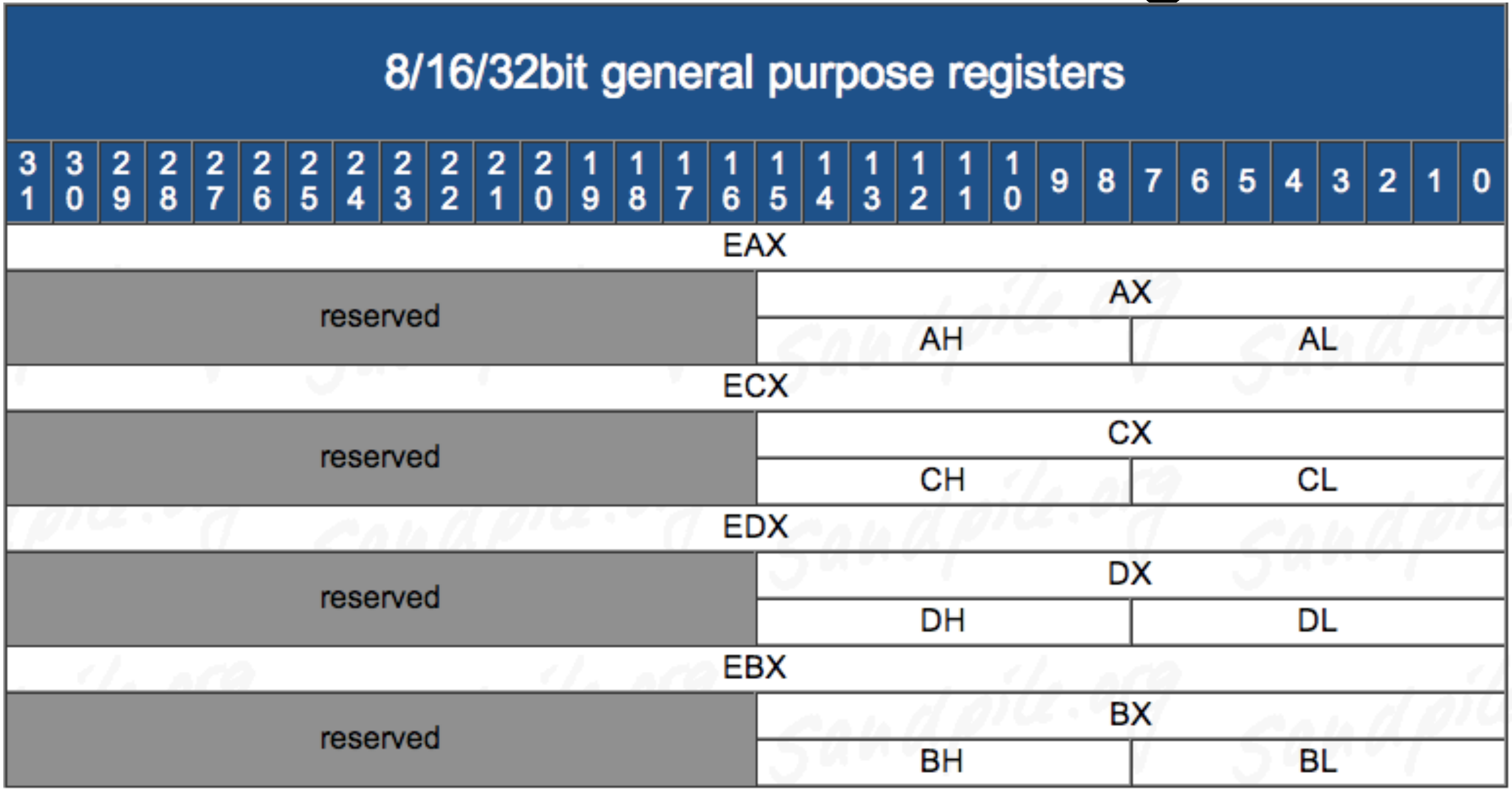
Architecture - Registers Conventions 2

- **ESI** – Source pointer for string operations
- **EDI** – Destination pointer for string operations
- **ESP** – Stack pointer
- **EBP** – Stack frame base pointer
- **EIP** - Pointer to next instruction to execute (“instruction pointer”)

Architecture - Registers Conventions 3

- Caller-save registers - eax, edx, ecx
  - If the caller has anything in the registers that it cares about, the caller is in charge of saving the value before a call to a subroutine, and restoring the value after the call returns
  - Put another way - the callee can (and is highly likely to) modify values in caller-save registers
- Callee-save registers - ebp, ebx, esi, edi
  - If the callee needs to use more registers than are saved by the caller, the callee is responsible for making sure the values are stored/restored
  - Put another way - the callee must be a good citizen and not modify registers which the caller didn't save, unless the callee itself saves and restores the existing values

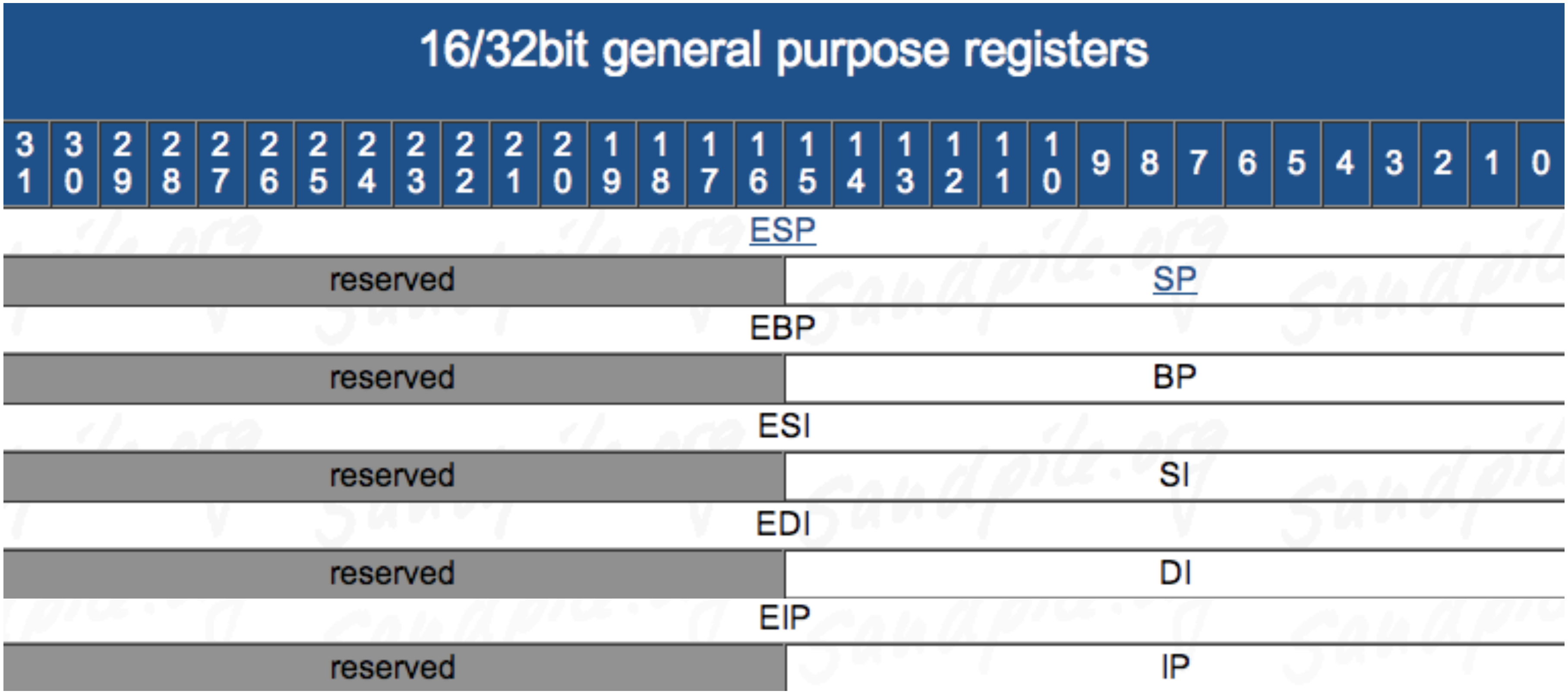
Architecture - Registers - 8/16/32 bit addressing 1



<table>
<tr><th colspan="32">8/16/32bit general purpose registers</th></tr>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="32">EAX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">AX</td></tr>
<tr><td colspan="8">AH</td><td colspan="8">AL</td></tr>
<tr><td colspan="32">ECX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">CX</td></tr>
<tr><td colspan="8">CH</td><td colspan="8">CL</td></tr>
<tr><td colspan="32">EDX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">DX</td></tr>
<tr><td colspan="8">DH</td><td colspan="8">DL</td></tr>
<tr><td colspan="32">EBX</td></tr>
<tr><td colspan="16" rowspan="2">reserved</td><td colspan="16">BX</td></tr>
<tr><td colspan="8">BH</td><td colspan="8">BL</td></tr>
</table>

http://www.sandpile.org/ia32/reg.htm

Architecture - Registers - 8/16/32 bit addressing 2



<table>
<tr><th colspan="32">16/32bit general purpose registers</th></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">ESP</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">SP</td></tr>
<tr><td colspan="32">EBP</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">BP</td></tr>
<tr><td colspan="32">ESI</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">SI</td></tr>
<tr><td colspan="32">EDI</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">DI</td></tr>
<tr><td colspan="32">EIP</td></tr>
<tr><td colspan="16">reserved</td><td colspan="16">IP</td></tr>
</table>

http://www.sandpile.org/ia32/reg.htm

Architecture - EFLAGS

- EFLAGS register holds many single bit flags. Will only ask you to remember the following for now.
  - Zero Flag (ZF) - Set if the result of some instruction is zero; cleared otherwise.
  - Sign Flag (SF) - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

Intel Vol 1 Sec 3.4.3 - page 3-20

1

Your first x86 instruction: NOP

- NOP - No Operation! No registers, no values, no nothin'!
- Just there to pad/align bytes, or to delay time
- Bad guys use it to make simple exploits more reliable. But that's another class ;)

Extra! Extra! Late-breaking NOP news!

- Amaze those who know x86 by citing this interesting bit of trivia:
- “The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.”
  - I had never looked in the manual for NOP apparently :)
- Every other person I had told this to had never heard it either.
- Thanks to Jon Erickson for cluing me in to this.
- XCHG instruction is not officially in this class. But if I hadn't just told you what it does, I bet you would have guessed right anyway.

XCHG in book p. 112

The Stack

- The stack is a conceptual area of main memory (RAM) which is designated by the OS when a program is started.
  - Different OS start it at different addresses by convention
- A stack is a Last-In-First-Out (LIFO/FILO) data structure where data is "pushed" on to the top of the stack and "popped" off the top.
- By convention the stack grows toward lower memory addresses. Adding something to the stack means the top of the stack is now at a lower memory address.

The Stack 2

- As already mentioned, esp points to the top of the stack, the lowest address which is being used
  - While data will exist at addresses beyond the top of the stack, it is considered undefined
- The stack keeps track of which functions were called before the current one, it holds local variables and is frequently used to pass arguments to the next function to be called.
- A firm understanding of what is happening on the stack is ***essential*** to understanding a program's operation.

2

PUSH - Push Word, Doubleword or Quadword onto the Stack

- For our purposes, it will always be a DWORD (4 bytes).
  - Can either be an immediate (a numeric constant), or the value in a register
- The push instruction automatically decrements the stack pointer, esp, by 4.

Book p. 120

push eax

Registers Before

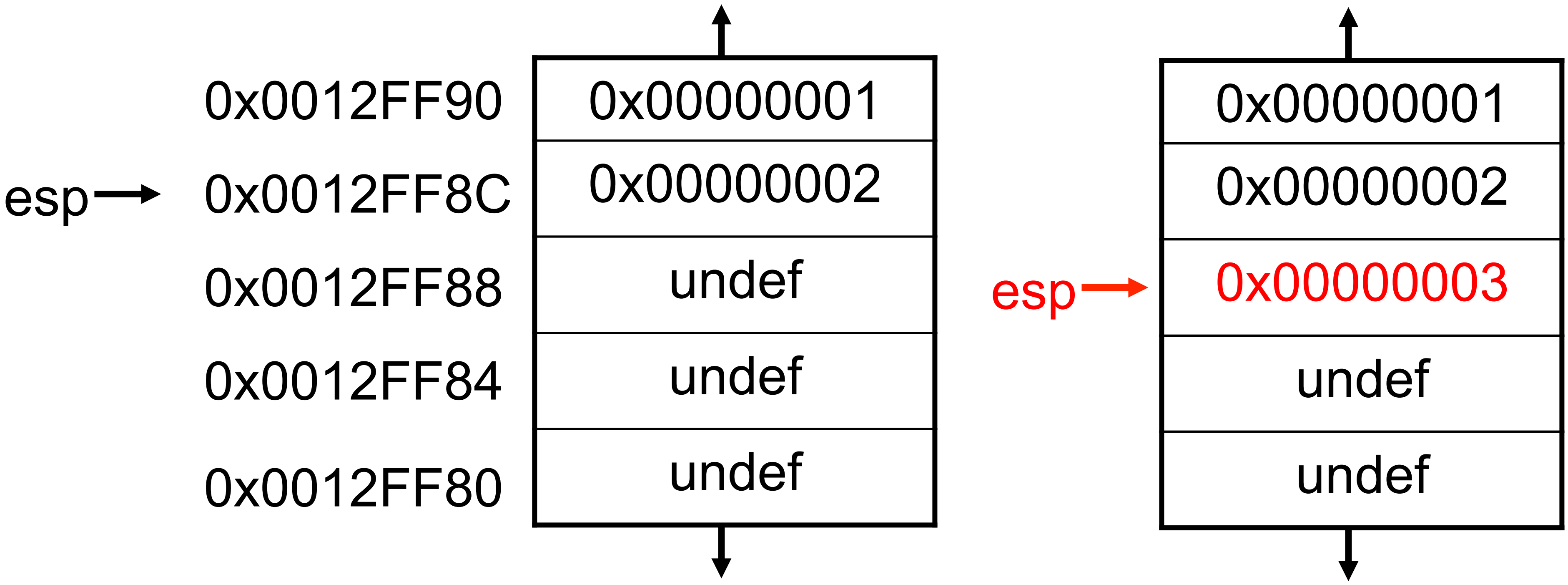
eax	0x00000003
esp	0x0012FF8C

Registers After

eax	0x00000003
esp	0x0012FF88

Stack Before

Stack After



	Stack Before	Stack After
0x0012FF90	0x00000001	0x00000001
esp → 0x0012FF8C	0x00000002	0x00000002
0x0012FF88	undef	esp → 0x00000003
0x0012FF84	undef	undef
0x0012FF80	undef	undef

3

POP- Pop a Value from the Stack

- Take a DWORD off the stack, put it in a register, and increment esp by 4

Book p. 120

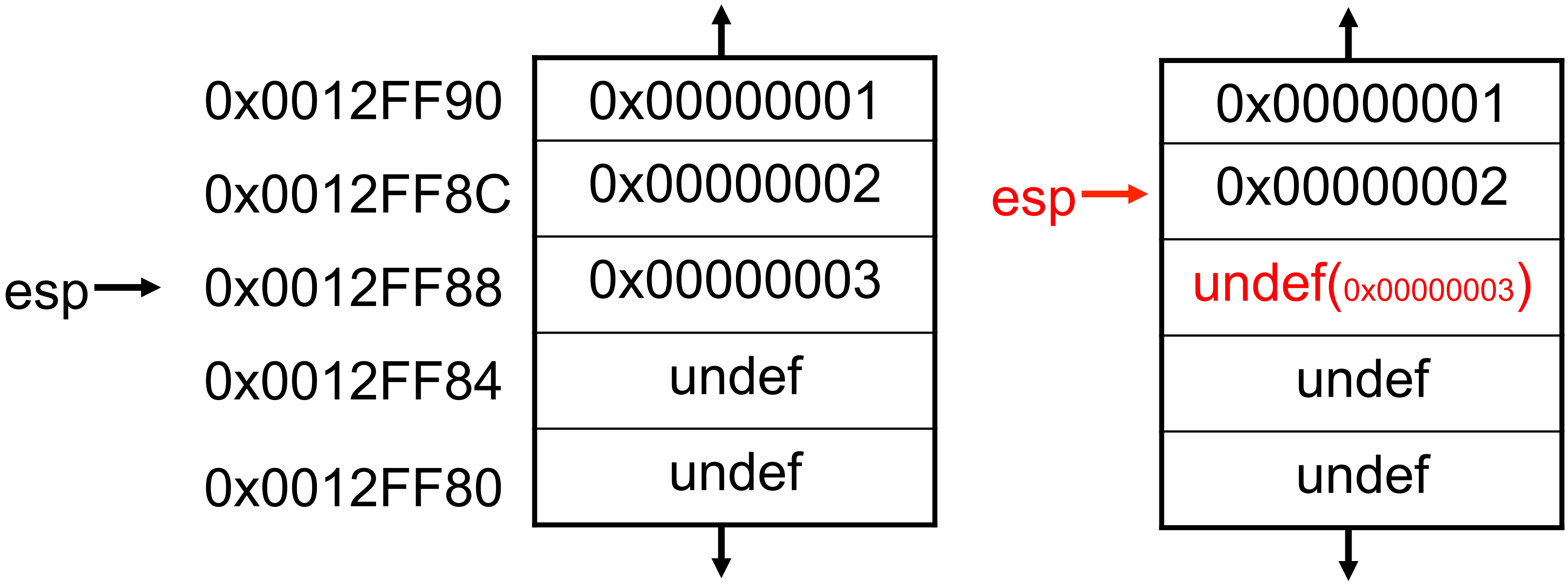
pop eax

Registers Before

eax	0xFFFFFFFF
esp	0x0012FF88

Registers After

eax	0x00000003
esp	0x0012FF8C



Stack Before

	0x0012FF90	0x00000001
	0x0012FF8C	0x00000002
esp →	0x0012FF88	0x00000003
	0x0012FF84	undef
	0x0012FF80	undef

Stack After

	0x00000001
esp →	0x00000002
	undef(0x00000003)
	undef
	undef

Calling Conventions

- How code calls a subroutine is compiler-dependent and configurable. But there are a few conventions.
- We will only deal with the “cdecl” and “stdcall” conventions.
- More info at
  - http://en.wikipedia.org/wiki/X86_calling_conventions
  - http://www.programmersheaven.com/2/Calling-conventions

Calling Conventions - cdecl

- “C declaration” - most common calling convention
- Function parameters pushed onto stack right to left
- Saves the old stack frame pointer and sets up a new stack frame
- eax or edx:eax returns the result for primitive data types
- Caller is responsible for cleaning up the stack

Calling Conventions - stdcall

- I typically only see this convention used by Microsoft C++ code - e.g. Win32 API
- Function parameters pushed onto stack right to left
- Saves the old stack frame pointer and sets up a new stack frame
- eax or edx:eax returns the result for primitive data types
- Callee responsible for cleaning up any stack parameters it takes
- Aside: typical MS, “If I call my new way of doing stuff 'standard' it must be true!”

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 eip 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)

Book p. 132

5

RET - Return from Procedure

- Two forms
  - – Pop the top of the stack into eip (remember pop increments stack pointer)
    - In this form, the instruction is just written as “ret”
    - Typically used by cdecl functions
  - – Pop the top of the stack into eip and add a constant number of bytes to esp
    - In this form, the instruction is written as “ret 0x8”, or “ret 0x20”, etc
    - Typically used by stdcall functions

Kinda book p. 133

6

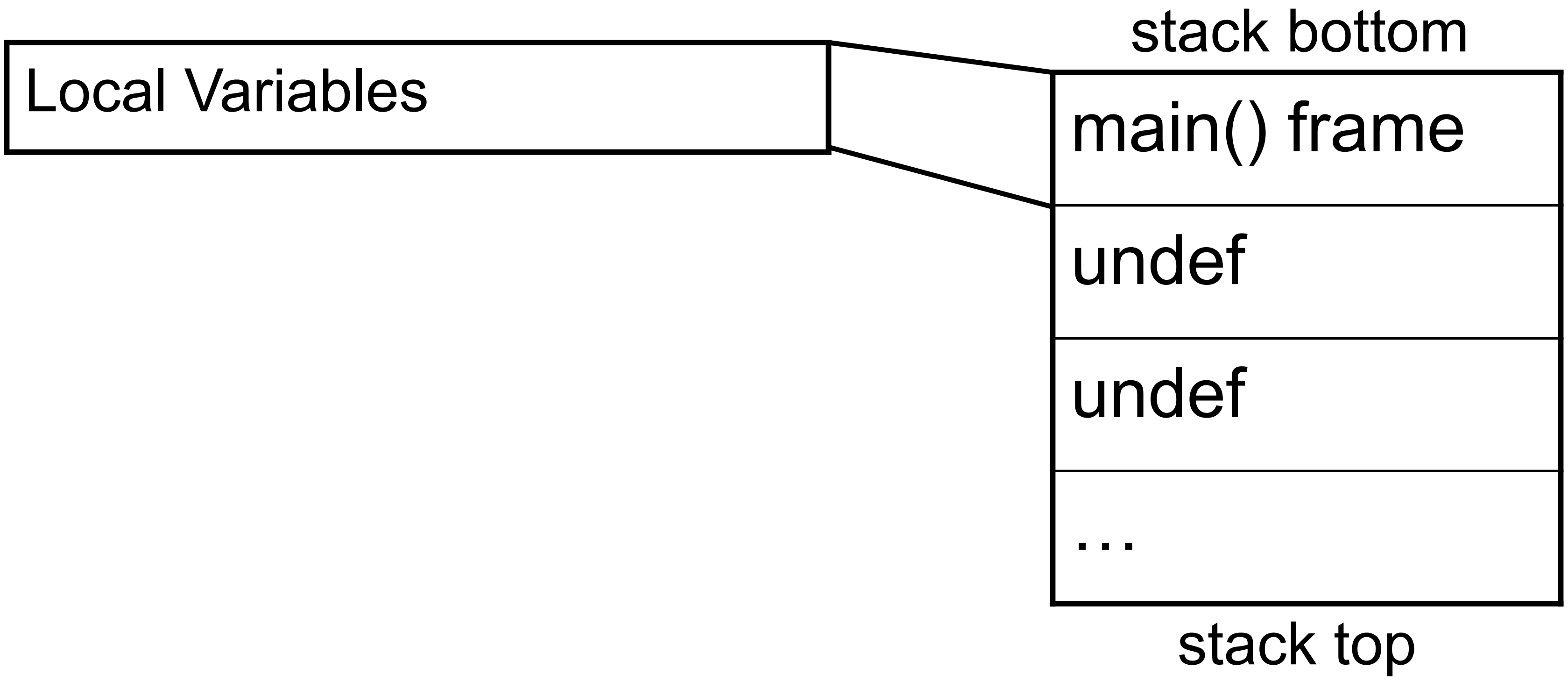
MOV - Move

- 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/m32 form talked about later

Book p. 97

General Stack Frame Operation

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).



Local Variables

stack bottom

main() frame

undef

undef

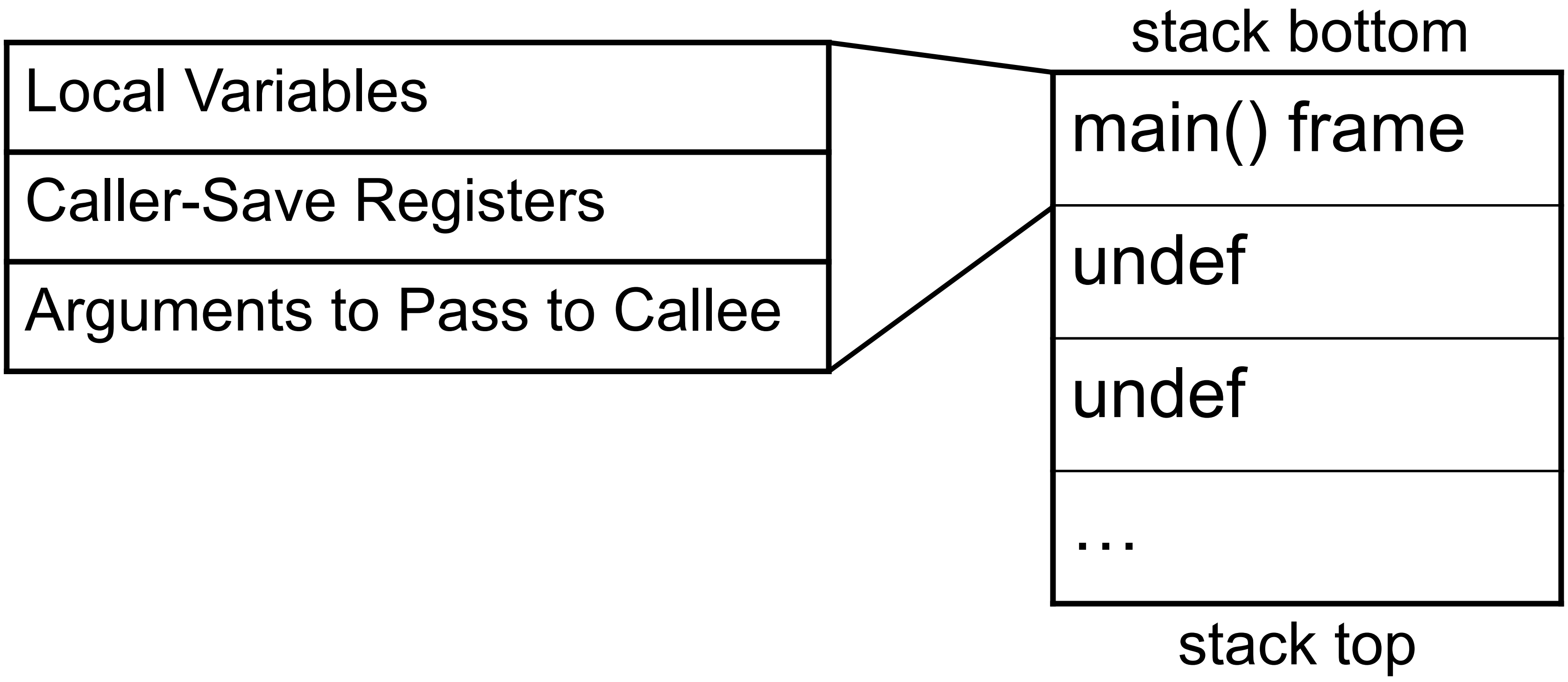
…

stack top

Book p. 306

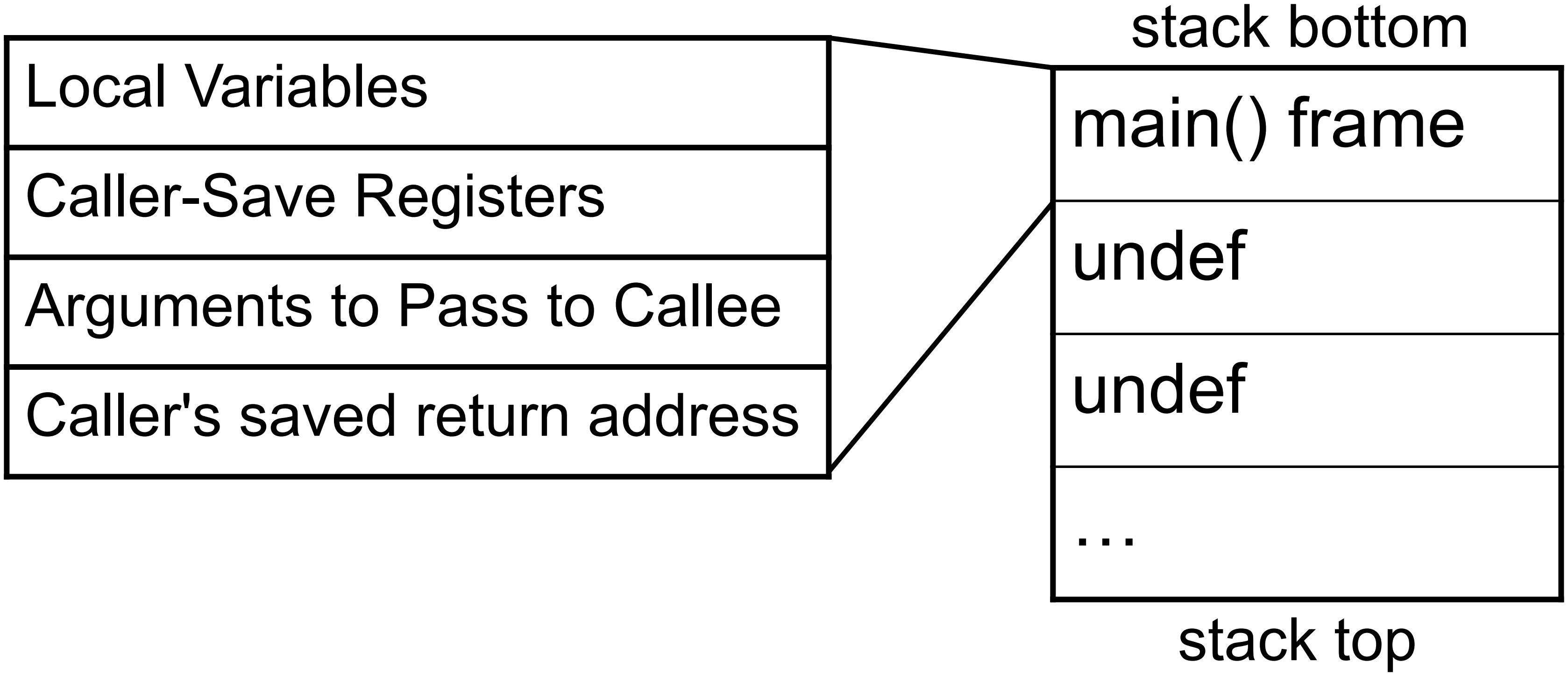
General Stack Frame Operation 2

When main() decides to call a subroutine, main() becomes “the caller”. We will assume main() has some registers it would like to remain the same, so it will save them. We will also assume that the callee function takes some input arguments.



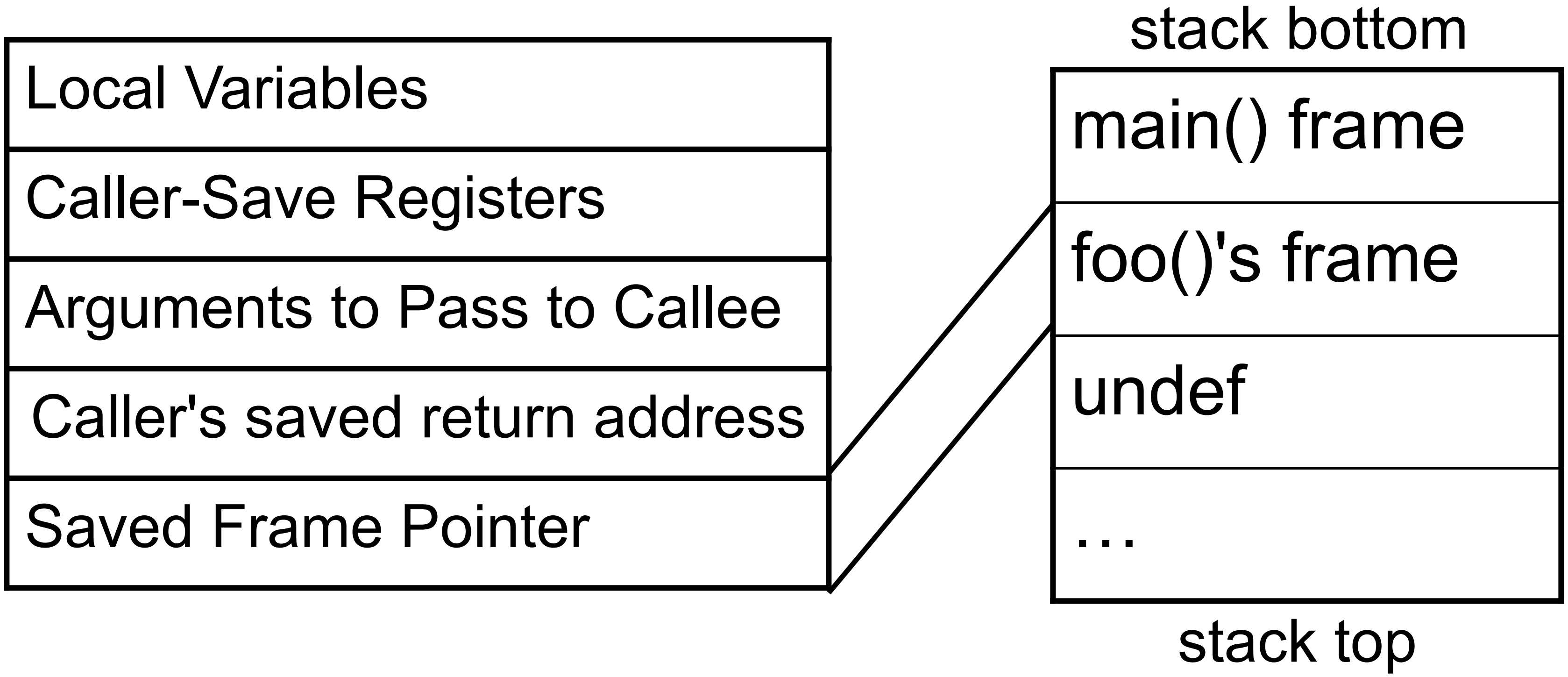
General Stack Frame Operation 3

When main() actually issues the CALL instruction, the return address gets saved onto the stack, and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.



General Stack Frame Operation 4

When foo() starts, the frame pointer (ebp) still points to main()'s frame. So the first thing it does is to save the old frame pointer on the stack and set the new value to point to its own frame.

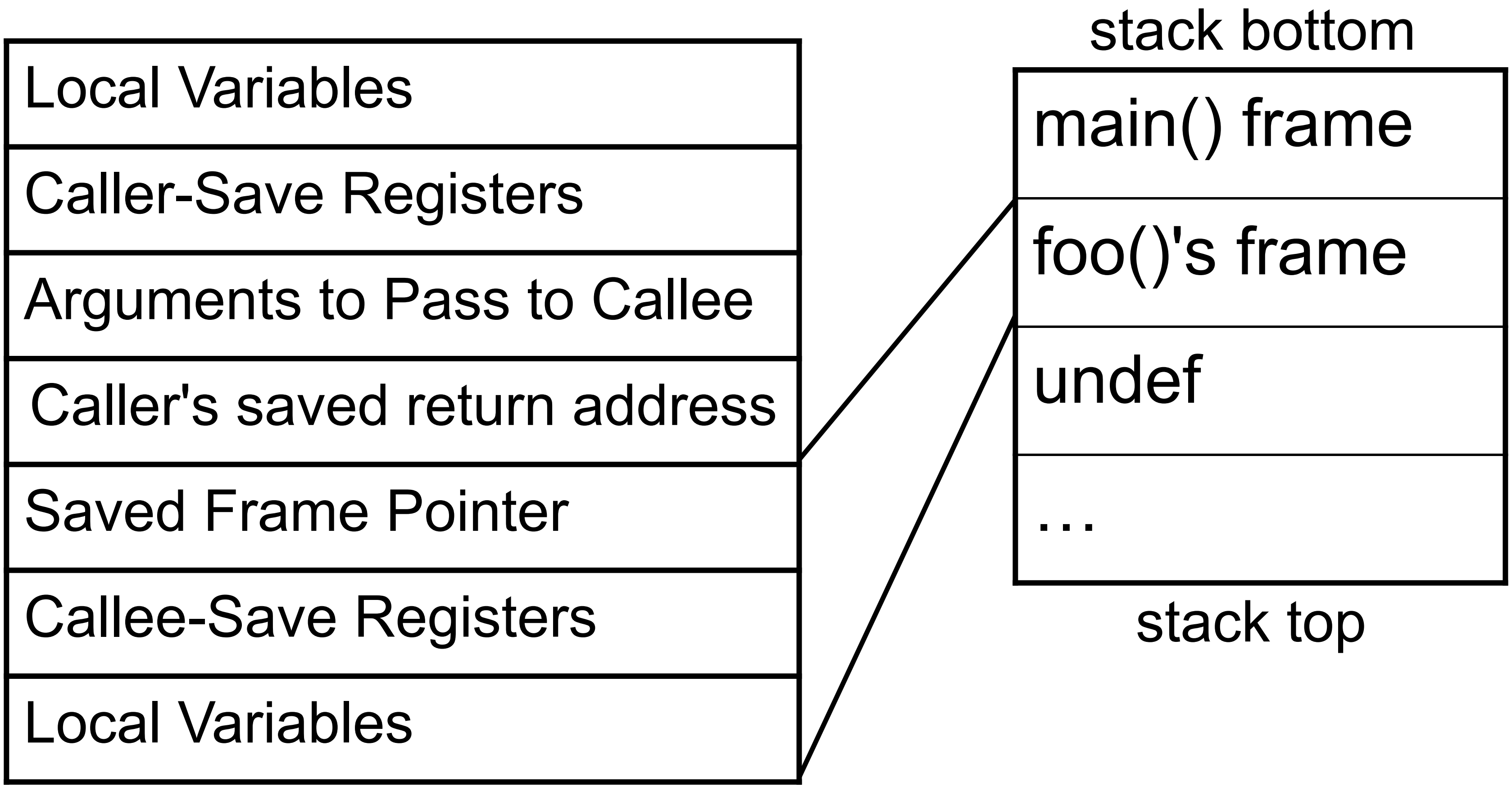


Local Variables
Caller-Save Registers
Arguments to Pass to Callee
Caller's saved return address
Saved Frame Pointer

stack bottom
main() frame
foo()'s frame
undef
…
stack top

General Stack Frame Operation 5

Next, we'll assume the the callee foo() would like to use all the registers, and must therefore save the callee-save registers. Then it will allocate space for its local variables.



Local Variables

Caller-Save Registers

Arguments to Pass to Callee

Caller's saved return address

Saved Frame Pointer

Callee-Save Registers

Local Variables

stack bottom

main() frame

foo()'s frame

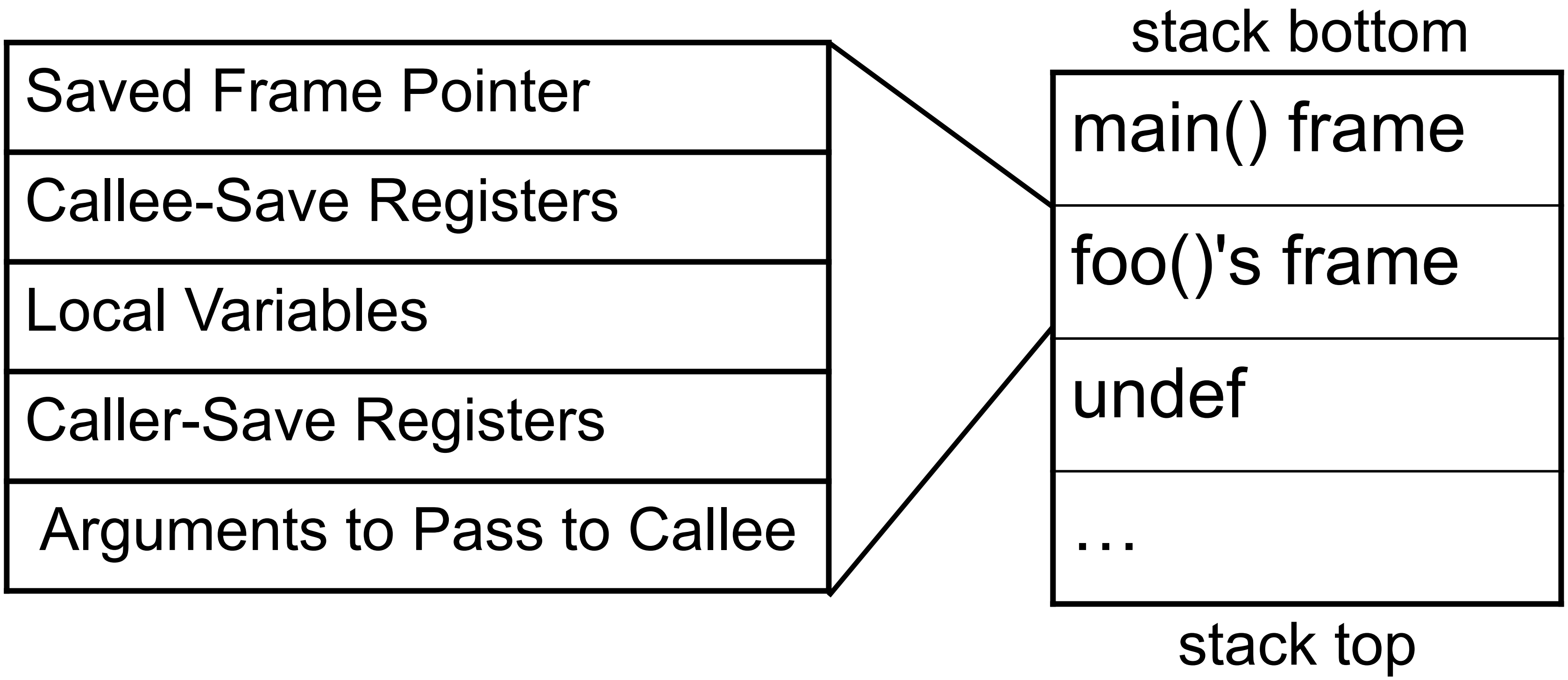
undef

…

stack top

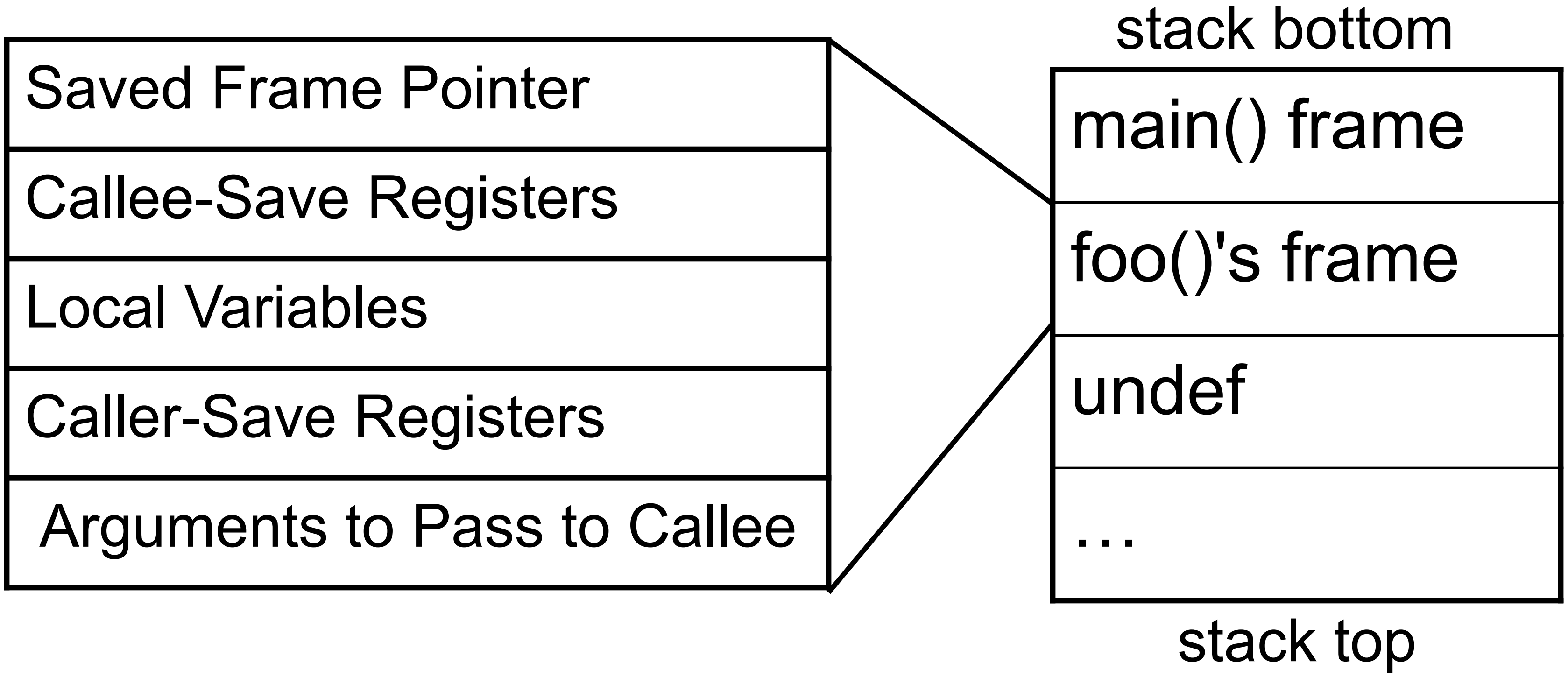
General Stack Frame Operation 6

At this point, foo() decides it wants to call bar(). It is still the callee-of-main(), but it will now be the caller-of-bar. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well.



General Stack Frame Layout

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.) But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omit frame pointers, changing calling convention to pass arguments in registers, etc.)



Saved Frame Pointer

Callee-Save Registers

Local Variables

Caller-Save Registers

Arguments to Pass to Callee

stack bottom

main() frame

foo()'s frame

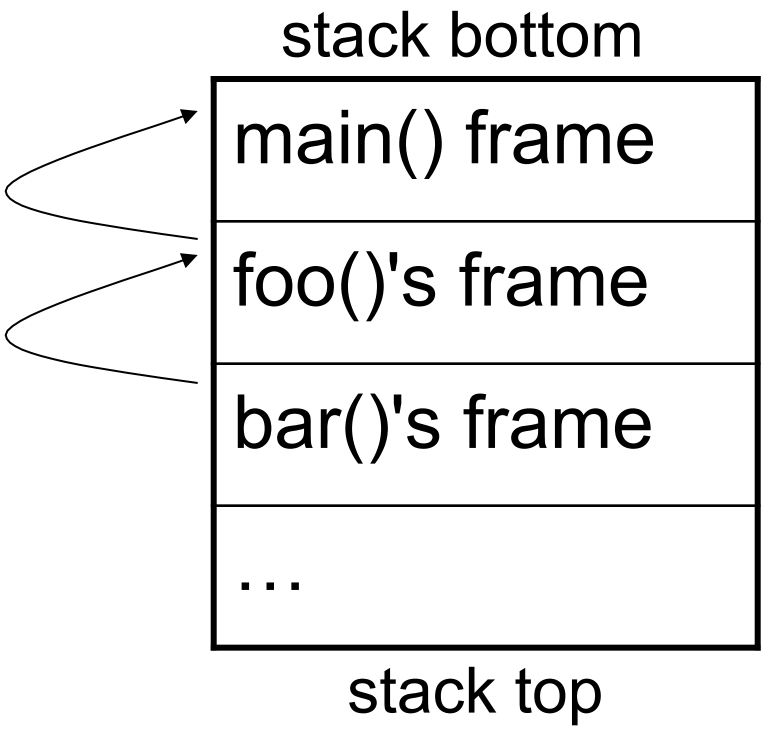
undef

…

stack top

Stack Frames are a Linked List!

The ebp in the current frame points at the saved ebp of the previous frame.



Example1.c

The stack frames in this example will be very simple.
Only saved frame pointer (ebp) and saved return addresses (eip).

```
//Example1 - using the stack
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int sub(){
    return 0xbeef;
}
int main(){
    sub();
    return 0xf00d;
}
```

```
sub:
00401000  push         ebp
00401001  mov          ebp,esp
00401003  mov          eax,0BEEFh
00401008  pop          ebp
00401009  ret
main:
00401010  push         ebp
00401011  mov          ebp,esp
00401013  call         sub (401000h)
00401018  mov          eax,0F00Dh
0040101D  pop          ebp
0040101E  ret
```

Example1.c 1:

EIP = 00401010, but no instruction yet executed

eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF6C ⌘

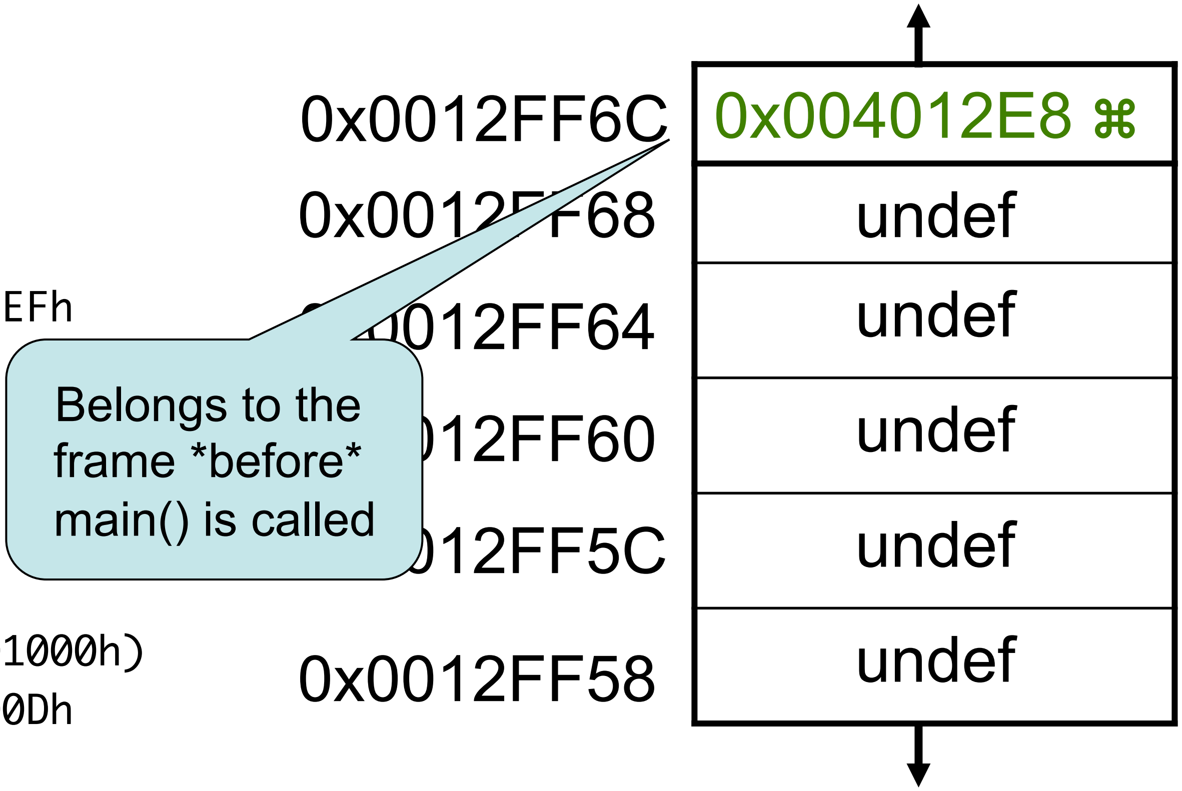
Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```



0x0012FF6C	0x004012E8 ⌘
0x0012FF68	undef
0012FF64	undef
12FF60	undef
12FF5C	undef
0x0012FF58	undef

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

Example1.c 2

eax	0x003435C0 ⌘
ebp	0x0012FFB8 ⌘
esp	0x0012FF68 ♍

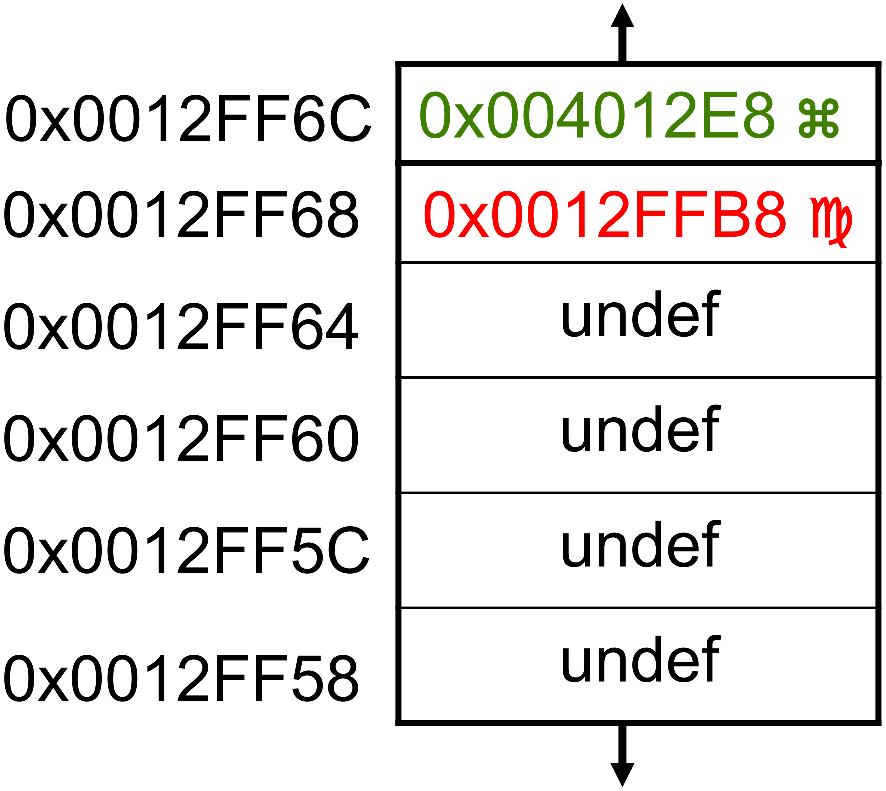
Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
sub:
00401000   push      ebp
00401001   mov       ebp,esp
00401003   mov       eax,0BEEFh
00401008   pop       ebp
00401009   ret
main:
00401010   push      ebp  ☒
00401011   mov       ebp,esp
00401013   call      sub (401000h)
00401018   mov       eax,0F00Dh
0040101D   pop       ebp
0040101E   ret
```



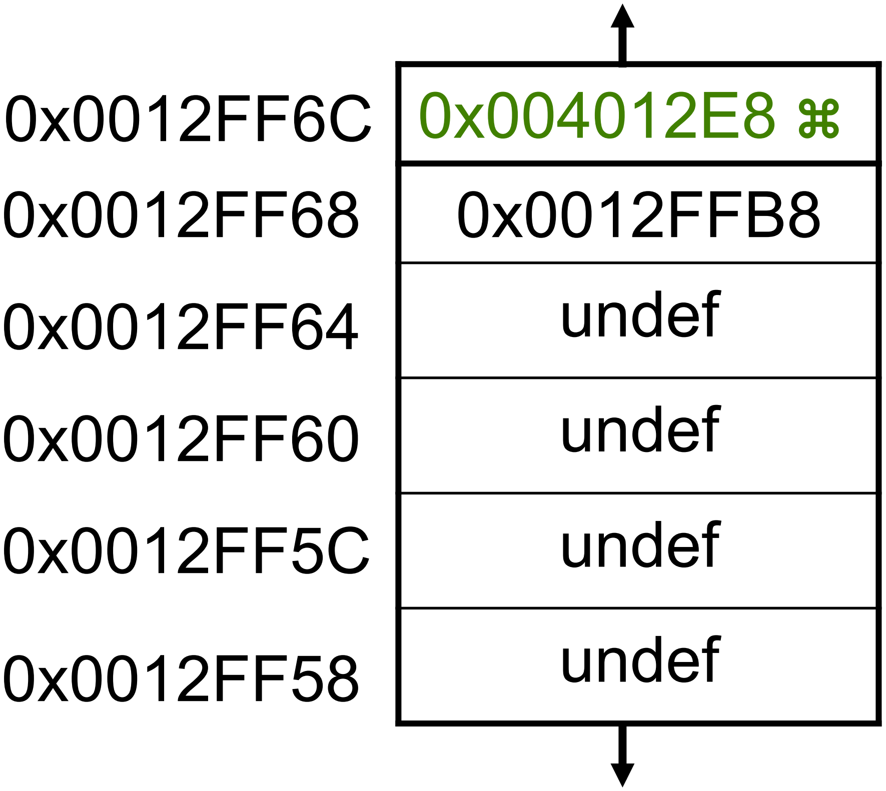
0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8 ♍
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 3

eax	0x003435C0 ⌘
ebp	0x0012FF68 ♍
esp	0x0012FF68

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
sub:
00401000   push        ebp
00401001   mov         ebp,esp
00401003   mov         eax,0BEEFh
00401008   pop         ebp
00401009   ret
main:
00401010   push        ebp
00401011   mov         ebp,esp ☒
00401013   call        sub (401000h)
00401018   mov         eax,0F00Dh
0040101D   pop         ebp
0040101E   ret
```



0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 4

eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF64 ♍

Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
sub:
00401000  push    ebp
00401001  mov     ebp,esp
00401003  mov     eax,0BEEFh
00401008  pop     ebp
00401009  ret
main:
00401010  push    ebp
00401011  mov     ebp,esp
00401013  call    sub (401000h) ☒
00401018  mov     eax,0F00Dh
0040101D  pop     ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018 ♍
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 5

eax	0x003435C0 ⌘
ebp	0x0012FF68
esp	0x0012FF60 ♍

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

```
sub:
00401000  push        ebp ☒
00401001  mov         ebp,esp
00401003  mov         eax,0BEEFh
00401008  pop         ebp
00401009  ret
main:
00401010  push        ebp
00401011  mov         ebp,esp
00401013  call        sub (401000h)
00401018  mov         eax,0F00Dh
0040101D  pop         ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68 ♍
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 6

eax	0x003435C0 ⌘
ebp	0x0012FF60 ♍
esp	0x0012FF60

Key:

☒ **executed instruction,**

♍ **modified value**

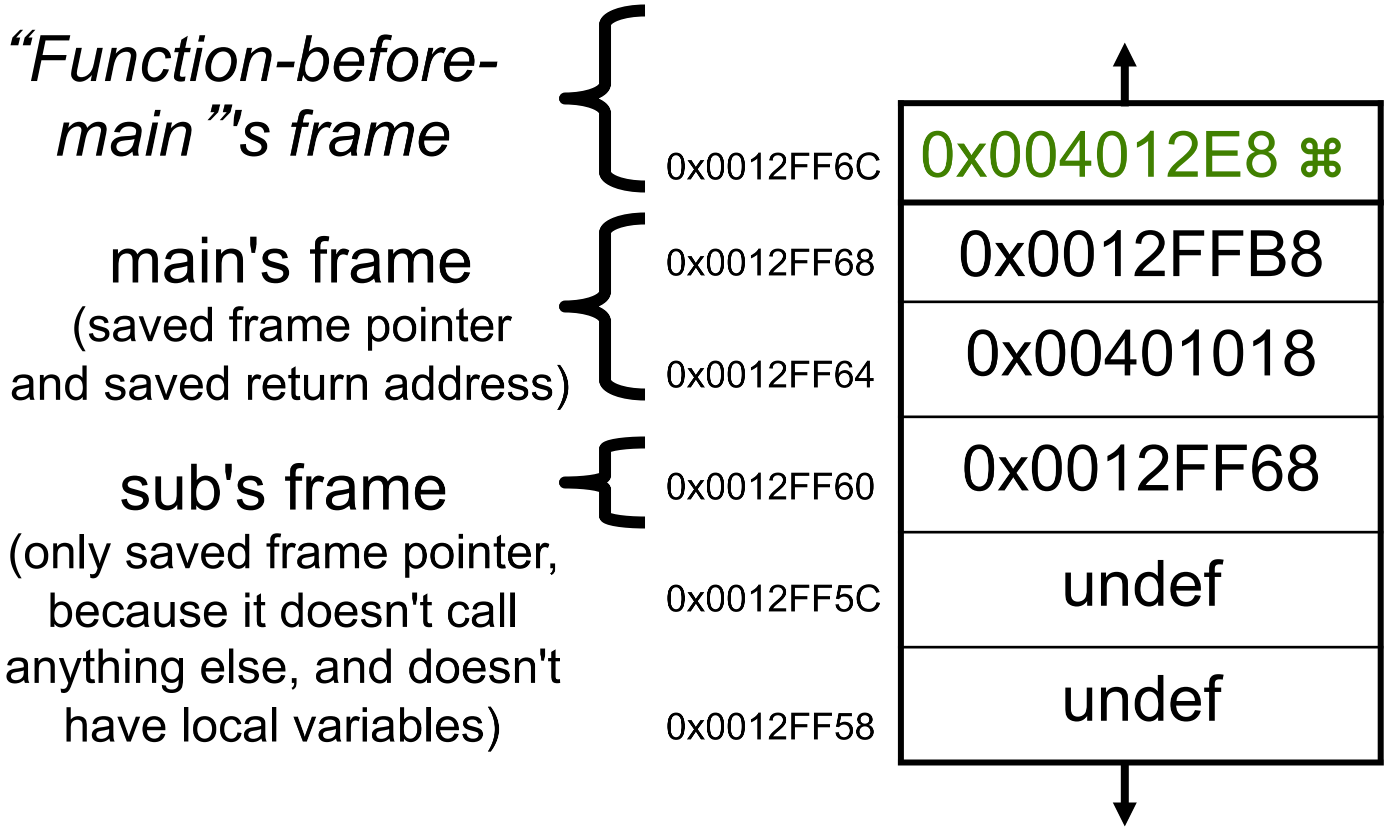
⌘ **start value**

```
sub:
00401000   push        ebp
00401001   mov         ebp,esp ☒
00401003   mov         eax,0BEEFh
00401008   pop         ebp
00401009   ret
main:
00401010   push        ebp
00401011   mov         ebp,esp
00401013   call        sub (401000h)
00401018   mov         eax,0F00Dh
0040101D   pop         ebp
0040101E   ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 6
STACK FRAME TIME OUT

```
 sub
push    ebp
mov     ebp, esp
mov     eax, 0BEEFh
pop     ebp
retn
main
push    ebp
mov     ebp, esp
call    _sub
mov     eax, 0F00Dh
pop     ebp
retn
```



"Function-before-main"'s frame

main's frame
(saved frame pointer and saved return address)

sub's frame
(only saved frame pointer, because it doesn't call anything else, and doesn't have local variables)

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 7

eax	0x0000BEEF
ebp	0x0012FF60
esp	0x0012FF60

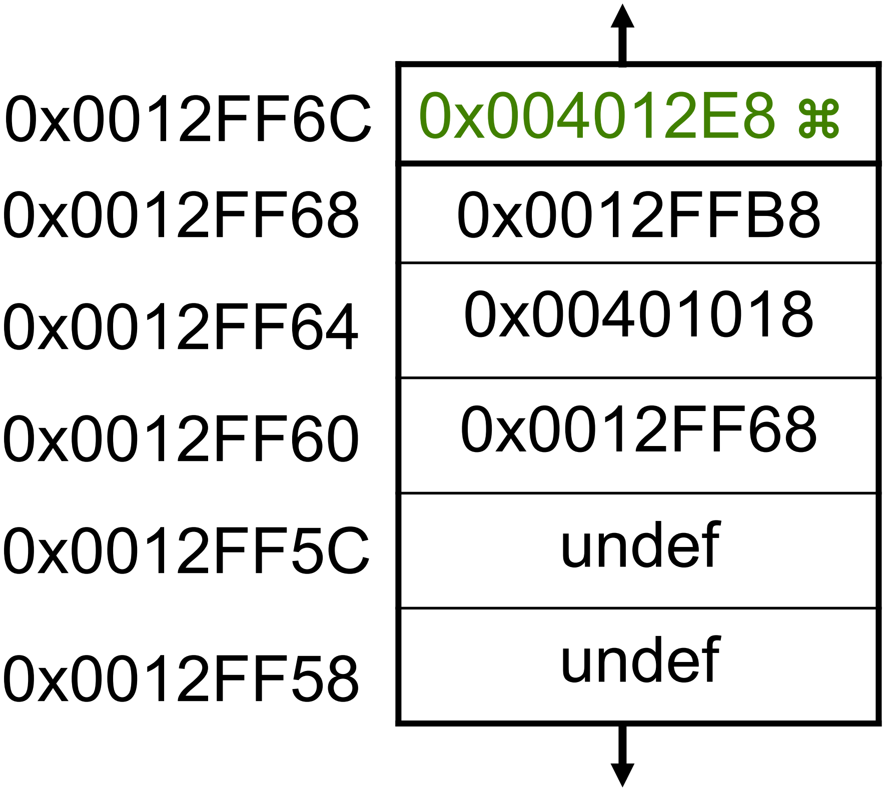
Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
sub:
00401000  push        ebp
00401001  mov         ebp,esp
00401003  mov         eax,0BEEFh ☒
00401008  pop         ebp
00401009  ret
main:
00401010  push        ebp
00401011  mov         ebp,esp
00401013  call        sub (401000h)
00401018  mov         eax,0F00Dh
0040101D  pop         ebp
0040101E  ret
```



0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 8

eax	0x0000BEEF
ebp	0x0012FF68 ♍
esp	0x0012FF64 ♍

Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp ☒
00401009  ret
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	undef ♍
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 9

eax	0x0000BEEF
ebp	0x0012FF68
esp	0x0012FF68 ♍

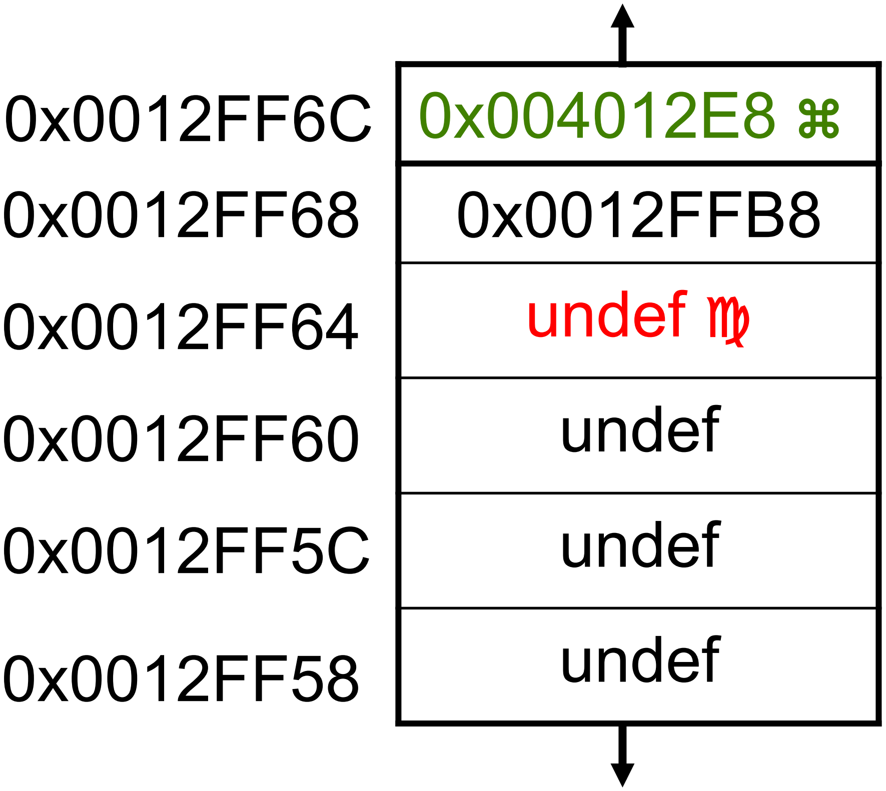
Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
sub:
00401000  push      ebp
00401001  mov       ebp,esp
00401003  mov       eax,0BEEFh
00401008  pop       ebp
00401009  ret ☒
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  call      sub (401000h)
00401018  mov       eax,0F00Dh
0040101D  pop       ebp
0040101E  ret
```



0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	undef ♍
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 9

eax	0x0000F00D ♍
ebp	0x0012FF68
esp	0x0012FF68

Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
sub:
00401000  push        ebp
00401001  mov         ebp,esp
00401003  mov         eax,0BEEFh
00401008  pop         ebp
00401009  ret
main:
00401010  push        ebp
00401011  mov         ebp,esp
00401013  call        sub (401000h)
00401018  mov         eax,0F00Dh ☒
0040101D  pop         ebp
0040101E  ret
```

0x0012FF6C	0x004012E8 ⌘
0x0012FF68	0x0012FFB8
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 10

eax	0x0000F00D
ebp	0x0012FFB8 ♍
esp	0x0012FF6C ♍

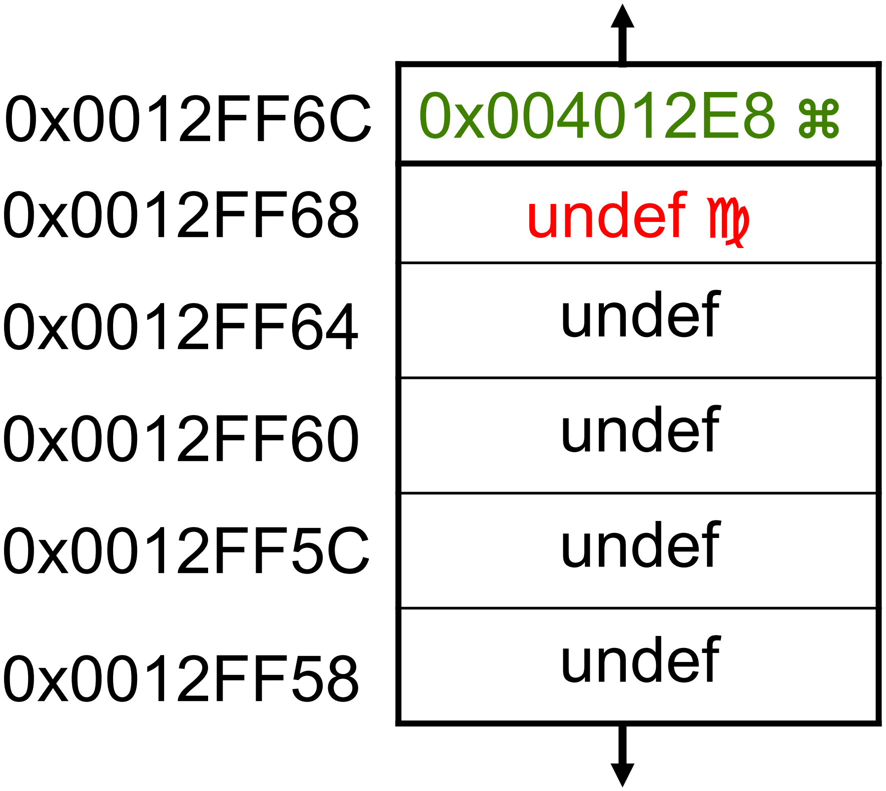
Key:

☒ executed instruction,

♍ modified value

⌘ start value

```
sub:
00401000  push    ebp
00401001  mov     ebp,esp
00401003  mov     eax,0BEEFh
00401008  pop     ebp
00401009  ret
main:
00401010  push    ebp
00401011  mov     ebp,esp
00401013  call    sub (401000h)
00401018  mov     eax,0F00Dh
0040101D  pop     ebp ☒
0040101E  ret
```



0x0012FF6C	0x004012E8 ⌘
0x0012FF68	undef ♍
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 11

eax	0x0000F00D
ebp	0x0012FFB8
esp	0x0012FF70 ♍

Key:

☒ **executed instruction,**

♍ **modified value**

⌘ **start value**

```
sub:
00401000  push       ebp
00401001  mov        ebp,esp
00401003  mov        eax,0BEEFh
00401008  pop        ebp
00401009  ret
main:
00401010  push       ebp
00401011  mov        ebp,esp
00401013  call       sub (401000h)
00401018  mov        eax,0F00Dh
0040101D  pop        ebp
0040101E  ret ☒
```

0x0012FF6C	undef ♍
0x0012FF68	undef
0x0012FF64	undef
0x0012FF60	undef
0x0012FF5C	undef
0x0012FF58	undef

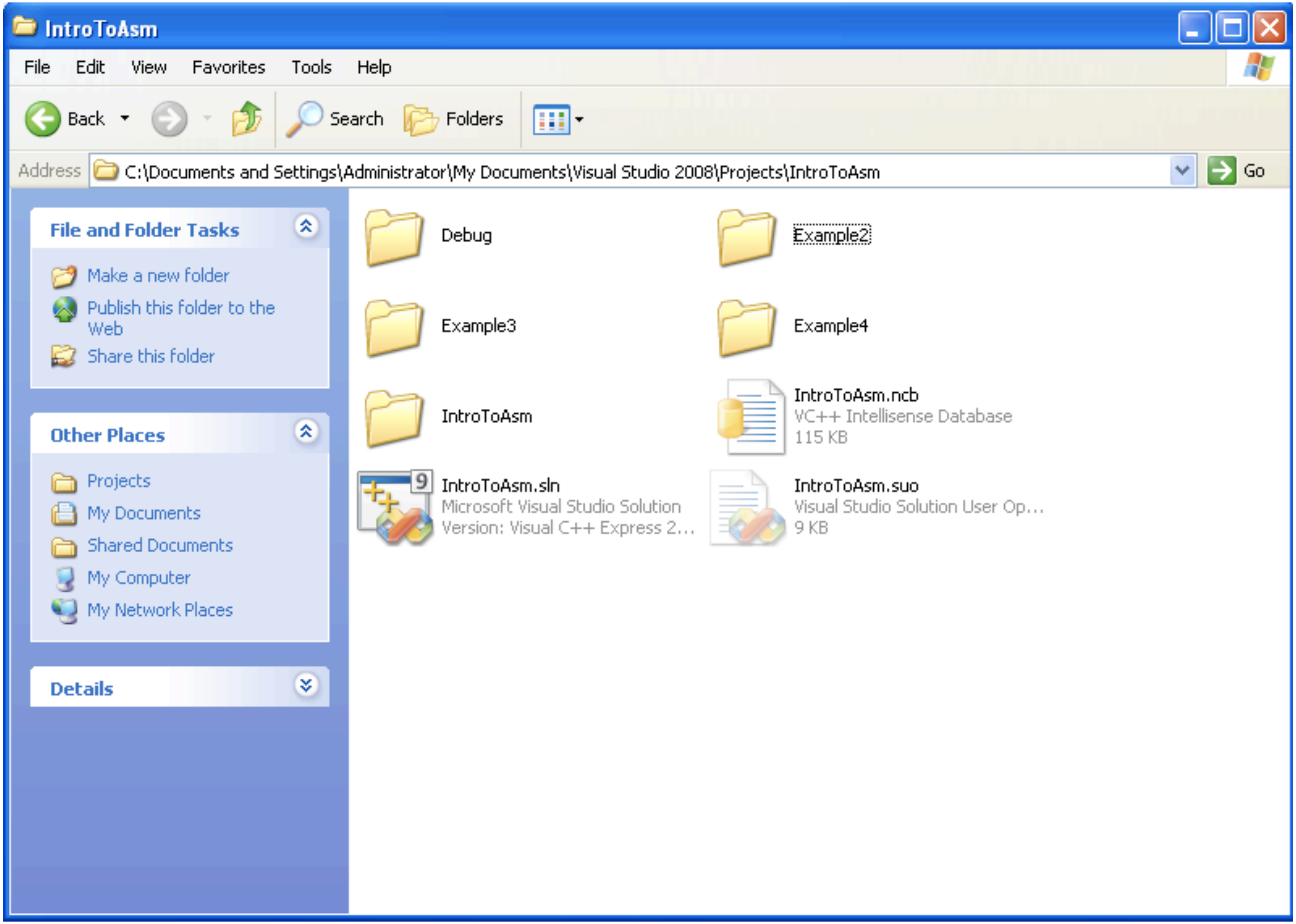
Execution would continue at the value ret removed from the stack: 0x004012E8

Example1 Notes

- sub() is deadcode - its return value is not used for anything, and main always returns 0xF00D. If optimizations are turned on in the compiler, it would remove sub()
- Because there are no input parameters to sub(), there is no difference whether we compile as cdecl vs stdcall calling conventions

Let's do that in a tool

- Visual C++ 2008 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
- Can't move applications to other systems without installing the “redistributable libraries”



C:\Documents and Settings\Administrator\My Documents\Visual Studio 2008\Projects\IntroToAsm

Debug

Example2

Example3

Example4

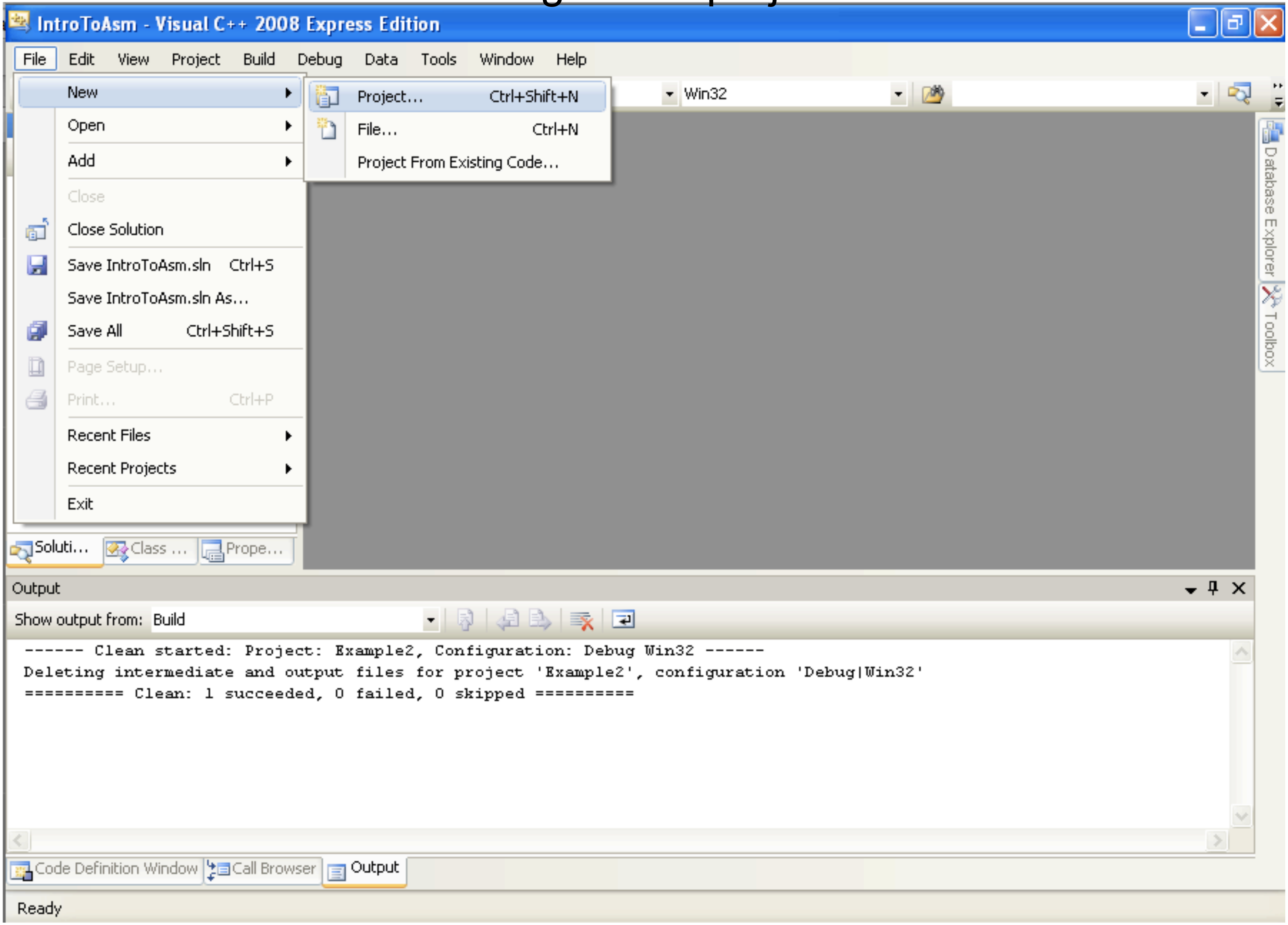
IntroToAsm

IntroToAsm.ncb
VC++ Intellisense Database
115 KB

IntroToAsm.sln
Microsoft Visual Studio Solution
Version: Visual C++ Express 2…

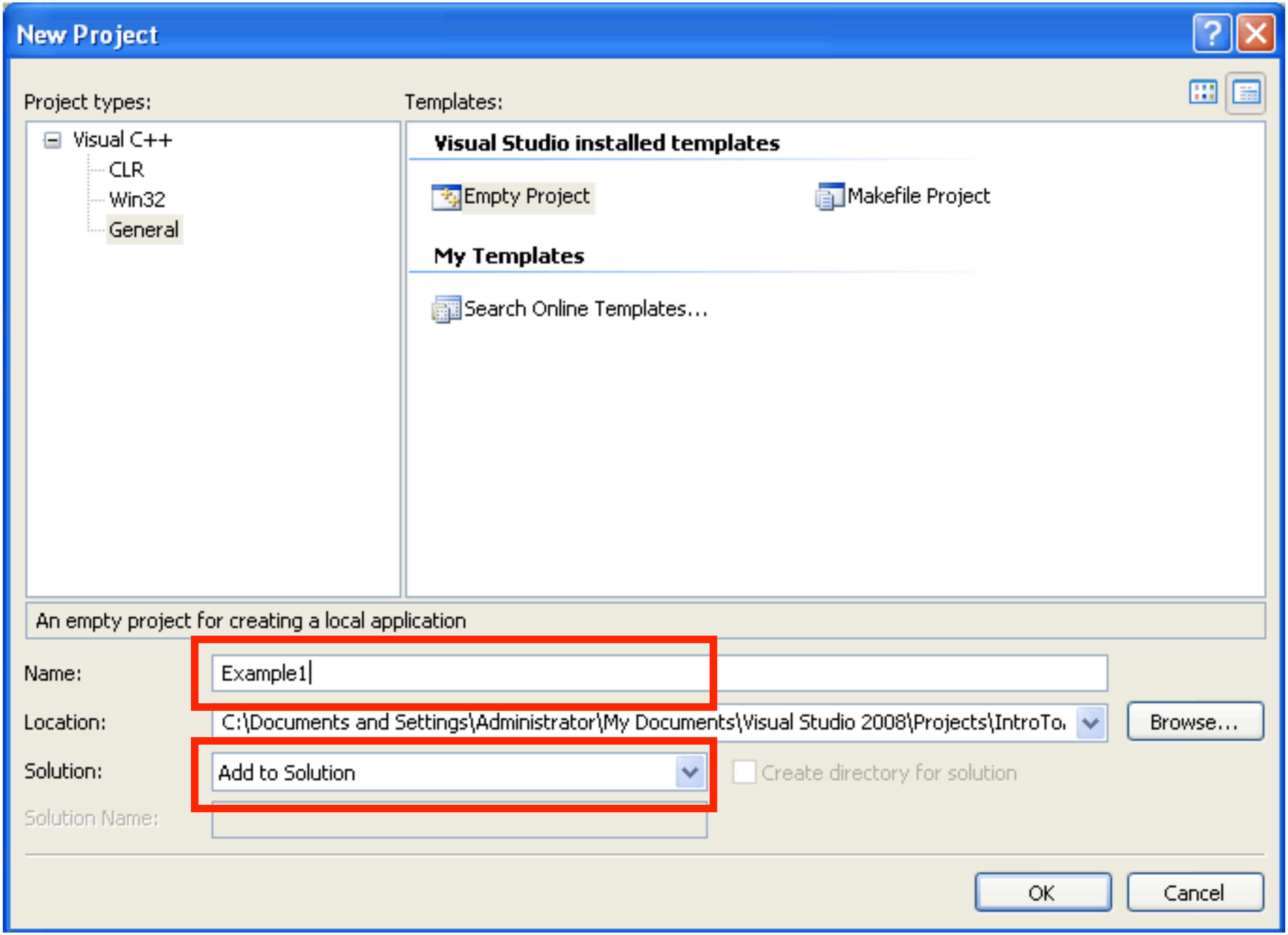
IntroToAsm.suo
Visual Studio Solution User Op…
9 KB

Creating a new project - 1



```
------ Clean started: Project: Example2, Configuration: Debug Win32 ------
Deleting intermediate and output files for project 'Example2', configuration 'Debug|Win32'
========== Clean: 1 succeeded, 0 failed, 0 skipped ==========
```

Creating a new project - 2



An empty project for creating a local application

Name: Example1

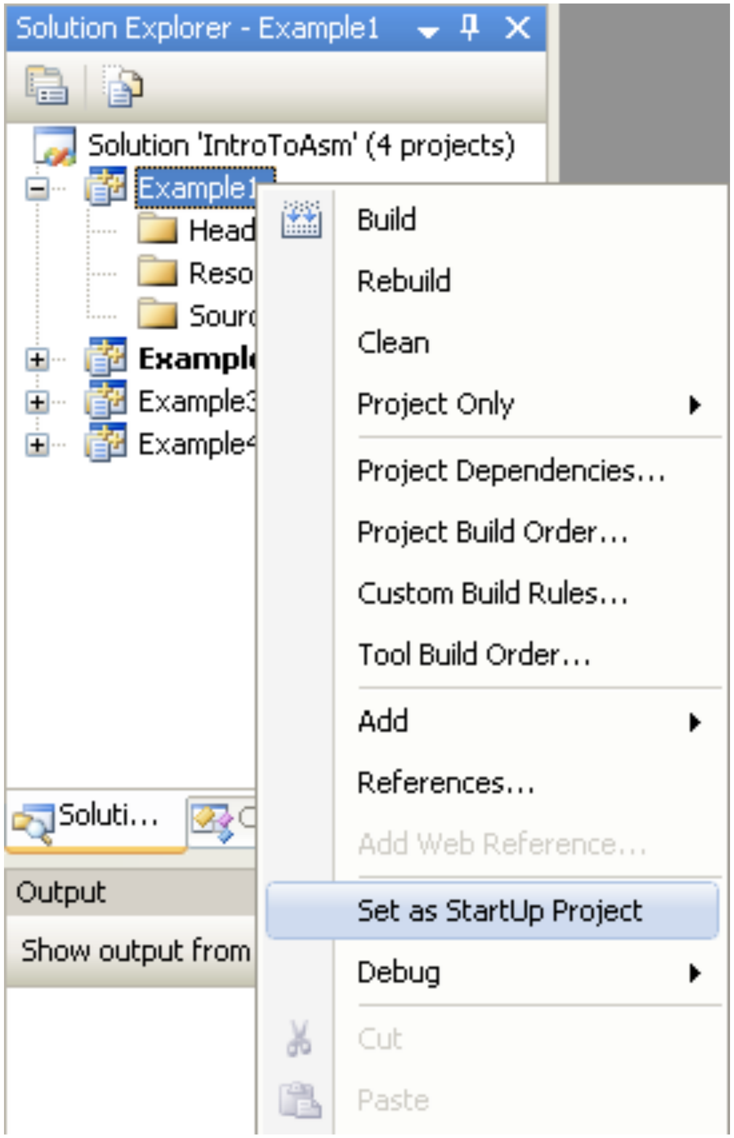
Location: C:\Documents and Settings\Administrator\My Documents\Visual Studio 2008\Projects\IntroTo

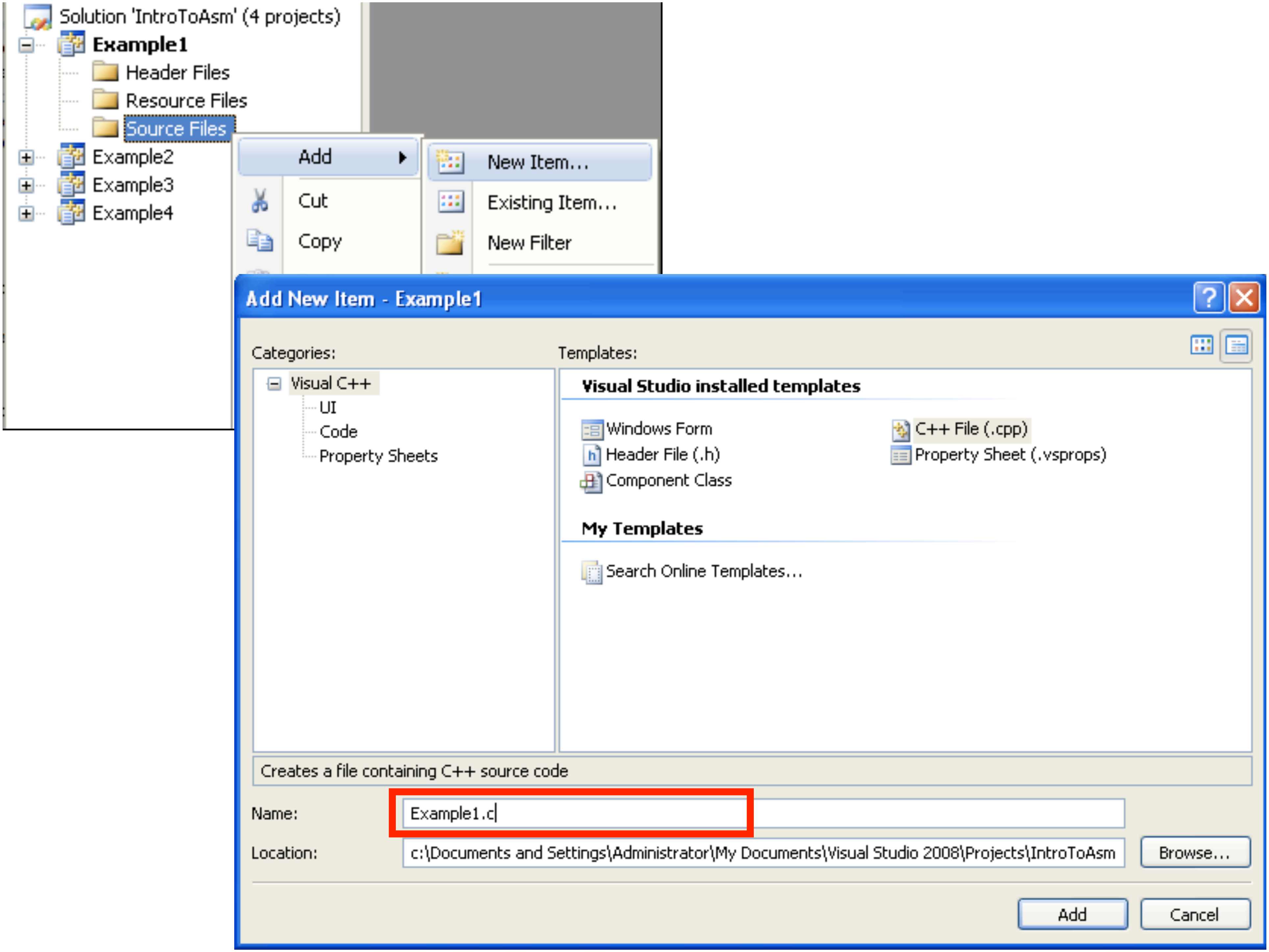
Solution: Add to Solution

☐ Create directory for solution

Solution Name:

Creating a new project - 3



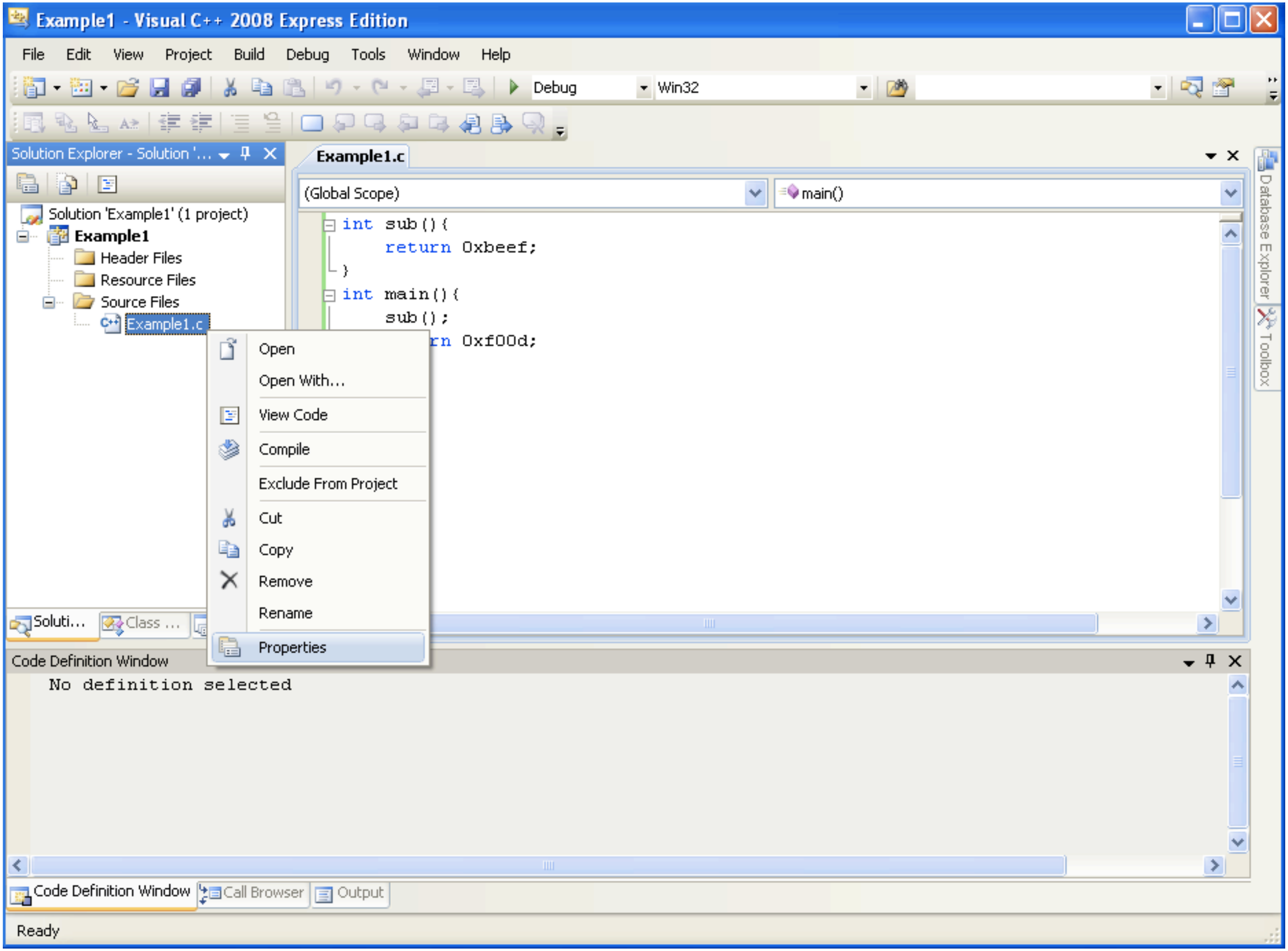


Creates a file containing C++ source code

Name: Example1.c

Location: c:\Documents and Settings\Administrator\My Documents\Visual Studio 2008\Projects\IntroToAsm

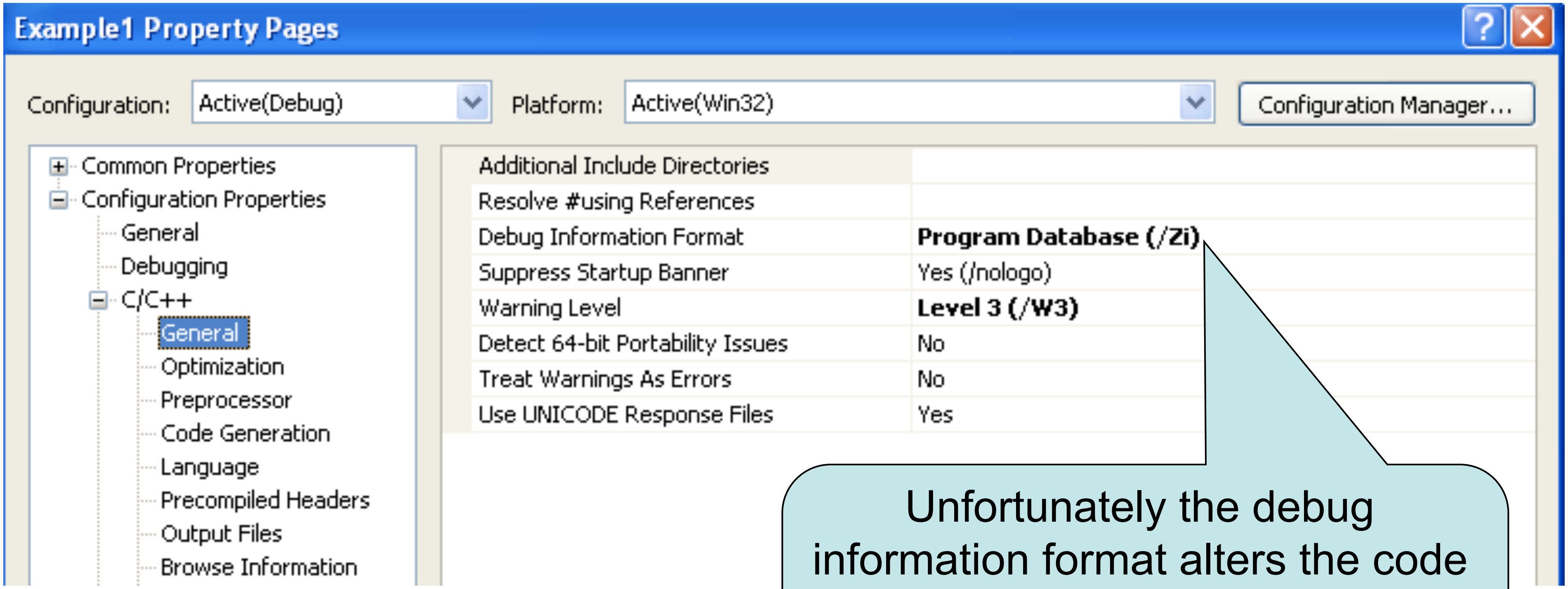
Setting project properties - 1



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

```
No definition selected
```

Setting project properties - 2



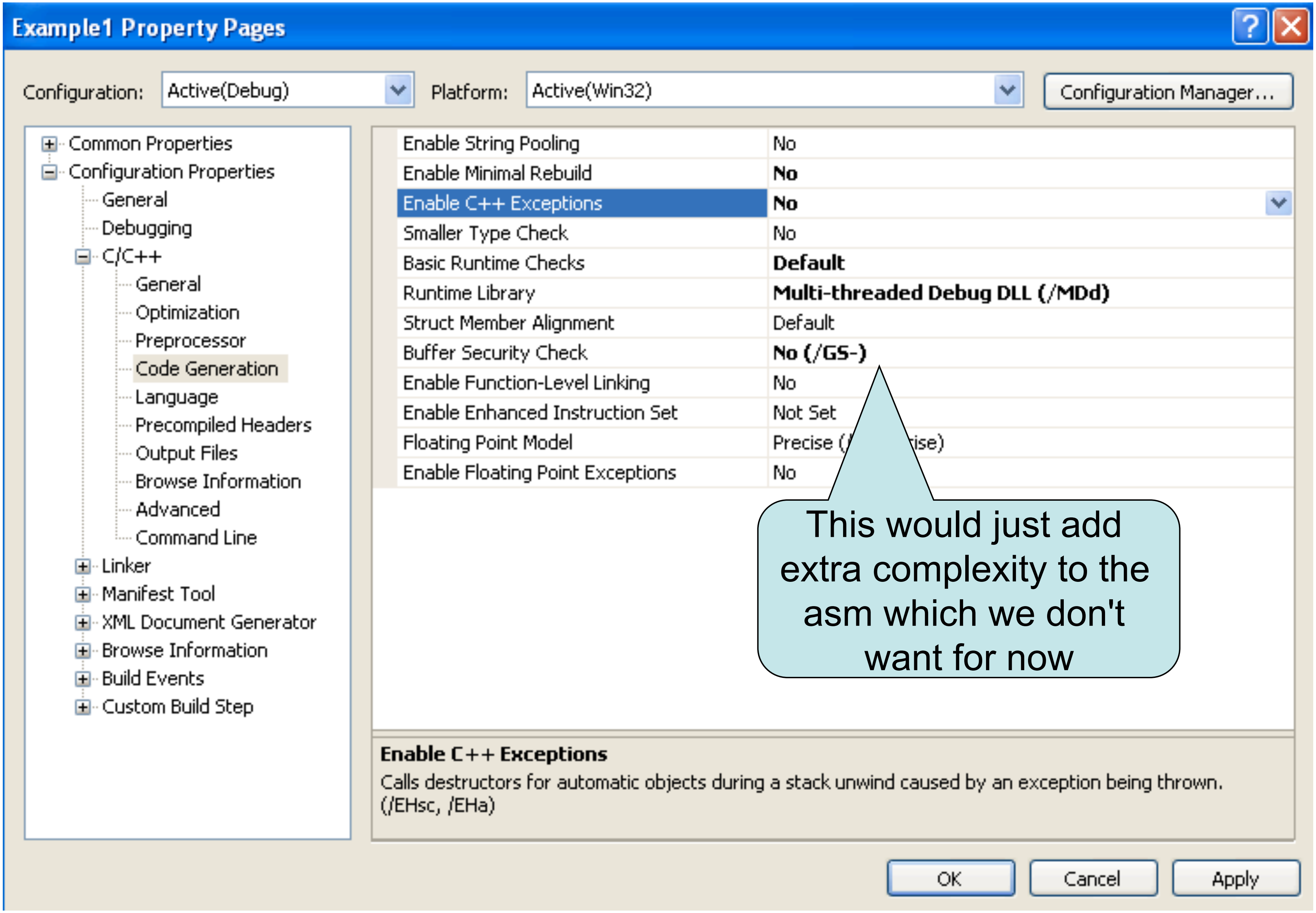
Configuration: Active(Debug)

Platform: Active(Win32)

Additional Include Directories	
Resolve #using References	
Debug Information Format	**Program Database (/Zi)**
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level 3 (/W3)**
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

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)

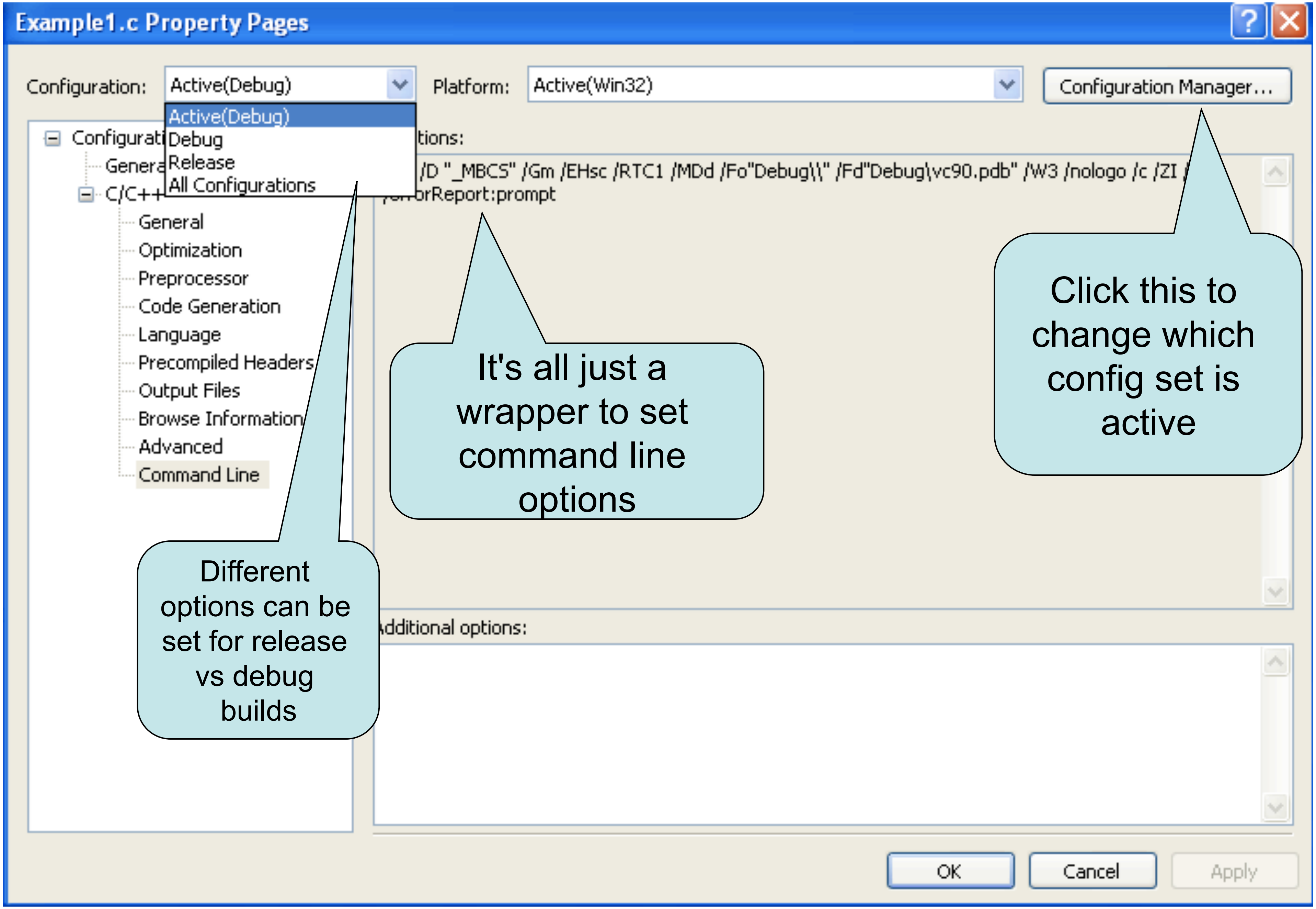
Enable String Pooling	No
Enable Minimal Rebuild	**No**
Enable C++ Exceptions	**No**
Smaller Type Check	No
Basic Runtime Checks	**Default**
Runtime Library	**Multi-threaded Debug DLL (/MDd)**
Struct Member Alignment	Default
Buffer Security Check	**No (/GS-)**
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/ ise)
Enable Floating Point Exceptions	No

Enable C++ Exceptions

Calls destructors for automatic objects during a stack unwind caused by an exception being thrown. (/EHsc, /EHa)

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

Setting project properties - 4



Configuration: Active(Debug)

Platform: Active(Win32)

tions:

```
/D "_MBCS" /Gm /EHsc /RTC1 /MDd /Fo"Debug\\" /Fd"Debug\vc90.pdb" /W3 /nologo /c /ZI /
/errorReport:prompt
```

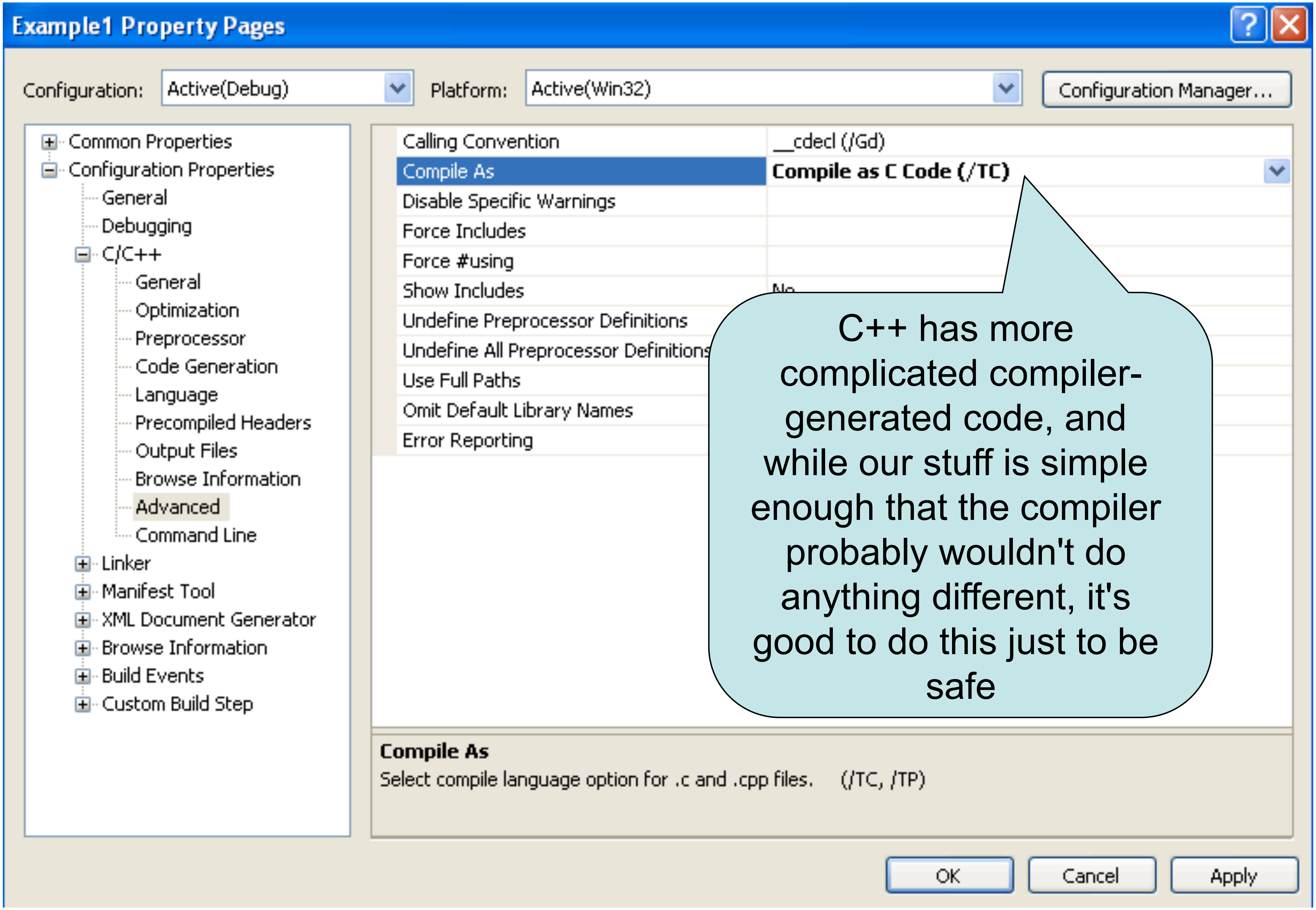
Additional options:

Different options can be set for release vs debug builds

It's all just a wrapper to set command line options

Click this to change which config set is active

Setting project properties - 5



Configuration: Active(Debug)

Platform: Active(Win32)

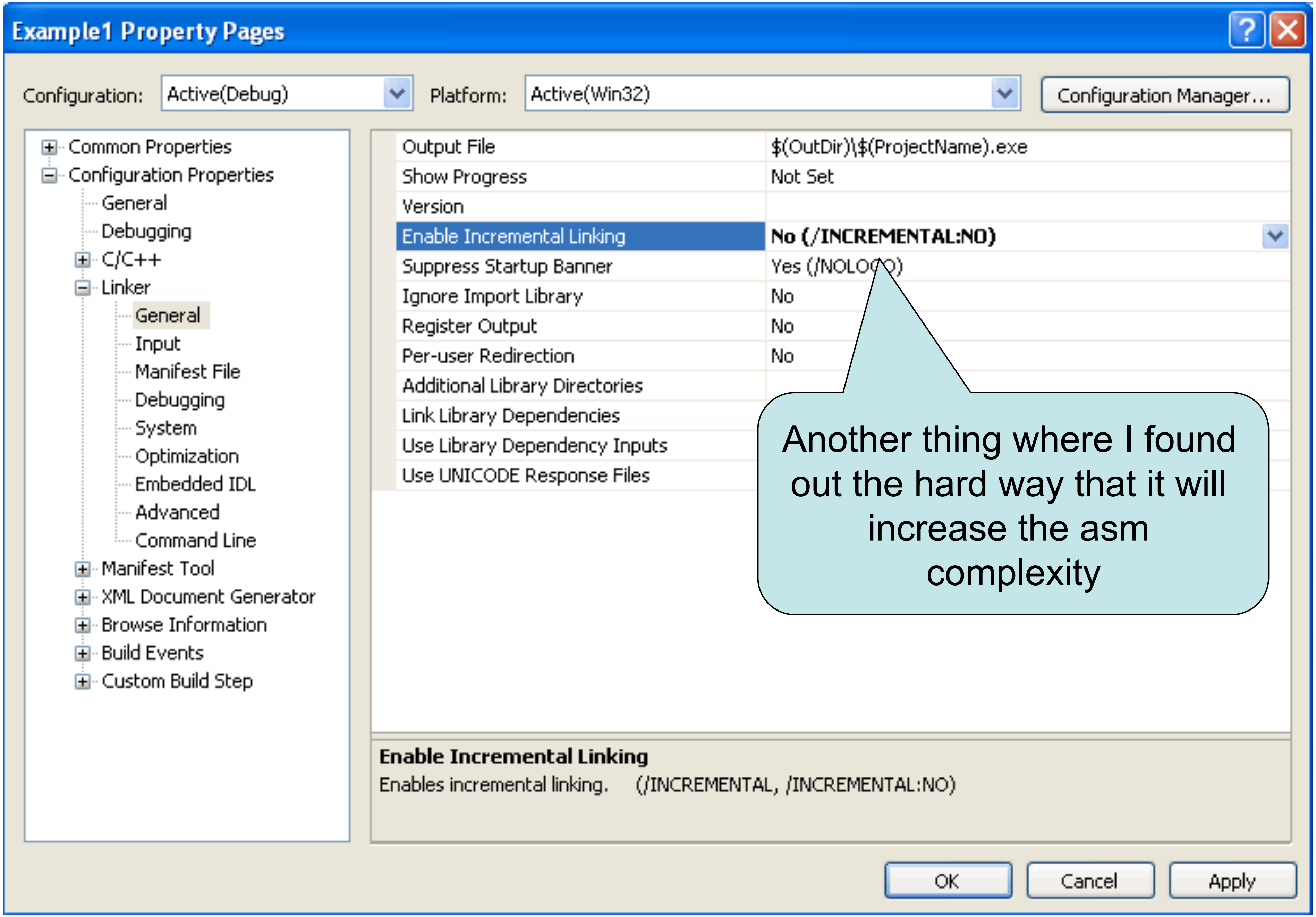
Calling Convention	__cdecl (/Gd)
Compile As	**Compile as C Code (/TC)**
Disable Specific Warnings	
Force Includes	
Force #using	
Show Includes	No
Undefine Preprocessor Definitions	
Undefine All Preprocessor Definitions	
Use Full Paths	
Omit Default Library Names	
Error Reporting	

Compile As

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

C++ has more complicated compiler-generated code, and while our stuff is simple enough that the compiler probably wouldn't do anything different, it's good to do this just to be safe

Setting project properties - 6



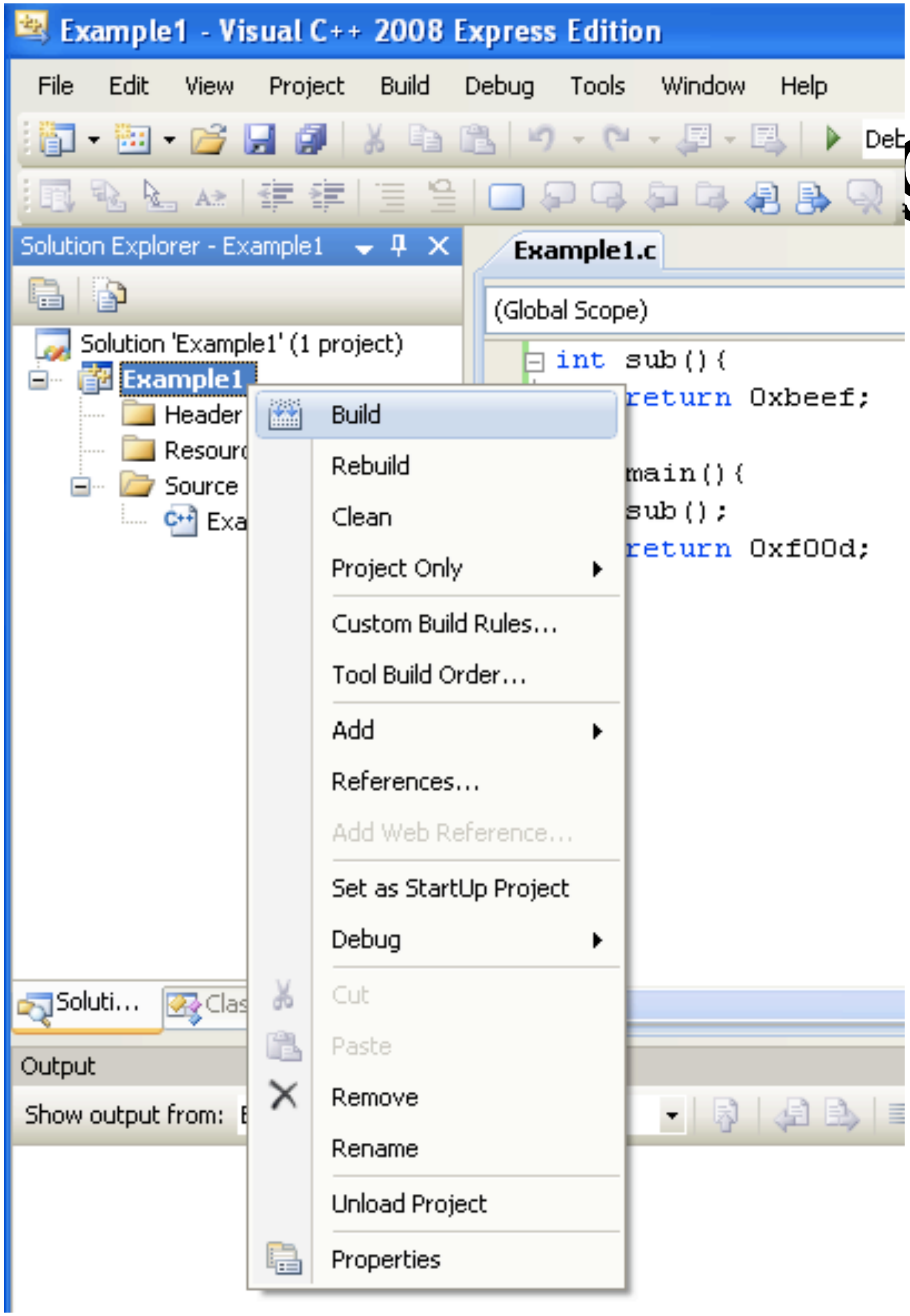
Configuration: Active(Debug)
Platform: Active(Win32)

Output File	$(OutDir)\$(ProjectName).exe
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	No
Additional Library Directories	
Link Library Dependencies	
Use Library Dependency Inputs	
Use UNICODE Response Files	

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

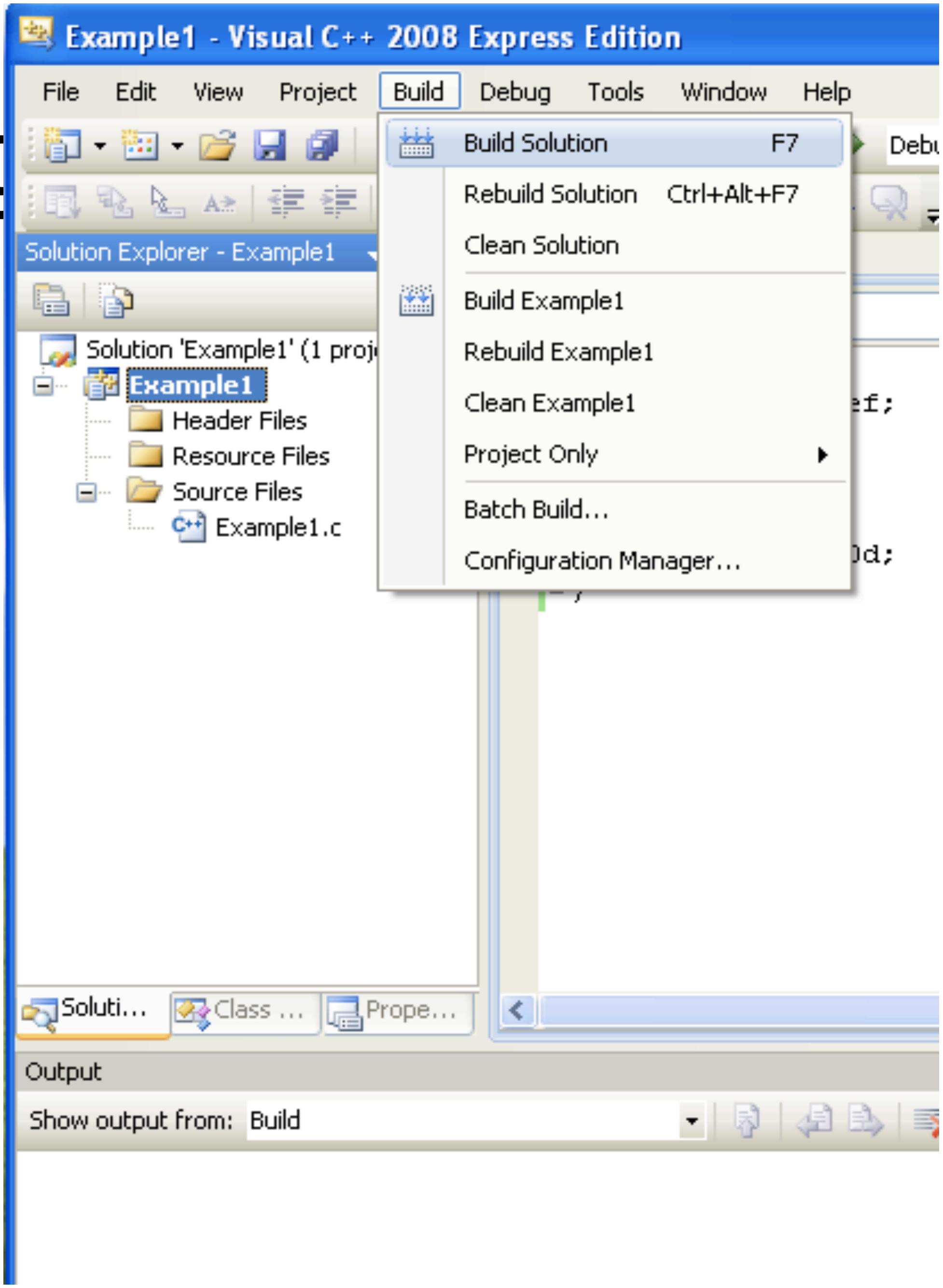
Another thing where I found out the hard way that it will increase the asm complexity

Building the project - 1

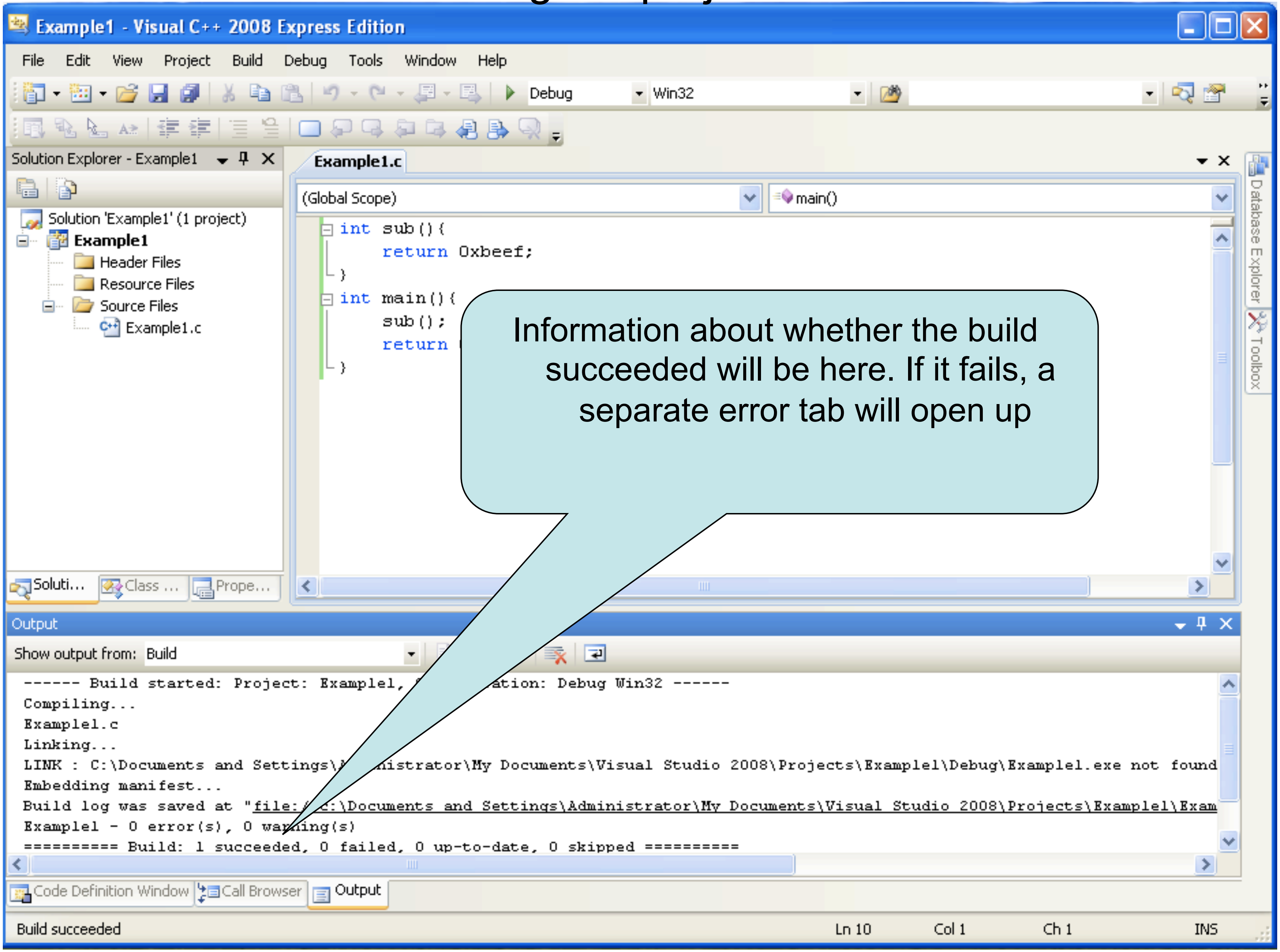


```
int sub(){
   return 0xbeef;

   main(){
   sub();
   return 0xf00d;
```



Building the project - 2



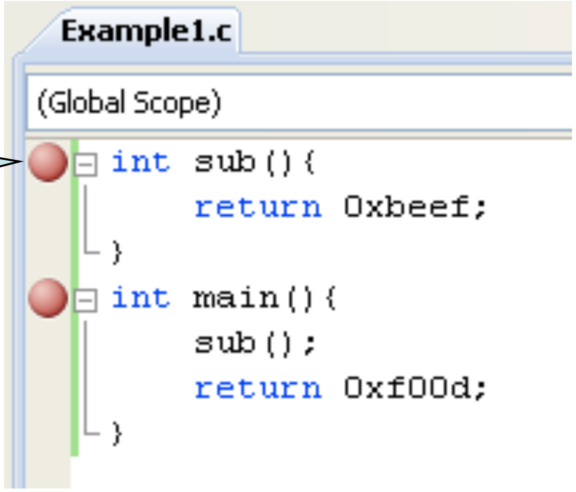
```
int sub(){
    return 0xbeef;
}
int main(){
    sub();
    return
}
```

```
------ Build started: Project: Example1, Configuration: Debug Win32 ------
Compiling...
Example1.c
Linking...
LINK : C:\Documents and Settings\Administrator\My Documents\Visual Studio 2008\Projects\Example1\Debug\Example1.exe not found
Embedding manifest...
Build log was saved at "file://c:\Documents and Settings\Administrator\My Documents\Visual Studio 2008\Projects\Example1\Exam
Example1 - 0 error(s), 0 warning(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

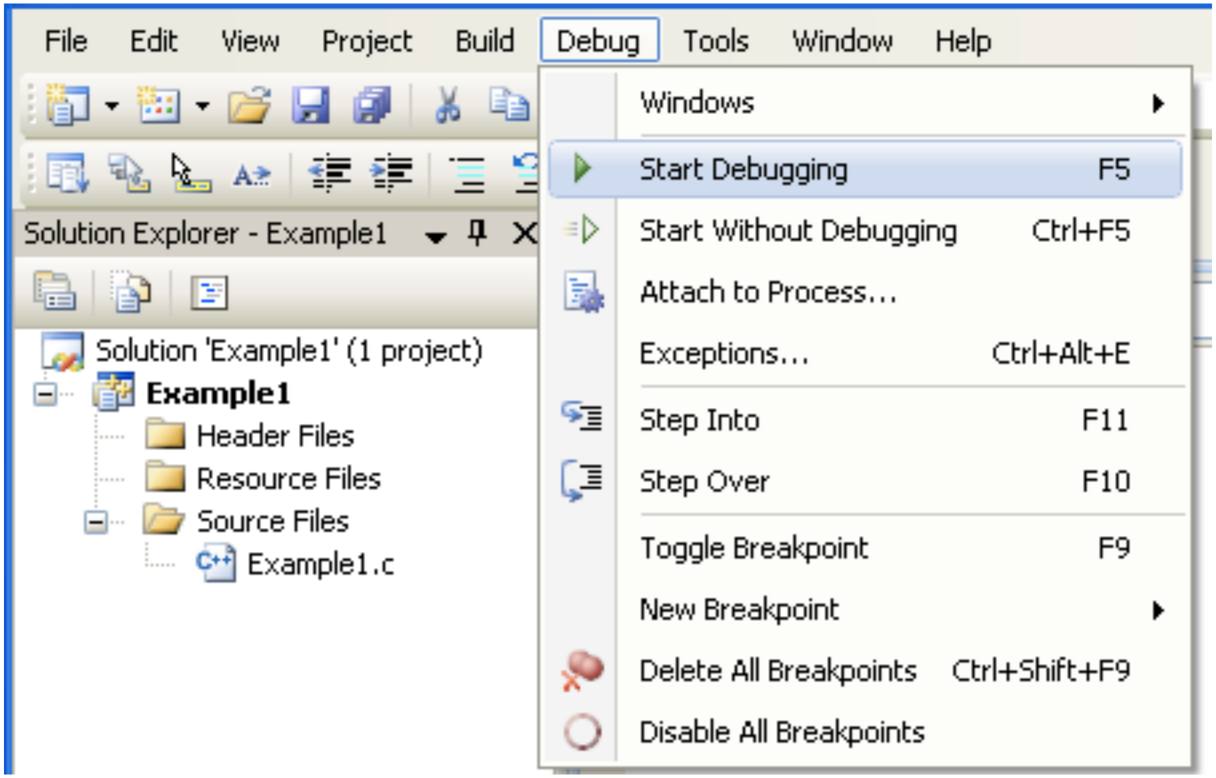
Information about whether the build succeeded will be here. If it fails, a separate error tab will open up

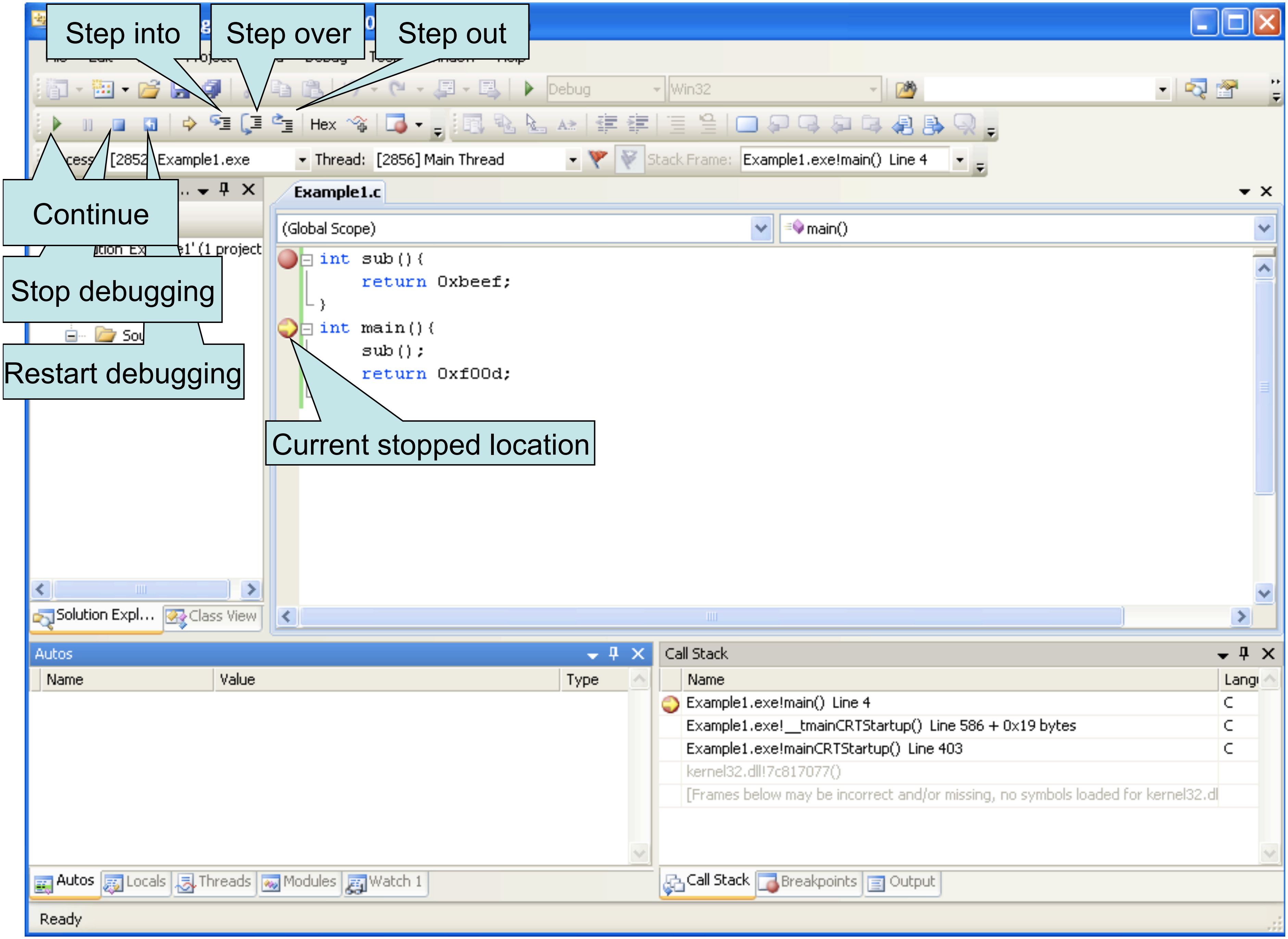
Setting breakpoints & start debugger

Click to the left of the line to break at.



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





Step into

Step over

Step out

Continue

Stop debugging

Restart debugging

Current stopped location

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

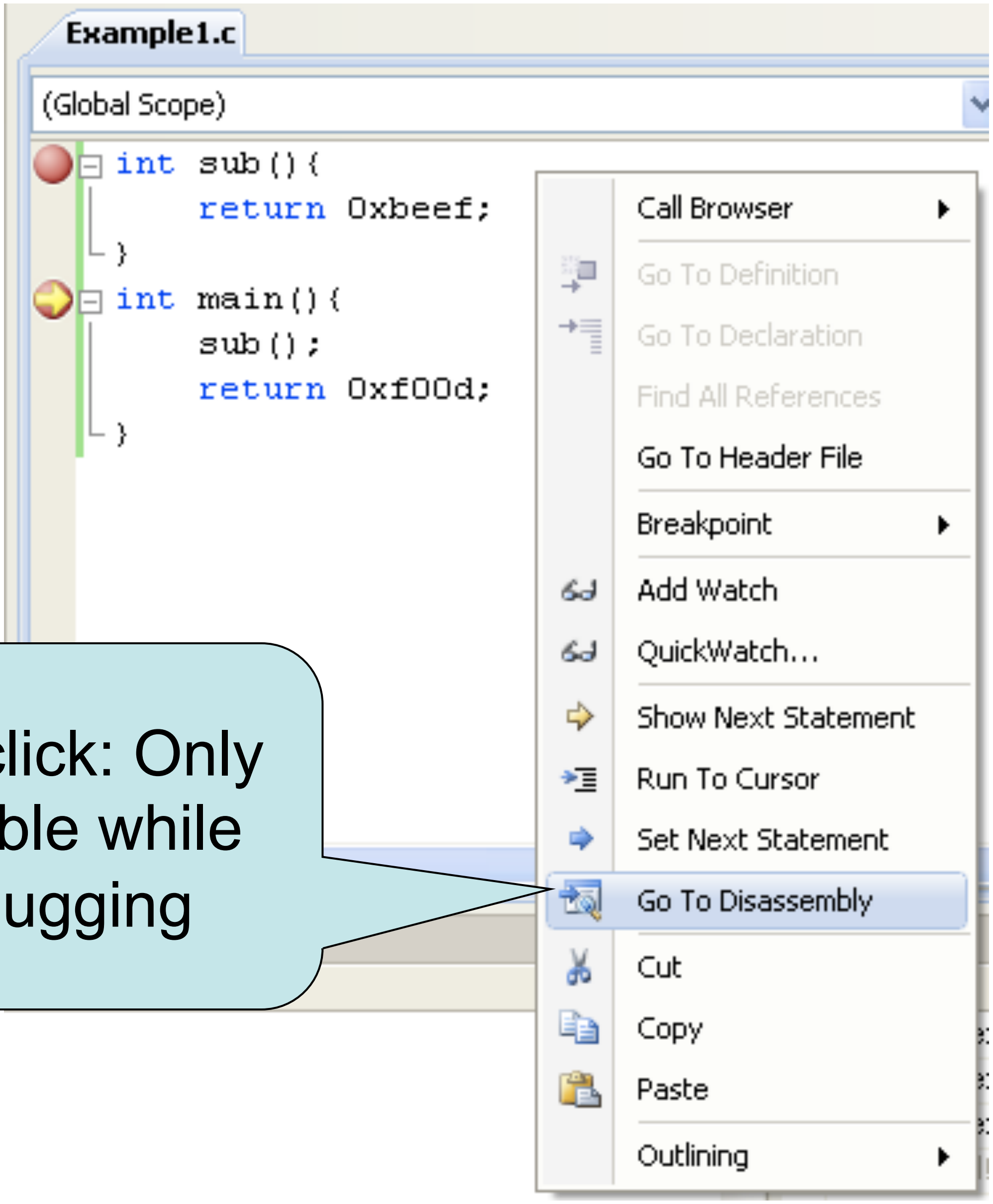
Autos

Name	Value	Type

Call Stack

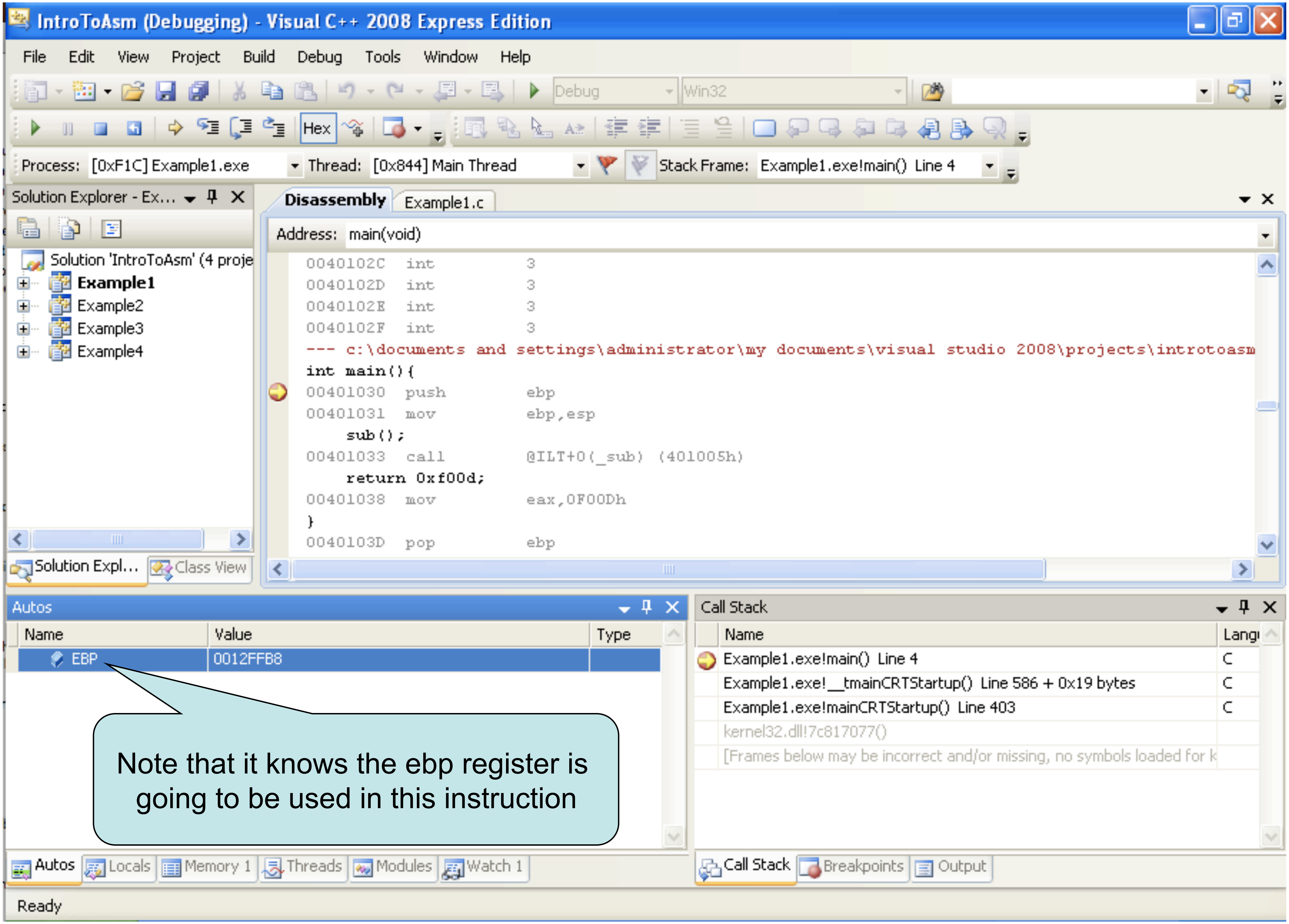
Name	Lang
Example1.exe!main() Line 4	C
Example1.exe!__tmainCRTStartup() Line 586 + 0x19 bytes	C
Example1.exe!mainCRTStartup() Line 403	C
kernel32.dll!7c817077()	
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dl	

Showing assembly



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

Right click: Only available while debugging



Process: [0xF1C] Example1.exe Thread: [0x844] Main Thread Stack Frame: Example1.exe!main() Line 4

Address: main(void)

```
0040102C  int         3
0040102D  int         3
0040102E  int         3
0040102F  int         3
--- c:\documents and settings\administrator\my documents\visual studio 2008\projects\introtoasm
int main(){
00401030  push        ebp
00401031  mov         ebp,esp
    sub();
00401033  call        @ILT+0(_sub) (401005h)
    return 0xf00d;
00401038  mov         eax,0F00Dh
}
0040103D  pop         ebp
```

Autos

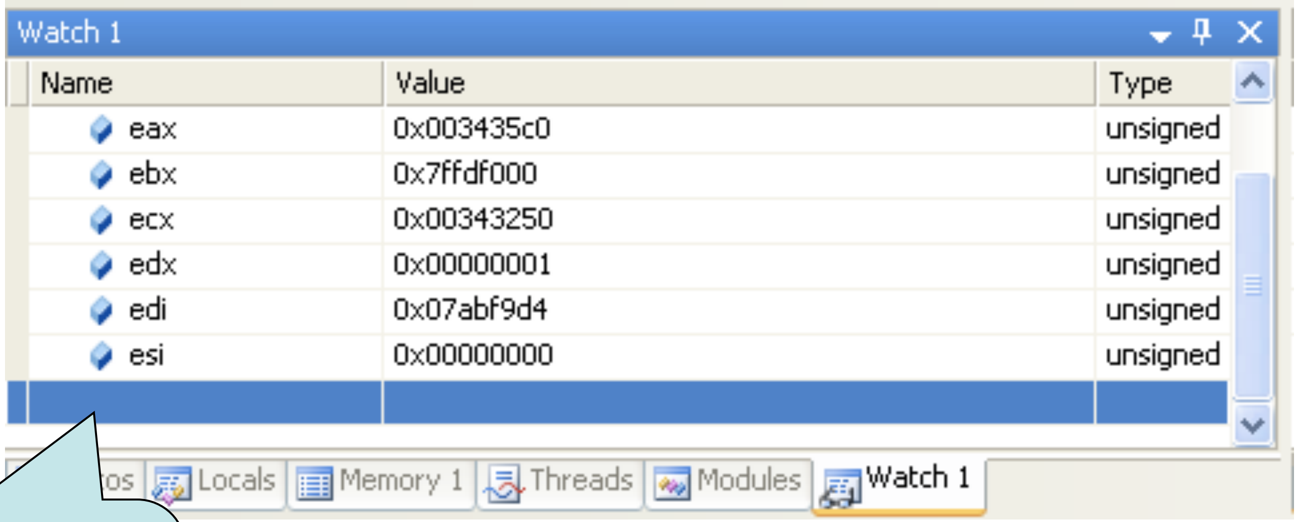
Name	Value	Type
EBP	0012FFB8	

Call Stack

Name	Lang
Example1.exe!main() Line 4	C
Example1.exe!__tmainCRTStartup() Line 586 + 0x19 bytes	C
Example1.exe!mainCRTStartup() Line 403	C
kernel32.dll!7c817077()	
[Frames below may be incorrect and/or missing, no symbols loaded for k	

Note that it knows the ebp register is going to be used in this instruction

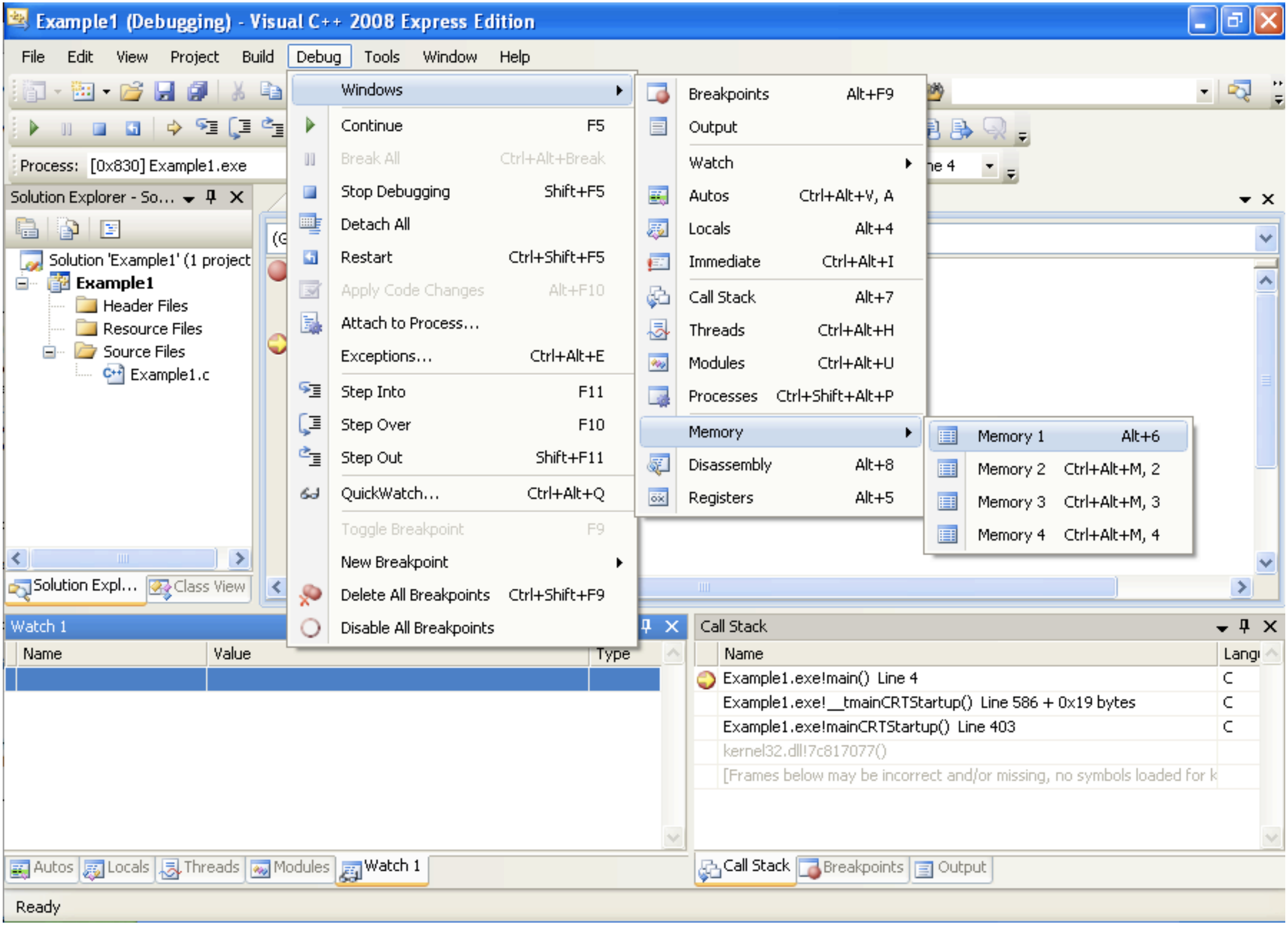
Showing registers



Name	Value	Type
eax	0x003435c0	unsigned
ebx	0x7ffdf000	unsigned
ecx	0x00343250	unsigned
edx	0x00000001	unsigned
edi	0x07abf9d4	unsigned
esi	0x00000000	unsigned

Here you can enter register names or variable names

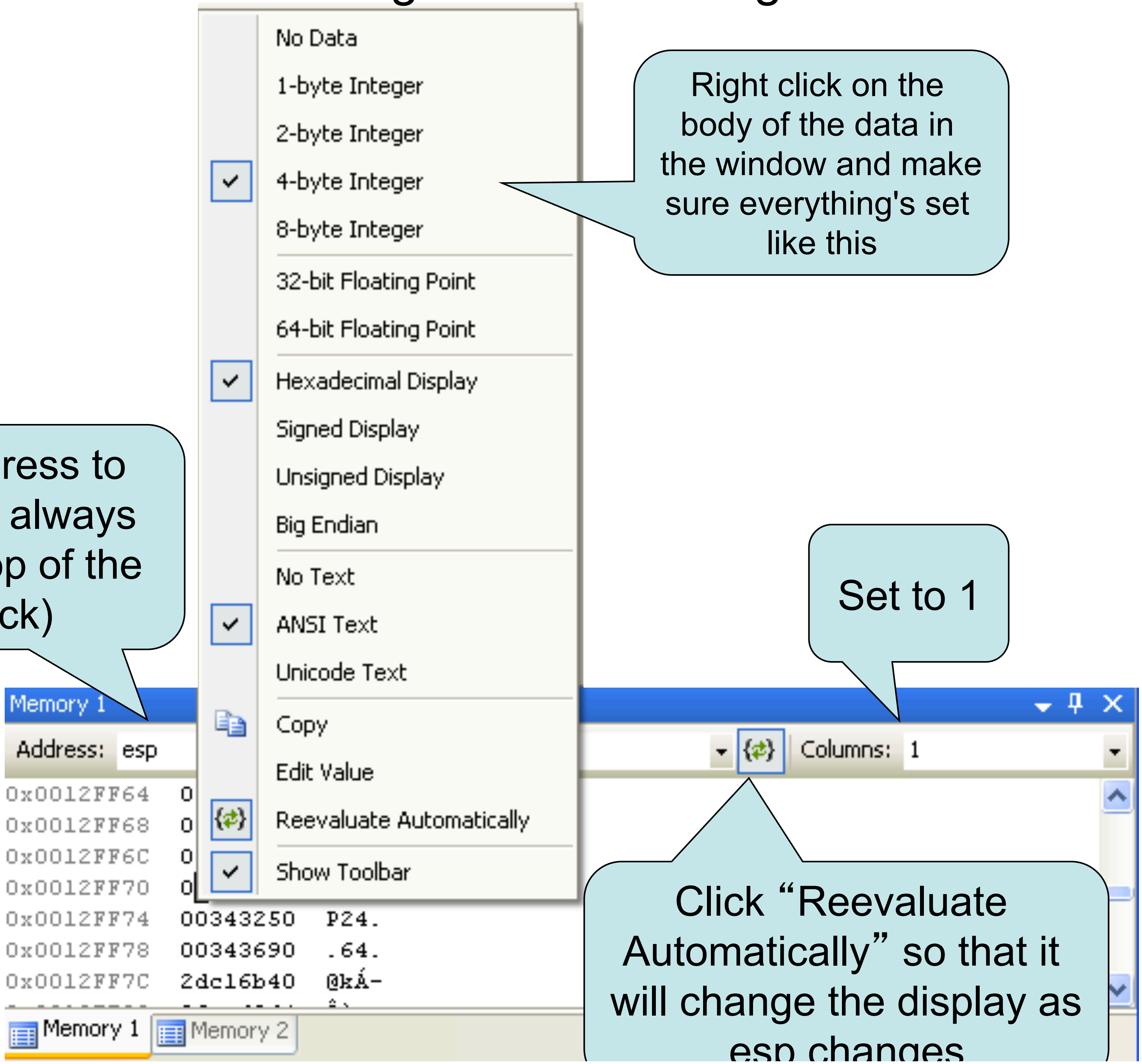
Watching the stack change - 1



Call Stack

Name	Lang
Example1.exe!main() Line 4	C
Example1.exe!__tmainCRTStartup() Line 586 + 0x19 bytes	C
Example1.exe!mainCRTStartup() Line 403	C
kernel32.dll!7c817077()	
[Frames below may be incorrect and/or missing, no symbols loaded for k	

Watching the stack change - 2



Address: esp

Columns: 1

```
0x0012FF64  0
0x0012FF68  0
0x0012FF6C  0
0x0012FF70  0
0x0012FF74  00343250  P24.
0x0012FF78  00343690  .64.
0x0012FF7C  2dc16b40  @kÁ-
```

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

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

Set to 1

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

Going through Example1.c in Visual Studio

```
sub:
  push      ebp
  mov       ebp,esp
  mov       eax,0BEEFh
  pop      ebp
  ret
main:
  push      ebp
  mov       ebp,esp
  call    sub
  mov       eax,0F00Dh
  pop      ebp
  ret
```

Example2.c with Input parameters and Local Variables

```
#include <stdlib.h>
int sub(int x, int y){
        return 2*x+y;
}

int main(int argc, char ** argv){
        int a;
        a = atoi(argv[1]);
        return sub(argc,a);
}
```

```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp
.text:0000000D           retn
.text:00000010 _main:    push   ebp
.text:00000011           mov    ebp, esp
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4]
.text:0000001A           push   ecx
.text:0000001B           call   dword ptr ds:__imp__atoi
.text:00000021           add    esp, 4
.text:00000024           mov    [ebp-4], eax
.text:00000027           mov    edx, [ebp-4]
.text:0000002A           push   edx
.text:0000002B           mov    eax, [ebp+8]
.text:0000002E           push   eax
.text:0000002F           call   _sub
.text:00000034           add    esp, 8
.text:00000037           mov    esp, ebp
.text:00000039           pop    ebp
.text:0000003A           retn
```

"r/m32" Addressing Forms

- Anywhere you see an r/m32 it means it could be taking a value either from a register, or a memory address.
- I'm just calling these “r/m32 forms” because anywhere you see “r/m32” 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 eax, ebx
  - mov eax, [ebx]
  - mov eax, [ebx+ecx*X] (X=1, 2, 4, 8)
  - mov eax, [ebx+ecx*X+Y] (Y= one byte, 0-255 or 4 bytes, 0-2^32-1)
- Most complicated form is: `[base + index*scale + disp]`

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

7

LEA - Load Effective Address

- Frequently used with pointer arithmetic, sometimes for just arithmetic in general
- Uses the r/m32 form but **is the exception to the rule** that the square brackets [ ] syntax means dereference (“value at”)
- Example: ebx = 0x2, edx = 0x1000
  - lea eax, [edx+ebx*2]
  - eax = 0x1004, not the value at 0x1004

Not covered in book

8 9

ADD and SUB

- Adds or Subtracts, just as expected
- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate
- No source ***and*** destination as r/m32s, 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
- add esp, 8
- sub eax, [ebx*2]

Add p. 202, Sub p. 210

Example2.c - 1

```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp
.text:0000000D           retn
.text:00000010 _main:    push   ebp ☒
.text:00000011           mov    ebp, esp
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4]
.text:0000001A           push   ecx
.text:0000001B           call   dword ptr ds:__imp__atoi
.text:00000021           add    esp, 4
.text:00000024           mov    [ebp-4], eax
.text:00000027           mov    edx, [ebp-4]
.text:0000002A           push   edx
.text:0000002B           mov    eax, [ebp+8]
.text:0000002E           push   eax
.text:0000002F           call   _sub
.text:00000034           add    esp, 8
.text:00000037           mov    esp, ebp
.text:00000039           pop    ebp
.text:0000003A           retn
```

eax	**0xcafe ⌘**
ecx	**0xbabe ⌘**
edx	**0xfeed ⌘**
ebp	**0x0012FF50 ⌘**
esp	**0x0012FF24 ♍**

0x0012FF30	0x12FFB0 (char ** argv)⌘
0x0012FF2C	0x2 (int argc) ⌘
0x0012FF28	Addr after "call _main" ⌘
0x0012FF24	0x0012FF50(saved ebp)♍
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Key: executed instruction ☒, modified value ♍, arbitrary example start value ⌘

Example2.c - 2

```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp
.text:0000000D           retn
.text:00000010 _main:    push   ebp
.text:00000011           mov    ebp, esp ⊠
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4]
.text:0000001A           push   ecx
.text:0000001B           call   dword ptr ds:__imp__atoi
.text:00000021           add    esp, 4
.text:00000024           mov    [ebp-4], eax
.text:00000027           mov    edx, [ebp-4]
.text:0000002A           push   edx
.text:0000002B           mov    eax, [ebp+8]
.text:0000002E           push   eax
.text:0000002F           call   _sub
.text:00000034           add    esp, 8
.text:00000037           mov    esp, ebp
.text:00000039           pop    ebp
.text:0000003A           retn
```

eax	0xcafe
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24 ♍
esp	0x0012FF24

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2.c - 3

```
.text:00000000 _sub:     push    ebp
.text:00000001           mov     ebp, esp
.text:00000003           mov     eax, [ebp+8]
.text:00000006           mov     ecx, [ebp+0Ch]
.text:00000009           lea     eax, [ecx+eax*2]
.text:0000000C           pop     ebp
.text:0000000D           retn
.text:00000010 _main:    push    ebp
.text:00000011           mov     ebp, esp
.text:00000013           push    ecx ☒
                         mov     eax, [ebp+0Ch]
                         mov     ecx, [eax+4]
                         push    ecx
                         call    dword ptr ds:__imp__atoi
                         add     esp, 4
                         mov     [ebp-4], eax
                         mov     edx, [ebp-4]
                         push    edx
                         mov     eax, [ebp+8]
                         push    eax
                         call    _sub
                         add     esp, 8
                         mov     esp, ebp
                         pop     ebp
                         retn
```

Caller-save, or space for local var? This time it turns out to be space for local var since there is no corresponding pop, and the address is used later to refer to the value we know is stored in a.

eax	0xcafe
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a) ♍
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2.c - 4

```
.text:00000000 _sub:    push   ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp
.text:0000000D          retn
.text:00000010 _main:   push   ebp
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
.text:00000014          mov    eax, [ebp+0Ch] ☒
                        mov    ecx, [eax+4]
                        push   ecx
                        call   dword ptr ds:__imp__atoi
                        add    esp, 4
                        mov    [ebp-4], eax
                        mov    edx, [ebp-4]
                        push   edx
                        mov    eax, [ebp+8]
                        push   eax
.text:0000002F          call   _sub
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp
.text:0000003A          retn
```

Getting the base of the argv char * array (aka argv[0])

eax	0x12FFB0 ♍
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 5

```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp
.text:0000000D           retn
.text:00000010 _main:    push   ebp
.text:00000011           mov    ebp, esp
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4] ☒
                         push   ecx
                         call   dword ptr ds:__imp__atoi
                         add    esp, 4
                         mov    [ebp-4], eax
                         mov    edx, [ebp-4]
                         push   edx
                         mov    eax, [ebp+8]
                         push   eax
                         call   _sub
                         add    esp, 8
                         mov    esp, ebp
                         pop    ebp
                         retn
```

Getting the char * at argv[1]
(I chose 0x12FFD4 arbitrarily since it's out of the stack scope we're currently looking at)

eax	0x12FFB0
ecx	0x12FFD4♍(arbitrary⌘)
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 6

```
.text:00000000 _sub:    push   ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
                        lea    eax, [ecx+eax*2]
                        pop    ebp
                        retn
                        push   ebp
                        mov    ebp, esp
                        push   ecx
                        mov    eax, [ebp+0Ch]
                        mov    ecx, [eax+4]
                        push   ecx ⊠
                        call   dword ptr ds:__imp__atoi ⊠
                        add    esp, 4 ⊠
                        mov    [ebp-4], eax
                        mov    edx, [ebp-4]
                        push   edx
                        mov    eax, [ebp+8]
                        push   eax
                        call   _sub
                        add    esp, 8
                        mov    esp, ebp
                        pop    ebp
                        retn
```

Saving some slides…
This will push the address of the string at argv[1] (0x12FFD4). atoi() will read the string and turn in into an int, put that int in eax, and return. Then the adding 4 to esp will negate the having pushed the input parameter and make 0x12FF1C undefined again (this is indicative of cdecl)

eax	0x100♍ (arbitrary⌘)
ecx	0x12FFD4
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef ♍
0x0012FF18	undef ♍
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 7

```
.text:00000000 _sub:    push   ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp
.text:0000000D          retn
.text:00000010 _main:   push   ebp
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
                        mov    eax, [ebp+0Ch]
                        mov    ecx, [eax+4]
                        push   ecx
                        call   dword ptr ds:__imp__atoi
                        add    esp, 4
                        mov    [ebp-4], eax ☒
                        mov    edx, [ebp-4] ☒
                        push   edx ☒
                        mov    eax, [ebp+8]
                        push   eax
                        call   _sub
                        add    esp, 8
                        mov    esp, ebp
.                       pop    ebp
.text:0000003A          retn
```

First setting “a” equal to the return value. Then pushing “a” as the second parameter in sub(). We can see an obvious optimization would have been to replace the last two instructions with “push eax”.

eax	0x100
ecx	0x12FFD4
edx	0x100 ♍
ebp	0x0012FF24
esp	0x0012FF1C ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a) ♍
0x0012FF1C	0x100 (int y) ♍
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 8

```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp
.text:0000000D           retn
.text:00000010 _main:    push   ebp
.text:00000011           mov    ebp, esp
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4]
.text:0000001A           push   ecx
.text:0000001B           call   dword ptr ds:__imp__atoi
.text:00000021           add    esp, 4
.text:00000024           mov    [ebp-4], eax
.text:00000027           mov    edx, [ebp-4]
.text:0000002A           push   edx
                         mov    eax, [ebp+8] ☒
                         push   eax ☒
                         call   _sub
                         add    esp, 8
                         mov    esp, ebp
                         pop    ebp
.text:0000003A           retn
```

Pushing argc as the first parameter (int x) to sub()

eax	0x2 ♍
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF24
esp	0x0012FF18 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x) ♍
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 9

```
.text:00000000 _sub:    push   ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp
.text:0000000D          retn
.text:00000010 _main:   push   ebp
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push   ecx
.text:0000001B          call   dword ptr ds:__imp__atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push   edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push   eax
.text:0000002F          call   _sub ☒
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp
.text:0000003A          retn
```

eax	0x2
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF24
esp	0x0012FF14 ♍

↑

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034 ♍
0x0012FF10	undef
0x0012FF0C	undef

↓

Example2 - 10

```
.text:00000000 _sub:      push   ebp ☒
.text:00000001            mov    ebp, esp ☒
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

eax	0x2
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF10 ♍
esp	0x0012FF10 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24(saved ebp)♍
0x0012FF0C	undef

Example2 - 11

```
.text:00000000 _sub:       push    ebp
.text:00000001             mov     ebp, esp
                           mov     eax, [ebp+8] ☒
                           mov     ecx, [ebp+0Ch] ☒
                           lea     eax, [ecx+eax*2]
                           pop     ebp
                           retn
.text:00000010 _main:      push    ebp
.text:00000011             mov     ebp, esp
.text:00000013             push    ecx
.text:00000014             mov     eax, [ebp+0Ch]
.text:00000017             mov     ecx, [eax+4]
.text:0000001A             push    ecx
.text:0000001B             call    dword ptr ds:__imp__atoi
.text:00000021             add     esp, 4
.text:00000024             mov     [ebp-4], eax
.text:00000027             mov     edx, [ebp-4]
.text:0000002A             push    edx
.text:0000002B             mov     eax, [ebp+8]
.text:0000002E             push    eax
.text:0000002F             call    _sub
.text:00000034             add     esp, 8
.text:00000037             mov     esp, ebp
.text:00000039             pop     ebp
.text:0000003A             retn
```

Move "x" into eax, and "y" into ecx.

eax	0x2 ♍ (no value change)
ecx	0x100 ♍
edx	0x100
ebp	0x0012FF10
esp	0x0012FF10

↑

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24 (saved ebp)
0x0012FF0C	undef

↓

Example2 - 12

```
.text:00000000 _sub:        push    ebp
.text:00000001              mov     ebp, esp
.text:00000003              mov     eax, [ebp+8]
                            mov     ecx, [ebp+0Ch]
                            lea     eax, [ecx+eax*2] ☒
                            pop     ebp
                            retn
                            push    ebp
                            mov     ebp, esp
                            push    ecx
                            mov     eax, [ebp+0Ch]
                            mov     ecx, [eax+4]
                            push    ecx
.text:0000001B              call    dword ptr ds:__imp__atoi
.text:00000021              add     esp, 4
.text:00000024              mov     [ebp-4], eax
.text:00000027              mov     edx, [ebp-4]
.text:0000002A              push    edx
.text:0000002B              mov     eax, [ebp+8]
.text:0000002E              push    eax
.text:0000002F              call    _sub
.text:00000034              add     esp, 8
.text:00000037              mov     esp, ebp
.text:00000039              pop     ebp
.text:0000003A              retn
```

Set the return value (eax) to 2*x + y. Note: neither pointer arith, nor an "address" which was loaded. Just an afficient way to do a calculation.

eax	0x104 ♍
ecx	0x100
edx	0x100
ebp	0x0012FF10
esp	0x0012FF10

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24 (saved ebp)
0x0012FF0C	undef

Example2 - 13

```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp ☒
.text:0000000D           retn
.text:00000010 _main:    push   ebp
.text:00000011           mov    ebp, esp
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4]
.text:0000001A           push   ecx
.text:0000001B           call   dword ptr ds:__imp__atoi
.text:00000021           add    esp, 4
.text:00000024           mov    [ebp-4], eax
.text:00000027           mov    edx, [ebp-4]
.text:0000002A           push   edx
.text:0000002B           mov    eax, [ebp+8]
.text:0000002E           push   eax
.text:0000002F           call   _sub
.text:00000034           add    esp, 8
.text:00000037           mov    esp, ebp
.text:00000039           pop    ebp
.text:0000003A           retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24 ♍
esp	0x0012FF14 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	undef ♍
0x0012FF0C	undef

Example2 - 14

```
.text:00000000 _sub:      push   ebp
.text:00000001            mov    ebp, esp
.text:00000003            mov    eax, [ebp+8]
.text:00000006            mov    ecx, [ebp+0Ch]
.text:00000009            lea    eax, [ecx+eax*2]
.text:0000000C            pop    ebp
.text:0000000D            retn ☒
.text:00000010 _main:     push   ebp
.text:00000011            mov    ebp, esp
.text:00000013            push   ecx
.text:00000014            mov    eax, [ebp+0Ch]
.text:00000017            mov    ecx, [eax+4]
.text:0000001A            push   ecx
.text:0000001B            call   dword ptr ds:__imp__atoi
.text:00000021            add    esp, 4
.text:00000024            mov    [ebp-4], eax
.text:00000027            mov    edx, [ebp-4]
.text:0000002A            push   edx
.text:0000002B            mov    eax, [ebp+8]
.text:0000002E            push   eax
.text:0000002F            call   _sub
.text:00000034            add    esp, 8
.text:00000037            mov    esp, ebp
.text:00000039            pop    ebp
.text:0000003A            retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF18 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	undef ♍
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 15

```
.text:00000000 _sub:      push    ebp
.text:00000001            mov     ebp, esp
.text:00000003            mov     eax, [ebp+8]
.text:00000006            mov     ecx, [ebp+0Ch]
.text:00000009            lea     eax, [ecx+eax*2]
.text:0000000C            pop     ebp
.text:0000000D            retn
.text:00000010 _main:     push    ebp
.text:00000011            mov     ebp, esp
.text:00000013            push    ecx
.text:00000014            mov     eax, [ebp+0Ch]
.text:00000017            mov     ecx, [eax+4]
.text:0000001A            push    ecx
.text:0000001B            call    dword ptr ds:__imp__atoi
.text:00000021            add     esp, 4
.text:00000024            mov     [ebp-4], eax
.text:00000027            mov     edx, [ebp-4]
.text:0000002A            push    edx
.text:0000002B            mov     eax, [ebp+8]
.text:0000002E            push    eax
.text:0000002F            call    _sub
.text:00000034            add     esp, 8 ☒
.text:00000037            mov     esp, ebp
.text:00000039            pop     ebp
.text:0000003A            retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF20 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	undef ♍
0x0012FF18	undef ♍
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 16

```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp
.text:0000000D           retn
.text:00000010 _main:    push   ebp
.text:00000011           mov    ebp, esp
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4]
.text:0000001A           push   ecx
.text:0000001B           call   dword ptr ds:__imp__atoi
.text:00000021           add    esp, 4
.text:00000024           mov    [ebp-4], eax
.text:00000027           mov    edx, [ebp-4]
.text:0000002A           push   edx
.text:0000002B           mov    eax, [ebp+8]
.text:0000002E           push   eax
.text:0000002F           call   _sub
.text:00000034           add    esp, 8
.text:00000037           mov    esp, ebp ☒
.text:00000039           pop    ebp
.text:0000003A           retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF24 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	undef ♍
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 17

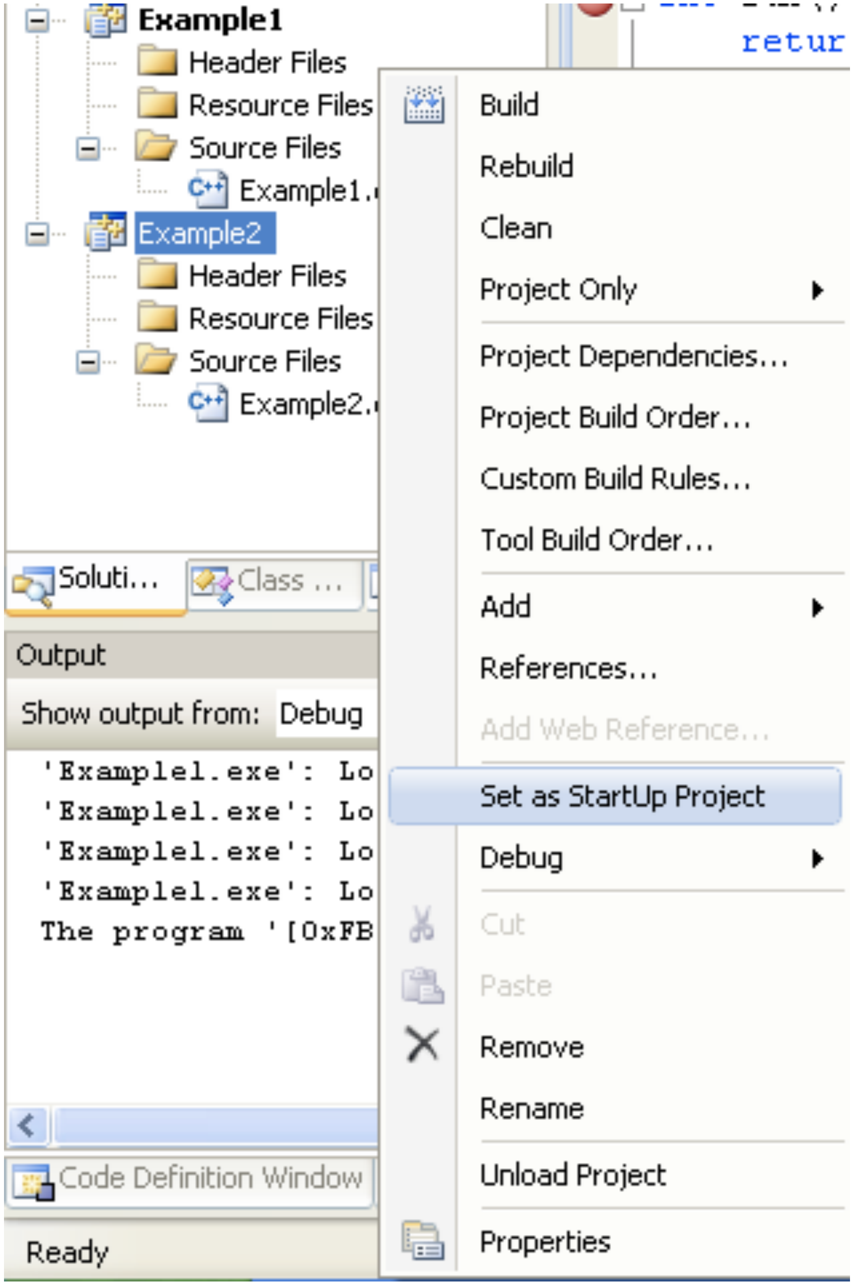
```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp
.text:0000000D           retn
.text:00000010 _main:    push   ebp
.text:00000011           mov    ebp, esp
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4]
.text:0000001A           push   ecx
.text:0000001B           call   dword ptr ds:__imp__atoi
.text:00000021           add    esp, 4
.text:00000024           mov    [ebp-4], eax
.text:00000027           mov    edx, [ebp-4]
.text:0000002A           push   edx
.text:0000002B           mov    eax, [ebp+8]
.text:0000002E           push   eax
.text:0000002F           call   _sub
.text:00000034           add    esp, 8
.text:00000037           mov    esp, ebp
.text:00000039           pop    ebp ☒
.text:0000003A           retn
```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF50 ♍
esp	0x0012FF28 ♍

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	undef ♍
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

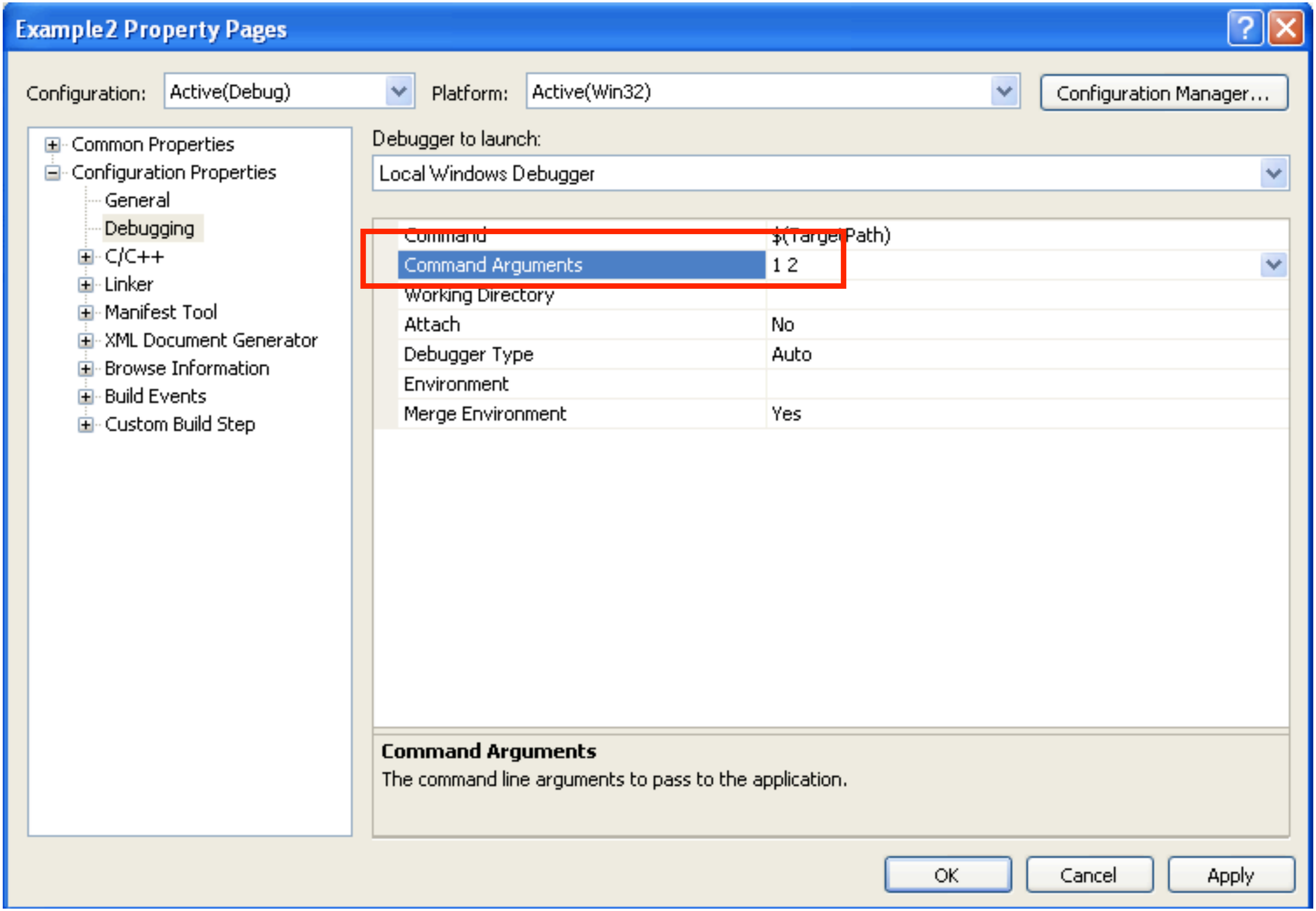
Going through Example2.c in Visual Studio

```
sub:
  push       ebp
  mov        ebp,esp
  mov        eax,0BEEFh
  pop        ebp
  ret
main:
  push       ebp
  mov        ebp,esp
  call       sub
  mov        eax,0F00Dh
  pop        ebp
  ret
```



```
'Example1.exe': Lo
'Example1.exe': Lo
'Example1.exe': Lo
'Example1.exe': Lo
The program '[0xFB
```

Setting command line arguments



Configuration: Active(Debug)

Platform: Active(Win32)

Debugger to launch: Local Windows Debugger

Command	$(TargetPath)
Command Arguments	1 2
Working Directory	
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes

Command Arguments

The command line arguments to pass to the application.

Instructions we now know (9)

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

Back to Hello World

```
.text:00401730 main
.text:00401730          push    ebp
.text:00401731          mov     ebp, esp
.text:00401733          push    offset aHelloWorld ; "Hello world\n“
.text:00401738          call    ds:__imp__printf
.text:0040173E          add     esp, 4
.text:00401741          mov     eax, 1234h
.text:00401746          pop     ebp
.text:00401747          retn
```

Are we all comfortable with this now?

Windows Visual C++ 2005, /GS (buffer overflow protection) option turned off
Disassembled with IDA Pro 4.9 Free Version