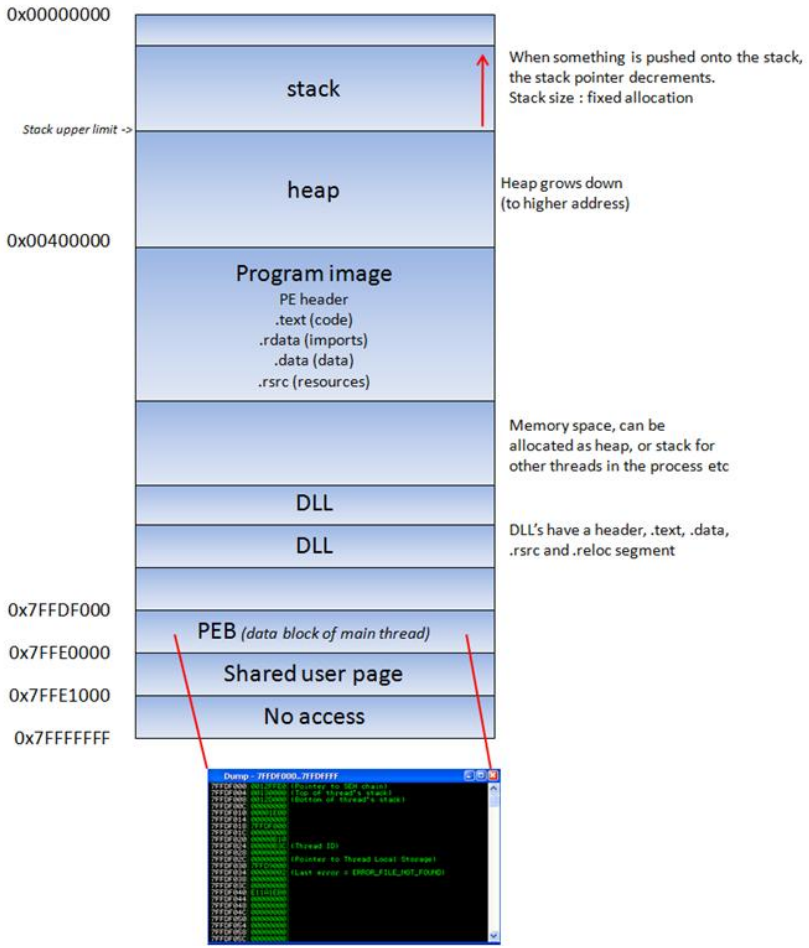
Stack Management & Calling Conventions

Information Security

Mihai-Lica Pura
2020.03.18

Memory segments of a process



Memory segments of a process

- code/text segment
  - instructions that the processor executes
  - read-only
  - fixed size
  - EIP keeps track of the next instruction
- data segment
  - global variables
  - static variables
  - read/write access
  - fixed in size

Memory segments of a process

- heap segment
  - all other variables
  - read/write access
  - can grow larger or smaller
- stack segment
  - used to pass data/arguments to functions
  - space for variables
  - read/write access

The stack

- data structure that works LIFO (Last in first out)
  - PUSH
  - POP
- when a thread is created, a stack gets allocated by the OS
- when the thread ends, the stack is cleared as well
- the size of the stack is fixed
  - defined when it gets created and doesn't change

The stack

Stack	Dynamic/Heap memory
Fast and automatic allocation	Manual allocation
Fast and automatic reclaiming	Manual reclaiming
Suitable for temporary data/local data	Suitable for data required outside of function
Required data must be saved before function/subroutine exists	No saving needed for required data
Limited in size	The only limit is the size of the memory

The stack

- the bottom of the stack starts from the very end of the virtual memory of a page and grows down (to a lower address)
- ESP (Stack Pointer)
  - access the stack memory directly
  - points at the top (so the lowest memory address) of the stack

The stack

- PUSH
  - adds something to the top of the stack
  - ESP points to a lower memory address
  - address is decremented with the size of the pushed data
  - 4 bytes in case of addresses/pointers
  - decrements usually happen before the item is placed on the stack
  - if ESP already points at the next free location in the stack, the decrement happens after placing data on the stack

The stack

- POP
  - removes one item (4 bytes) from the stack and puts it in a register
  - ESP points to a higher address
  - address is incremented
  - 4 bytes in case of addresses/pointers
  - increments happen after an item is removed from the stack

The stack

- When a function/subroutine is entered
  - a stack frame is created
    - it keeps the parameters of the parent procedure together
    - the saved values of registers (EBP, EIP)
    - is used to pass arguments to the function/subroutine
  - the current location of the stack – the stack pointer (ESP)
- When a function/subroutine returns
  - the saved values of EIP and EBP are retrieved from the stack and placed back in EIP and EBP
  - the normal application flow can be resumed

Example

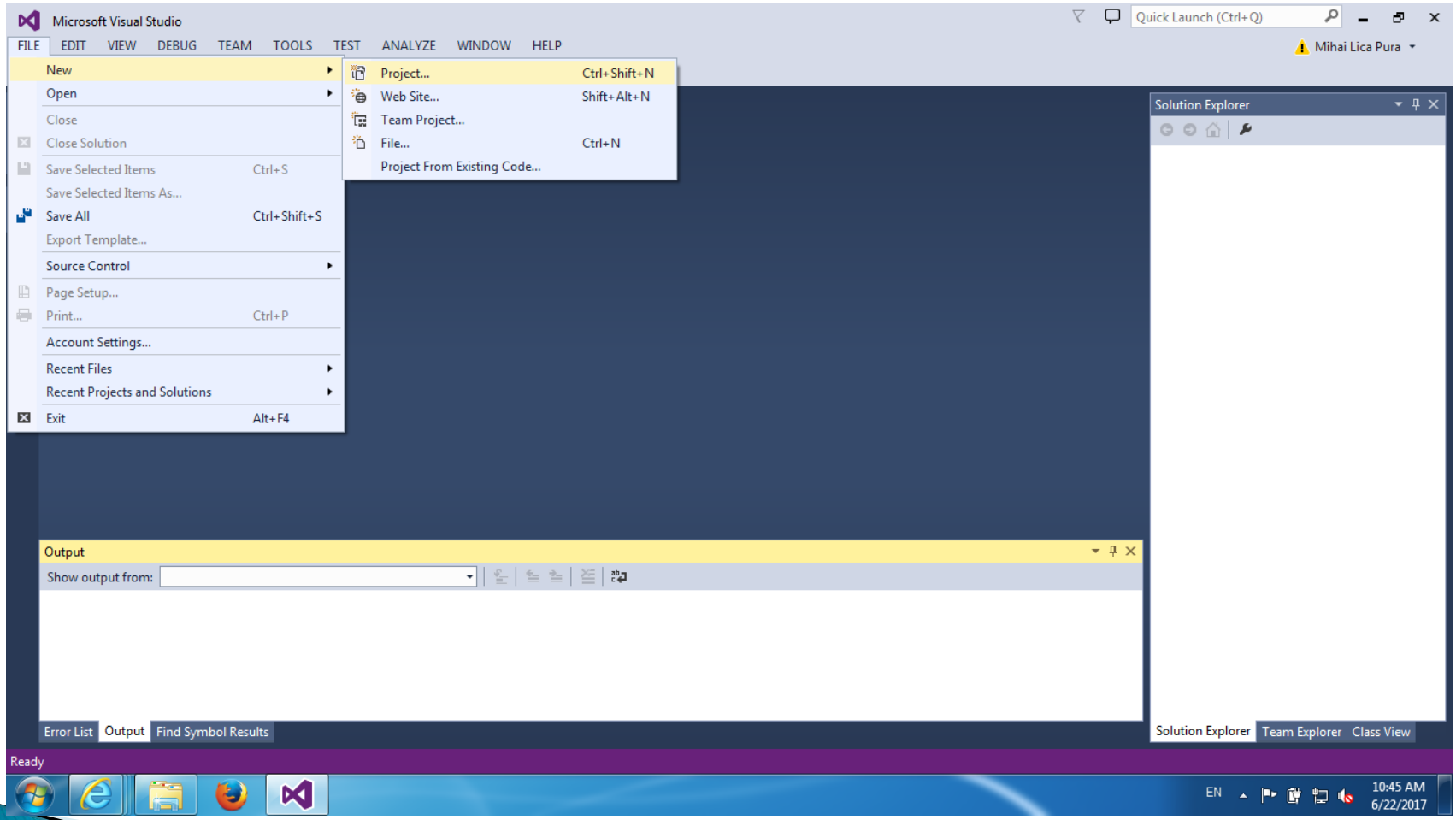
Solution from folder 3_Code Samples, Source.c

```
#include <string.h>

void do_something(char* Buffer)
{
   char MyVar[128];
   strcpy(MyVar, Buffer);
}

void main(int argc, char* argv[])
{
   do_something(argv[1]);
}
```

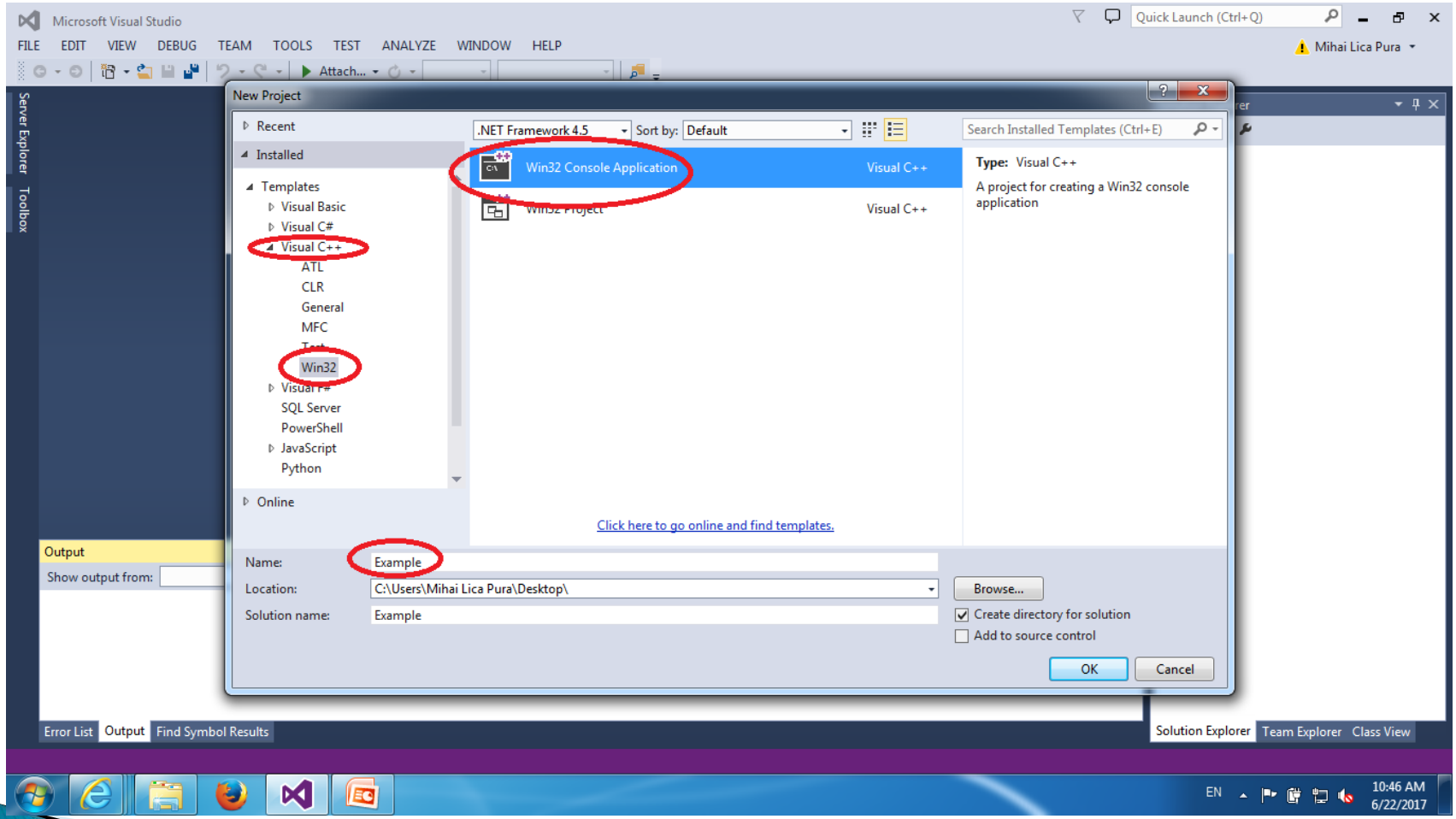
Example

- main():
  - takes an argument (argv[1]) and passes it to function do_something()
- do_something():
  - the argument is copied into a local variable
  - Local variable has a maximum of 128 bytes
  - if the argument is longer than 127 bytes (+ a null byte to terminate the string), the buffer may get overflown

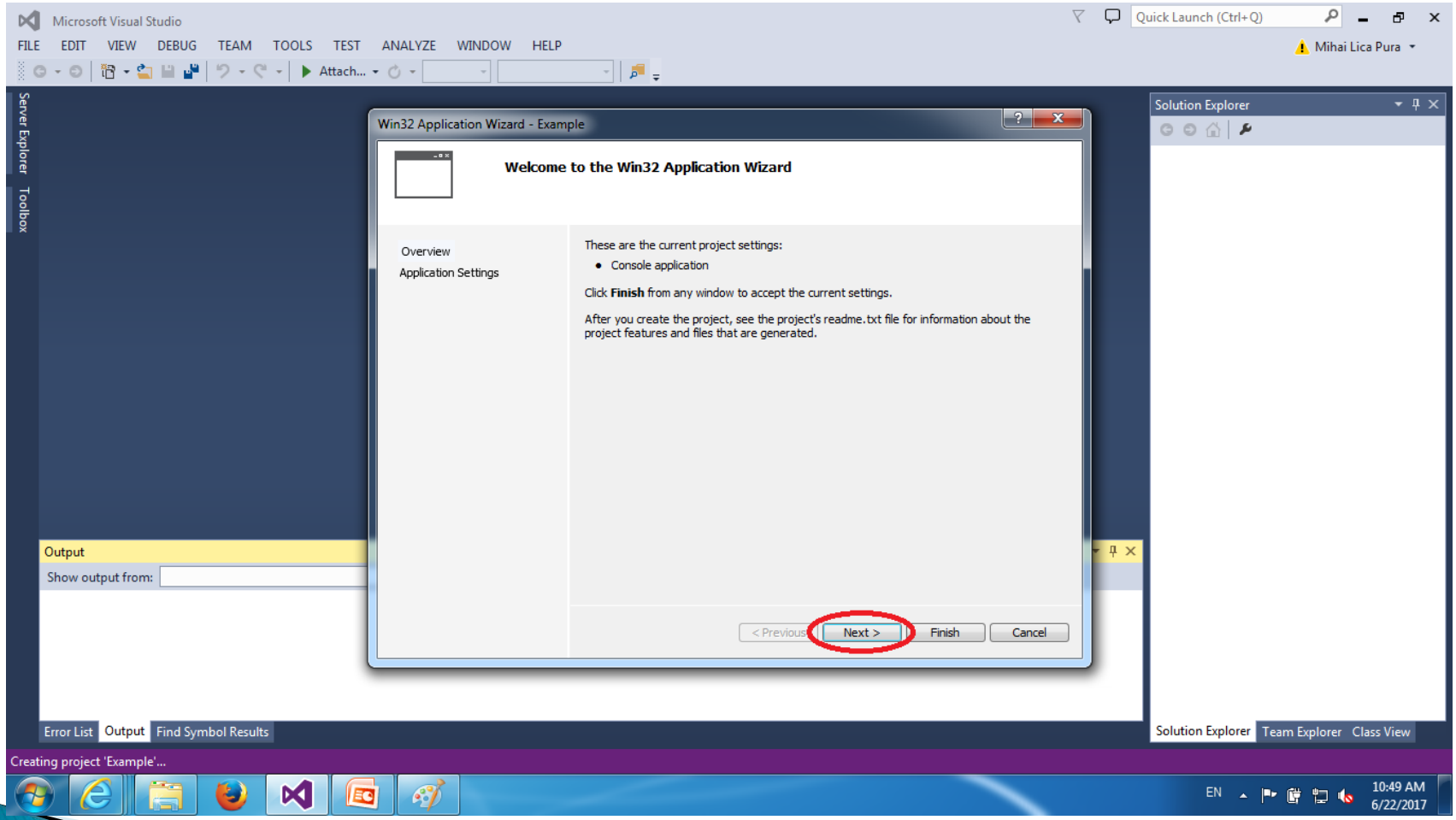
Visual Studio 2013 – Console application



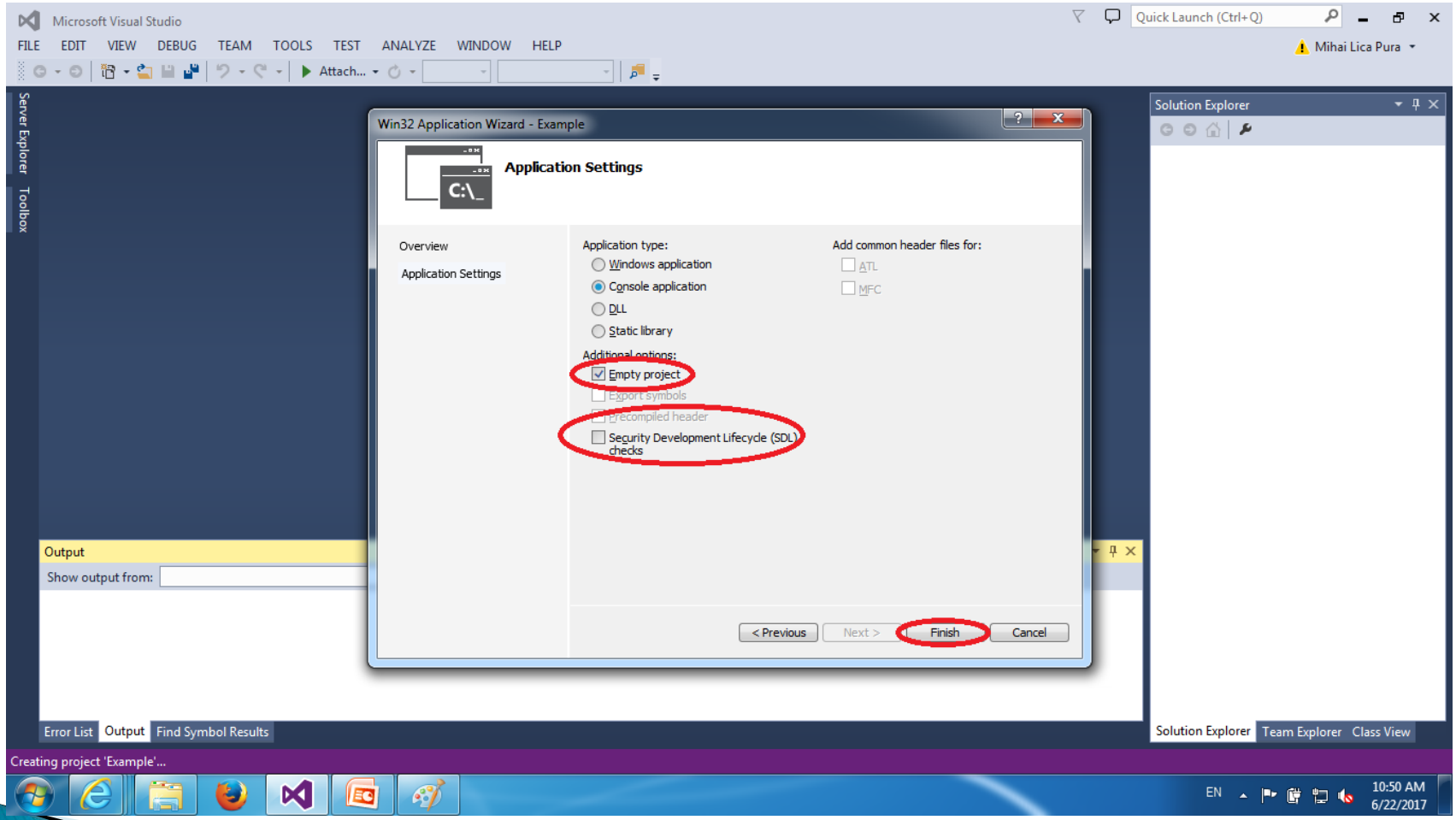
Visual Studio 2013 – Console application



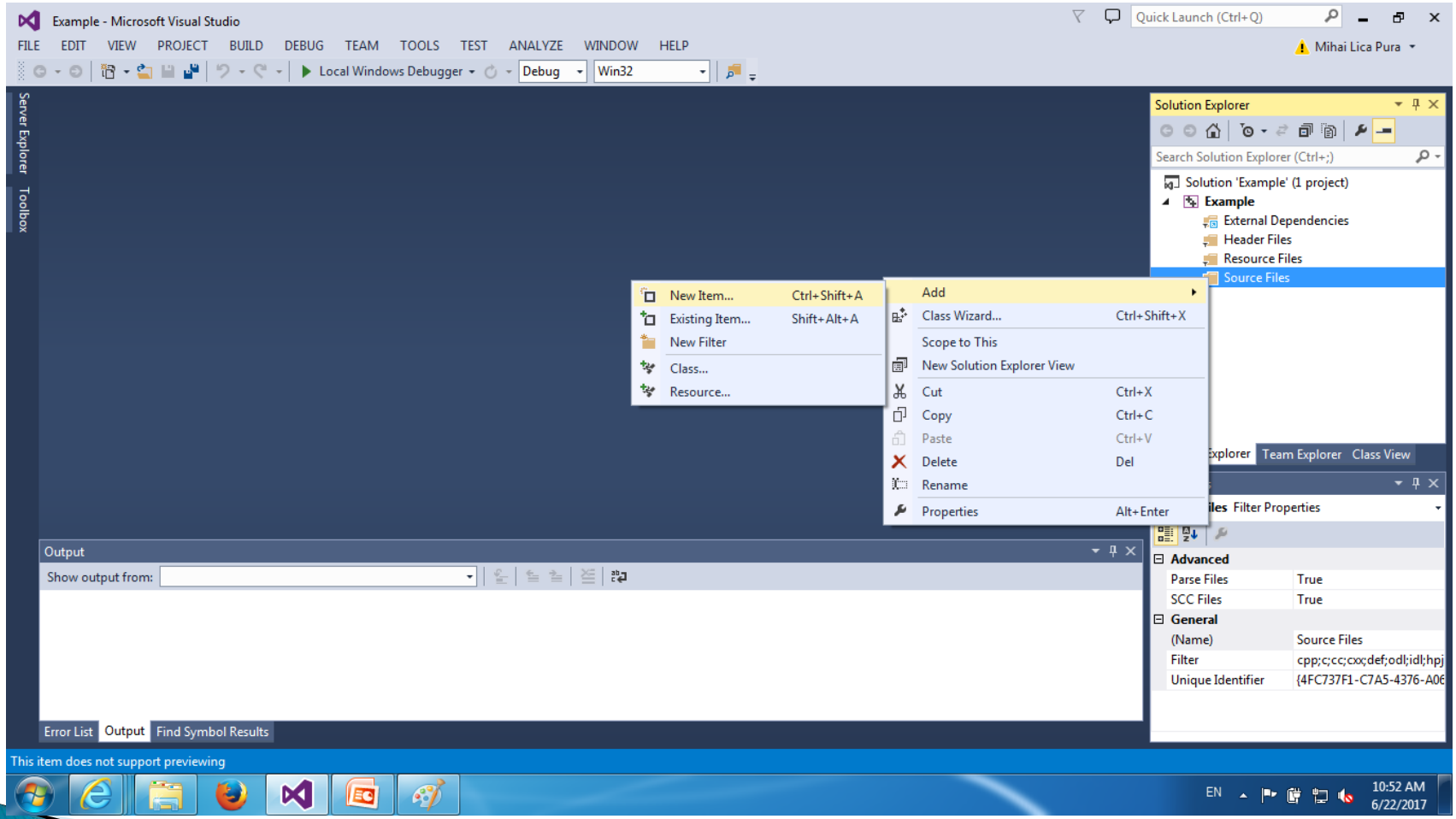
Visual Studio 2013 – Console application



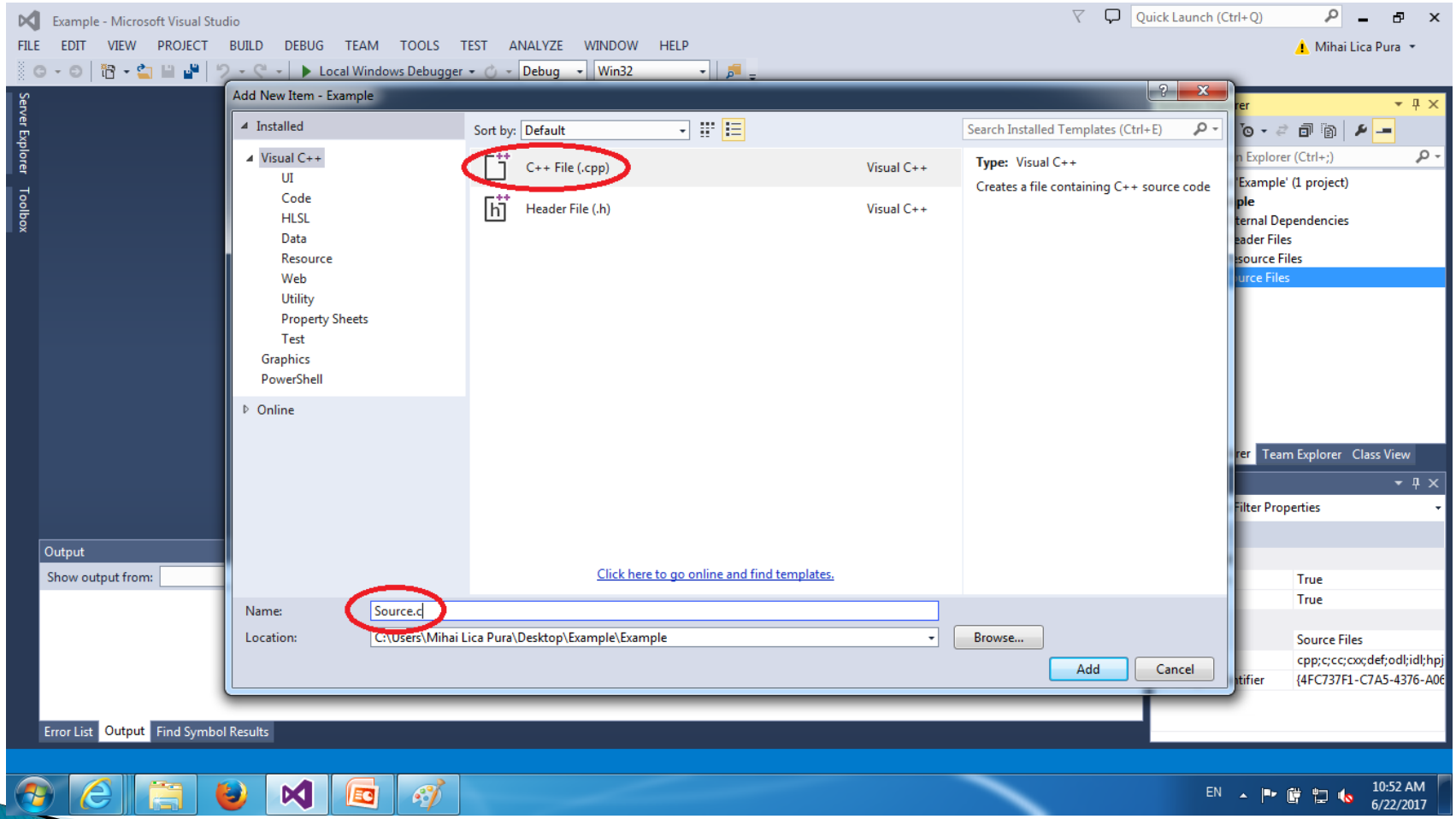
Visual Studio 2013 – Console application



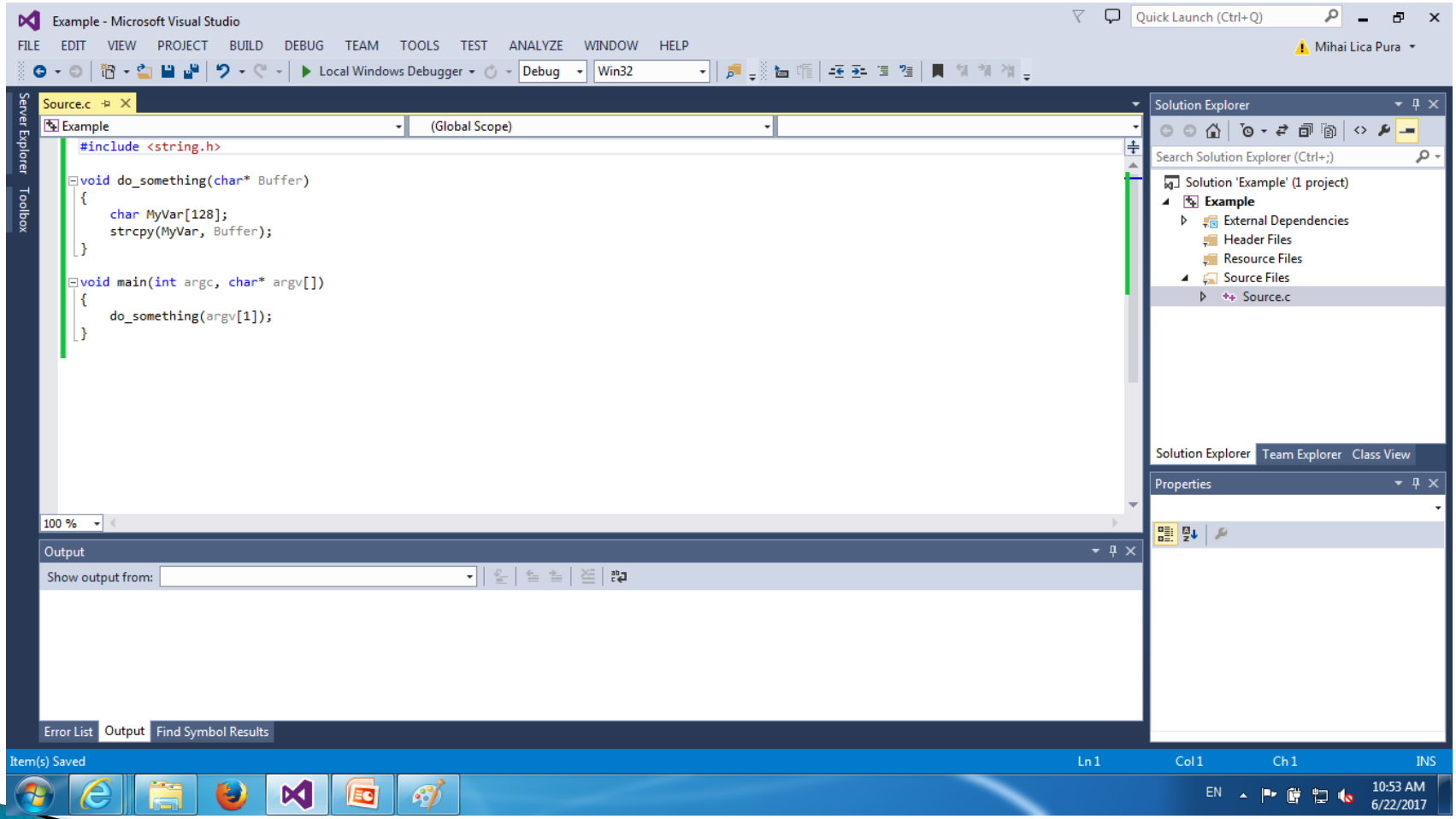
Visual Studio 2013 – Console application



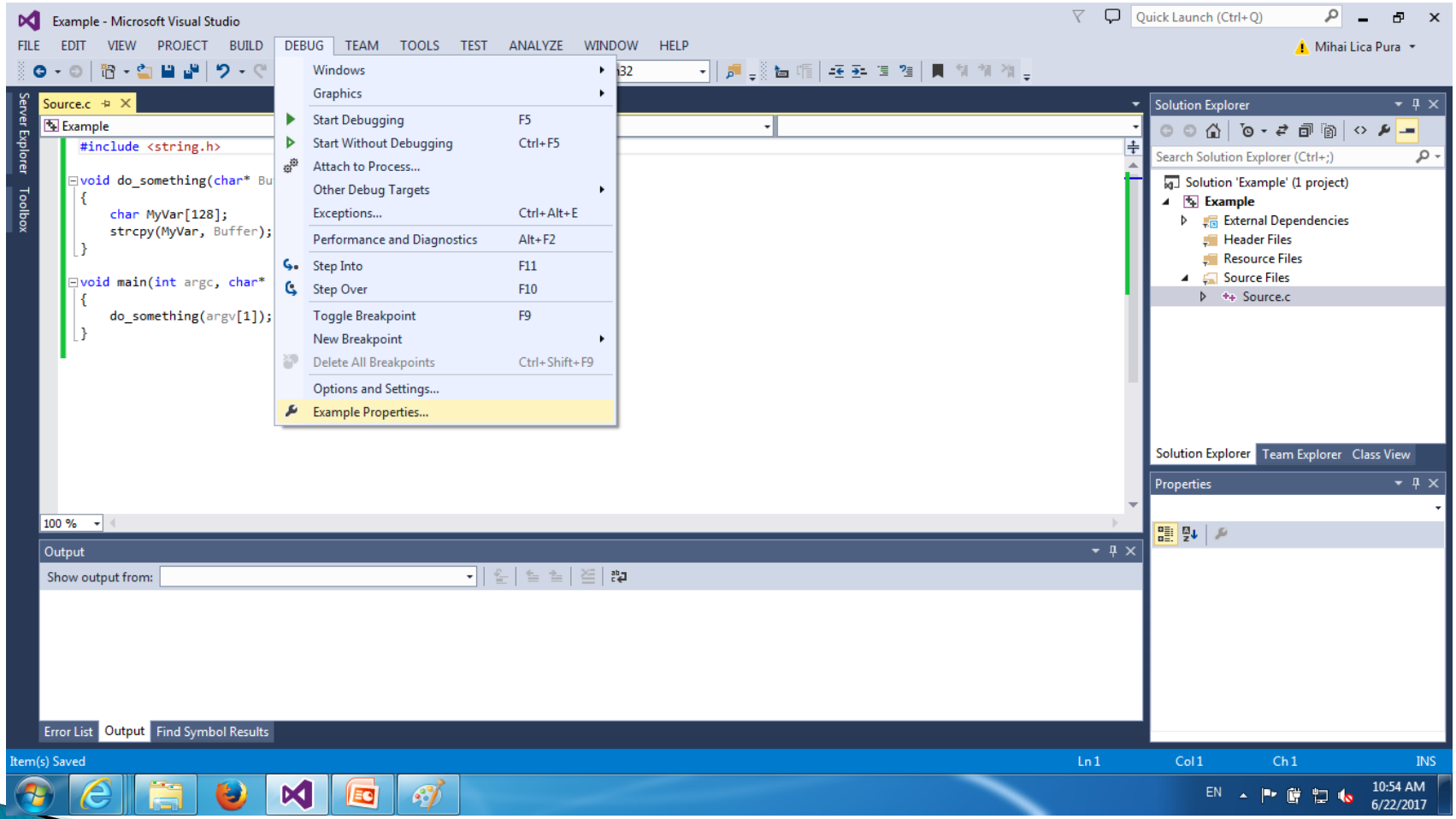
Visual Studio 2013 – Console application



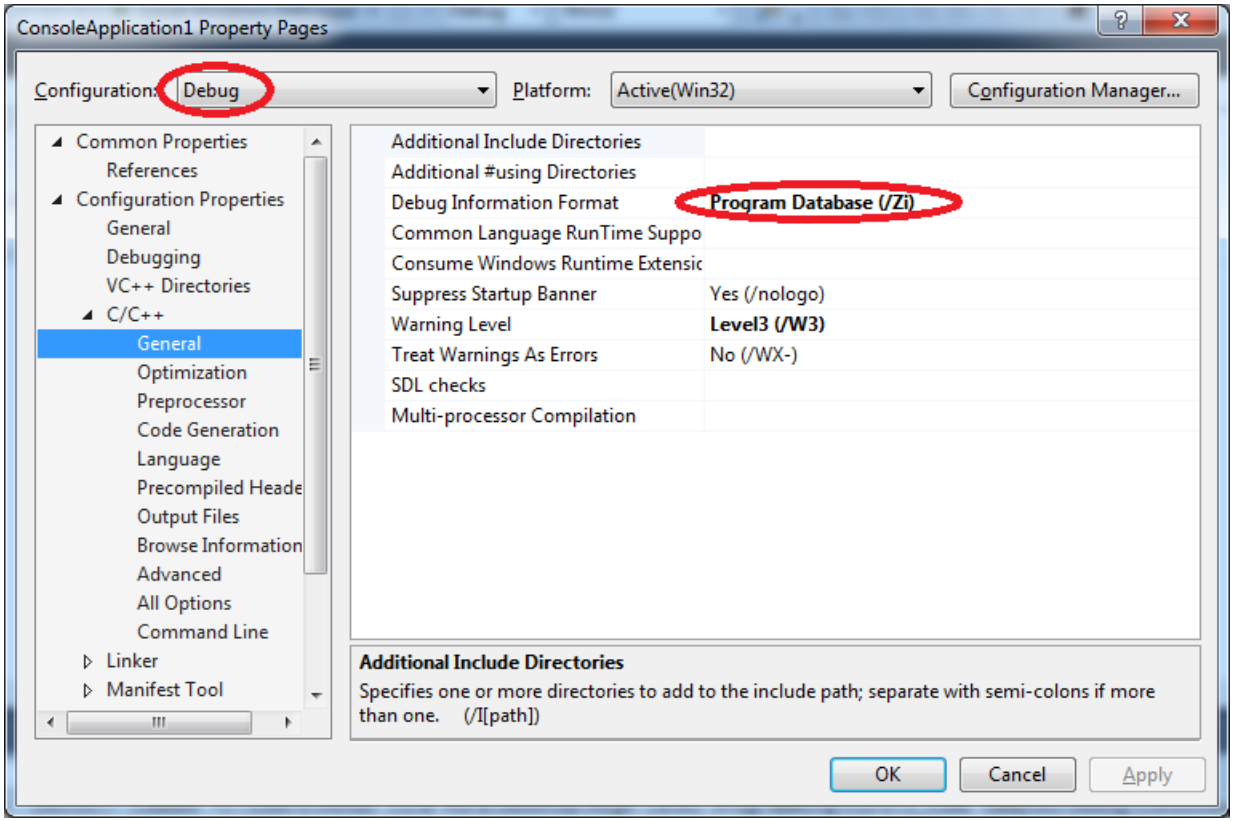
Visual Studio 2013 – Console application



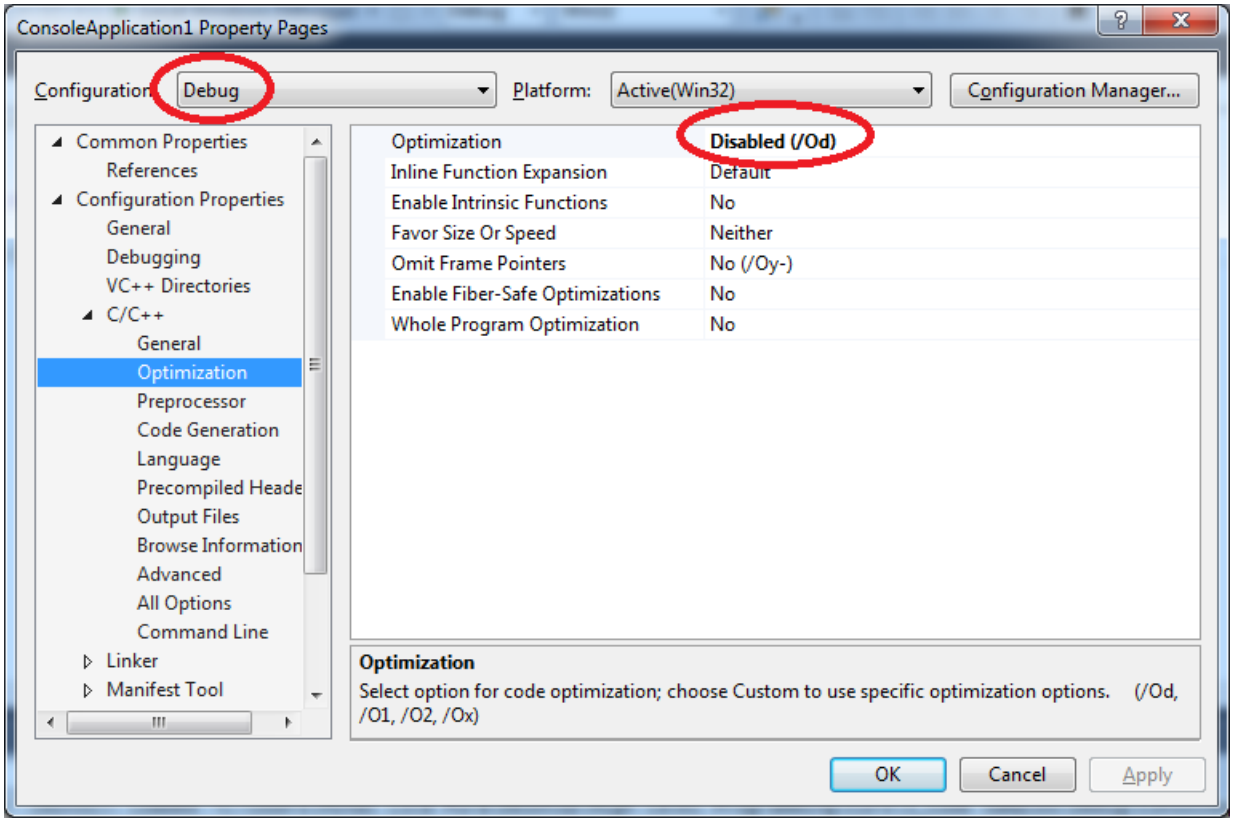
Visual Studio 2013 – Compiler configuration



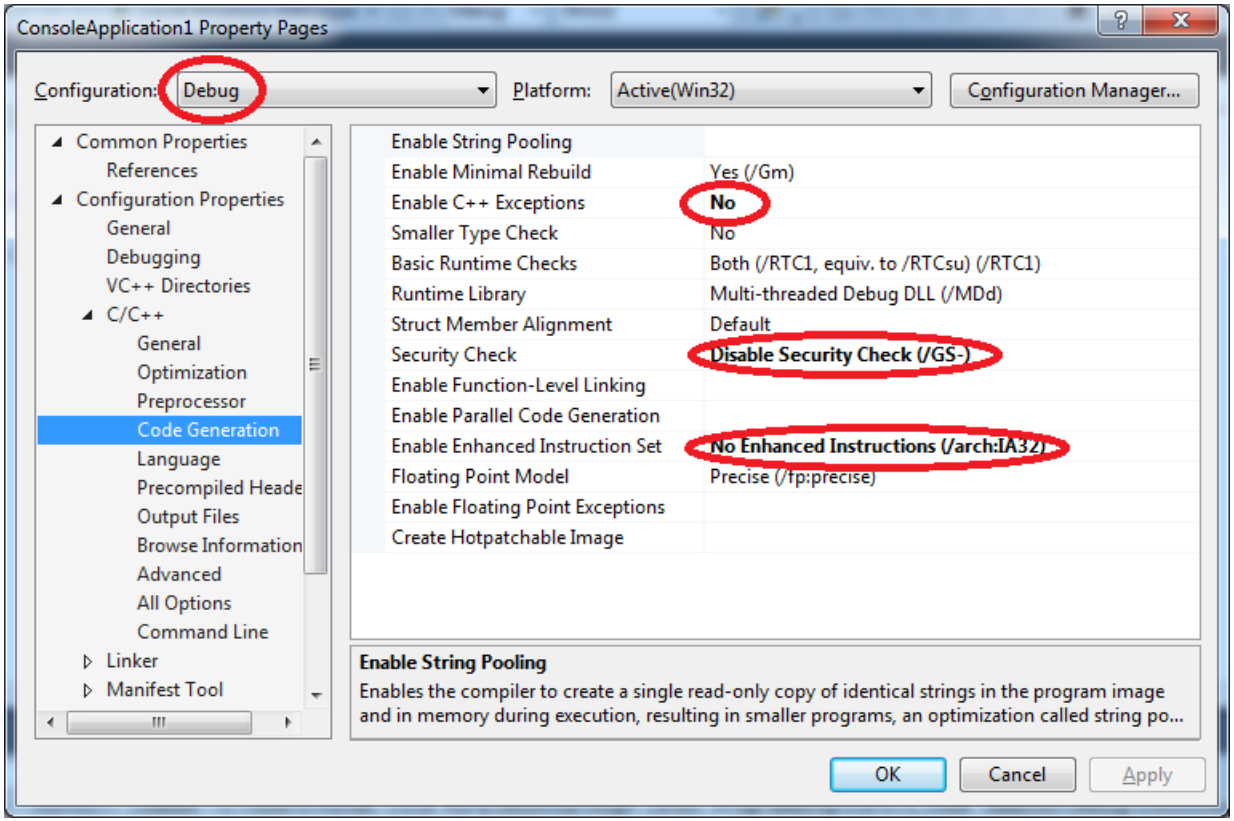
Visual Studio 2013 – Compiler configuration



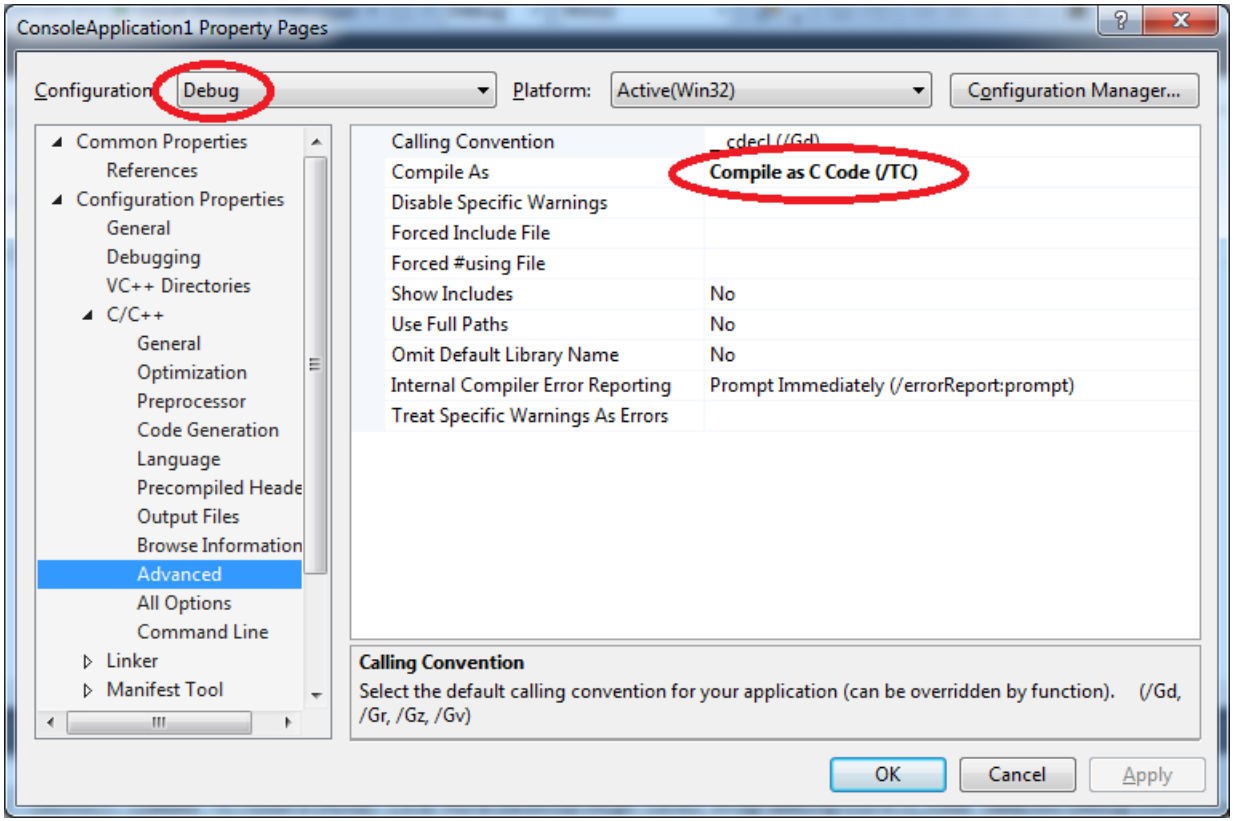
Visual Studio 2013 – Compiler configuration



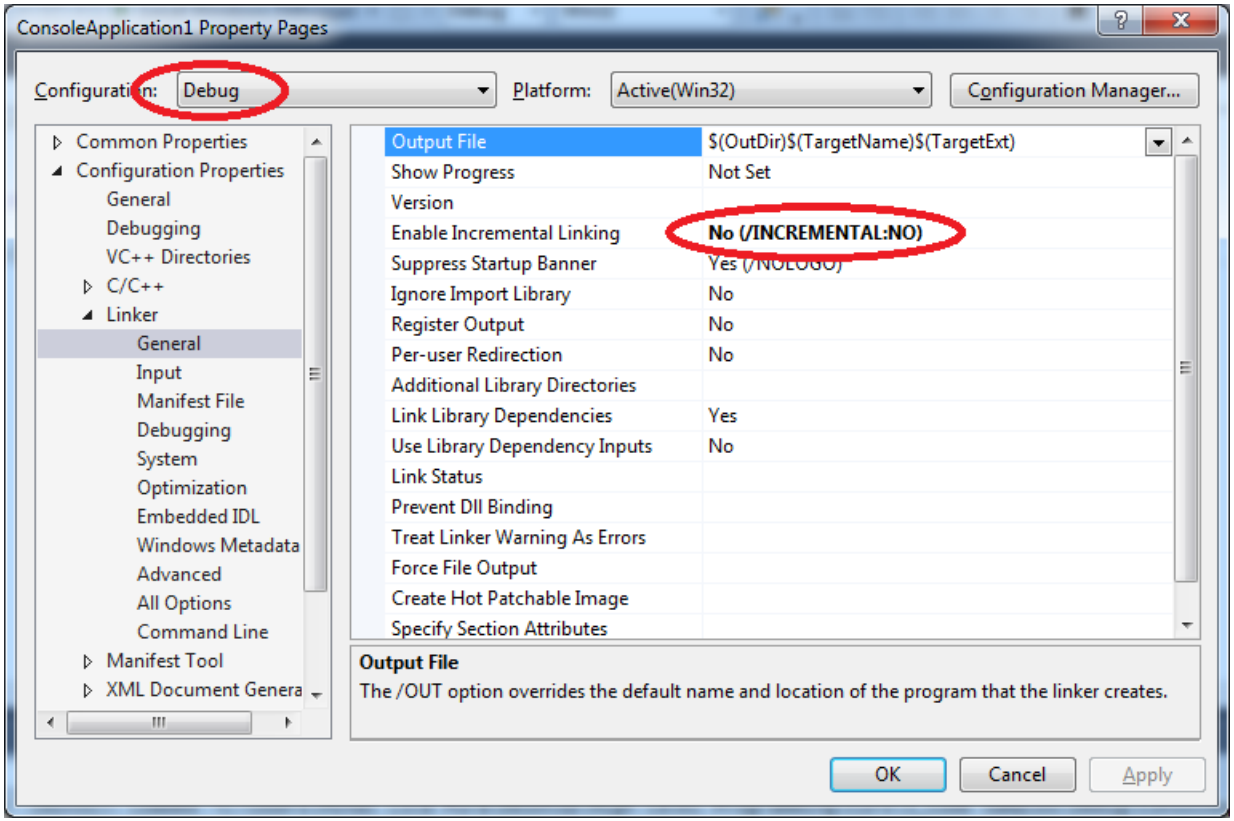
Visual Studio 2013 – Compiler configuration



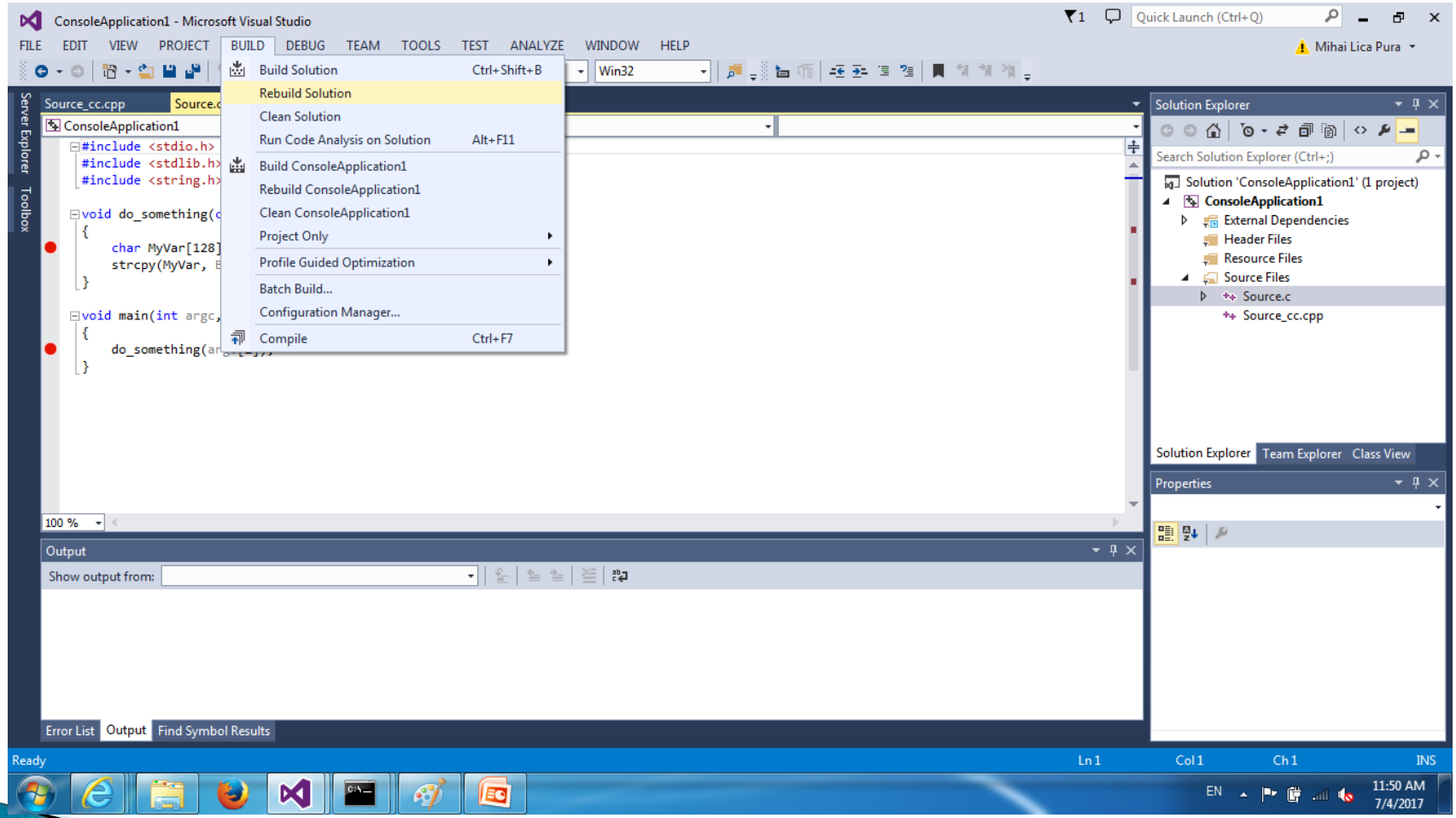
Visual Studio 2013 – Compiler configuration



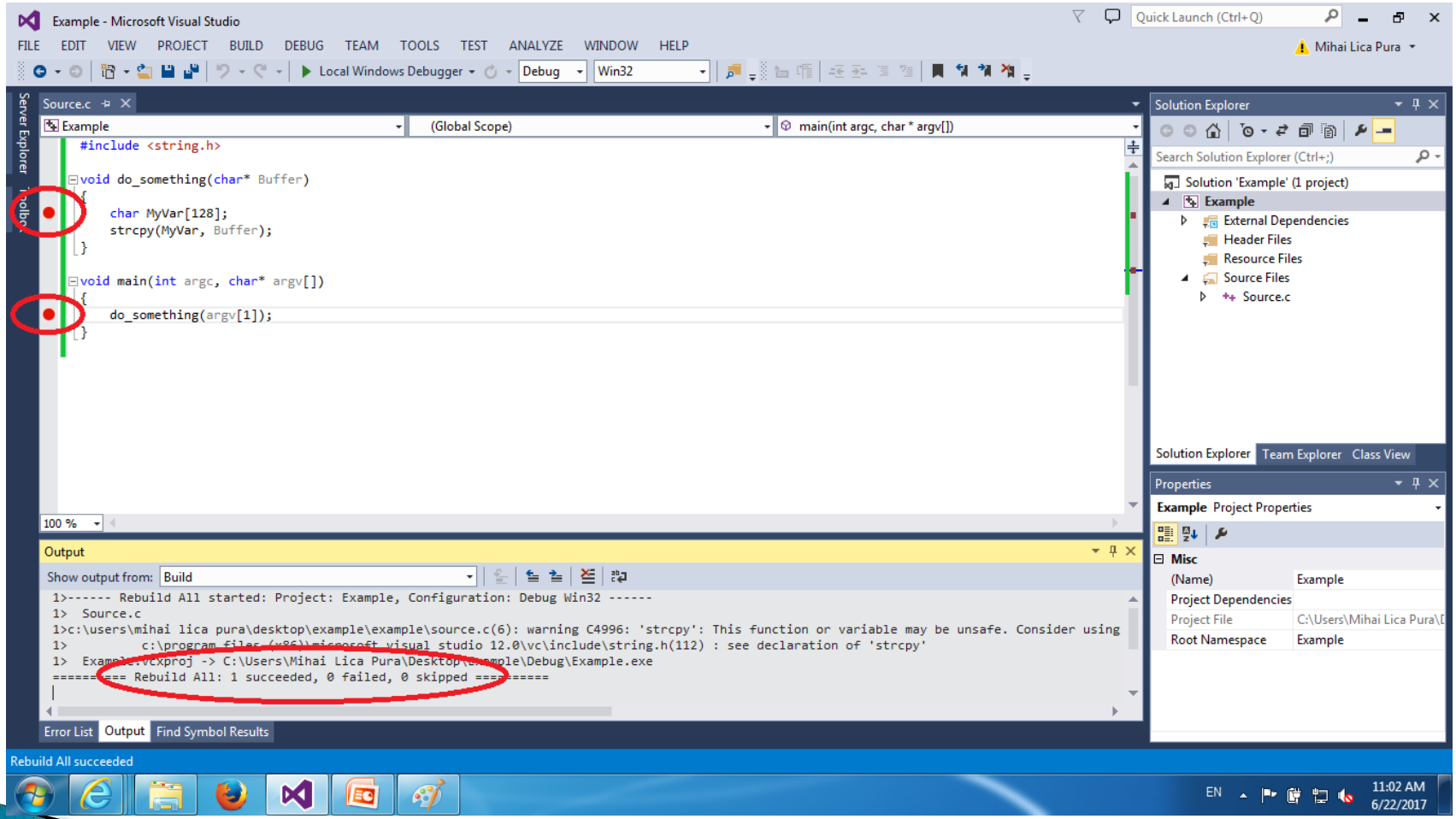
Visual Studio 2013 – Linker configuration



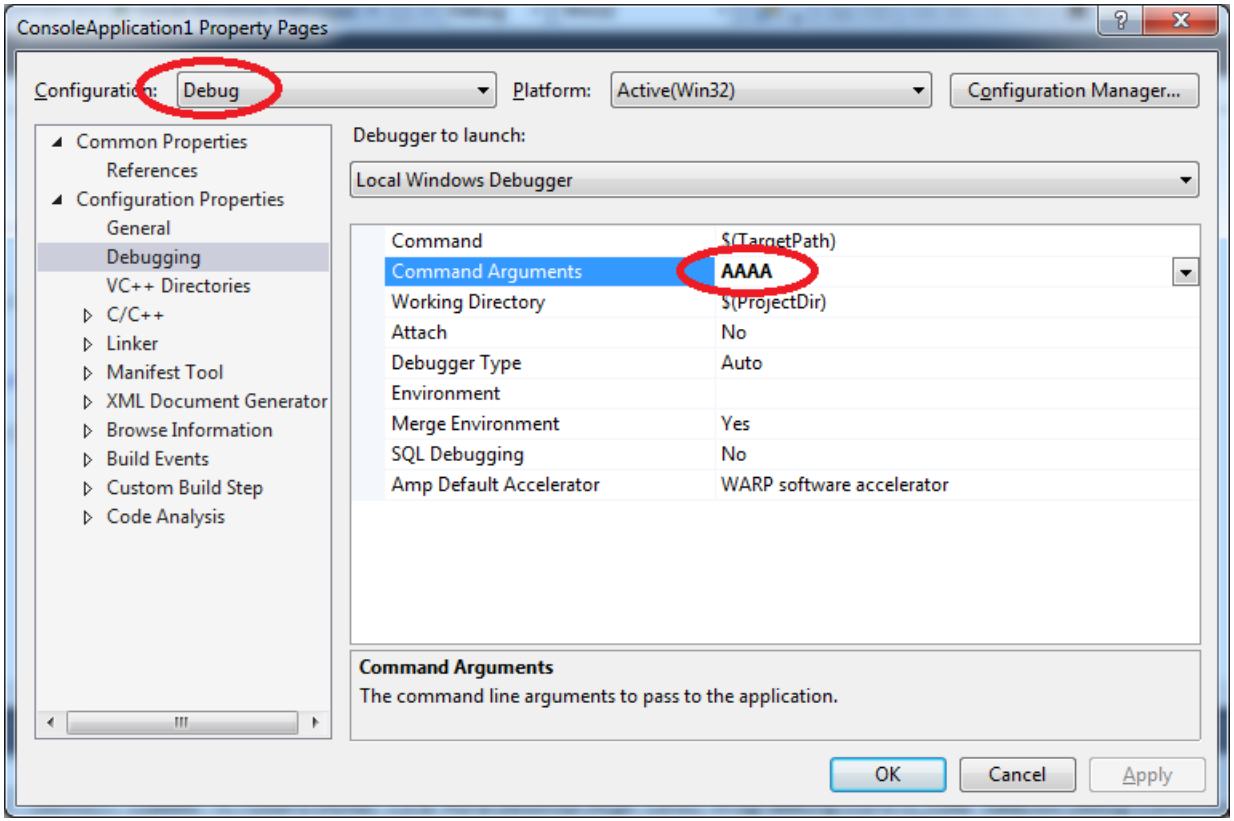
Visual Studio 2013 – Build application



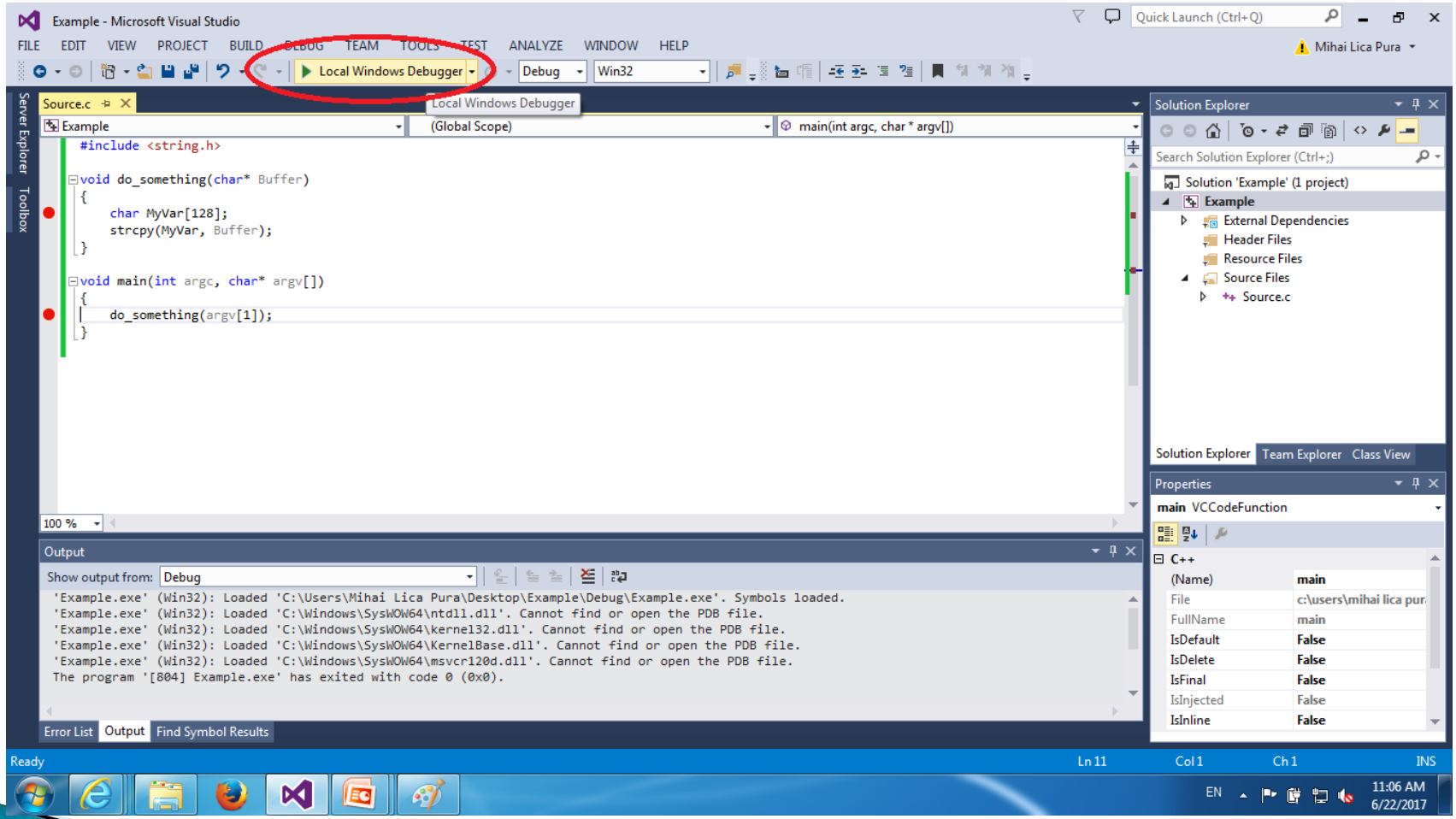
Visual Studio 2013 – Place breakpoints



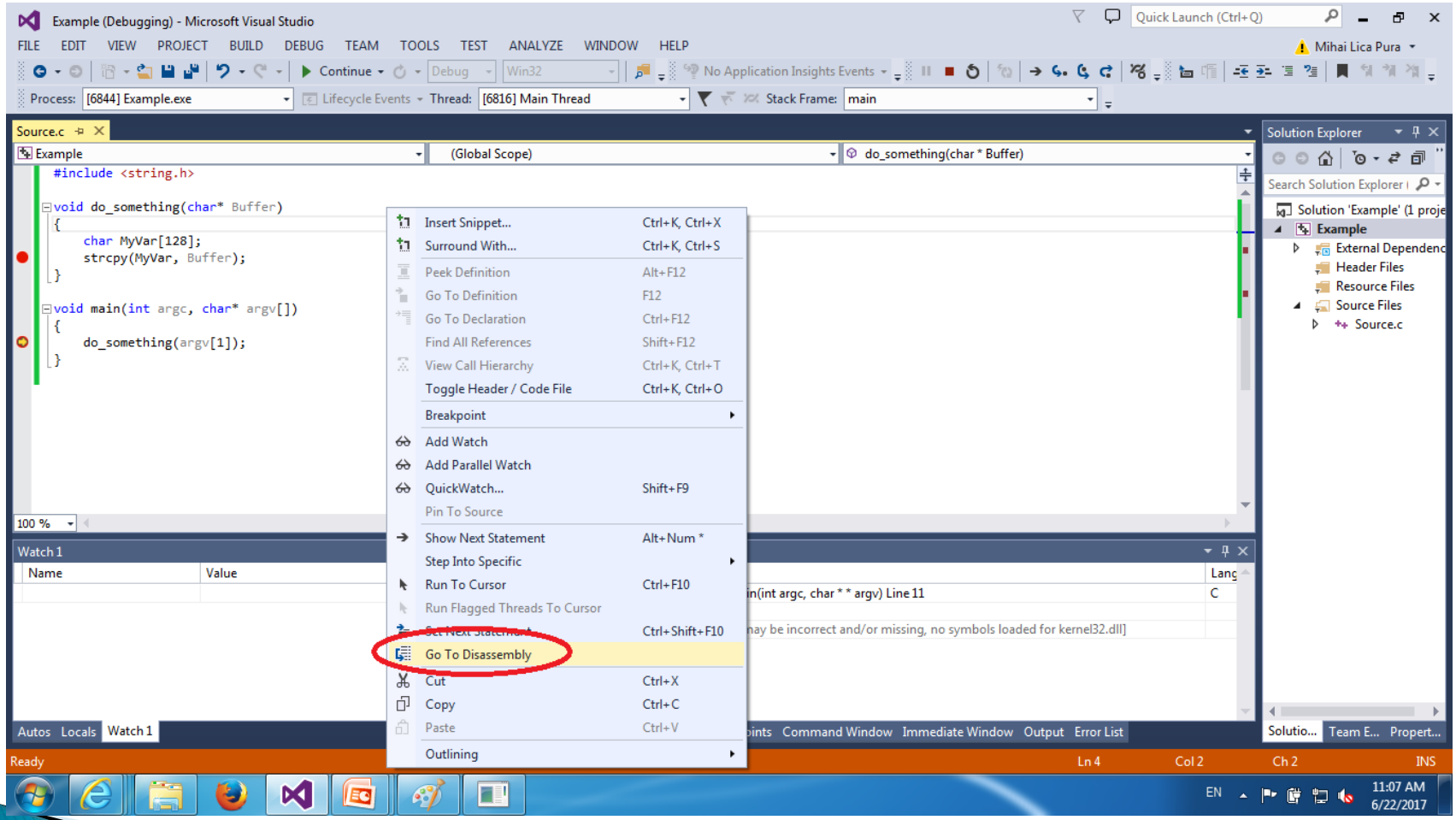
Visual Studio 2013 – Command line arguments



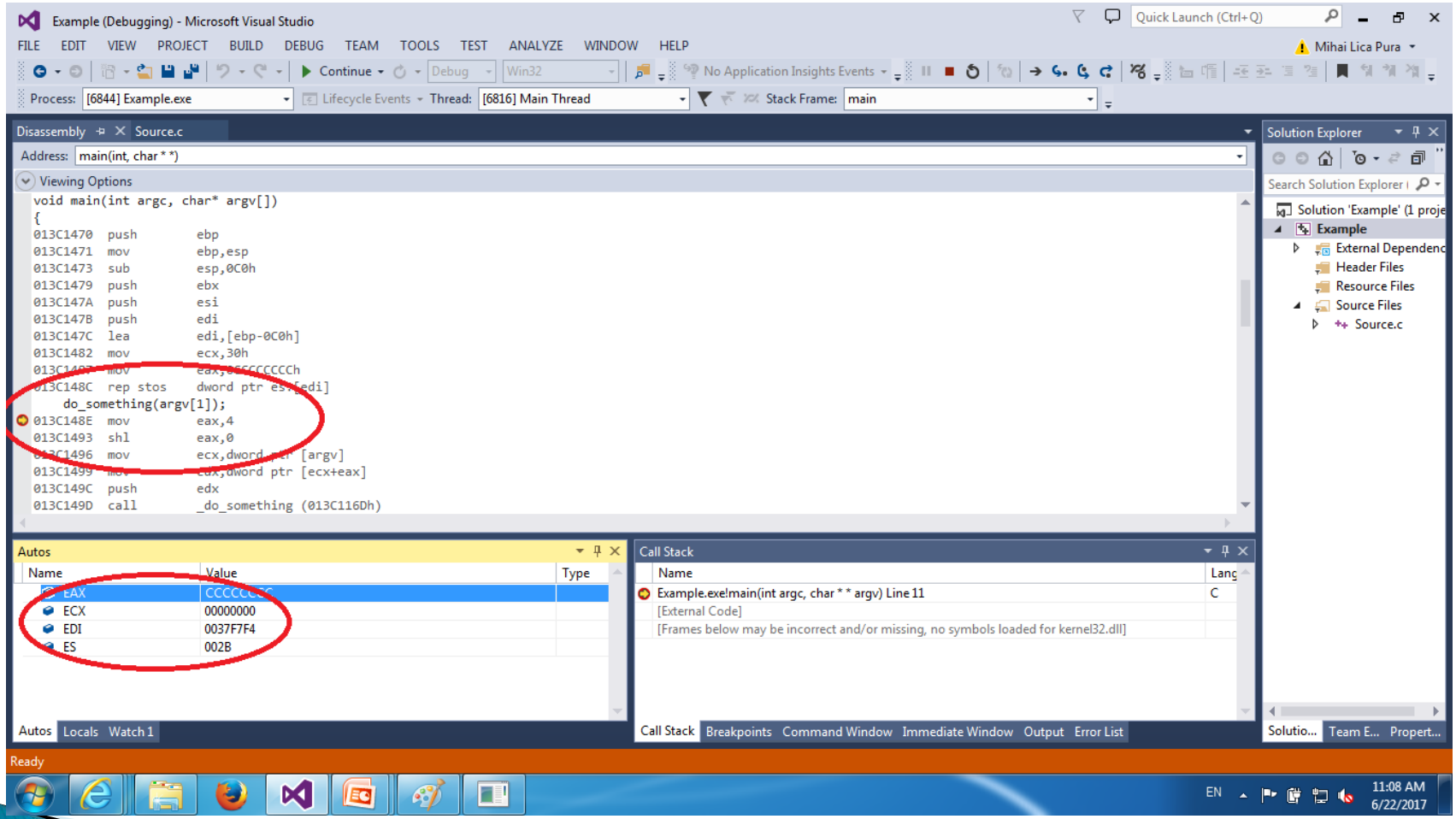
Visual Studio 2013 – Start debugging



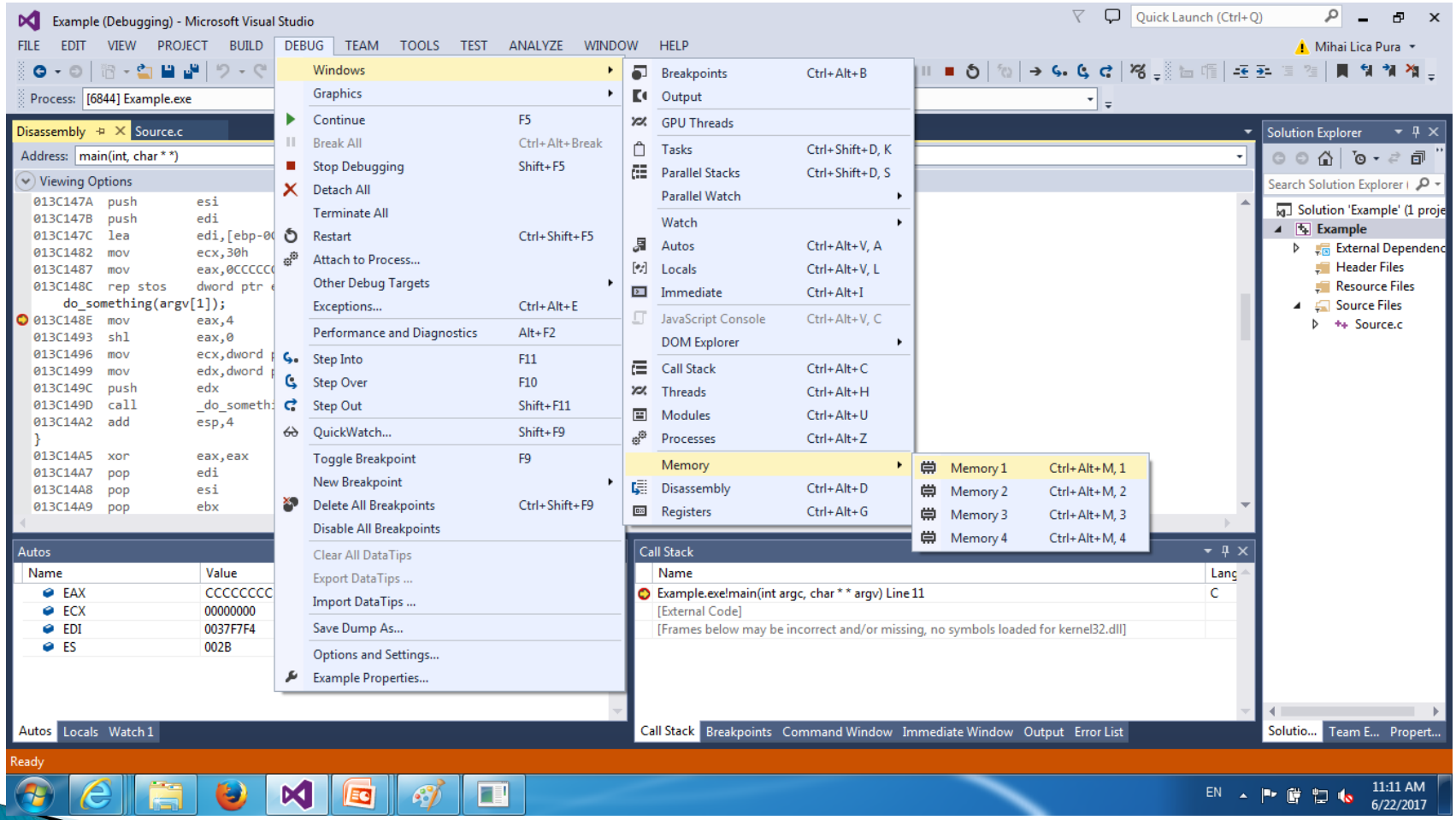
Visual Studio 2013 – Show disassembly



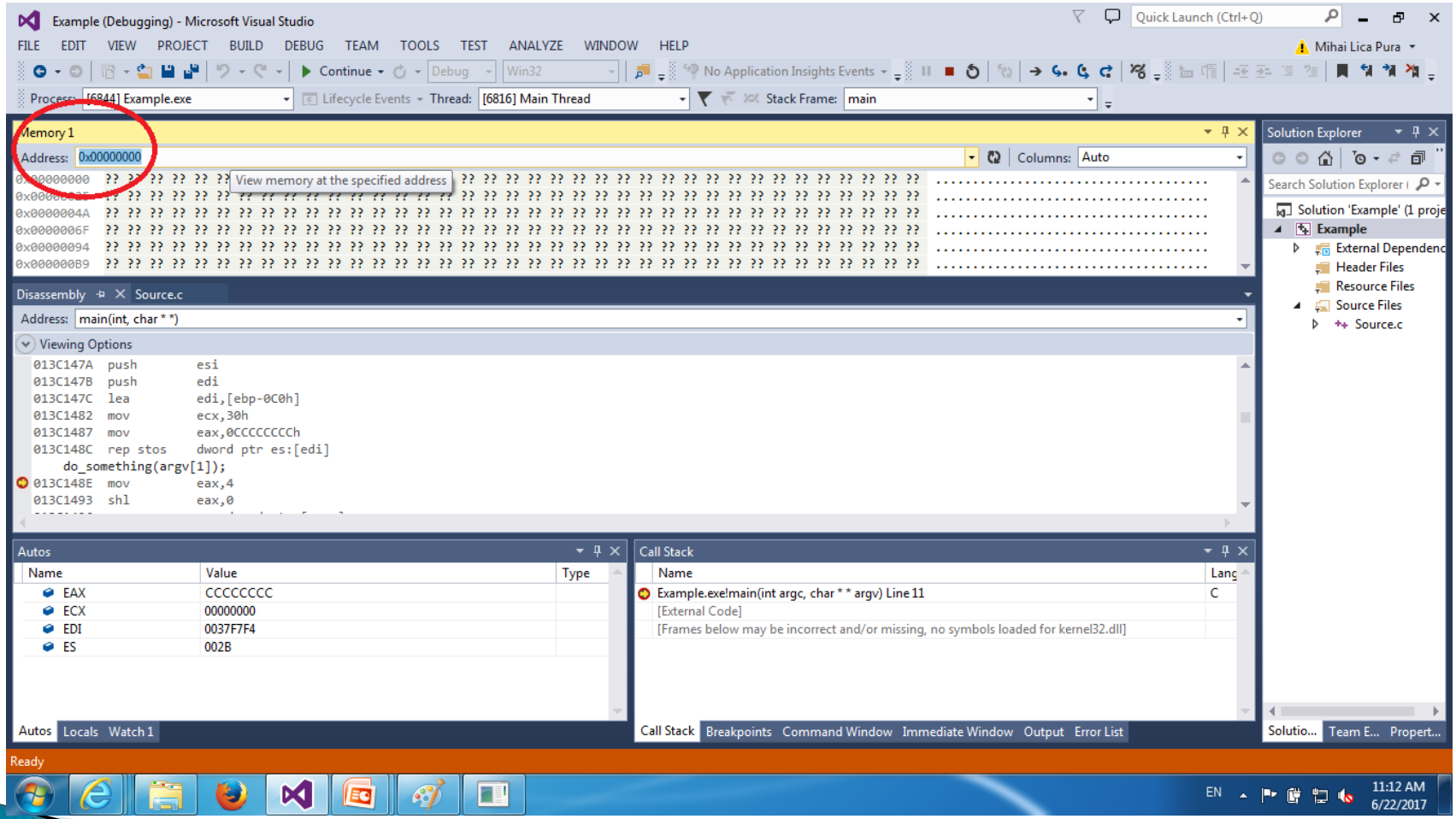
Visual Studio 2013 – Watch



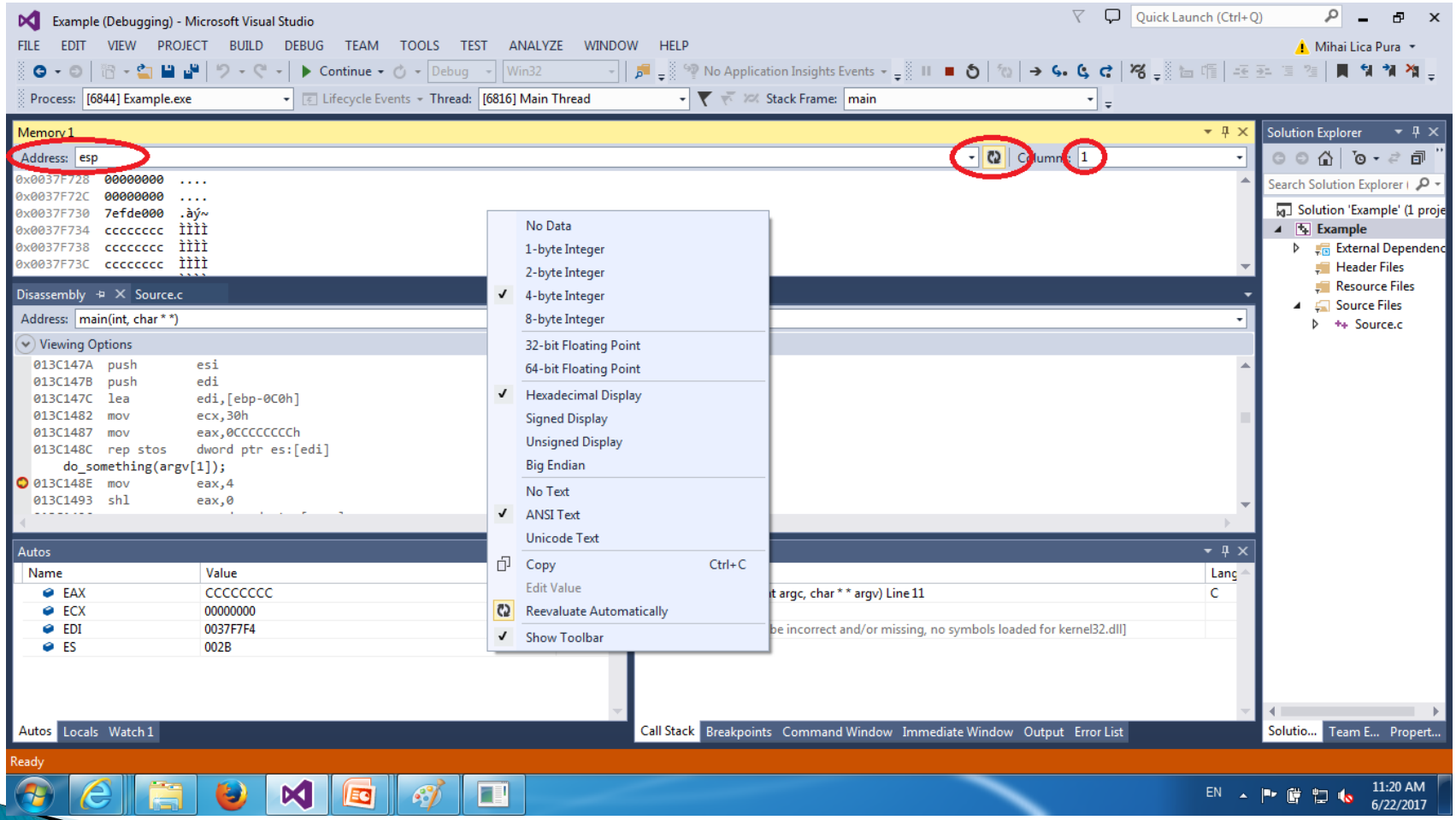
Visual Studio 2013 – Show memory



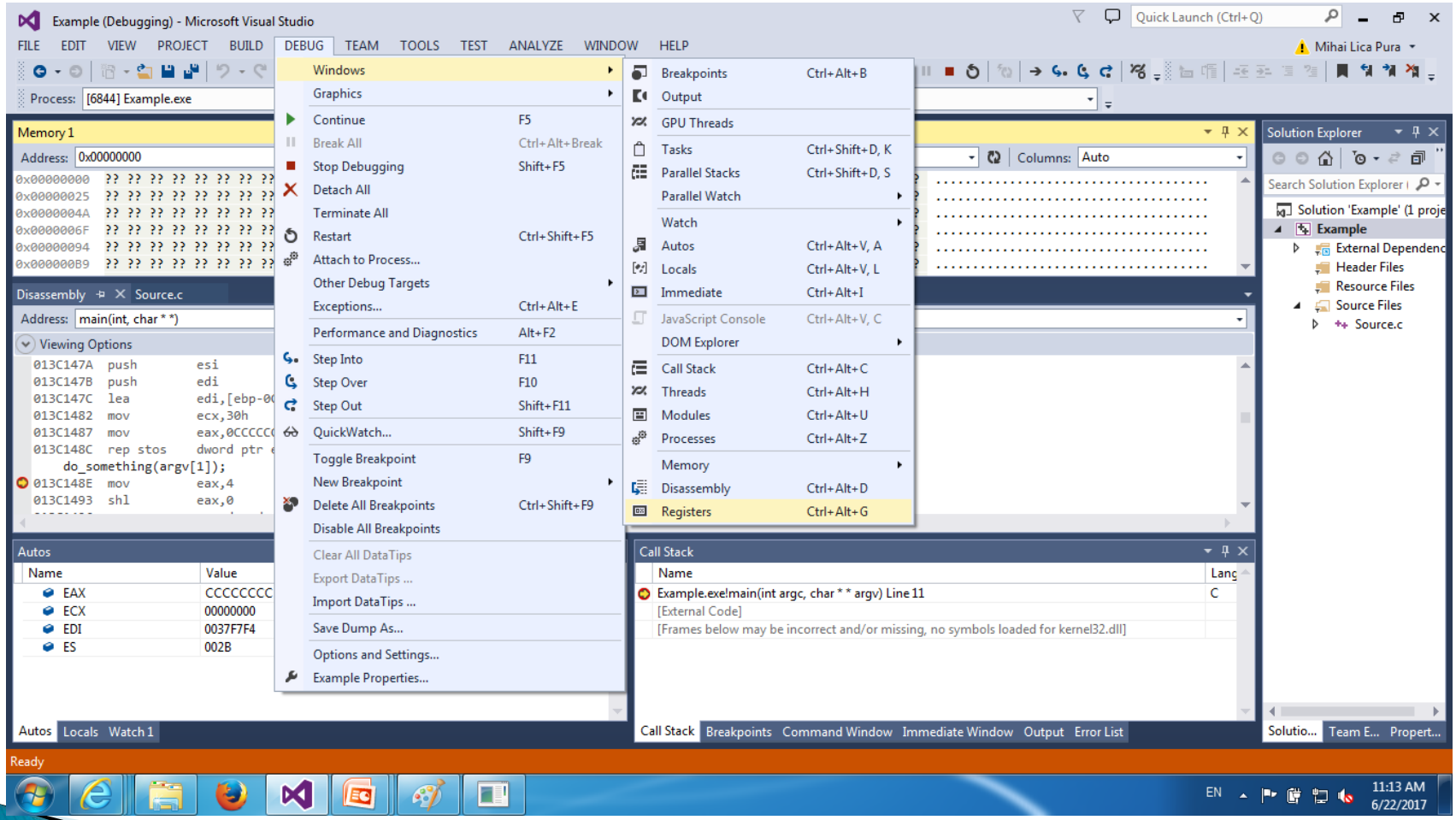
Visual Studio 2013 – Show memory



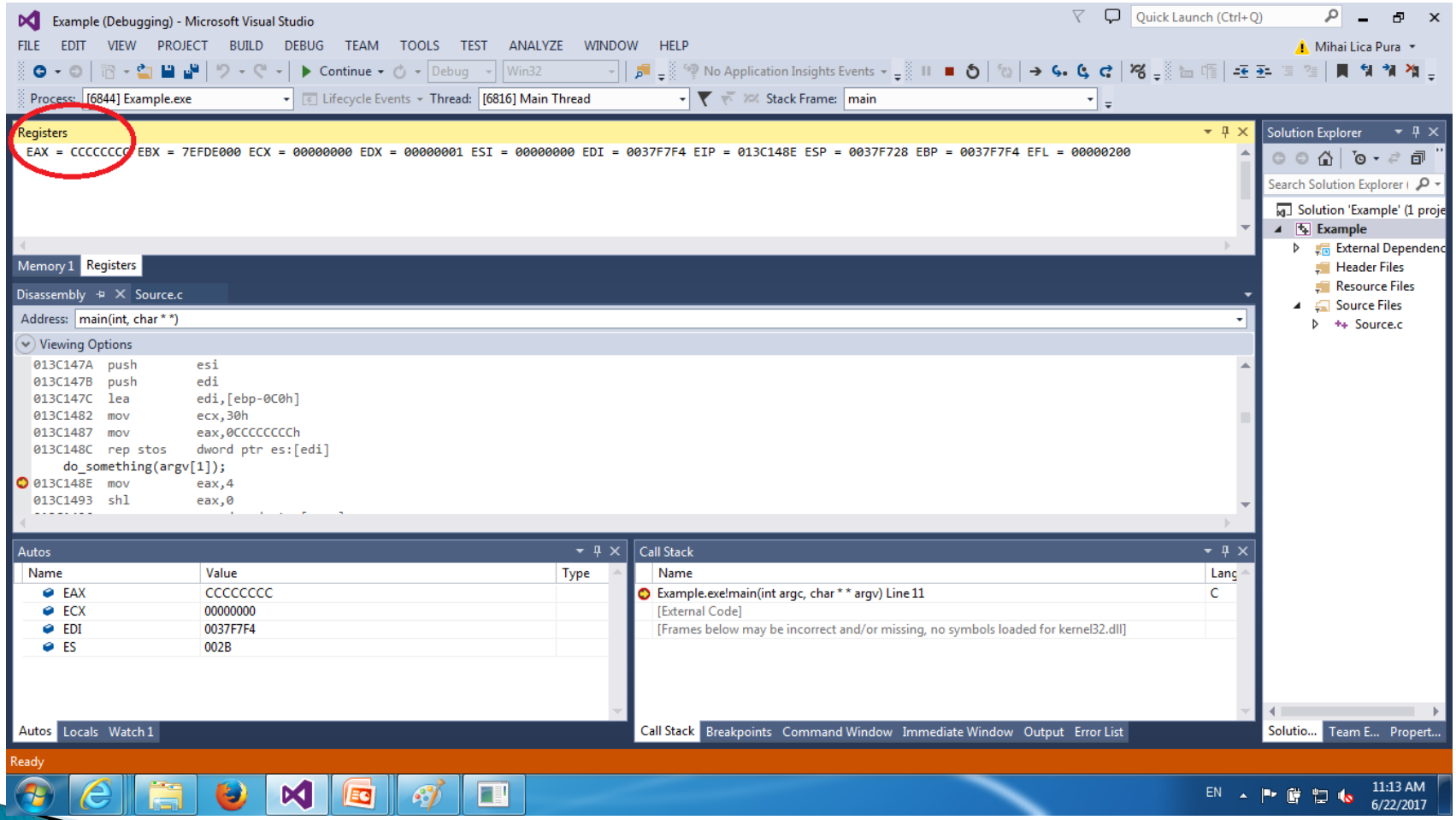
Visual Studio 2013 – Watching the stack



Visual Studio 2013 – Show registers



Visual Studio 2013 – Show registers



Example – main

```
; before the call for do_something() – standard
  prologue
push    ebp                     ; preserve registers on
  the stack
mov     ebp,esp                 ; highest address of
  new stack frame
```

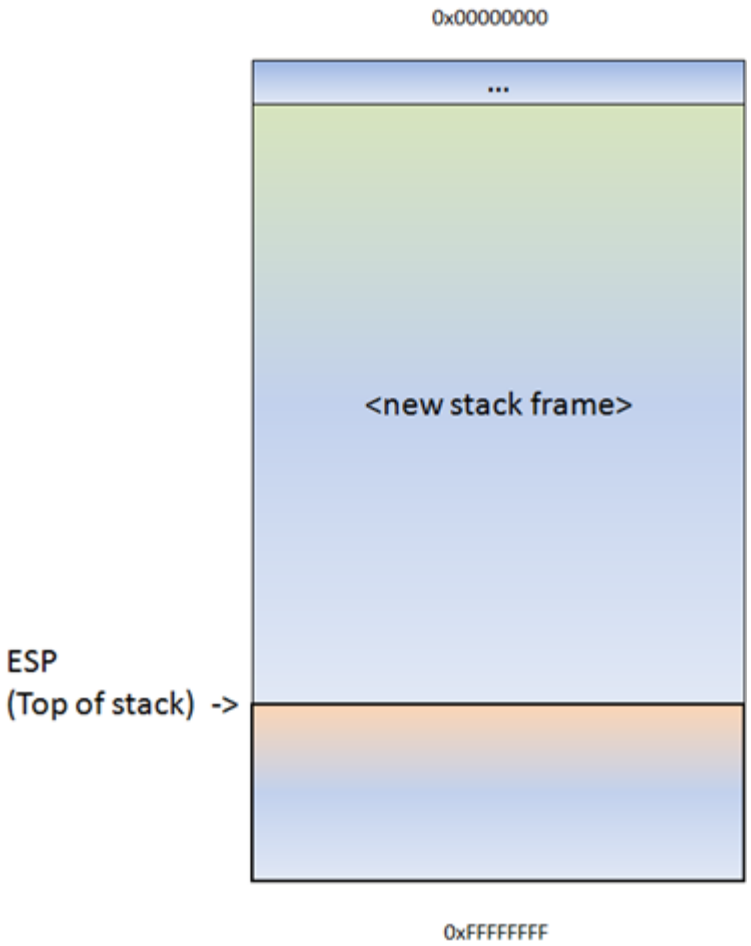
Example – main

```
; the call for do_something()
mov         eax,4
shl         eax,0
mov         ecx,dword ptr [argv]                ; base
  address of argv
mov         edx,dword ptr [ecx+eax]  ; &argv[1] =
  *argv + 4
push        edx                                 ; push address
  on stack for
       do_something()
call        _do_something (019116Dh)
```

Example – do_something()

- a new stack frame will be created
  - on top of the ‘parent’ stack
  - the stack pointer (ESP) points to the highest address of the newly created stack
  - this is the "top of the stack"

Example – do_something()

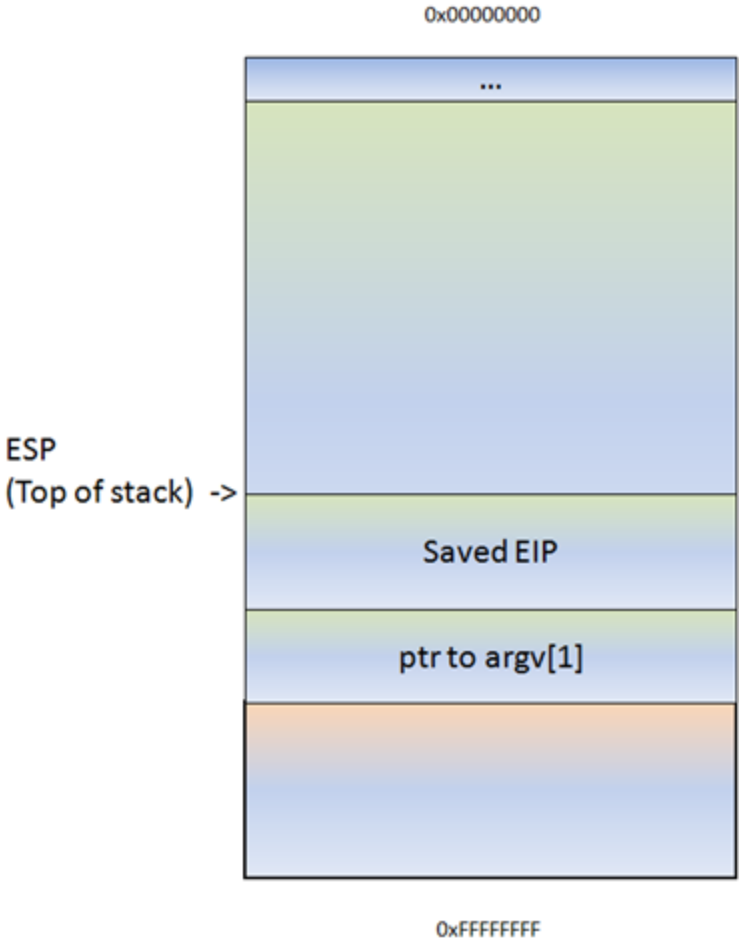


Example – do_something()

- before do_something() is called
  - a pointer to the argument(s) gets pushed to the stack
  - in this case – a pointer to argv[1]

- call function do_something()
  - CALL instruction pushes the current instruction pointer onto the stack
    - so it knows where to return to if the function ends
  - then will jump to the function code

Example – do_something()



Example – do_something()

```
; standard prologue (same as for main)
push    ebp                 ; preserve registers on the
  stack
mov     ebp,esp             ; highest address of new
  stack frame
; declares/allocates space for local variables (MyVar)
sub     esp,88h                 ; allocates 136 bytes
  on the                                stack
```

Example – do_something()

```
push      edi
lea       edi,[ebp-88h]
mov       ecx,22h
mov       eax,0CCCCCCCCh
rep stos  dword ptr es:[edi]
; writes the value in EAX (0CCCCCCCCh)
  starting at the address pointed to by EDI
  (local stack), ECX (34) times
; 0xcccccccc is a magic number chosen by
  Microsoft to indicate uninitialized memory
; this extra diagnostic checking is enabled by
  the /RTCu option
```

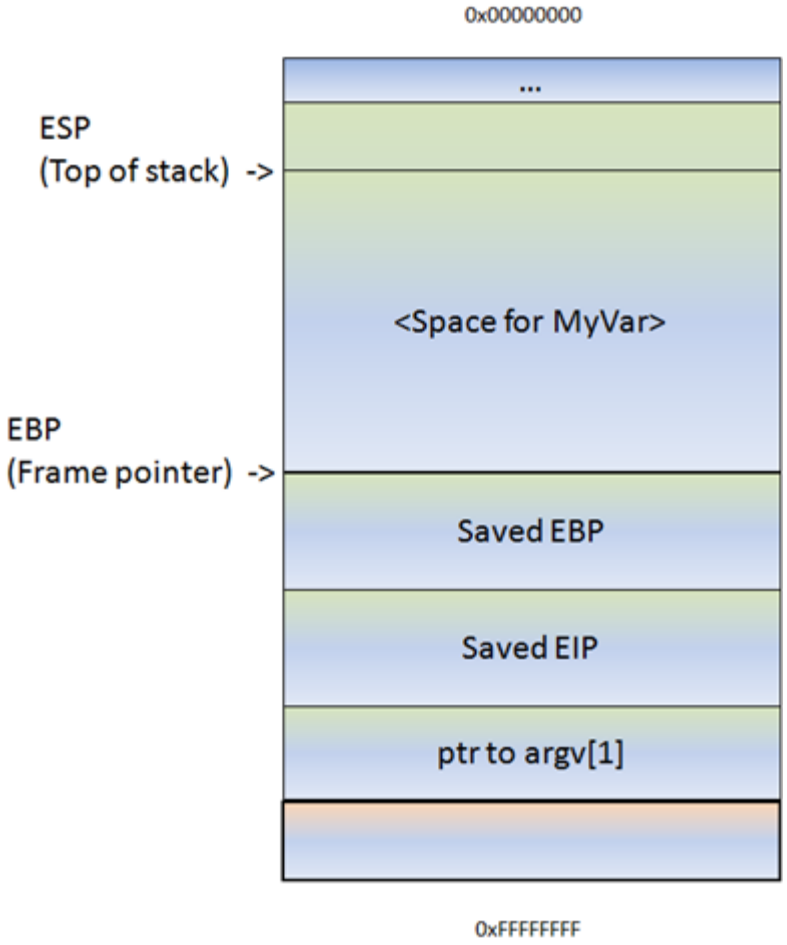
Example – do_something()

- the prolog of do_something() executes
  - after the push of EBP, ESP is put in EBP
  - both ESP and EBP point at the top of the current stack
  - from that point on, the stack will be referenced by ESP (top of the stack at any time) and EBP (the base pointer of the current stack).
  - this way, the application can reference variables by using an offset to EBP

Example – do_something()

- stack space for the variable MyVar (128bytes) is declared/allocated.
  - some space is allocated on the stack to hold data in this variable
  - ESP is decremented by a number of bytes
  - this number of bytes will most likely be more than 128 bytes, because of an allocation routine determined by the compiler.

Example – do_something()



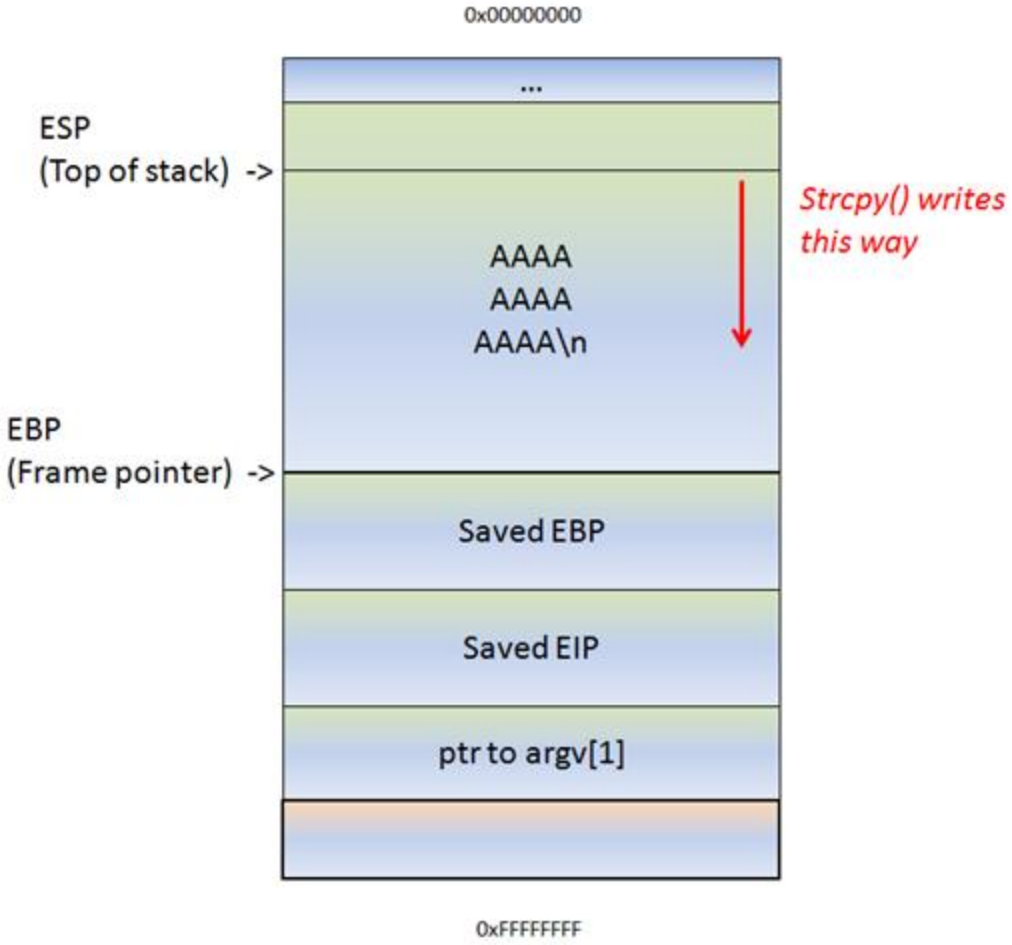
Example – do_something()

```
; the call for strcpy()
mov     eax,dword ptr [Buffer]      ; base address of
  Buffer
push    eax                         ; push address on
  stack for strcpy
lea     ecx,[MyVar]                 ; base address of
  MyVar
push    ecx                         ; push address on
  stack for strcpy
call    _strcpy (01910AAh)
```

Example – do_something()

- before strcpy() is called
  - a pointer to the argument(s) gets pushed to the stack
  - in this case – a pointer to Buffer and MyVar
- Call for strcpy() function
  - reads data from the address pointed to by [Buffer] and stores it in [MyVar]
  - reads all data until it sees a null byte (string terminator)
  - while copying, ESP stays where it is

Example – do_something()



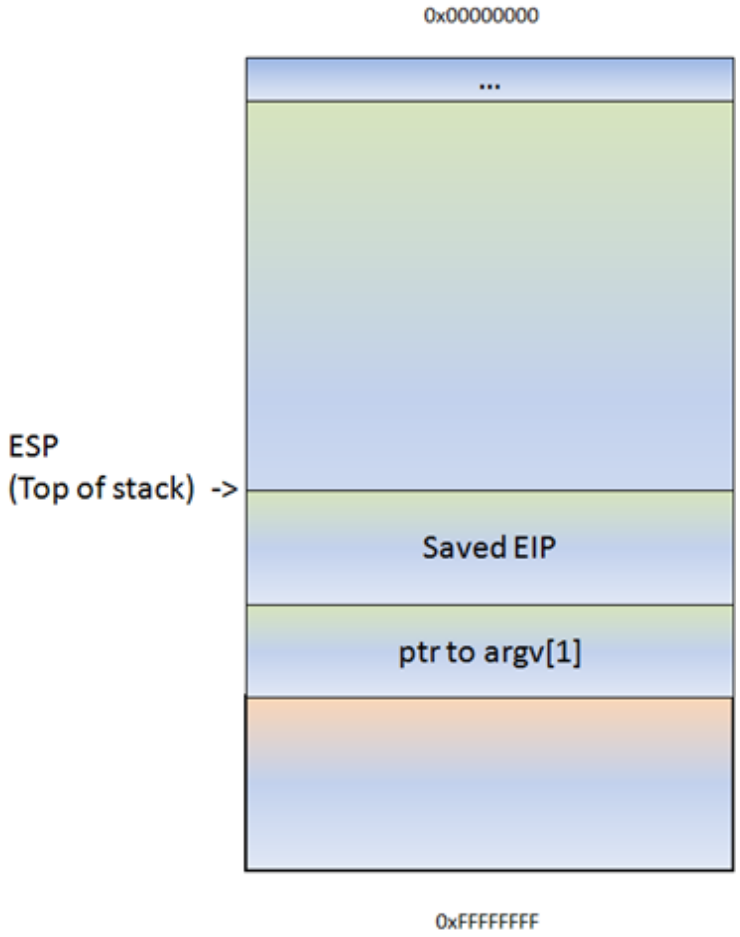
Example – do_something()

```
; after the call for strcpy()
add      esp,8          ; cleanup (8 – two previous
  pushes)
add      esp,88h   ; deallocs the 132 bytes allocated
  on                      the stack for MyVar local
  variable
; standard epilogue
cmp      ebp,esp   ; run-time error checks
call     __RTC_CheckEsp (019113Bh)

pop      edi
mov      esp,ebp
pop      ebp            ; restore registers from the
```

Example – do_something()

- end of function do_something()
  - "unwinds" the stack
  - moves ESP back to the location where saved EIP was
  - issues a RET instruction
    - will pick up the saved EIP pointer from the stack and jump to it
    - thus it goes back to the main function, right after where do_something() was called

Example – do_something()



Example – main

```
; after the call for do_something()
add         esp,4                    ; cleanup (4 – one
  previous push)
xor         eax,eax                  ; make EAX equal 0 as
  return value

; standard epilogue
cmp         ebp,esp                  ; run-time error checks
call        __RTC_CheckEsp (019113Bh)

mov         esp,ebp
pop         ebp                      ; restore registers from
  the stack
```

Standard prologue vs. standard epilogue

Standard prologue	Standard epilogue

push ebp
mov ebp,esp
sub esp,0C0h
push ebx
push esi
push edi
	pop edi
	pop esi
	pop ebx
	add esp,0C0h
	mov esp,ebp
	pop ebp
	ret

Calling conventions

- Who cleans the arguments from the stack?
- Which registers are guaranteed to retain their values after a subroutine call?

- Caller function cleanup
  - cdecl
- Called function cleanup
  - stdcall, fastcall, vectorcall
- Either caller function or called function cleanup
  - thiscall

cdecl

- subroutine arguments are passed on the stack
- returned values:
  - integer values and memory addresses – EAX register
  - floating point values – ST0 x87 register
- Processor registers:
  - EAX, ECX, and EDX are caller-saved
  - all the other registers are callee-saved
- x87 floating point registers:
  - ST0 to ST7 must be empty (popped or freed)
  - ST1 to ST7 must be empty on exiting a function.
  - ST0 must also be empty when not used for returning a value

cdecl

Solution from folder 3_Code Samples, Source_cc.c

```
int __cdecl calee(int, int, int) {
    return 0;
}

int caller(void) {
    int ret;
    ret  =  calee(1, 2, 3);
    return ret;
}
```

cdecl – caller

```
; standard prologue
push    ebp
mov     ebp,esp
push    ecx
mov     dword ptr [ret],0CCCCCCCCh
```

cdecl – caller

```
push     3                  ; push calee arguments on
  the stack
push     2                  (in reverse order)
push     1

call     calee (013B1159h)  ; call calee

add      esp,0Ch            ; cleanup the calee
  stack (12 – 3                    pushes for int)
mov      dword ptr [ret],eax ; save returned value
  from EAX in ret

mov      eax,dword ptr [ret]
```

cdecl – callee

```
push    ebp
mov     ebp,esp

xor     eax,eax         ; make EAX equal 0 as return
  value

pop     ebp
ret
```

stdcall

- the called function cleans up the arguments from the stack
  - this will be done after returning
  - the x86 ret instruction allows an optional 16-bit parameter that specifies the number of stack bytes to release after returning to the caller
- the other aspects are the same as for cdecl

- is the standard calling convention for Microsoft Win32 API

stdcall

Solution from folder 3_Code Samples, Source_cc.c

```
int __stdcall calee(int, int, int) {
    return 0;
}

int caller(void) {
    int ret;
    ret  =  calee(1, 2, 3);
    return ret;
}
```

stdcall – caller

```
;standard prologue

push    3
push    2
push    1

call    calee (010D1104h)    ; call calee

mov     dword ptr [ret],eax   ; save returned value
  from EAX in ret
mov     eax,dword ptr [ret]   ; save return value in
  EAX
```

stdcall – calee

```
; standard prologue
xor     eax,eax     ; make EAX equal 0 as return value
; standard epilogue
ret     0Ch         ; pop 12 bytes from the stack after returning (12 – 3 pushes for int)
```

fastcall

- the cleaning of the stack is done as for stdcall
- passes the first two arguments (evaluated left to right) that fit into ECX and EDX through these registers
- remaining arguments are pushed onto the stack from right to left
- the other aspects are the same as for cdecl

- Microsoft compiler:
  - when compiles for x64 (IA64, AMD64), __fastcall keyword is ignored and uses the 64-bit calling convention instead

fastcall – caller

```
; standard prologue
push    3                   ; the rest are pushed on the
  stack
mov     edx,2
mov     ecx,1                       ; first two arguments
  (left to right) fit
                                into ECX and EDX

call    calee (0CC110Eh)    ; call calee

mov     dword ptr [ret],eax ; save returned value
  from EAX in ret
mov     eax,dword ptr [ret] ; save return value in
  EAX
```

fastcall – calee

```
; standard prologue
sub     esp,8
mov     dword ptr [b],0CCCCCCCCh
mov     dword ptr [a],0CCCCCCCCh
mov     dword ptr [b],edx       ; second argument
  from EDX
mov     dword ptr [a],ecx       ; first argument from
  ECX

xor     eax,eax                 ; make EAX equal 0 as return
  value

; standard epilogue
```

Bibliography

- Olve Maudal, Jon Jagger, Deep C (and C++)

http://www.pvv.org/~oma/DeepC_slides_oct2011.pdf

- Olve Maudal, Insecure coding in C and C++

https://www.slideshare.net/olvemaudal/insecure-coding-in-c-and-c

- eLearn Hacking, Stack Based Overflow Exploit: Basic

https://elearnhacking.wordpress.com/2014/09/21/tutorial-1-stack-based-overflow-

Bibliography

- Wikipedia, x86 calling conventions

https://en.wikipedia.org/wiki/X86_calling_conventions

- x86 Disassembly/Calling Conventions

https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions