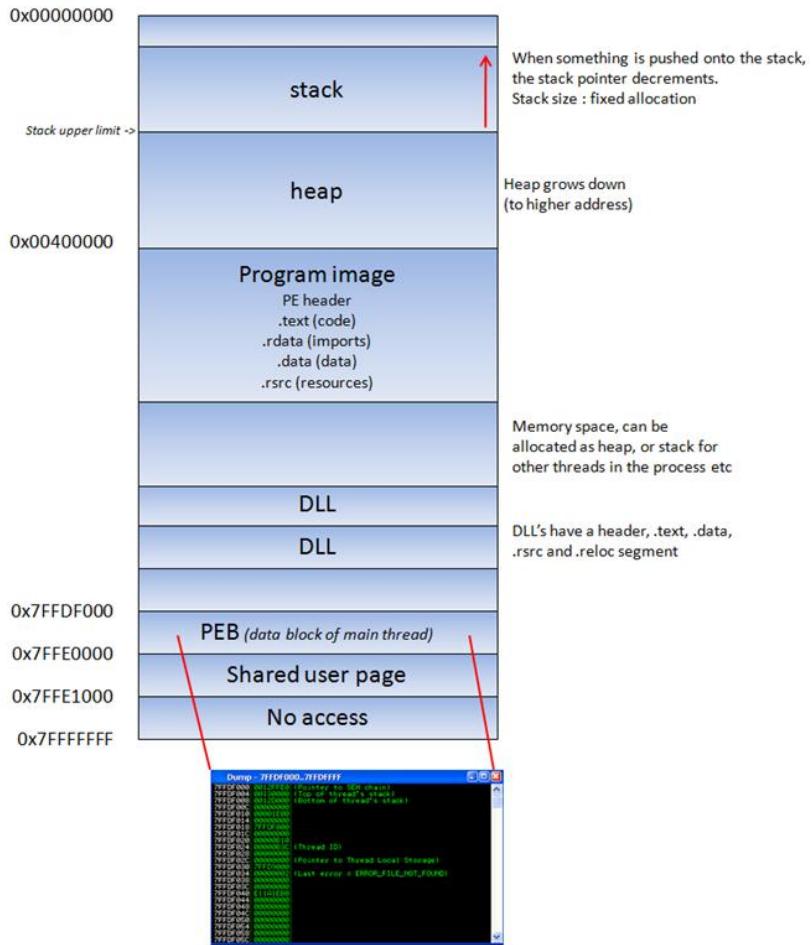




# **Stack Management & Calling Conventions Information Security**

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 overflow

# Visual Studio 2013 – Console application



Microsoft Visual Studio

FILE EDIT VIEW DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Quick Launch (Ctrl+Q)

Mihai Lica Pura

Solution Explorer

New

- Project... Ctrl+Shift+N
- Web Site... Shift+Alt+N
- Team Project...
- File... Ctrl+N
- Project From Existing Code...

Save Selected Items Ctrl+S

Save Selected Items As...

Save All Ctrl+Shift+S

Export Template...

Source Control

Page Setup...

Print... Ctrl+P

Account Settings...

Recent Files

Recent Projects and Solutions

Exit Alt+F4

Output

Show output from:

Error List Output Find Symbol Results

Solution Explorer Team Explorer Class View

Ready

EN 10:45 AM 6/22/2017



# Visual Studio 2013 - Console application

Microsoft Visual Studio  
FILE EDIT VIEW DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP  
Attach... Quick Launch (Ctrl+Q) Mihai Lica Pura

New Project  
Recent .NET Framework 4.5 Sort by: Default  
Installed Win32 Console Application Visual C++  
Templates Visual Basic Visual C# Visual C++ ATL CLR General MFC T... Win32 Project Visual C++  
Visual C++ Win32  
Visual F# SQL Server PowerShell JavaScript Python  
Online Click here to go online and find templates.  
Name: Example  
Location: C:\Users\Mihai Lica Pura\Desktop Browse...  
Solution name: Example Create directory for solution Add to source control  
OK Cancel

Error List Output Find Symbol Results Solution Explorer Team Explorer Class View  
10:46 AM 6/22/2017

The screenshot shows the 'New Project' dialog in Microsoft Visual Studio 2013. The 'Win32 Console Application' template is selected and highlighted with a red circle. The 'Visual C++' category under 'Templates' is also circled in red. The 'Name' field contains 'Example' and the 'Location' field shows the user's desktop path. The 'OK' button is visible at the bottom right of the dialog.

# Visual Studio 2013 - Console application



Microsoft Visual Studio

FILE EDIT VIEW DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Quick Launch (Ctrl+Q)

Mihai Lica Pura

Solution Explorer

Server Explorer Toolbox

Output

Show output from:

Welcome to the Win32 Application Wizard

These are the current project settings:

- Console application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

< Previous Next > Finish Cancel

Error List Output Find Symbol Results

Solution Explorer Team Explorer Class View

Creating project 'Example'...

Windows icon Internet Explorer icon File icon Mozilla Firefox icon Visual Studio icon Task Manager icon Paint icon

EN 10:49 AM 6/22/2017

# Visual Studio 2013 - Console application



Microsoft Visual Studio

FILE EDIT VIEW DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Quick Launch (Ctrl+Q)

Mihai Lica Pura

Solution Explorer

Server Explorer Toolbox

Win32 Application Wizard - Example

Application Settings

Overview Application Settings

Application type:

- Windows application
- Console application
- DLL
- Static library

Add common header files for:

- ATL
- MFC

Additional options:

- Empty project
- Export symbols
- Precompiled header
- Security Development Lifecycle (SDL) checks

Output

Show output from:

< Previous Next > **Finish** Cancel

Error List Output Find Symbol Results

Creating project 'Example'...

Solution Explorer Team Explorer Class View

EN 10:50 AM 6/22/2017

16



# Visual Studio 2013 - Console application

Example - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Local Windows Debugger Debug Win32

Solution Explorer

Search Solution Explorer (Ctrl+.)

Solution 'Example' (1 project)

Example

- External Dependencies
- Header Files
- Resource Files
- Source Files

Output

Show output from:

Error List Output Find Symbol Results

This item does not support previewing

New Item... Ctrl+Shift+A

Existing Item... Shift+Alt+A

New Filter

Class...

Resource...

Add

Class Wizard... Ctrl+Shift+X

Scope to This

New Solution Explorer View

Cut Ctrl+X

Copy Ctrl+C

Paste Ctrl+V

Delete Del

Rename

Properties Alt+Enter

Advanced

Parse Files True

SCC Files True

General

(Name) Source Files

Filter cpp;c;c++;def;odl;idl;hpj

Unique Identifier {4FC737F1-C7A5-4376-A06

EN 10:52 AM 6/22/2017

The screenshot shows the Microsoft Visual Studio 2013 IDE. The title bar reads "Example - Microsoft Visual Studio". The menu bar includes FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, ANALYZE, WINDOW, and HELP. The toolbar has icons for various tools like Server Explorer, Toolbox, and Solution Explorer. The status bar at the bottom shows "EN 10:52 AM 6/22/2017".  
The main window contains several panes:

- Solution Explorer**: Shows a single project named "Example" with its files: External Dependencies, Header Files, Resource Files, and Source Files (which is currently selected).
- Properties**: Shows settings for the "Source Files" filter, including "Parse Files" and "SCC Files" set to "True". It also lists "General" settings with "(Name)" set to "Source Files", "Filter" set to "cpp;c;c++;def;odl;idl;hpj", and a unique identifier.
- Output**: A pane for displaying build output, currently empty.
- Error List**: A pane for displaying build errors, currently empty.
- Find Symbol Results**: A pane for displaying search results, currently empty.

A context menu is open over the "Source Files" node in the Solution Explorer, displaying options like "New Item...", "Existing Item...", "New Filter", "Class...", and "Resource...". Other menu items visible include "Add", "Class Wizard...", "Scope to This", "New Solution Explorer View", "Cut", "Copy", "Paste", "Delete", "Rename", and "Properties".

# Visual Studio 2013 - Console application



Example - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Local Windows Debugger Debug Win32

Add New Item - Example

Installed

Visual C++

- UI
- Code
- HLSL
- Data
- Resource
- Web
- Utility
- Property Sheets
- Test
- Graphics
- PowerShell

Sort by Default

C++ File (.cpp)

Type: Visual C++  
Creates a file containing C++ source code

Header File (.h)

Online

Name: Source.cpp

Location: C:\Users\Mihai Lica Pura\Desktop\Example\Example

Add Cancel

Output

Show output from:

Click here to go online and find templates.

Error List Output Find Symbol Results

EN 10:52 AM 6/22/2017

Server Explorer Toolbox

File Explorer (Ctrl+;) Example (1 project)  
File  
External Dependencies  
Header Files  
Resource Files  
Source Files

Team Explorer Class View

Filter Properties

True True

Source Files  
cpp;c;c++;def;odl;idl;hpj  
Identifier {4FC737F1-C7A5-4376-A0E

# Visual Studio 2013 - Console application



The screenshot shows the Microsoft Visual Studio 2013 IDE interface. The main window displays the code for a C program named `Source.c`. The code includes an `#include <string.h>` directive and two functions: `do_something` and `main`. The `do_something` function copies the contents of `Buffer` into `MyVar`. The `main` function calls `do_something` with the first command-line argument. The Solution Explorer on the right shows a single project named `Example` containing a `Source Files` folder with the `Source.c` file. The Properties window is visible at the bottom right. The status bar at the bottom shows various icons and the date/time: 10:53 AM, 6/22/2017.

```
#include <string.h>

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

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

# Visual Studio 2013 - Compiler configuration

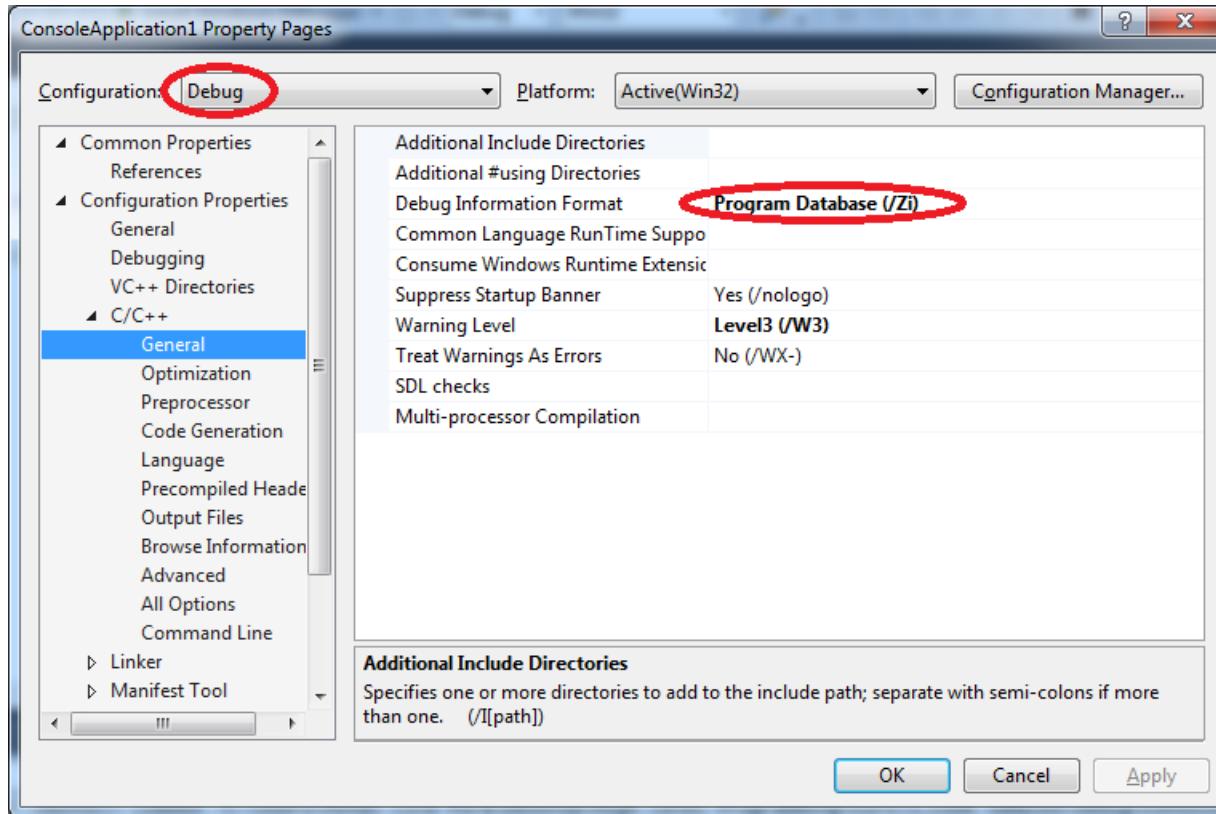


A screenshot of the Microsoft Visual Studio 2013 IDE. The main window shows a code editor with the file "Source.c" open, containing the following C code:

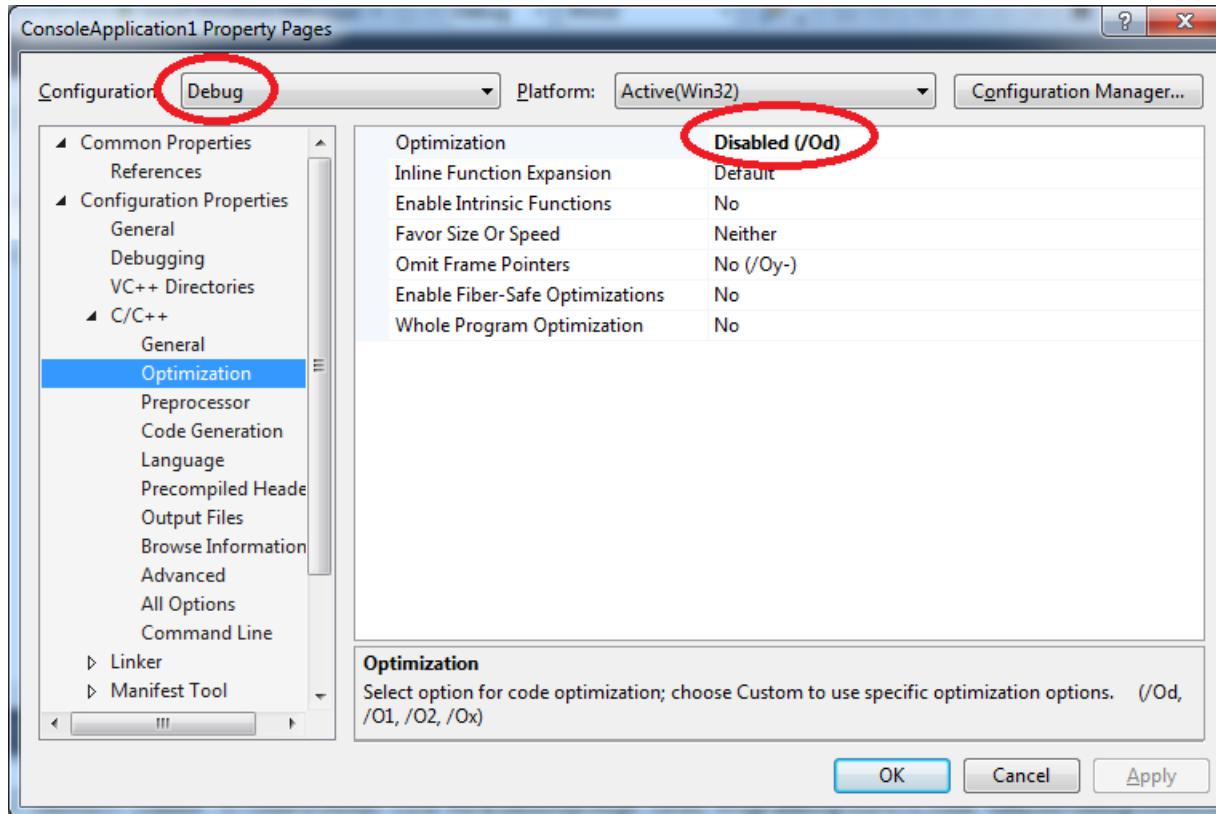
```
#include <string.h>
void do_something(char* Buffer)
{
    char MyVar[128];
    strcpy(MyVar, Buffer);
}
void main(int argc, char*
{
    do_something(argv[1]);
}
```

The "DEBUG" menu is open, highlighting the "Example Properties..." option. The "Solution Explorer" pane shows a single project named "Example" with a source file "Source.c". The "Properties" pane is visible at the bottom right. The status bar at the bottom right shows the date and time: "6/22/2017 10:54 AM".

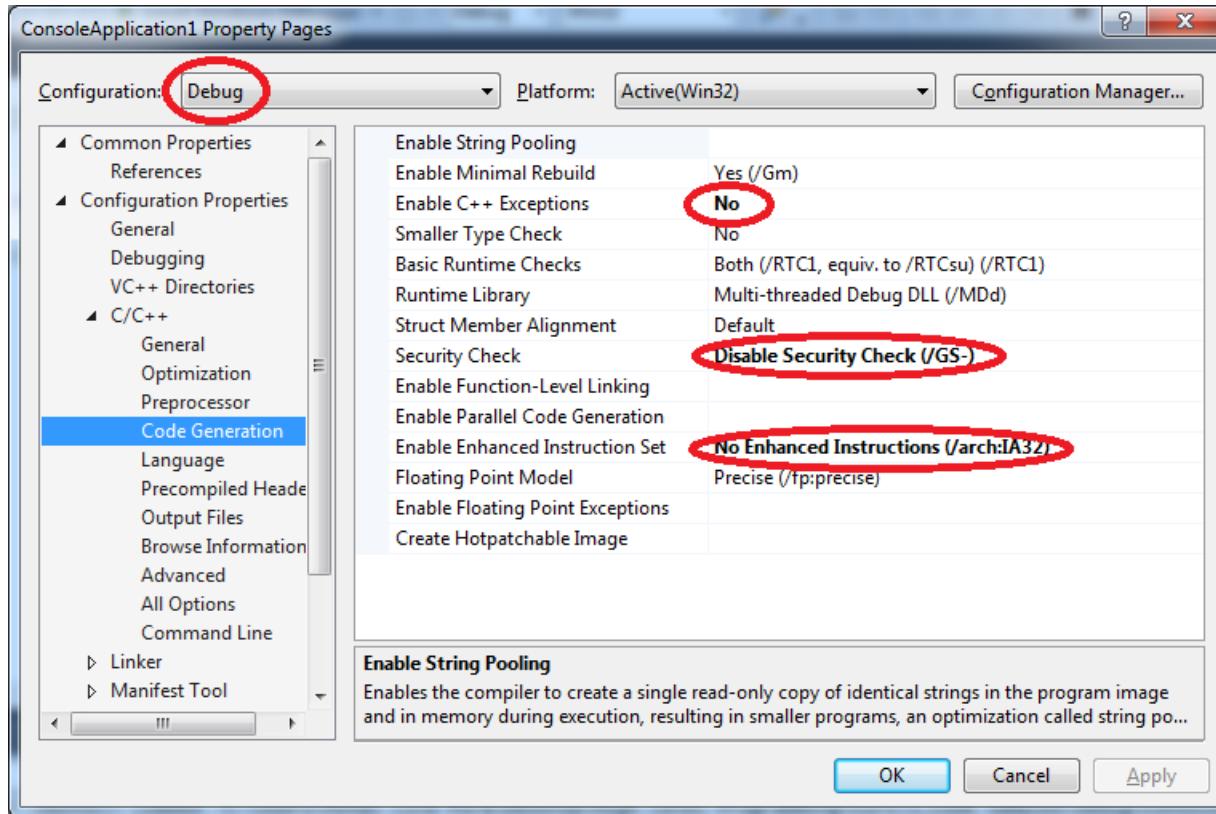
# Visual Studio 2013 – Compiler configuration



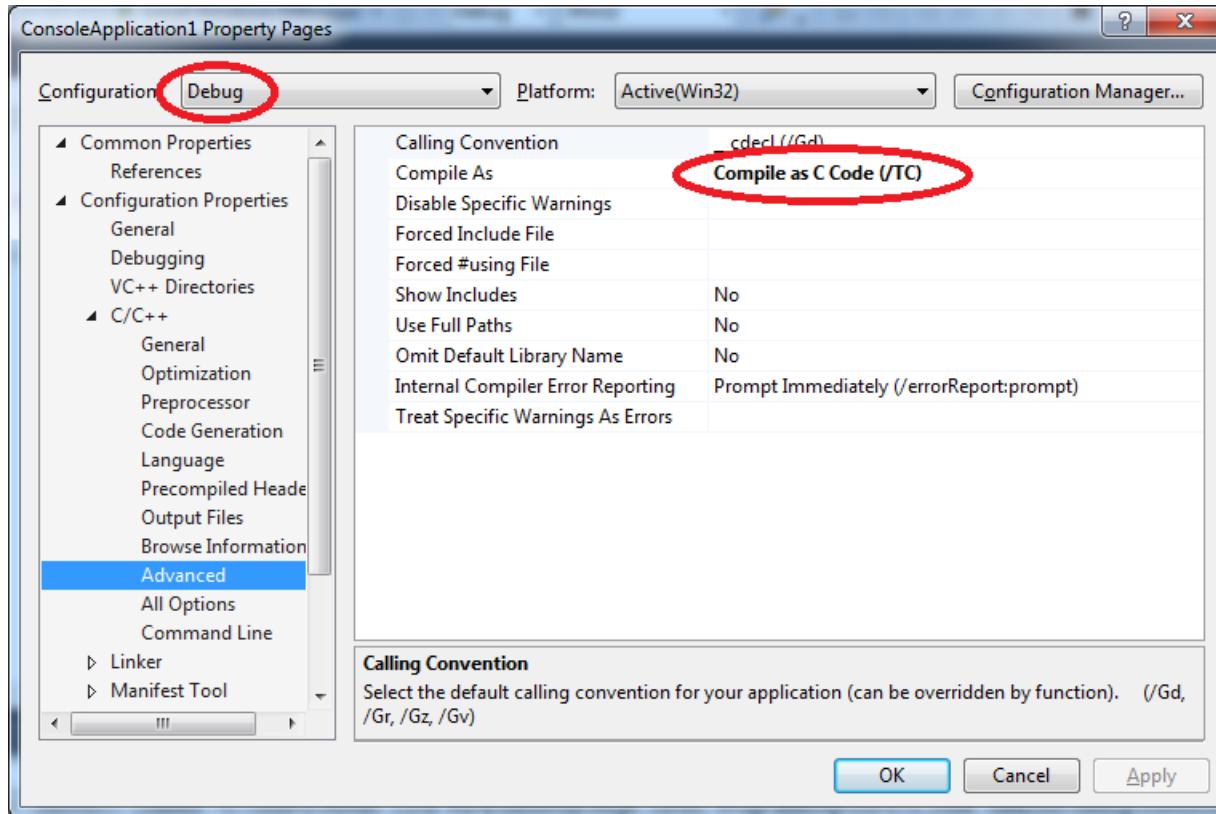
# Visual Studio 2013 – Compiler configuration



# Visual Studio 2013 – Compiler configuration

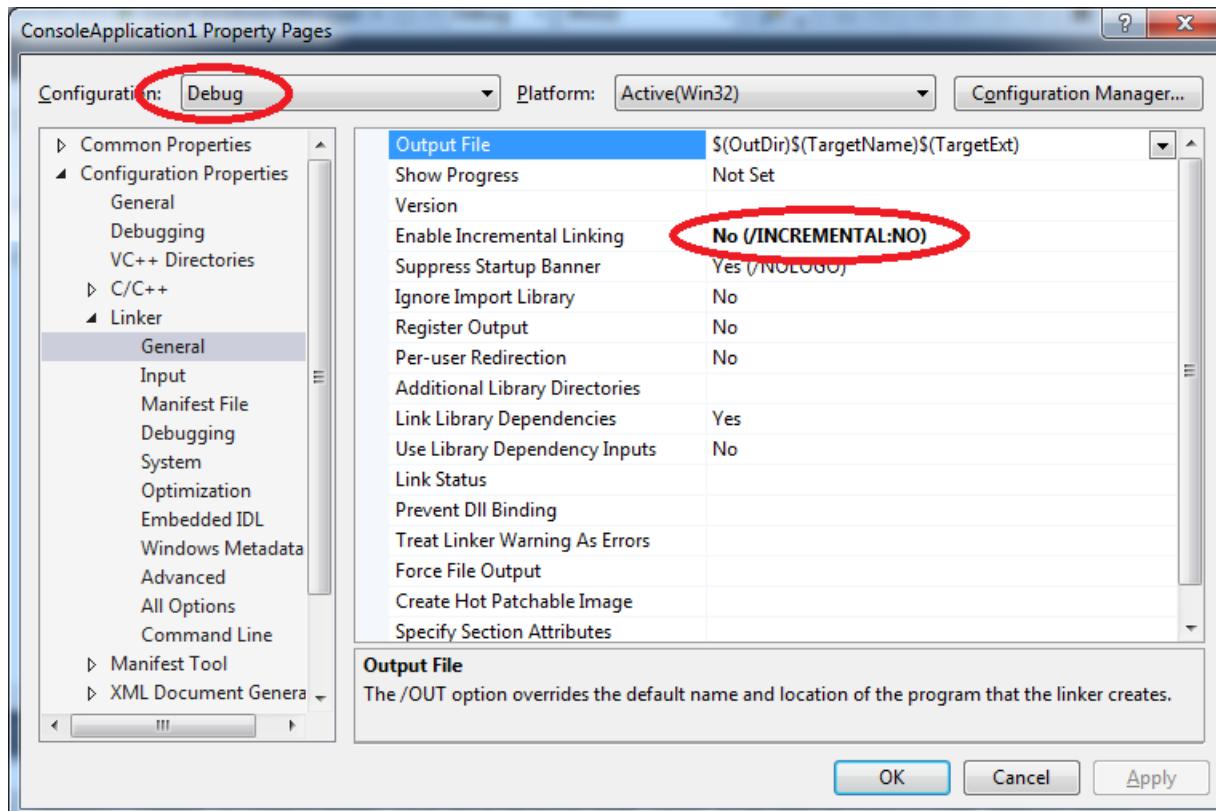


# Visual Studio 2013 – Compiler configuration





# Visual Studio 2013 – Linker configuration





# Visual Studio 2013 - Build application

The screenshot shows the Microsoft Visual Studio 2013 interface. The 'BUILD' menu is open, with 'Rebuild Solution' highlighted. The Solution Explorer on the right shows a project named 'ConsoleApplication1' containing 'External Dependencies', 'Header Files', 'Resource Files', and 'Source Files' with files 'Source.c' and 'Source\_cc.cpp'. The Properties window is also visible.



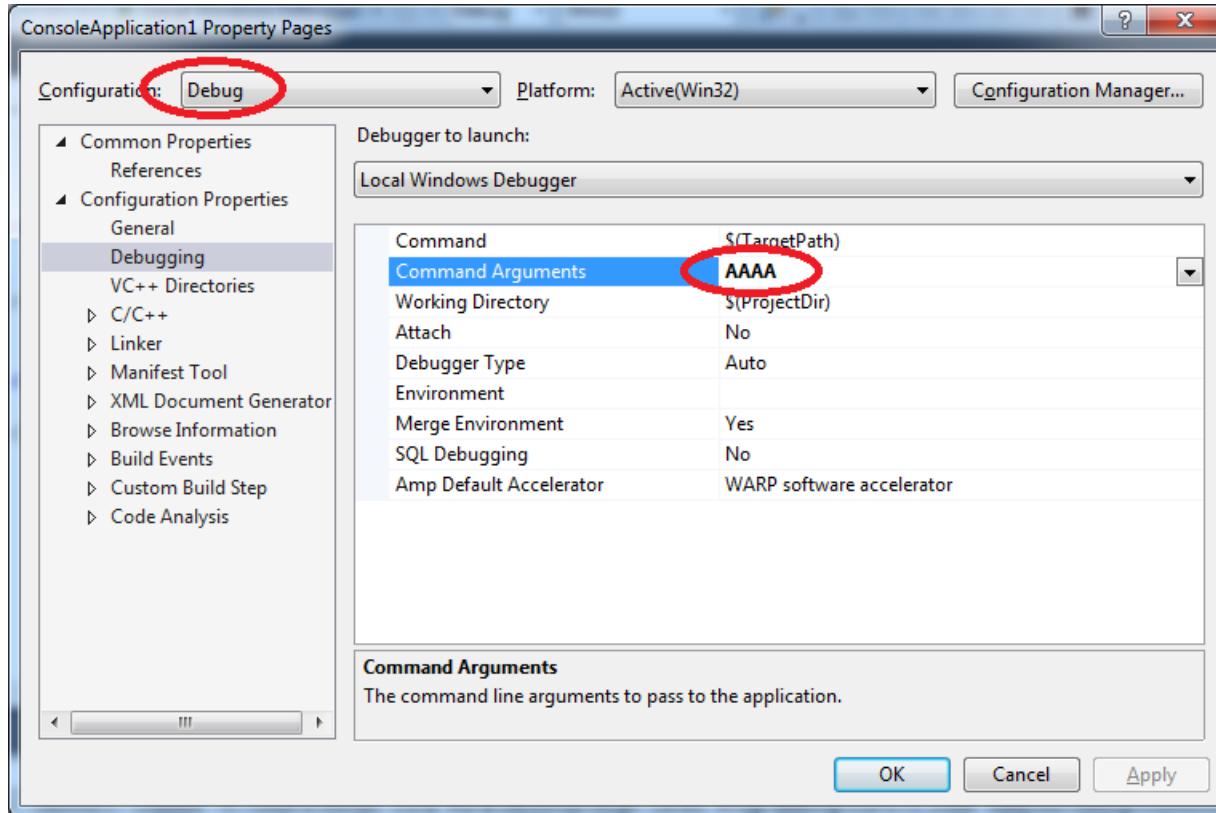
# Visual Studio 2013 – Place breakpoints

The screenshot shows the Microsoft Visual Studio 2013 interface with the following details:

- Title Bar:** Example - Microsoft Visual Studio
- Menu Bar:** FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
- Toolbars:** Standard, Debug, Win32
- Code Editor:** Shows `Source.c` with two red circles highlighting breakpoints set on the first line of `do_something` and the second line of `main`.
- Solution Explorer:** Shows the project 'Example' with files: External Dependencies, Header Files, Resource Files, Source Files, and Source.c.
- Properties Window:** Shows the project properties for 'Example' under 'Misc' tab.
- Output Window:** Shows the command-line output of a rebuild:

```
1>----- Rebuild All started: Project: Example, Configuration: Debug Win32 -----
1> Source.c
1>c:\users\mihai lica pura\Desktop\example\example\source.c(6): warning C4996: 'strcpy': This function or variable may be unsafe. Consider using
1>     c:\program files (\$)\microsoft visual studio 12.0\vc\include\string.h(112) : see declaration of 'strcpy'
1> Example.vcxproj -> C:\Users\Mihai Lica Pură\Desktop\Example\Debug\Example.exe
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped =====
```
- Status Bar:** Shows 'Rebuild All succeeded' and system icons.

# Visual Studio 2013 – Command line arguments





# Visual Studio 2013 – Start debugging

Example - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Local Windows Debugger (Debug Win32)

Source.c (Global Scope) main(int argc, char \* argv[])

Server Explorer Toolbox

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]);
}
```

100 %

Output

Show output from: Debug

'Example.exe' (Win32): Loaded 'C:\Users\Miha Lica Pura\Desktop\Example\Debug\Example.exe'. Symbols loaded.  
'Example.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'. Cannot find or open the PDB file.  
'Example.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'. Cannot find or open the PDB file.  
'Example.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'. Cannot find or open the PDB file.  
'Example.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcr120d.dll'. Cannot find or open the PDB file.  
The program '[804] Example.exe' has exited with code 0 (0x0).

Error List Output Find Symbol Results

Solution Explorer

Solution 'Example' (1 project)

- Example
  - External Dependencies
  - Header Files
  - Resource Files
- Source Files
  - Source.c

Solution Explorer Team Explorer Class View

Properties

main VCCodeFunction

C++ (Name) main

File	c:\users\mihai lica pur
FullName	main
IsDefault	False
IsDelete	False
IsFinal	False
IsInjected	False
IsInline	False

Ready Ln 11 Col 1 Ch 1 INS EN 11:06 AM 6/22/2017



# Visual Studio 2013 – Show disassembly

The screenshot shows the Microsoft Visual Studio 2013 interface during a debugging session. The main window displays the `Source.c` file:#include <string.h>

void do\_something(char\* Buffer)
{
 char MyVar[128];
 strcpy(MyVar, Buffer);
}

void main(int argc, char\* argv[])
{
 do\_something(argv[1]);
}A context menu is open at the cursor position, specifically over the line `do_something(argv[1]);`. The menu includes options like Insert Snippet..., Surround With..., Peek Definition, Go To Definition, Go To Declaration, Find All References, View Call Hierarchy, Toggle Header / Code File, Breakpoint, Add Watch, Add Parallel Watch, QuickWatch..., Pin To Source, Show Next Statement, Step Into Specific, Run To Cursor, Run Flagged Threads To Cursor, Set Next Statement, Go To Disassembly, Cut, Copy, Paste, and Outlining. The "Go To Disassembly" option is highlighted and circled in red.



# Visual Studio 2013 - Watch

The screenshot shows the Microsoft Visual Studio 2013 interface during a debug session. The title bar indicates "Example (Debugging) - Microsoft Visual Studio". The menu bar includes FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, ANALYZE, WINDOW, and HELP. The toolbar has standard icons for file operations and debugging. The status bar at the bottom shows "Ready", "EN", "11:08 AM", and the date "6/22/2017".

The main window displays the Disassembly view for the "main" function. The assembly code is shown with addresses from 013C1470 to 013C149D. A red circle highlights the instruction at address 013C148C, which is a "rep stos" instruction. Below the assembly view is the Autos window, which lists registers EAX, ECX, EDI, and ES with their current values: CCCCCCCC, 00000000, 0037F7F4, and 002B respectively. Another red circle highlights this row in the Autos window. To the right of the assembly view is the Call Stack window, which shows the current call stack frame: "Example.exe!main(int argc, char \*\* argv) Line 11 [External Code]".

The Solution Explorer on the right shows the project structure for "Solution 'Example' (1 project)". The "Example" folder contains "External Dependencies", "Header Files", "Resource Files", and "Source Files", with "Source.c" being the selected file.



# Visual Studio 2013 – Show memory

The screenshot shows the Microsoft Visual Studio 2013 interface during a debugging session. The title bar reads "Example (Debugging) - Microsoft Visual Studio". The menu bar is visible with options like FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, ANALYZE, WINDOW, and HELP. The "DEBUG" tab is selected.

The main window displays assembly code for "Source.c" at address "main(int, char \*\*)". Below the assembly window are three tabs: "Autos", "Locals", and "Watch1". The "Autos" tab shows register values:

Name	Value
EAX	CCCCCC
ECX	00000000
EDI	0037F7F4
ES	002B

The "DEBUG" menu is open, showing various debugging tools. The "Windows" submenu is currently selected, and the "Memory" option is highlighted. A submenu under "Memory" lists four memory windows: Memory 1, Memory 2, Memory 3, and Memory 4. The "Call Stack" window is also visible, showing the current call stack frame.

The Solution Explorer on the right shows a project named "Example" with files "Source.c" and "Source.h". The status bar at the bottom shows the time as "11:11 AM" and the date as "6/22/2017".



# Visual Studio 2013 – Show memory

Example (Debugging) - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Process: [6844] Example.exe Lifecycle Events Thread: [6816] Main Thread Stack Frame: main

Memory 1

Address: 0x00000000

Disassembly

Source.c

Address: main(int, char \*\*)

Viewing Options

```
013C147A push    esi
013C147B push    edi
013C147C lea     edi,[ebp-0Ch]
013C1482 mov     ecx,30h
013C1487 mov     eax,0CCCCCCCCh
013C148C rep stos dword ptr es:[edi]
    do_something(argv[1]);
013C148E mov     eax,4
013C1493 shr     eax,0
```

Autos

Name	Value	Type
EAX	CCCCCC	
ECX	00000000	
EDI	0037F7F4	
ES	002B	

Call Stack

Name	Lang
Example.exe!main(int argc, char ** argv) Line 11	C
[External Code]	
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll]	

Call Stack Breakpoints Command Window Immediate Window Output Error List

Solut... Team E... Propert...

Ready

EN 11:12 AM 6/22/2017

# Visual Studio 2013 – Watching the stack



Screenshot of Microsoft Visual Studio 2013 during debugging mode, showing the stack memory dump in the Memory window.

The **Memory** window displays memory starting at address **esp** (0x0037F728). The data is shown in a 4-byte Integer format, with values appearing as **CCCCCCC** (hex) and **ffff** (dec).

The **Viewing Options** context menu is open, with **4-byte Integer** selected. Other options include:

- No Data
- 1-byte Integer
- 2-byte Integer
- 4-byte Integer** (selected)
- 8-byte Integer
- 32-bit Floating Point
- 64-bit Floating Point
- Hexadecimal Display
- Signed Display
- Unsigned Display
- Big Endian
- No Text
- ANSI Text** (selected)
- Unicode Text

The **Autos** window shows local variable values:

Name	Value
EAX	CCCCCC
ECX	00000000
EDI	0037F7F4
ES	002B

The status bar at the bottom indicates the current time as **11:20 AM** on **6/22/2017**.



# Visual Studio 2013 – Show registers

The screenshot shows the Microsoft Visual Studio 2013 interface during a debugging session. The title bar reads "Example (Debugging) - Microsoft Visual Studio". The menu bar is visible with options FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, ANALYZE, WINDOW, and HELP. The DEBUG menu is open, highlighting the "Registers" option under the "Windows" submenu. The "Registers" option is highlighted with a yellow selection bar. The main workspace displays several windows: a "Memory" window showing memory dump data; a "Disassembly" window showing assembly code for the "main" function; an "Autos" window showing variable values for EAX, ECX, EDI, and ES; and a "Call Stack" window showing the current call stack frame. The Solution Explorer window on the right shows a project named "Example" with files like "Source.c". The status bar at the bottom indicates "Ready", shows system icons, and displays the date and time as "11:13 AM 6/22/2017".



# Visual Studio 2013 – Show registers

The screenshot shows the Microsoft Visual Studio 2013 interface during a debug session. The title bar reads "Example (Debugging) - Microsoft Visual Studio". The menu bar includes FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, ANALYZE, WINDOW, and HELP. The toolbar has various icons for debugging and navigation. The status bar at the bottom right shows "EN", "11:13 AM", "6/22/2017", and a battery icon.

The main window displays the Registers pane, which lists CPU register values. A red circle highlights the value of EAX, which is shown as "CCCCCCC". Other registers listed include EBX, ECX, EDX, ESI, EDI, EIP, ESP, EBP, and EFL. Below the Registers pane is the Disassembly pane, showing assembly code for the main function. The Solution Explorer pane on the right shows a project named "Example" with files like "Source.c".

Name	Type	Value
EAX		CCCCCCC
ECX		00000000
EDI		0037F7F4
ES		002B

Call Stack pane:

Name	Lang
Example.exe!main(int argc, char ** argv) Line 11	C
[External Code]	
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll]	

Toolbars and panes visible include: Registers, Memory1, Disassembly, Autos, Locals, Watch1, Call Stack, Breakpoints, Command Window, Immediate Window, Output, Error List, Solution Explorer, Team E..., Properties, and Ready.



# 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()*

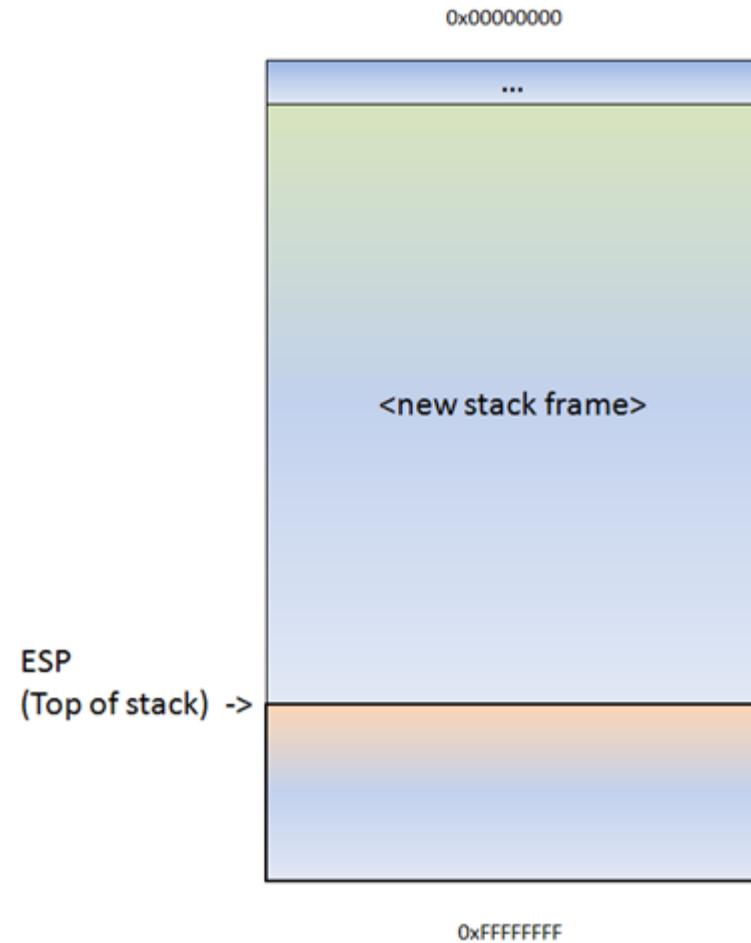
call \_do\_something (019116Dh)



# 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()

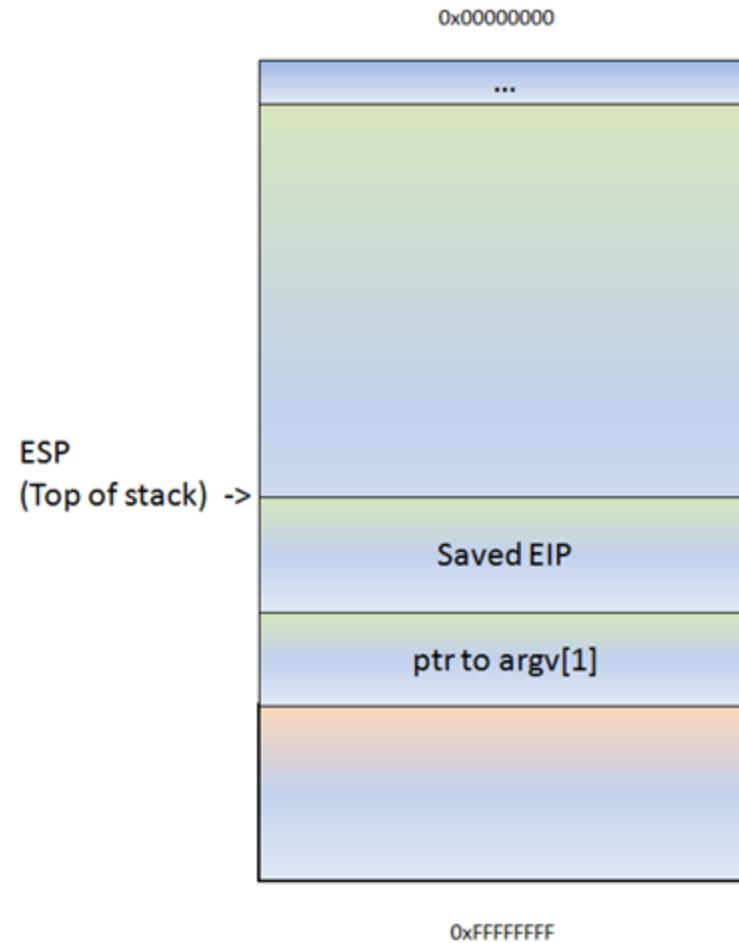




# Example - do\_something()

- ▶ 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,0CCCCCCCCCh
rep stos dword ptr es:[edi]
; writes the value in EAX (0CCCCCCCCCh)
; 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()

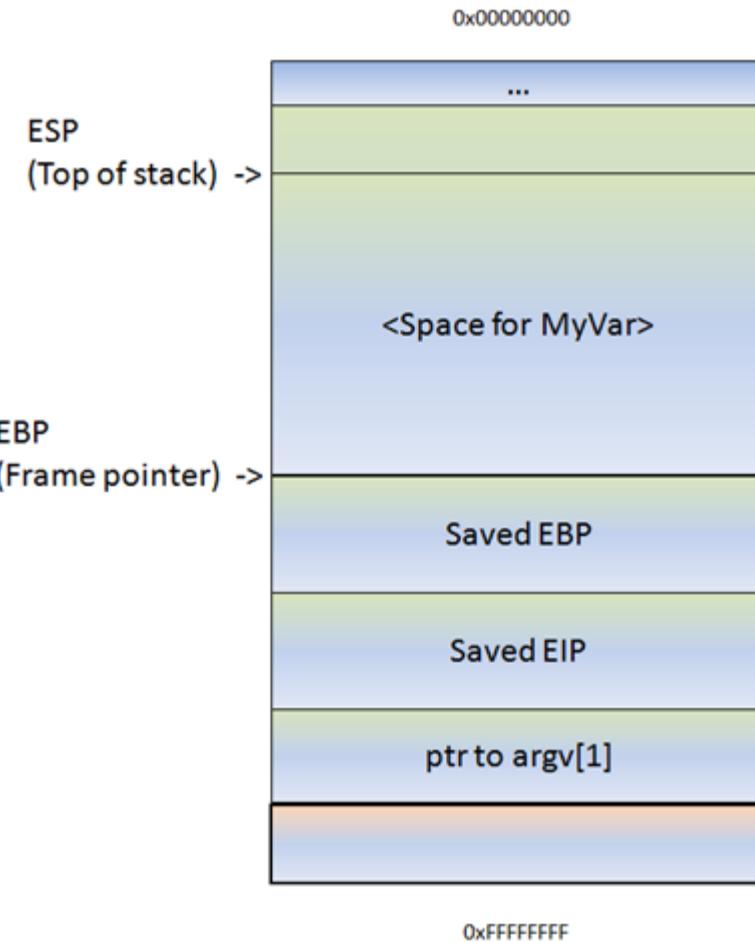
- ▶ the prolog of do\_something() executes
  - after the push of EBP, ESP is put in EBP
  - both ESP and EBP point at the top of the current stack
  - from that point on, the stack will be referenced by ESP (top of the stack at any time) and EBP (the base pointer of the current stack).
  - this way, 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*

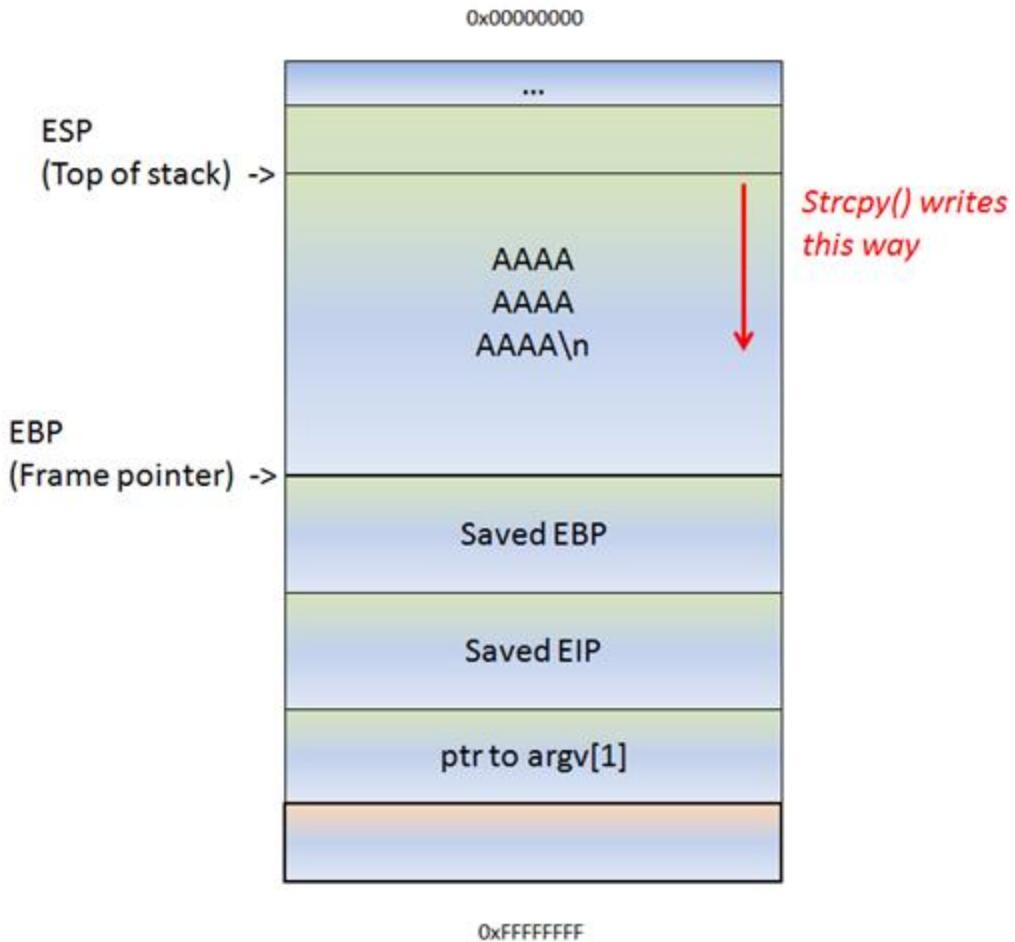
call \_strcpy (01910AAh)



# 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

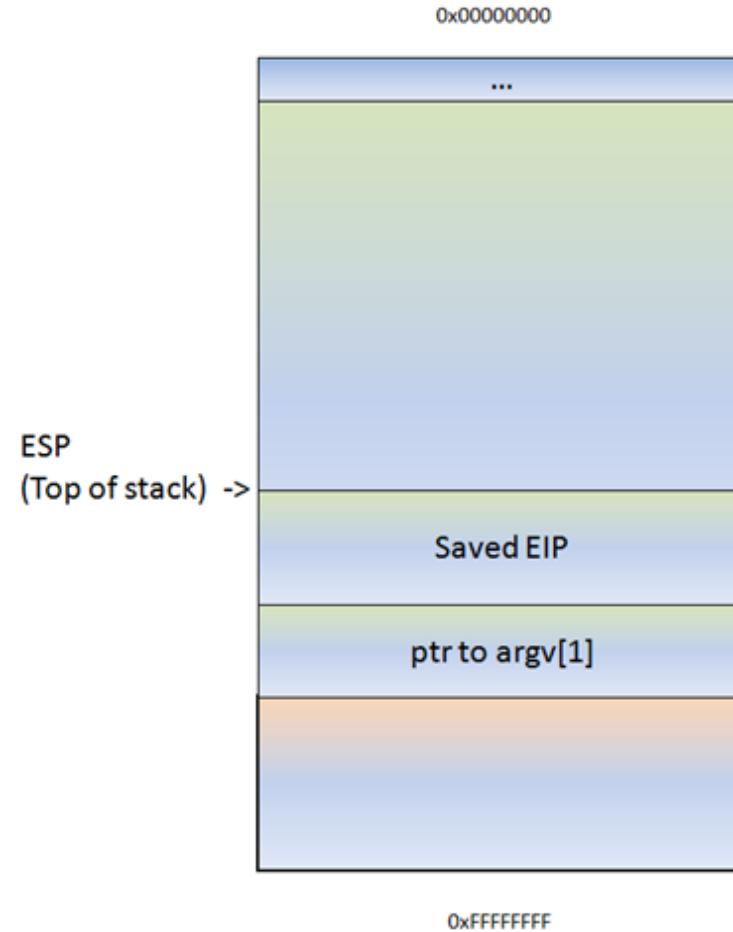
pop ebp ; restore registers from the



# 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	pop	edi
mov	ebp,esp	pop	esi
sub	esp,0C0h	pop	ebx
push	ebx	add	esp,0C0h
push	esi	mov	esp,ebp
push	edi	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 callee
```

```
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 callee

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],0CCCCCCCCCh
mov    dword ptr [a],0CCCCCCCCCh
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](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](https://en.wikipedia.org/wiki/X86_calling_conventions)

- ▶ x86 Disassembly/Calling Conventions

[https://en.wikibooks.org/wiki/X86\\_Disassembly/Calling\\_Conventions](https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions)