Weifan Lin

Csc342 Section G

02/17/2015

Lecture Summary

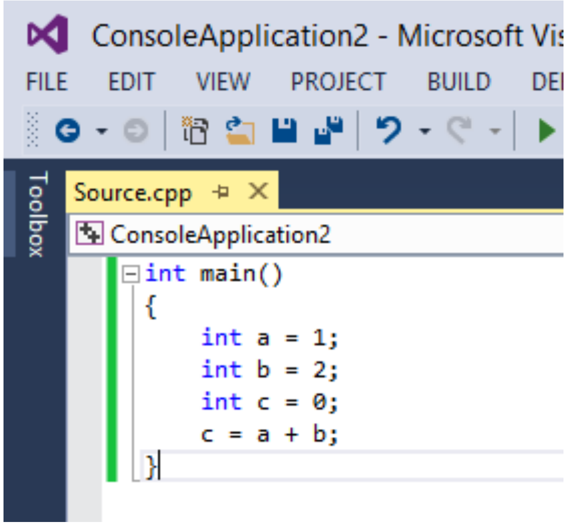
Title:

Debugging

Objective:

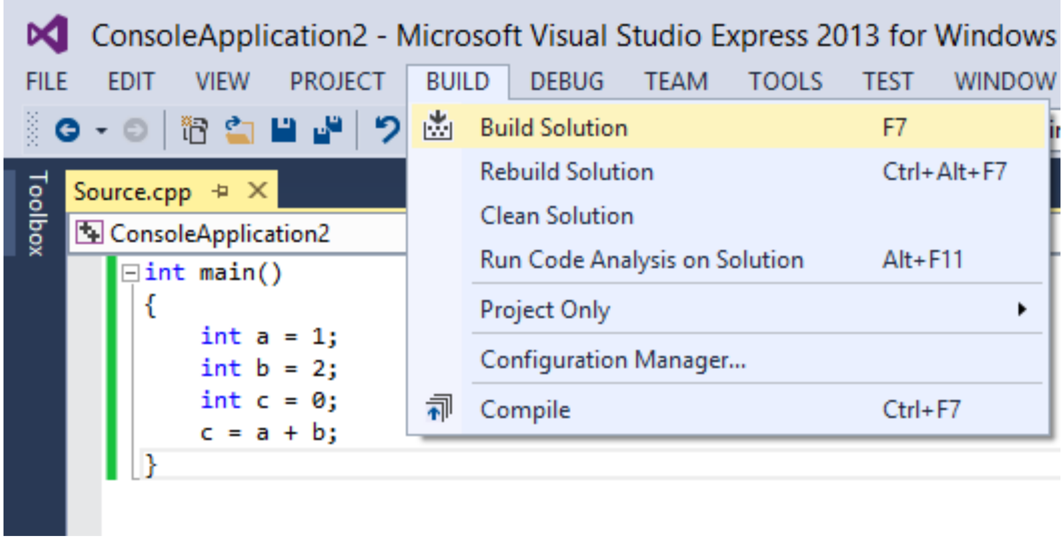
The goal of this lecture was to understand the memory address in the computer and how to allocate it through performance of debugging a piece of a program code.

In this performance, we will be using debugging tool in Microsoft Visual Studio. First we create a simple program in C++.



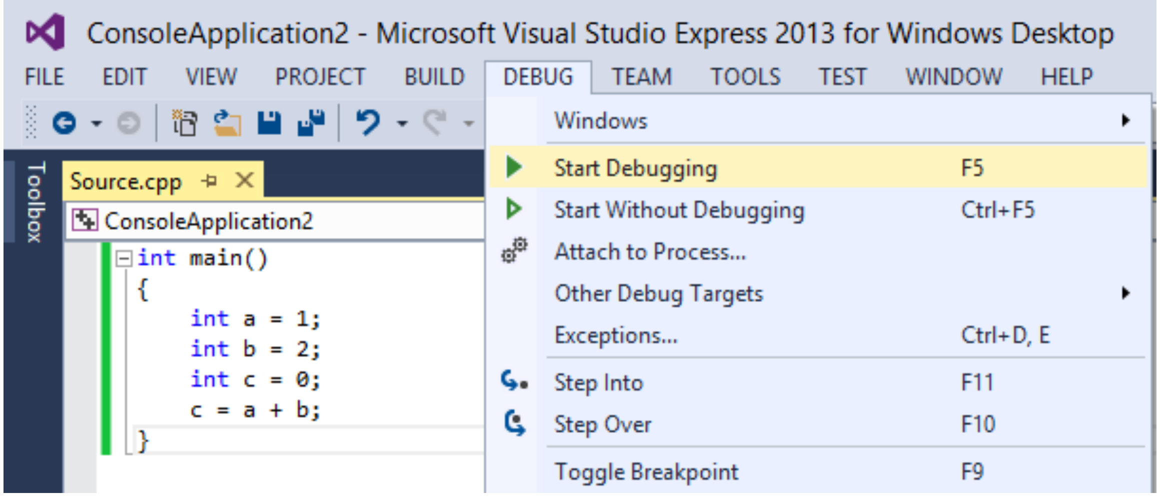
```
int main()
{
    int a = 1;
    int b = 2;
    int c = 0;
    c = a + b;
}
```

And build it as we click on “Build Solution” under BUILD menu.



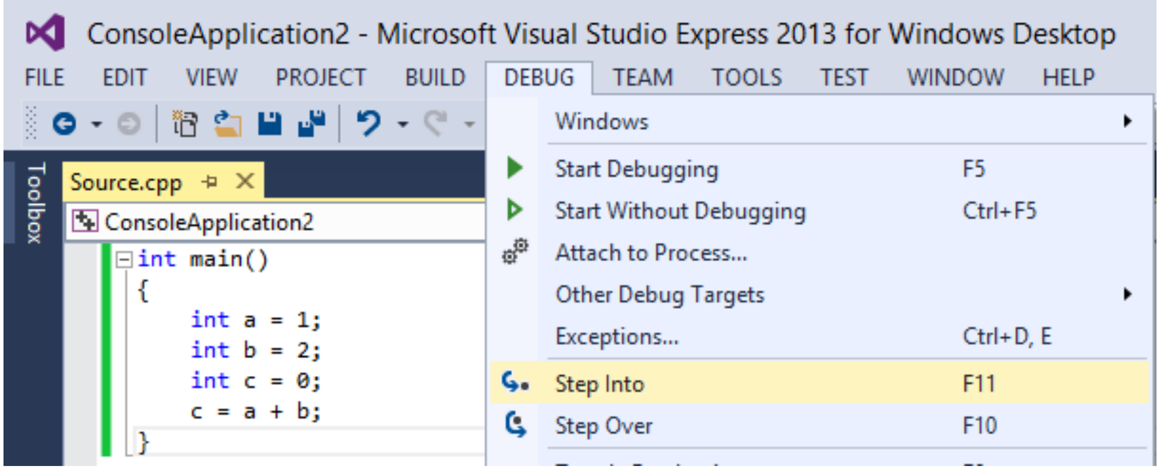
```
int main()
{
    int a = 1;
    int b = 2;
    int c = 0;
    c = a + b;
}
```

Then we go to DEBUG menu and start debugging.



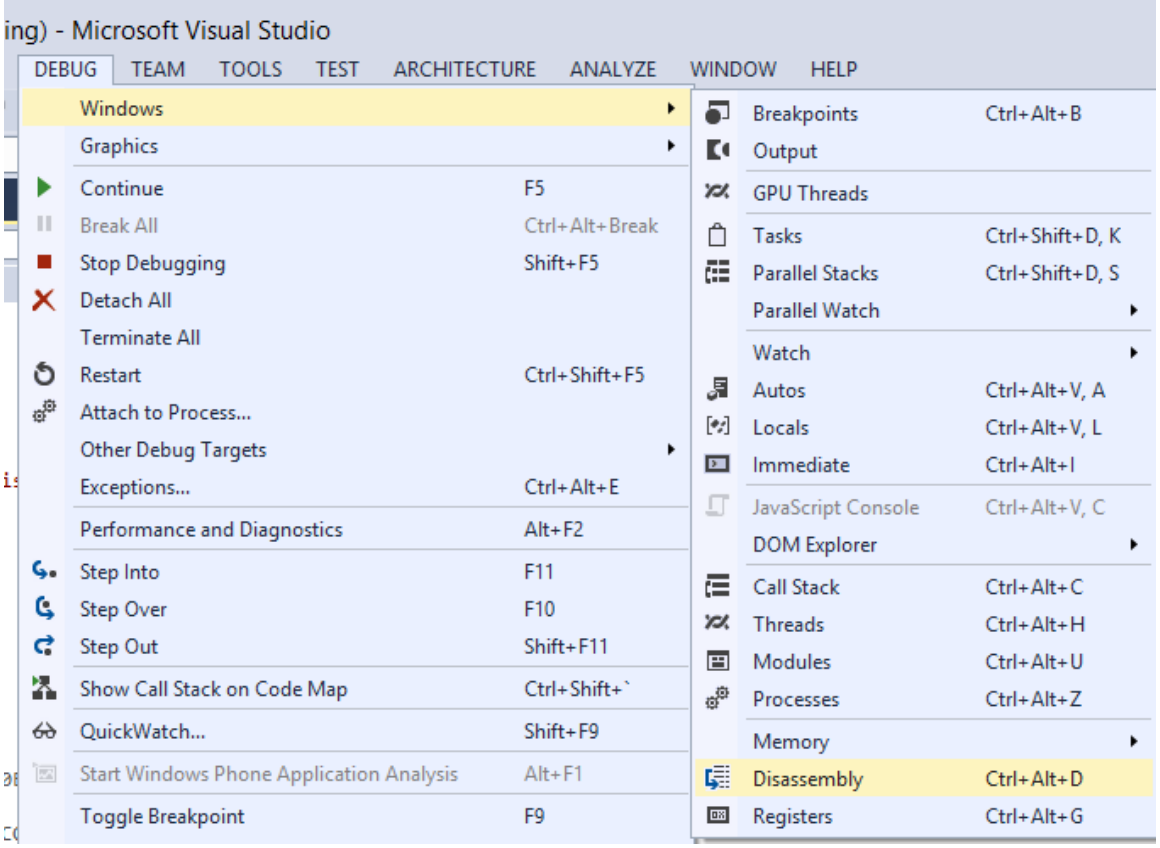
```
int main()
{
    int a = 1;
    int b = 2;
    int c = 0;
    c = a + b;
}
```

After finish debugging, click “Step Into”, this allows us to execute code one statement at a time.



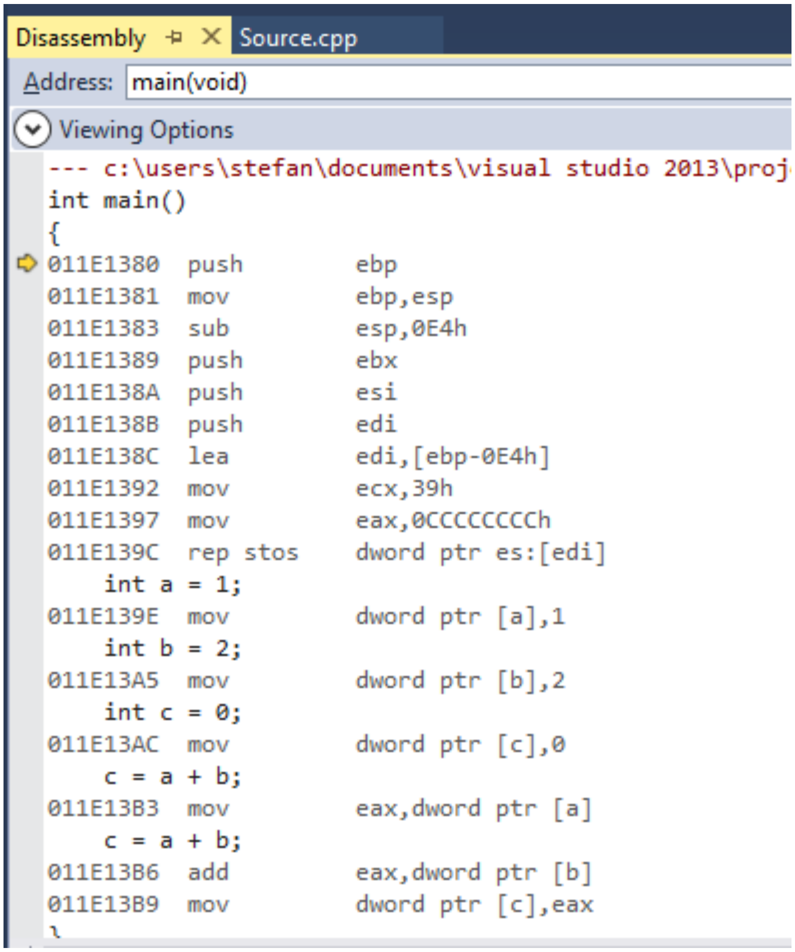
```
int main()
{
    int a = 1;
    int b = 2;
    int c = 0;
    c = a + b;
}
```

Now we can go to “Windows” under DEBUG menu to view Disassembly and Registers.



The Disassembly window shows the assembly code and memory address where each instruction is located. The Registers window displays register contents and change of register values as our code executes.

Disassembly window:



```
--- c:\users\stefan\documents\visual studio 2013\proj
int main()
{
011E1380  push        ebp
011E1381  mov         ebp,esp
011E1383  sub         esp,0E4h
011E1389  push        ebx
011E138A  push        esi
011E138B  push        edi
011E138C  lea         edi,[ebp-0E4h]
011E1392  mov         ecx,39h
011E1397  mov         eax,0CCCCCCCCh
011E139C  rep stos    dword ptr es:[edi]
    int a = 1;
011E139E  mov         dword ptr [a],1
    int b = 2;
011E13A5  mov         dword ptr [b],2
    int c = 0;
011E13AC  mov         dword ptr [c],0
    c = a + b;
011E13B3  mov         eax,dword ptr [a]
    c = a + b;
011E13B6  add         eax,dword ptr [b]
011E13B9  mov         dword ptr [c],eax
}
```

As we can see there is a yellow arrow points to “011E1380   push   ebp”, this indicates which instruction that we are excuted. We can press F10 to step over to next instruction, and the register values will change as well.

“011E1380 push ebp”

“012F52A0” is a address in hexadecimal refer to an instruction, in this instruction “push” means to save the value of current register, and “ebp” is base pointer register.

“011E1381 mov ebp,esp”

esp is stack pointer, and this instruction copys register from ebp to esp and ebp now points to the top of the stack.

“011E1383 sub esp,0E4h”

In this instruction is to allocate space for local variables. 0E4h is hexadecimal has value of 228 in decimal, sub means to subtract 228 bytes for local variables.

“011E1389 push ebx”

“011E138A push esi”

“011E138B push edi”

These three instructions are to save processor registers used for temporaries. ebx is base pointer, esi is source index register, and edi is destination index register.

“int a = 1;”

“011E139E   mov   dword ptr [a],1”

“int b = 2;”

“011E13A5   mov   dword ptr [b],2”

“int c = 0;”

“011E13AC   mov   dword ptr [c],0”

“c = a + b;”

“011E13B3   mov   eax,dword ptr [a]”

“011E13B6   add   eax,dword ptr [b]”

“011E13B9   mov   dword ptr [c],eax”

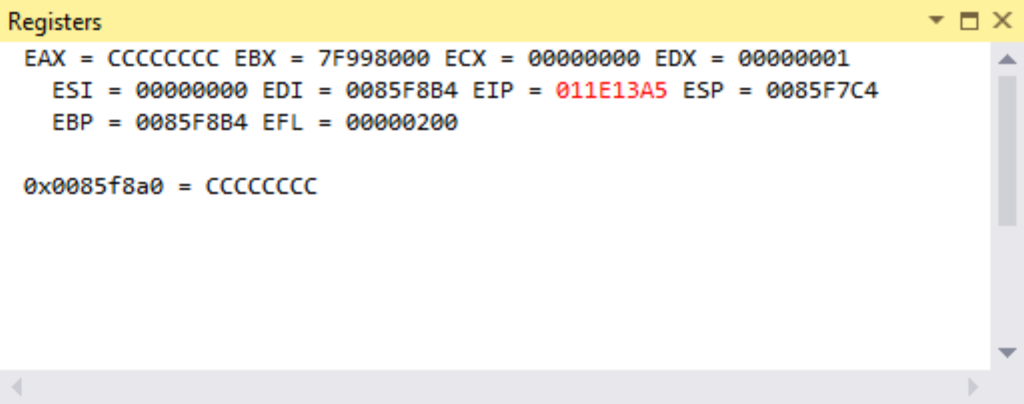
After these instructions, now the local variables are located on the stacks between ebp and esp.

Now we continue press F10 until the yellow arrow points to

“int b = 2;”

“011E13A5   mov   dword ptr [b],2”

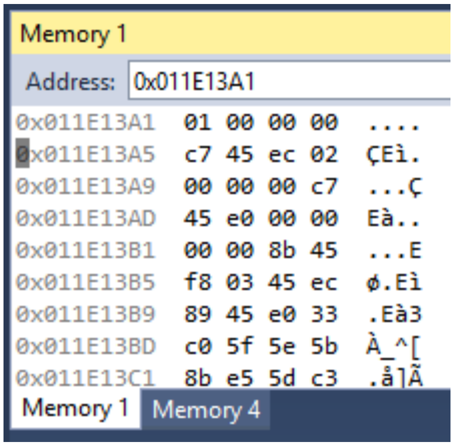
and we look at the register window:



```
EAX = CCCCCCCC EBX = 7F998000 ECX = 00000000 EDX = 00000001
  ESI = 00000000 EDI = 0085F8B4 EIP = 011E13A5 ESP = 0085F7C4
  EBP = 0085F8B4 EFL = 00000200

0x0085f8a0 = CCCCCCCC
```

EIP (instruction pointer) address is 011E13A5. We copy it and paste it into memory window:

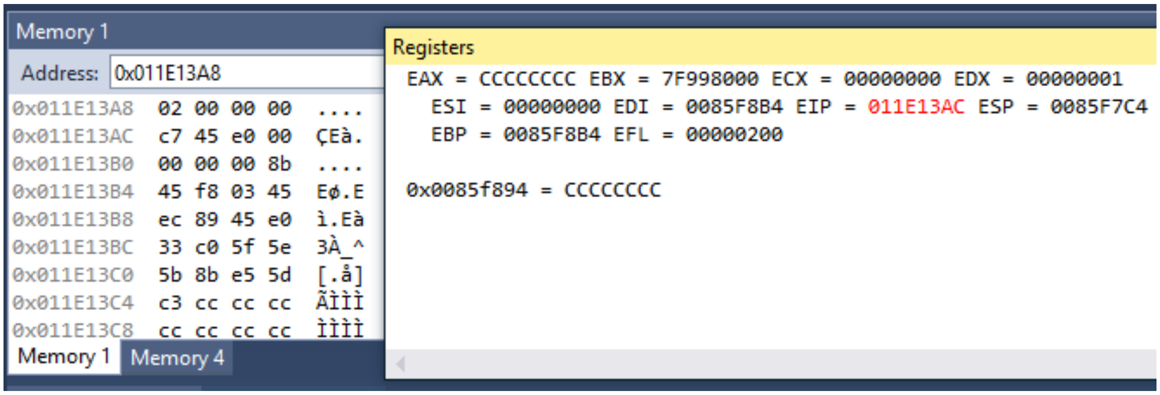


Address: 0x011E13A1

```
0x011E13A1  01 00 00 00  ....
0x011E13A5  c7 45 ec 02  ÇEì.
0x011E13A9  00 00 00 c7  ...Ç
0x011E13AD  45 e0 00 00  Eà..
0x011E13B1  00 00 8b 45  ...E
0x011E13B5  f8 03 45 ec  ø.Eì
0x011E13B9  89 45 e0 33  .Eà3
0x011E13BD  c0 5f 5e 5b  À_^[
0x011E13C1  8b e5 5d c3  .å]Ã
```

and we scroll up a little we can see the address 011E13A1 which is the address of variable “a” that has hexadecimal value of “01000000”, “01” are the least 2 significant bits and “a” has the deciaml value of 1.

If we press F10 one more time and we do the same thing we can find the variable “b” that has decimal value of 2:



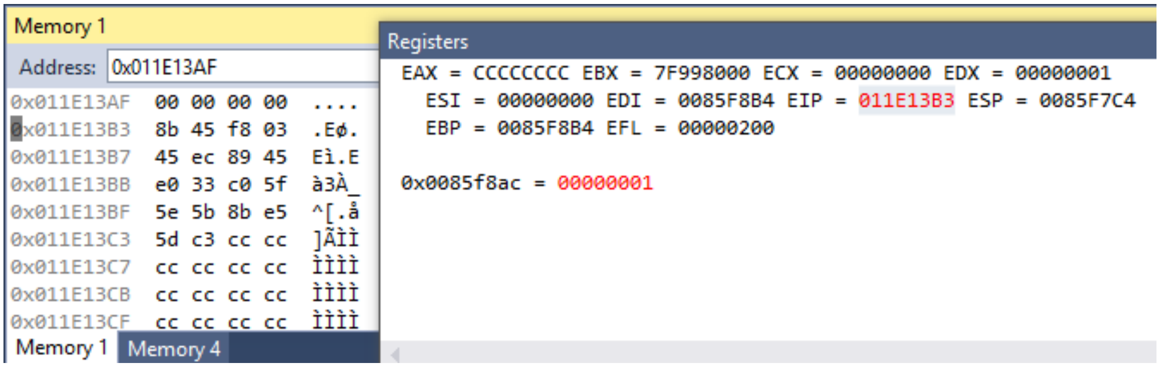
Address: 0x011E13A8

```
0x011E13A8  02 00 00 00  ....
0x011E13AC  c7 45 e0 00  ÇEà.
0x011E13B0  00 00 00 8b  ....
0x011E13B4  45 f8 03 45  Eø.E
0x011E13B8  ec 89 45 e0  ì.Eà
0x011E13BC  33 c0 5f 5e  3À_^
0x011E13C0  5b 8b e5 5d  [.å]
0x011E13C4  c3 cc cc cc  ÃÌÌÌ
0x011E13C8  cc cc cc cc  ÌÌÌÌ
```

```
EAX = CCCCCCCC EBX = 7F998000 ECX = 00000000 EDX = 00000001
  ESI = 00000000 EDI = 0085F8B4 EIP = 011E13AC ESP = 0085F7C4
  EBP = 0085F8B4 EFL = 00000200

0x0085f894 = CCCCCCCC
```

and the same thing finding the variable “c”:



```
0x011E13AF  00 00 00 00  ....
0x011E13B3  8b 45 f8 03  .Eø.
0x011E13B7  45 ec 89 45  Eì.E
0x011E13BB  e0 33 c0 5f  à3À_
0x011E13BF  5e 5b 8b e5  ^[.å
0x011E13C3  5d c3 cc cc  ]ÃÌÌ
0x011E13C7  cc cc cc cc  ÌÌÌÌ
0x011E13CB  cc cc cc cc  ÌÌÌÌ
0x011E13CF  cc cc cc cc  ÌÌÌÌ
```

```
EAX = CCCCCCCC EBX = 7F998000 ECX = 00000000 EDX = 00000001
  ESI = 00000000 EDI = 0085F8B4 EIP = 011E13B3 ESP = 0085F7C4
  EBP = 0085F8B4 EFL = 00000200

0x0085f8ac = 00000001
```