Lab 12

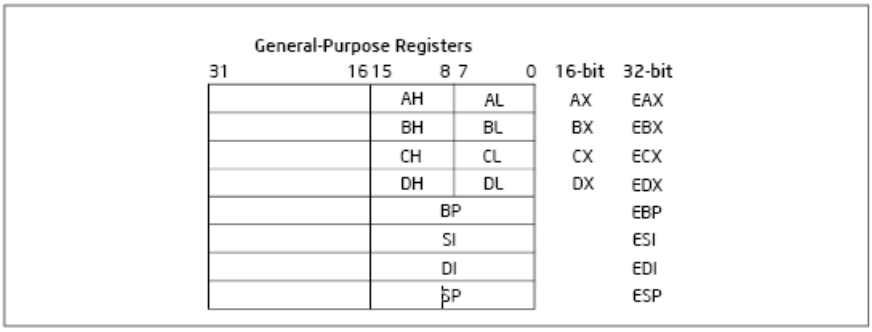
32-bit Inline Assembly Language Programming in Visual Studio

Learning outcome

- Students will be able to write 32-bit assembly language programs.
- Students will be able to run their assembly language code on the underlying processor natively, without any virtual environment.
- Students will learn about addressing modes in 32-bit programming.
- Students will realize how assembly language optimizes their code in terms of number of instructions and speed.
- Students will write relevant parts of the code in assembly language while keeping the rest in C/C++.

Introduction to 32-bit programming

Intel 8386 was the first 32-bit processor of the Intel family of microprocessors. It has 32-bit address and data buses. The protected mode was also introduced in this processor that limits the operation of user programs. It doesn't allow user programs to go beyond their memory limits assign to them and directly access IO devices. The size of existing registers, except the segment registers, were extended to 32-bit as shown below.



Addressing Modes

The addressing modes are the same.

In the 80386 and above, any register from EAX, EBX, ECX, EDX, EBP, EDI, ESI may be used in register indirect addressing mode. (Example: The MOV [EDX], CL instruction copies the byte sized contents of register CL into the data segment memory location addressed by EDX)

In the 80386 and above, any two registers from EAX, EBX, ECX, EDX, EBP, EDI, or ESI may be combined to generate the memory address. (Example: The MOV [EAX+EBX], CL instruction copies the byte sized contents of register CL into the data segment memory location addressed by EAX plus EBX.)

The 80386 and above use any 32-bit register except ESP to address memory. (Example: MOV AX, [ECX] or MOV AX, ARRAY[EBX]. The first instruction loads AX from the data segment address formed by ECX plus 4. The second instruction loads AX from the data segment memory location ARRAY plus the contents of EBX.

Inline Assembler *(Microsoft Specific)*

Assembly language serves many purposes, such as improving program speed, reducing memory needs, and controlling hardware when writing kernel modules. You can use the inline assembler to embed assembly-language instructions directly in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler, so you don't need a separate assembler such as the Microsoft Macro Assembler (MASM). Because the inline assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. Inline assembly code can use any C or C++ variable or function name that is in scope, so it is easy to integrate it with your program's C and C++ code. And because the assembly code can be mixed with C and C++ statements, it can do tasks that are cumbersome in C or C++ alone.

The__asm keyword invokes the inline assembler and can appear wherever a C or C++ statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces.

Example

__asm mov eax,ebx

or

__asm

{

Mov eax,ebx

}

Using and Preserving Registers in Inline Assembly *(Microsoft Specific)*

In general, you should not assume that a register will have a given value when an __asm block begins. Register values are not guaranteed to be preserved across separate __asm blocks. If you end a block of inline code and begin another, you cannot rely on the registers in the second block to retain their values from the first block. An __asm block inherits whatever register values result from the normal flow of control.

When using __asm to write assembly language in C/C++ functions, you don't need to preserve the EAX, EBX, ECX, EDX,ESI, or EDI registers. You should preserve other registers you use (such as DS,SS,ESP, EBP,and flags registers) for the scope of the __asm block.

Example#1: Program to add two integer numbers using the inline assembly.

```
#include<iostream>
using namespace std;


int main(void)
{
	int a = 10, b = 20, c = 0;
	__asm
		{
			mov ebx,a
			add ebx,b
			mov c,ebx
		}
		cout << "Sum of " << a << " and " << b << " is " << c << endl;

}
```

Example#2: Program to access array using inline 32-bit assembly. Both codes are incrementing every element of the array.

```
#include<iostream>
using namespace std;

int array1[] = {1,2,3,4,5};

int main(void)
{
	__asm
	{
		push ebp
		mov ebp, offset array1
		mov esi, 0
		mov ecx, 5
	l1:
		inc dword ptr[ebp + esi]
		add esi, 4
		loop l1
		pop ebp
	}
	for (int i = 0; i < 5; i++)
		cout << array1[i];
}
```

```
#include<iostream>
using namespace std;

int array1[] = {1,2,3,4,5};

int main(void)
{

	for (int i = 0; i < 20; i=i+4)
	{

		__asm
		{
		mov esi,i
		inc dword ptr [array1+esi]

		}
	}
	for (int i = 0; i < 5; i++)
		cout << array1[i];
}
```

Example#3: Program to rotate right each element of the array using 1) C++ and 2) inline assembly. Observe that assembly language makes our task simpler.

Rotate left in C++	Rotate left using inline assembly

#include<iostream>
using namespace std;
int main()
{
int array1[] = { 1,2,3,4,5 };
for (int i = 0; i < 5; i++)
{
array1[i] = (array1[i] << 1) | (array1[i] >>31);
}
for (int i = 0; i < 5; i++)
{
cout << array1[i] << endl;
}
return 0;
}
	#include<iostream>
	using namespace std;
	int main(void)
	{
	int array1[] = { 1,2,3,4,5 };
	for (int i = 0; i < 20; i=i+4)
	{
	__asm
	{
	mov eax,i
	rol dword ptr [array1+eax],1
	}
	}
	for (int i = 0; i < 5; i++)
	{
	cout << array1[i] << endl;
	}
	}

Example#3: Program to add 1 to each element of a 2D array

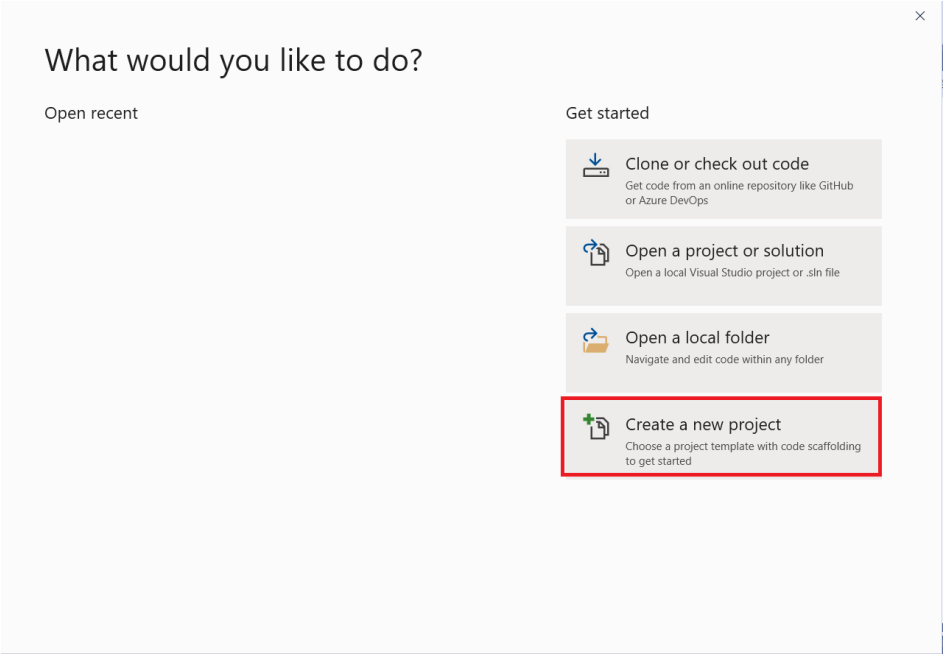
```
#include<iostream>
using namespace std;
int main(void)
{
	int array1[3][4] = { {1,2,3,1},{4,5,6,1},{7,8,9,1} };
	int index = 0;
	for (int i = 0; i < 3; i++)
		for (int j = 0; j < 4; j++)
		{
			index = (i * 4 + j) * sizeof(int);
			__asm
			{
				mov eax,index
				add dword ptr [array1+eax],1


			}
			cout << "(" << i << "," << j << ")" << "->" << array1[i][j] << endl;
		}

}
```

How to RUN 32-bit inline assembly language programs

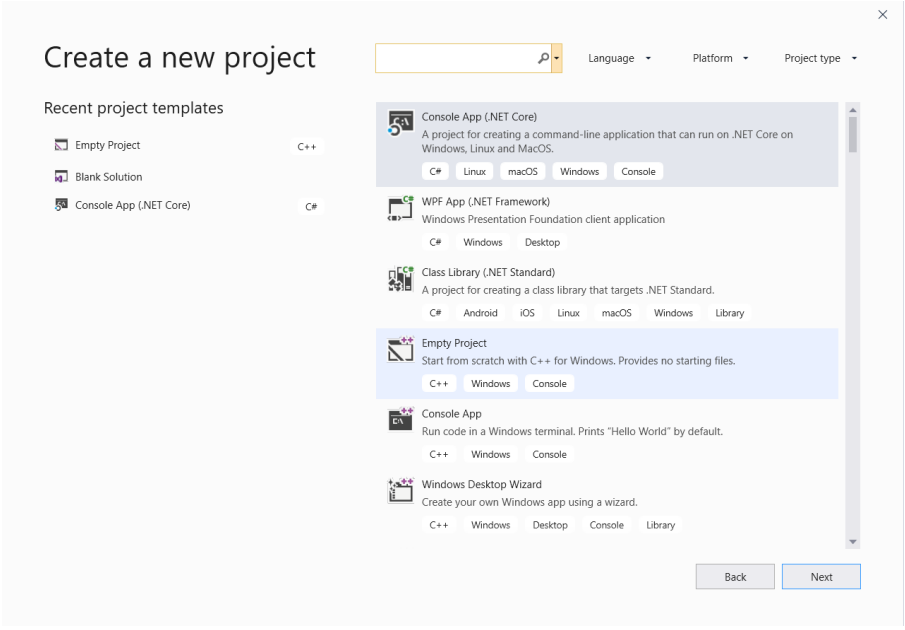
Step#1:

Open Visual Studio 2019

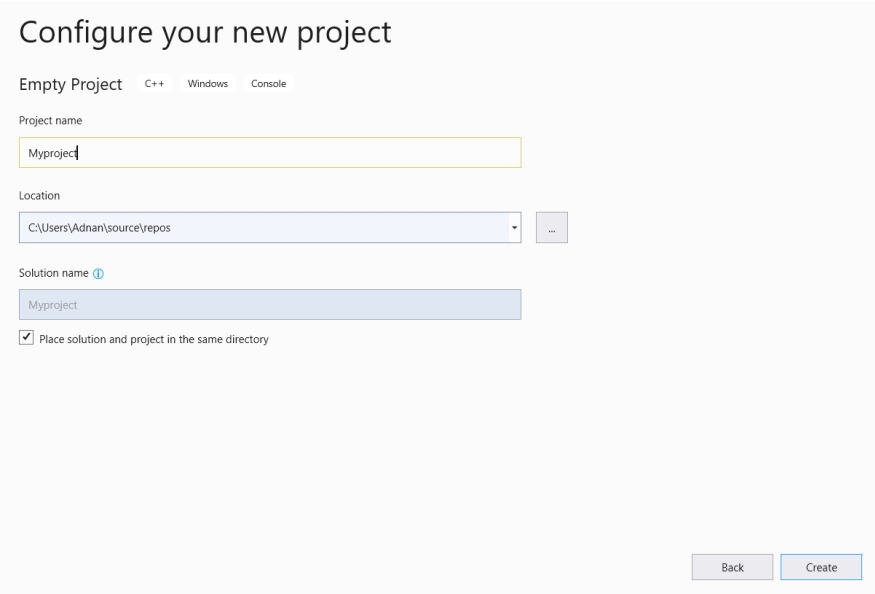
Step#2: Click on “Create a new project”



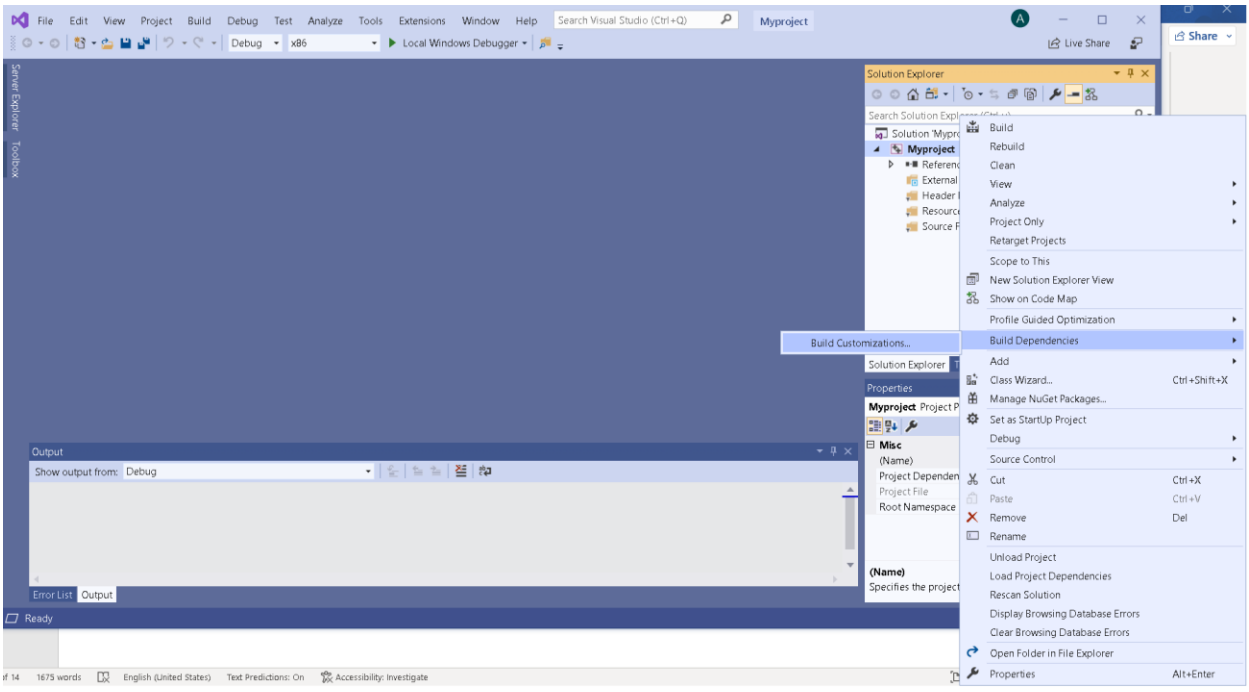
Step#3: Click on C++ under “Empty Project” and press Next



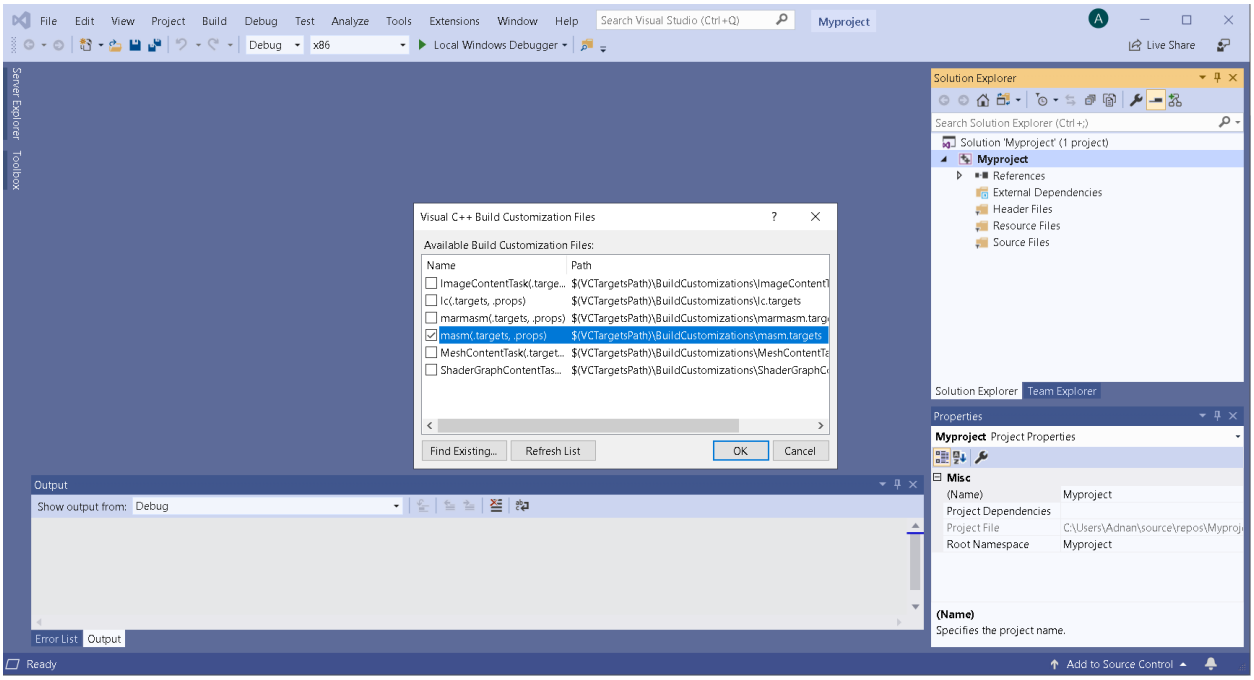
Step#4: Write your project name (e.g., Myproject) and click on Create



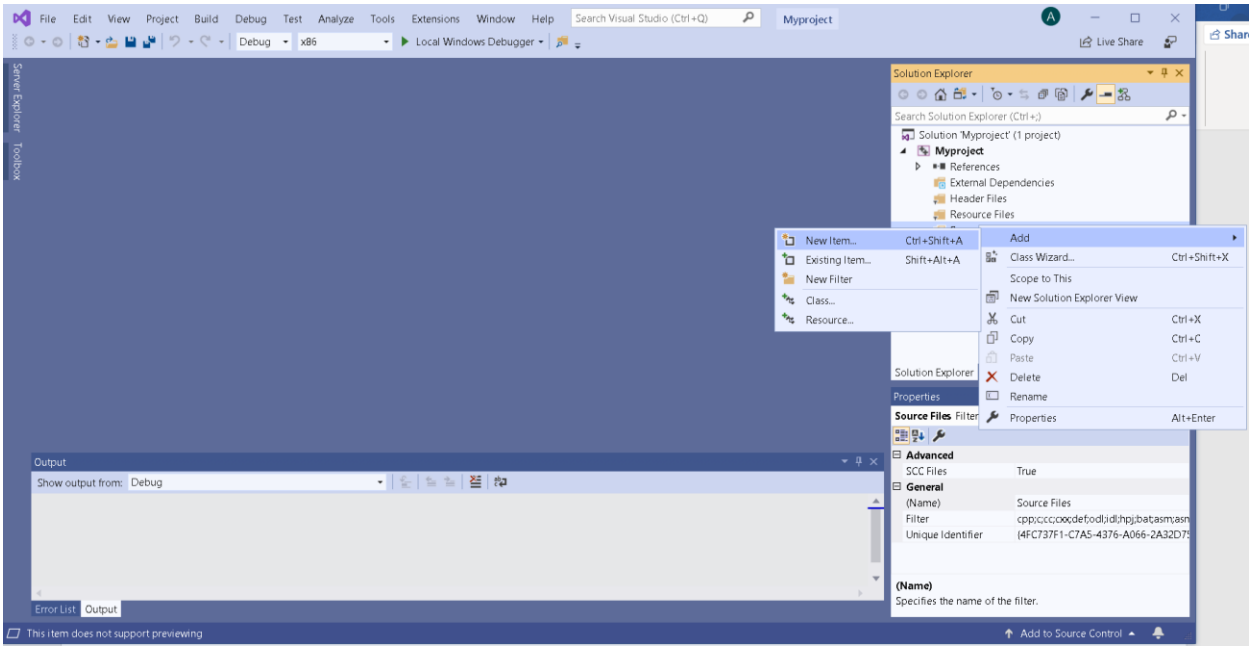
Step#5: Right click on project name (i.e., Myproject) → Build Dependencies →Build Customizations



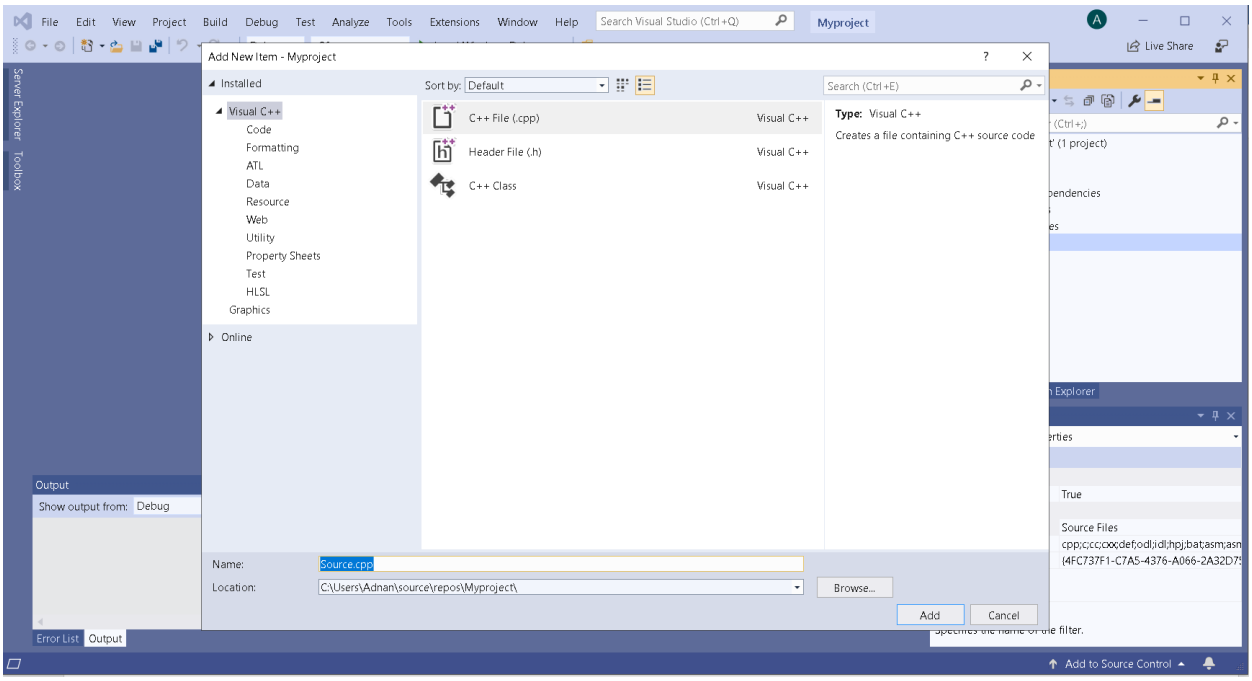
Step#5: Select “masm” and press OK



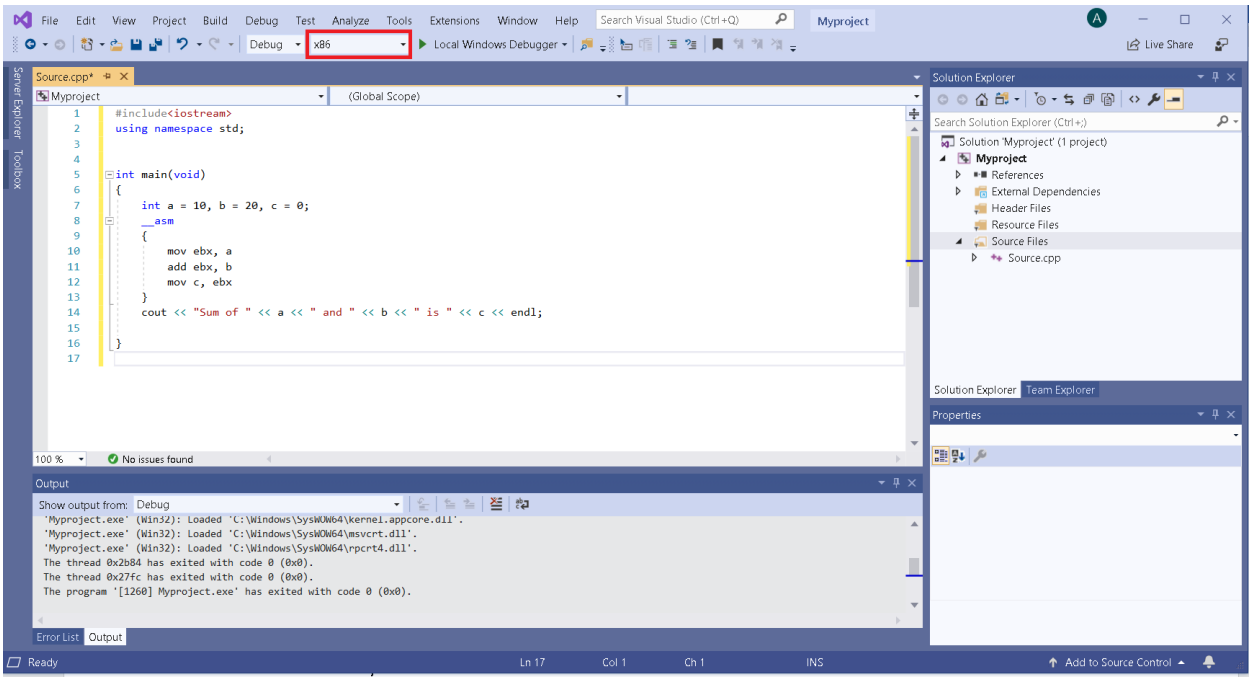
Step#6: Right click on “Source Files” → Add → New Item



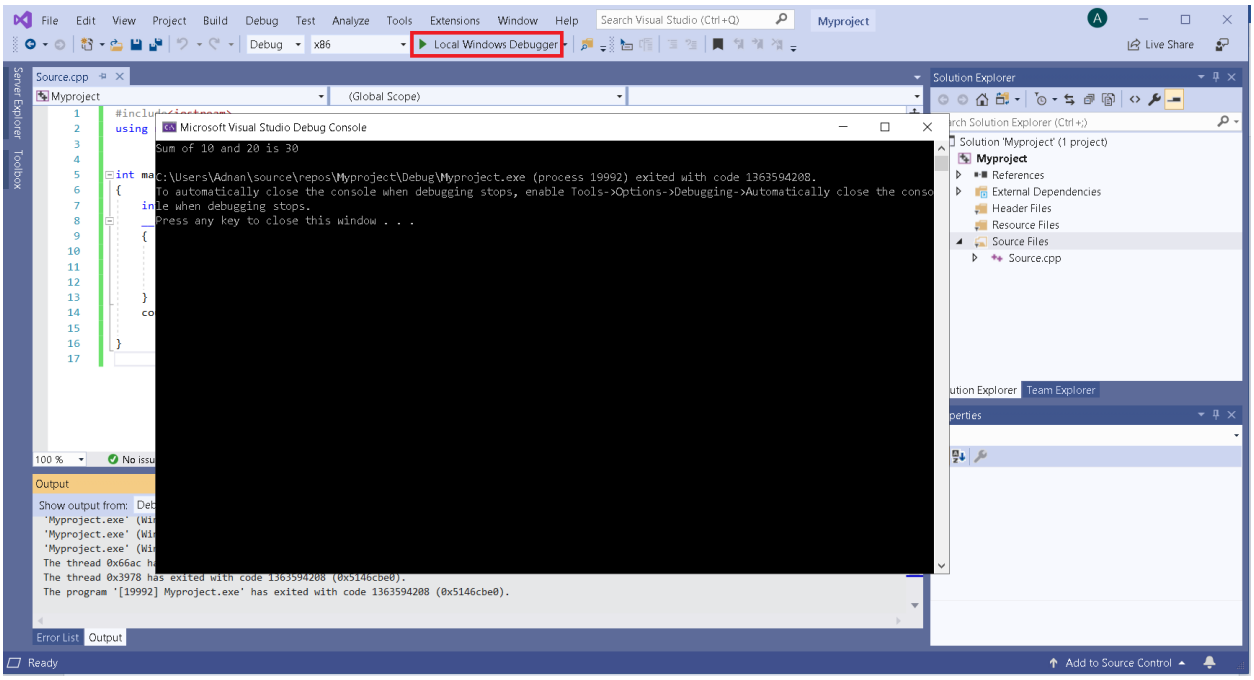
Step#7: Click in “C++ File (.cpp) , write a file name and click on Add.



Step#8: Paste code in Example#1 in the text window. Make sure that the environment is set to x86.

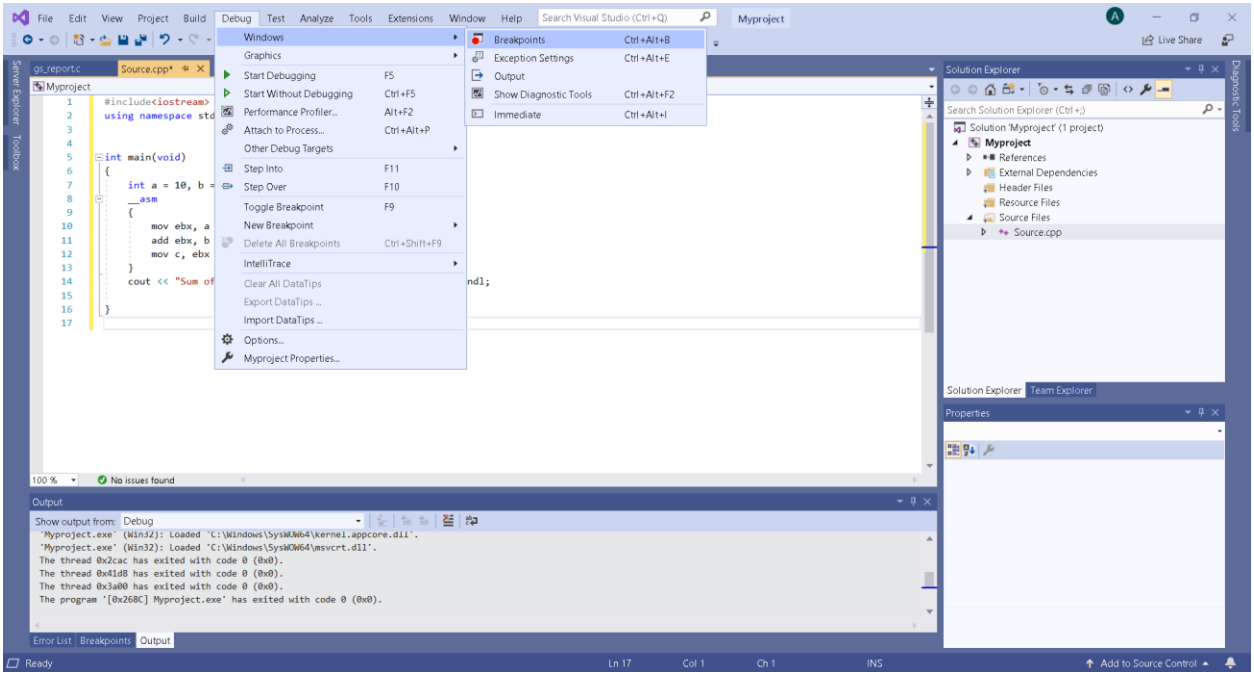


Step#8: Click on “Local Windows Debugger” to run the code.

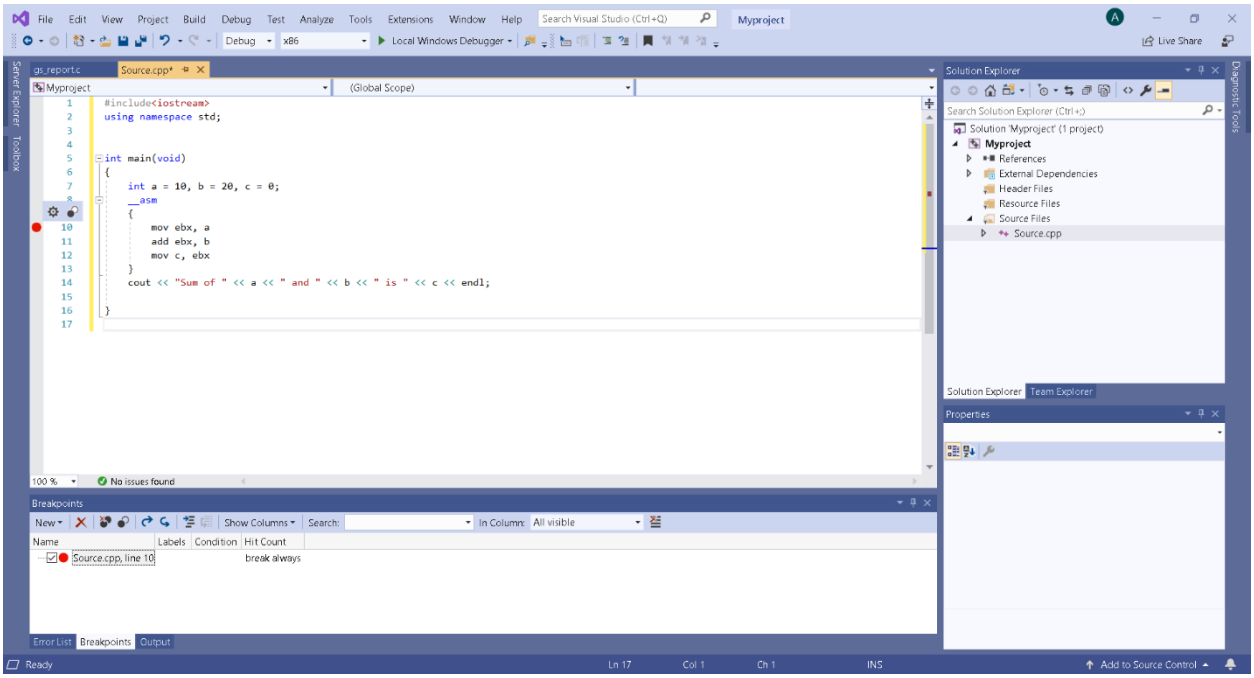


How to Debug a program

Step#1: Click on Debug→Windows→Breakpoints



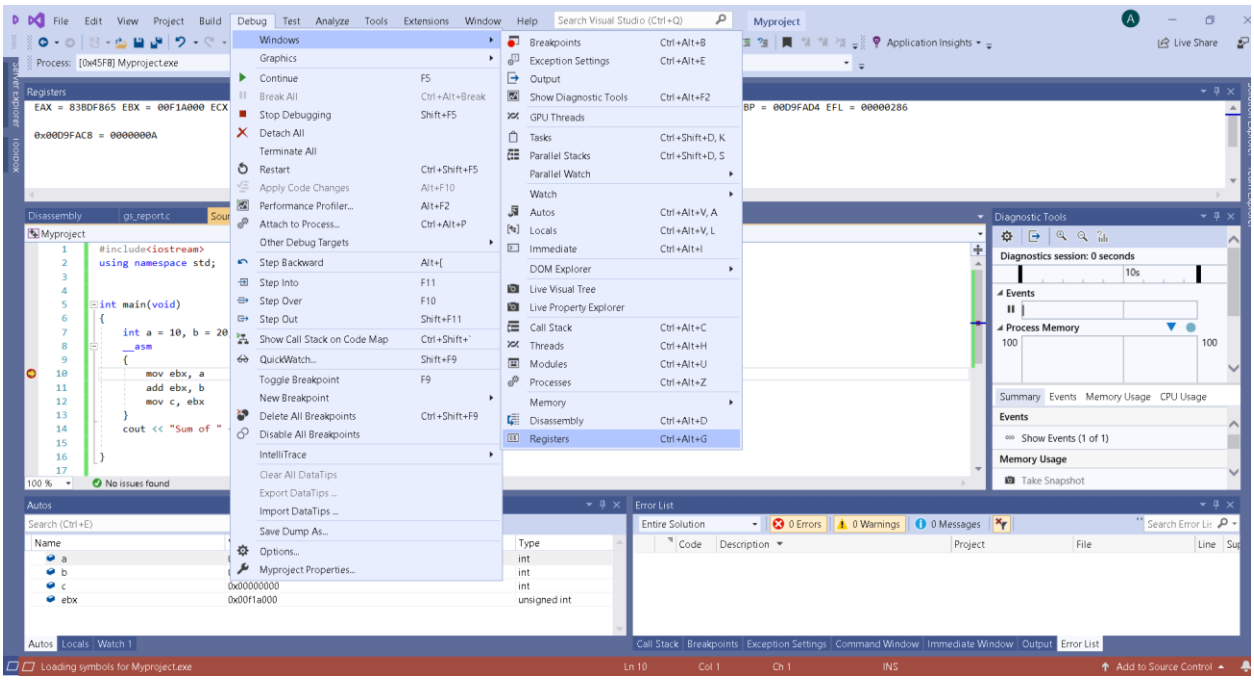
Step#2: Click at the left side of instruction to set breakpoints.



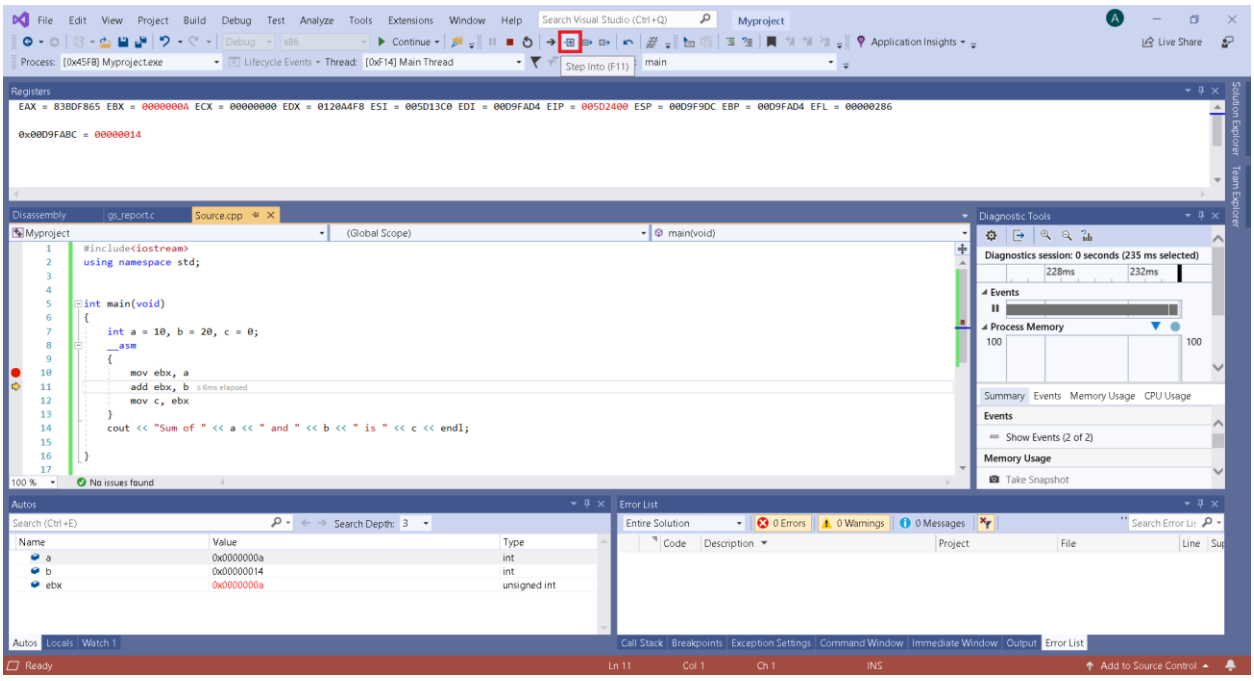
Step#3: Click on Local Windows Debugger

Step#4:

a) Click on Debug→Windows→Registers (to view status of registers)
b) Click on Debug→Windows→Disassembly (to view assembled code)
c) Click on Debug→Windows→Memory (to view memory)



Step#5: Click on “Step Into” (or F11) to debug program



Practice Tasks

Task-1

Write a program in C++ that declares and initializes an integer array of 5 elements. The program then swaps the least and most significant bytes and the nibbles of the second byte of each element of the array using inline assembly language programming.

Task-2

Write a program in C++ that declares and initializes an integer array of 5 elements. The program then rotates the least significant byte (i.e., byte #0) and most significant byte (i.e., byte #3) to the left and the byte #1 and byte #2 to the right of each element of the array using inline assembly language programming.