# CSC 3210

# Computer Organization and Programming

## Lab Work 10

Dr. Zulkar Nine

mnine@gsu.edu

Georgia State University

Fall 2021

# Learning Objective

- Logical instructions and loops

# Disclaimer

- The process shown in these slides might not work in every single computer due to Operating system version, Microsoft Visual Studio versions and everything.
- If you find any unusual error, you can inform the instructor.
- Instructor will help you resolve the issue.

# Attendance!

# Lab Work 10 Instructions

- Lab 10 : Conditional statements and loop

# Plan early …

- You have one week time to submit the lab
- Start early
- If you have issues
  - Email TA or instructor
  - Stop by during office hours
- Start working at the last moment is not a good idea.

- AppendixA shows how to check memory data and Appendix B shows how to create a new project.

# Problems in this lab

- You might see similar questions in the quizzes and exam.
- During the exam you might need solve similar problems without visual studio.

# A review

CMP and conditional jumps

# CMP Instruction

- The most common **boolean expressions** involve some type of **comparison**:

```
if A > B ...
while X > 0 and X < 200 ...
if check_for_error( N ) = true
```

- **CMP** instruction is used to compare integers (signed and unsigned)
  - Compares the **destination** operand to the **source** operand (HOW 🤔)
    - **CMP** performs **implied subtraction** of **source operand** from **destination operand**
    - **Nondestructive subtraction** of source from destination (destination operand is not changed)
  - **Syntax:**

    **CMP** *destination, source*

# *Jcond* Instruction

- A conditional jump instruction **branches** to a label
  - When specific **status flag** condition **is true**
- Syntax

**Jcond** destination

- **cond** refers to a flag condition identifying the state of one or more flags.

| | |
|---|---|
| JC | Jump if carry (Carry flag set) |
| JNC | Jump if not carry (Carry flag clear) |
| JZ | Jump if zero (Zero flag set) |
| JNZ | Jump if not zero (Zero flag clear) |

**y** and **Zero** flags:

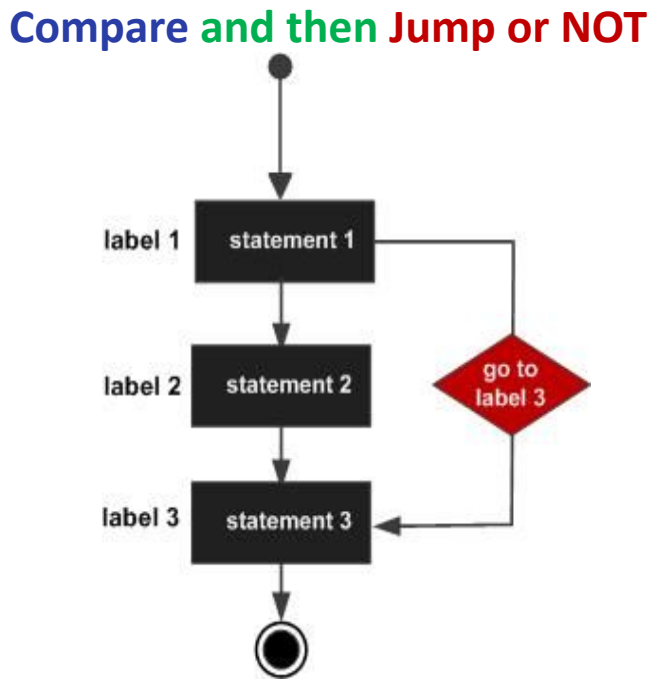
CPU status flags are most commonly **set** by arithmetic, comparison, and boolean instructions.

# Conditional Jumps (Example1)

- **First,** an operation such as CMP, AND, or SUB modifies the **CPU status flags**.
- **Second**, a conditional jump instruction **tests** the flags and **causes a branch** to a new address.

- The CMP instruction compares EAX to Zero.
- The JZ (Jump if zero) instruction jumps to label L1 **if** the Zero flag was set **by the CMP** instruction:

```
        cmp eax,0
        jz L1           ; jump if ZF = 1
        .
        .
    L1:
        .
        .
```



# Conditional Jumps (Example2)

- **First,** an operation such as CMP, AND, or SUB modifies the **CPU status flags**.
- **Second**, a conditional jump instruction **tests** the flags and **causes a branch** to a new address.

**Compare and then Jump**

- The AND instruction performs a bitwise AND on the DL register, **affecting** the Zero flag.
- The JNZ (jump if not Zero) instruction jumps if the Zero flag is clear:

```
        and dl,10110000b
        jnz L2                  ; jump if ZF = 0
        .
        .
L2:
```

# Types of Conditional Jumps Instructions

- **Conditional jump** instructions are able to
  - **Compare** signed and unsigned integers and
    - **Perform actions** based on the values of individual CPU flags.

- The conditional jump instructions can be divided into **four groups**:
  - Jumps based on **specific flag** values
  - Jumps based on **equality** between operands or the value of (E)CX
  - Jumps based on comparisons of **unsigned** operands
  - Jumps based on comparisons of **signed** operands

# Jumps Based on **Specific Flags**

| Mnemonic | Description | Flags |
|---|---|---|
| JZ | Jump if zero | ZF = 1 |
| JNZ | Jump if not zero | ZF = 0 |
| JC | Jump if carry | CF = 1 |
| JNC | Jump if not carry | CF = 0 |
| JO | Jump if overflow | OF = 1 |
| JNO | Jump if not overflow | OF = 0 |
| JS | Jump if signed | SF = 1 |
| JNS | Jump if not signed | SF = 0 |
| JP | Jump if parity (even) | PF = 1 |
| JNP | Jump if not parity (odd) | PF = 0 |

# Jumps Based on **Equality**

| Mnemonic | Description |
|---|---|
| JE | Jump if equal (*leftOp* = *rightOp*) |
| JNE | Jump if not equal (*leftOp* ≠ *rightOp*) |
| JCXZ | Jump if CX = 0 |
| JECXZ | Jump if ECX = 0 |
| JRCXZ | Jump if RCX = 0 (64-bit mode) |

# Jumps Based on **Unsigned Comparisons**

| Mnemonic | Description |
|---|---|
| JA | Jump if above (if *leftOp* > *rightOp*) |
| JNBE | Jump if not below or equal (same as JA) |
| JAE | Jump if above or equal (if *leftOp* >= *rightOp*) |
| JNB | Jump if not below (same as JAE) |
| JB | Jump if below (if *leftOp* < *rightOp*) |
| JNAE | Jump if not above or equal (same as JB) |
| JBE | Jump if below or equal (if *leftOp* <= *rightOp*) |
| JNA | Jump if not above (same as JBE) |

A and B

# Jumps Based on **Signed Comparisons**

| Mnemonic | Description |
|---|---|
| JG | Jump if greater (if *leftOp* > *rightOp*) |
| JNLE | Jump if not less than or equal (same as JG) |
| JGE | Jump if greater than or equal (if *leftOp* >= *rightOp*) |
| JNL | Jump if not less (same as JGE) |
| JL | Jump if less (if *leftOp* < *rightOp*) |
| JNGE | Jump if not greater than or equal (same as JL) |
| JLE | Jump if less than or equal (if *leftOp* <= *rightOp*) |
| JNG | Jump if not greater (same as JLE) |

G and L

# IF Statements: Example1 (IF-Then)

- IF statement is translated into assembly language with a
  - **CMP instruction** followed by
  - **Conditional jumps**.
- If op1 or op2 is a **memory operand** (a variable):
  - **one of them must be moved to a register** before executing **CMP**.

- Implement the following pseudocode in assembly language.
- All values are **unsigned**:

```
if( ebx == ecx )
{
  eax = 5;
  edx = 6;
}
```

```
    cmp ebx,ecx
    je  L1
    jmp L2
L1:
    mov eax,5
    mov edx,6
L2:
```

A and B

(There are multiple correct solutions to this problem.)

# IF Statements: Example1 (IF-Then)

- IF statement is translated into assembly language with a
  - **CMP instruction** followed by
  - **Conditional jumps**.
- If op1 or op2 is a **memory operand** (a variable):
  - **one of them must be moved to a register** before executing **CMP**.

- Implement the following pseudocode in assembly langua
- All values are **unsigned**:

```
if( ebx == ecx )
{
  eax = 5;
  edx = 6;
}
```

**Reverse The IF Condition**

```
    cmp ebx,ecx
    jne  next
    mov eax,5
    mov edx,6
next:
```

A and B

(There are multiple correct solutions to this problem.)

# Example: Check Larger of two integers

- Assume two signed integers stored in EAX and EBX register. Write a code that transfers largest integer into variable called MAXIMUM.

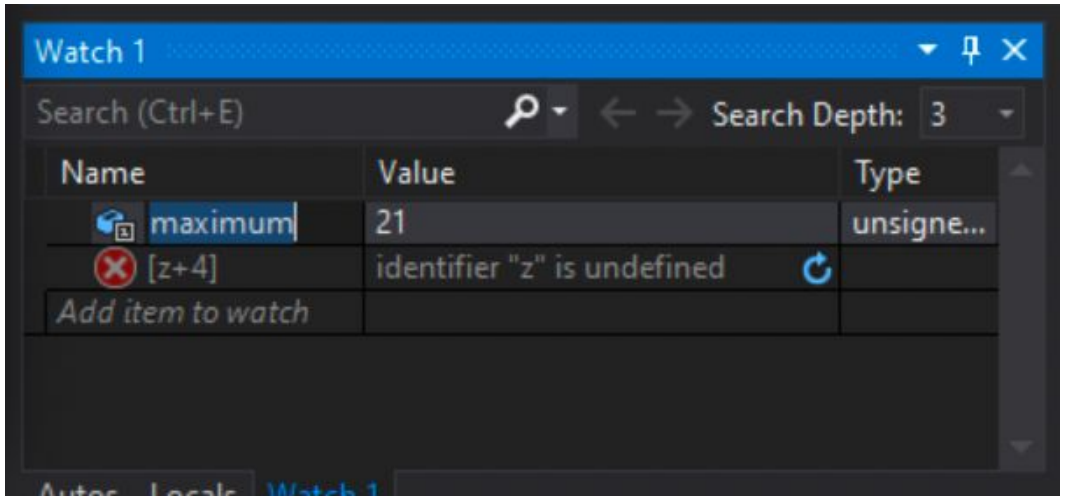
```
.data
    maximum DWORD ?
.code
    ; Store first item to EAX register
    MOV eax, 21
    ; Store the second item into EBX register
    MOV ebx, 19
    CMP eax, ebx
    JG L1
    JLE L2
        L1:  MOV MAXIMUM, eax
    JMP Next
    L2: MOV MAXIMUM, ebx
        Next:
```

Don’t need to turn in!

# Debug the code

- Enter the debug mode
- and open watch window. Go to Debug-> Windows -> Watch -> Watch1
- Enter the variable name maximum in watch window
- Then debug the code

  until you reach “INVOKE ExitProcess”



# Example: Find the smallest among three items

- Let's assume, the values are V1, V2, and V3 and you want to store the smallest item into AX register.

- Logic:

```
If V1 <= V2
    Mov minimum, V1
Else
    Mov minimum, V2
If V3 < minimum:
    Mov minimum, V3
```

# Find the smallest among three items

```
.data
  V1 SWORD 10
  V2 SWORD 13
  V3 SWORD -5
  minimum SWORD ?
```

```
.code
  mov ax, V1
  mov bx, V2
  mov cx, V3
  cmp ax, bx
  jle IF_Block  ;if V1 < V2, jump to IF block
  jg ELSE_Block
  IF_Block:
    mov minimum, ax
  jmp next
  ELSE_Block:
    mov minimum, bx
  next:
  cmp cx, minimum ; ;if V3 < minimum, jump to IF_block2
  jle IF_Block2
  jmp L1
  IF_Block2:
    mov minimum, cx
  L1:
```

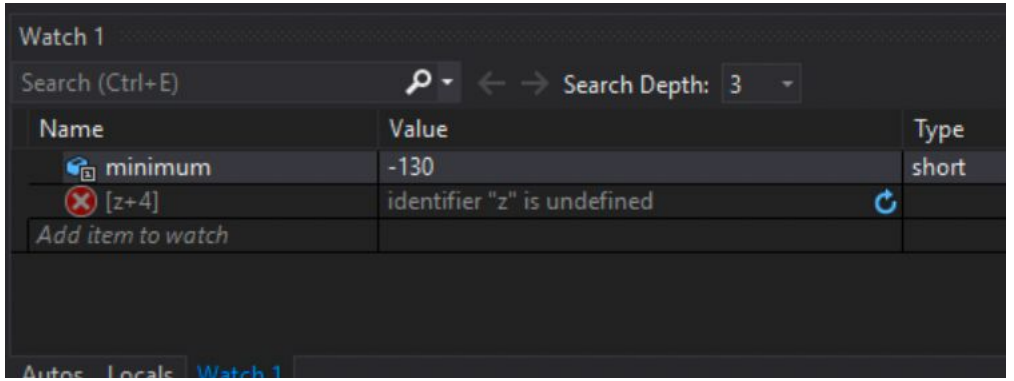
Logic:
If V1 < V2
    Mov minimum, V1
Else
    Mov minimum, V2
If V3 < AX:
    Mov minimum, V3

Don't need to turn in!

# Debug the code

- Enter the debug mode
- and open watch window. Go to Debug-> Windows -> Watch -> Watch1
- Enter the variable name minimum in the watch window
- Then debug the code

  until you reach “INVOKE ExitProcess”



# Lab 10

Submission

# Submission Instruction

- Write an assembly program to find the largest item in an array and store it in a variable named MAXIMUM.
- Hint: Use both Jump and loop instructions to write the program.
- logic:
- Assume that the first item of the array is the maximum and store it in variable MAXIMUM
- Write a loop. Inside the loop, compare the each array item with the maximum
- If the array item is greater than the MAXIMUM, update MAXIMUM with that array item.

```
.data
   Array WORD 10, 2, 23, 45, 21, 11
   MAXIMUM WORD ?
 .code
   ; write your code
```

# Submission Instruction

- Submit the screenshot of your code.
- Debug your code until you reach INVOKE ExitProcess, 0
- Take a screenshot of the watch window showing variable MAXIMUM.
  - Submit the screenshot.
- Also, Rename the asm file using your last name as Lastname.asm
  - Submit the ASM file as well.

# Appendix A

## Checking Memory Data

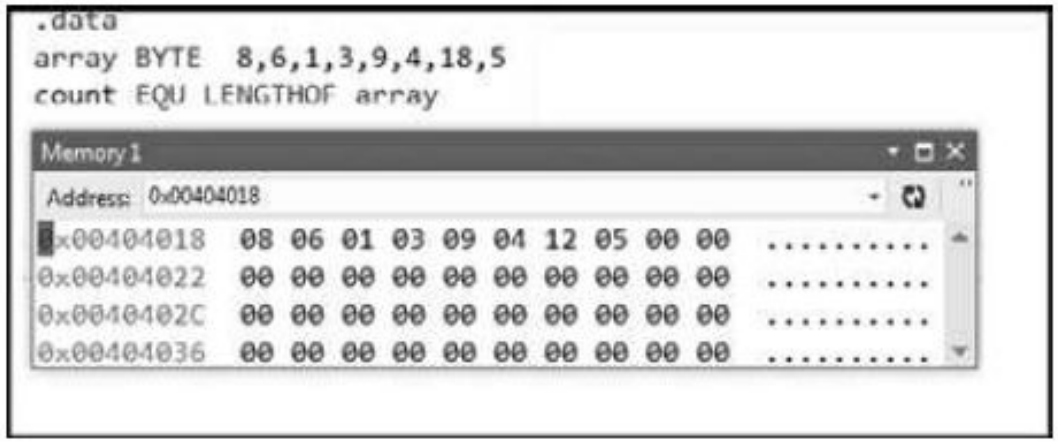
# Checking Memory Data

- Use **Memory window** to verify the values of memory locations.
  - **To activate Memory window,** run the debugger, go to debug menu and click on windows, open it, go to Memory then choose **Memory1**.
    - When you run your program and step over every line you will see the changed values marked with red color.

**You Must be in the Debugging Mode to see the memory or the register window**

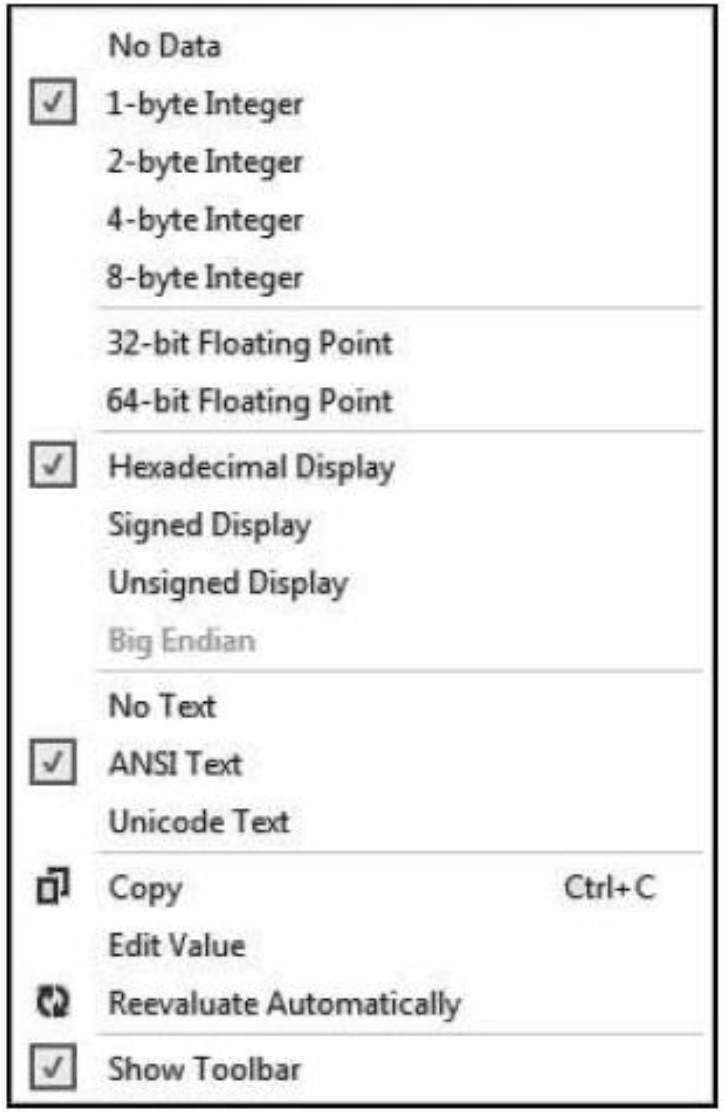
# Checking Memory Data

- **To activate Memory window,**
  - if you want to see the location of your variable in the memory,
    - Memory window search box (on the top of the memory window, Address:)
    - write **&** follow it with the variable name: example: **&myVal**l.
    - This will take you to the memory locations of your program (.data section).



# Checking Memory Data

- **To activate Memory window,**
  - You can right-click inside the memory window
  - You will see **Popup menu for the debugger's memory window**
  - You can choose how you want to group your bytes: by 1,2,4, or by 8
  - You can also presents data in **hexadecimal**, **signed**, or **unsigned** display



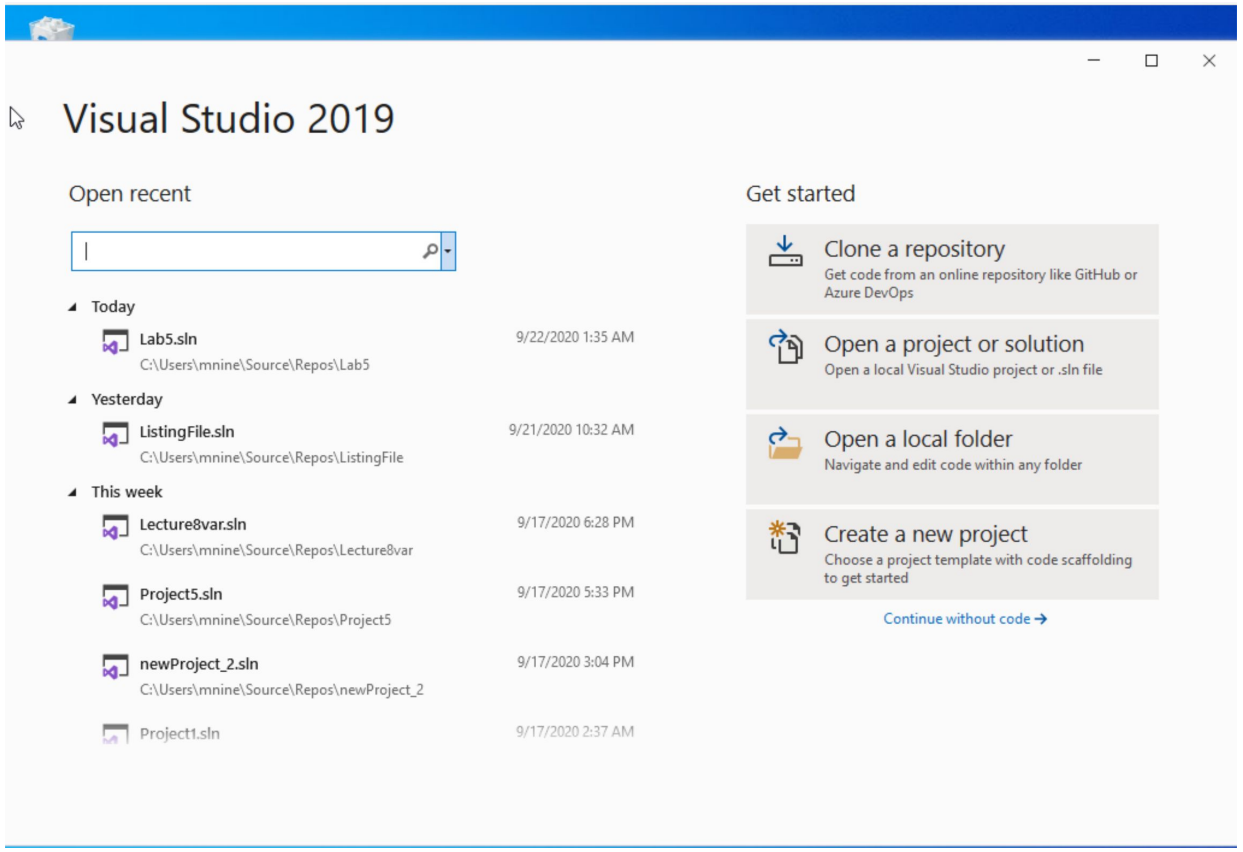
# Appendix B

## Create a Project

# Step 1: Create a project (1)

(1) Start Visual Studio

(2) Click Create a new Project

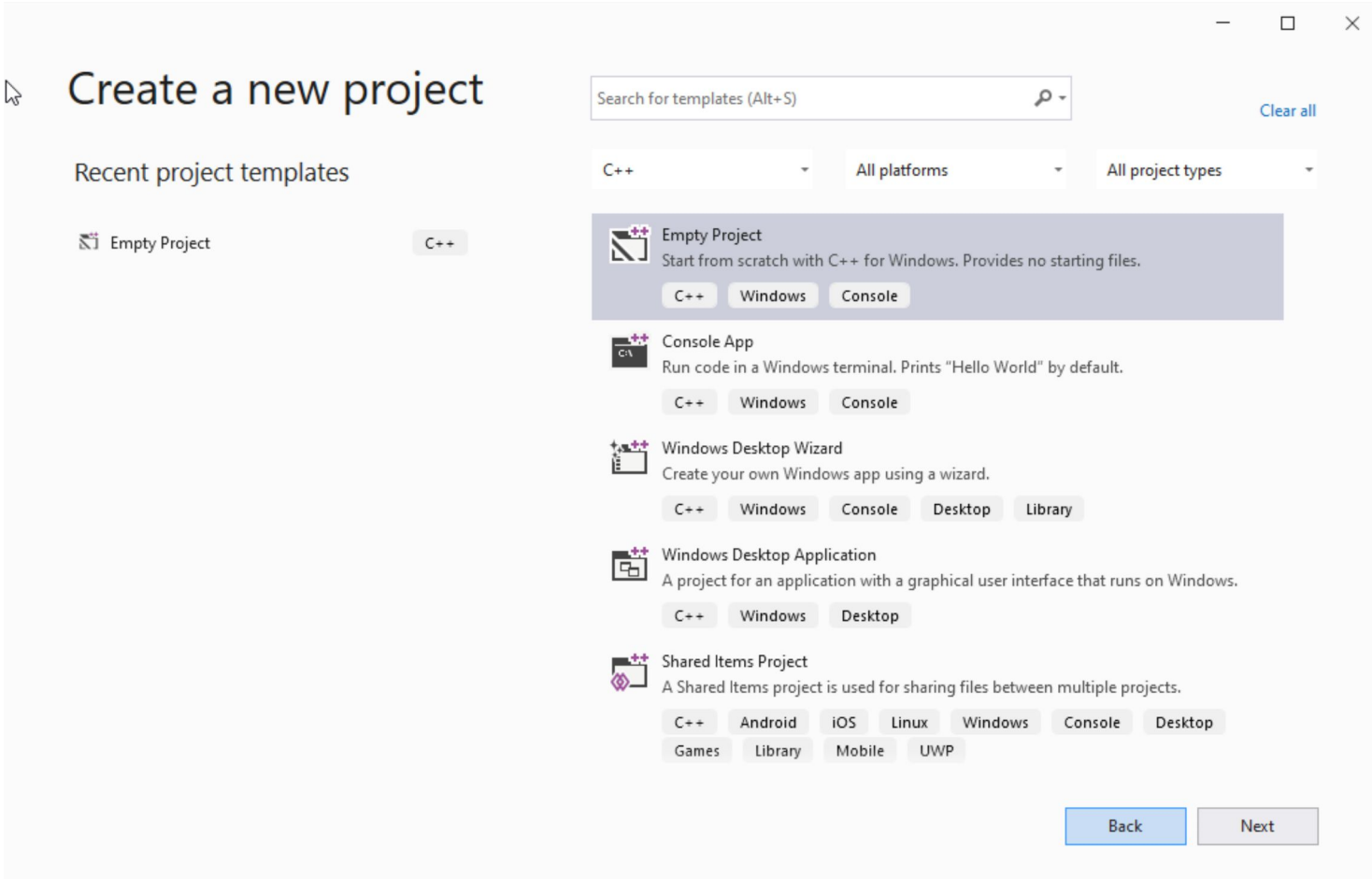


# Step 1: Create a project (2)

(1) Select C++ as language

(2) Select Empty Project

(3) Click Next

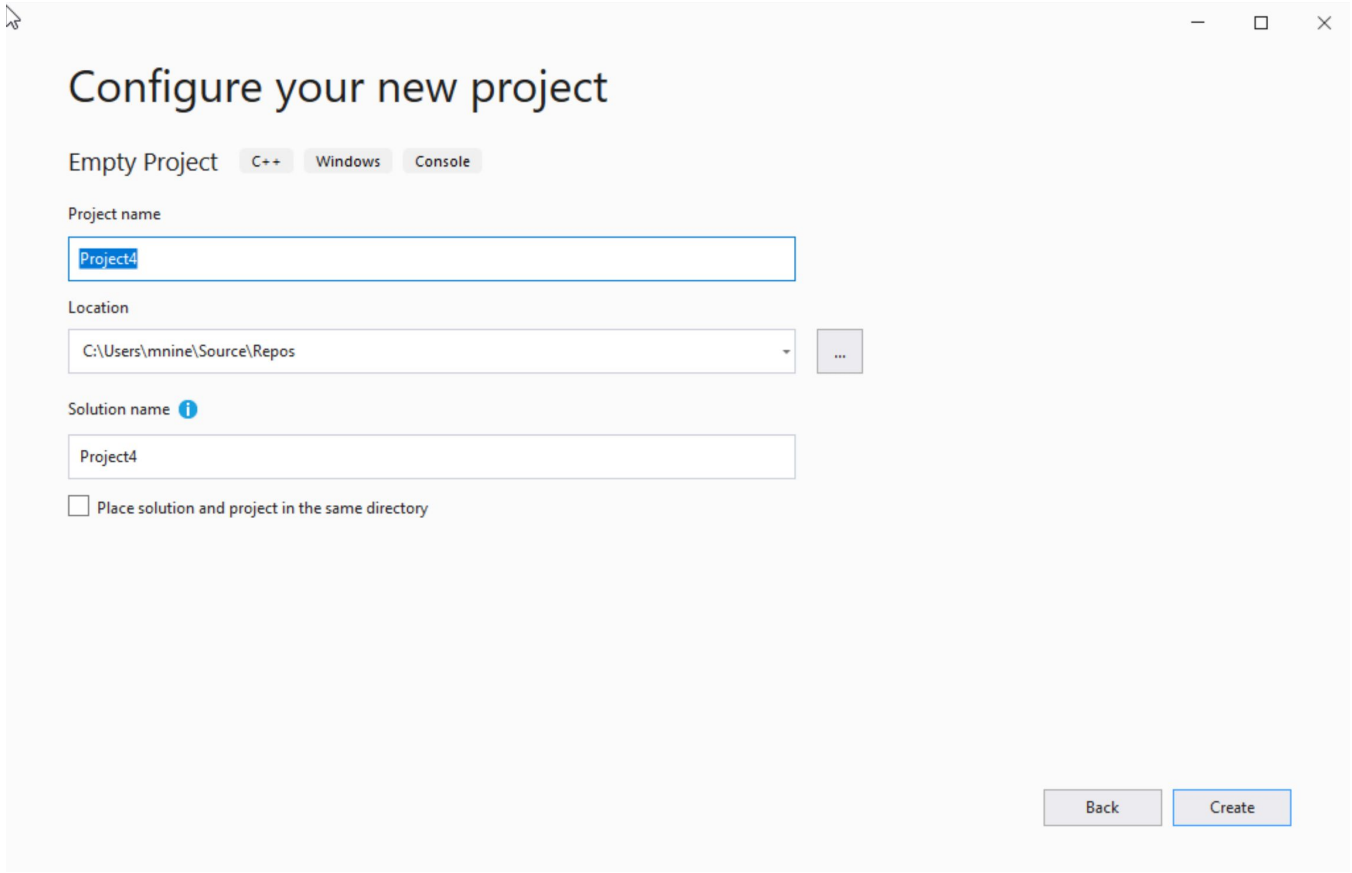


# Step 1: Create a project (3)

(1) You can change the project name as you like

(1) Also, you can change the project location

(2) Click Next

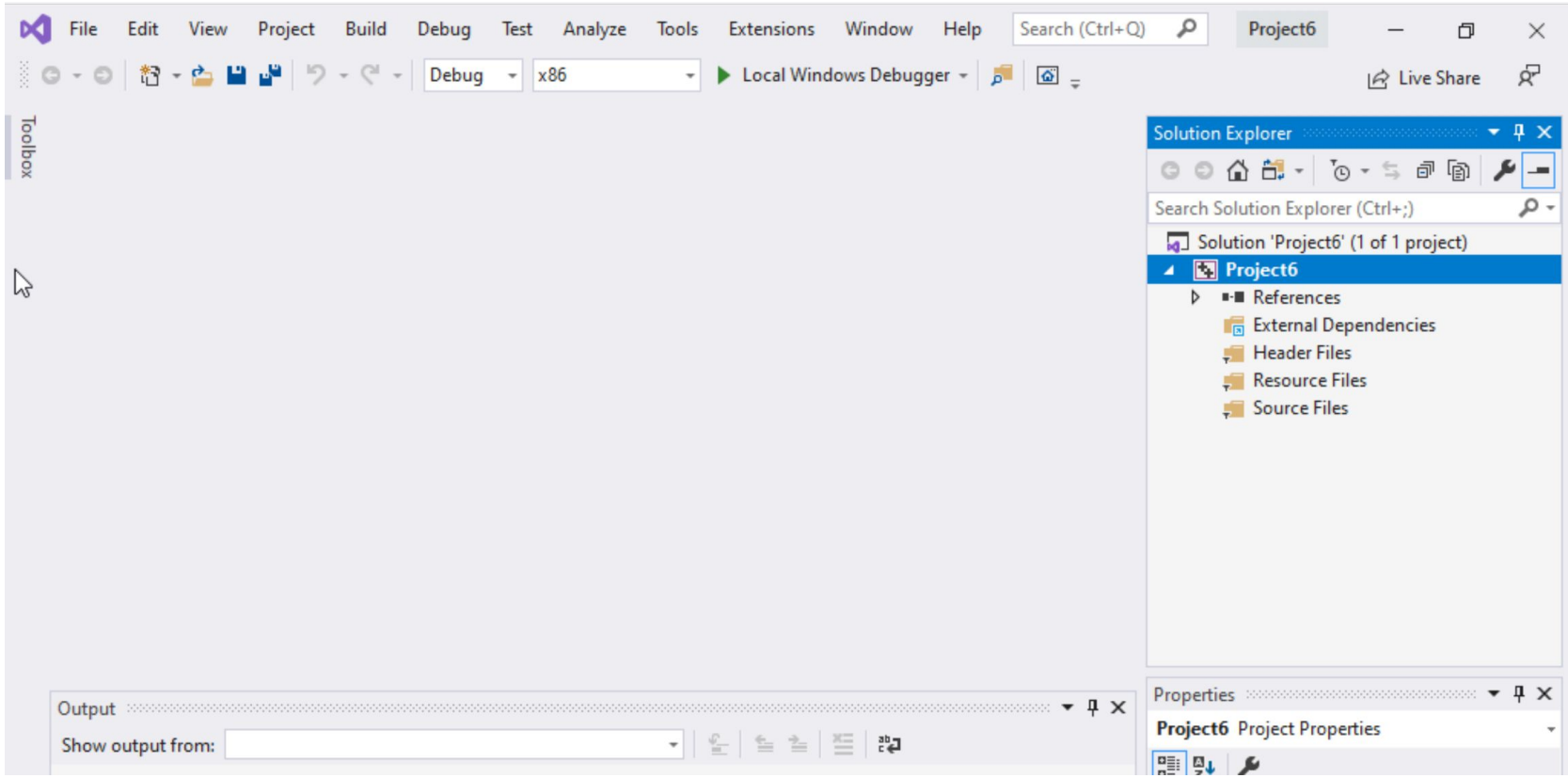


# Step 1: Create a project (4)

Delete the

Following folders:

- Header files
- Resources Files, and
- Source Files



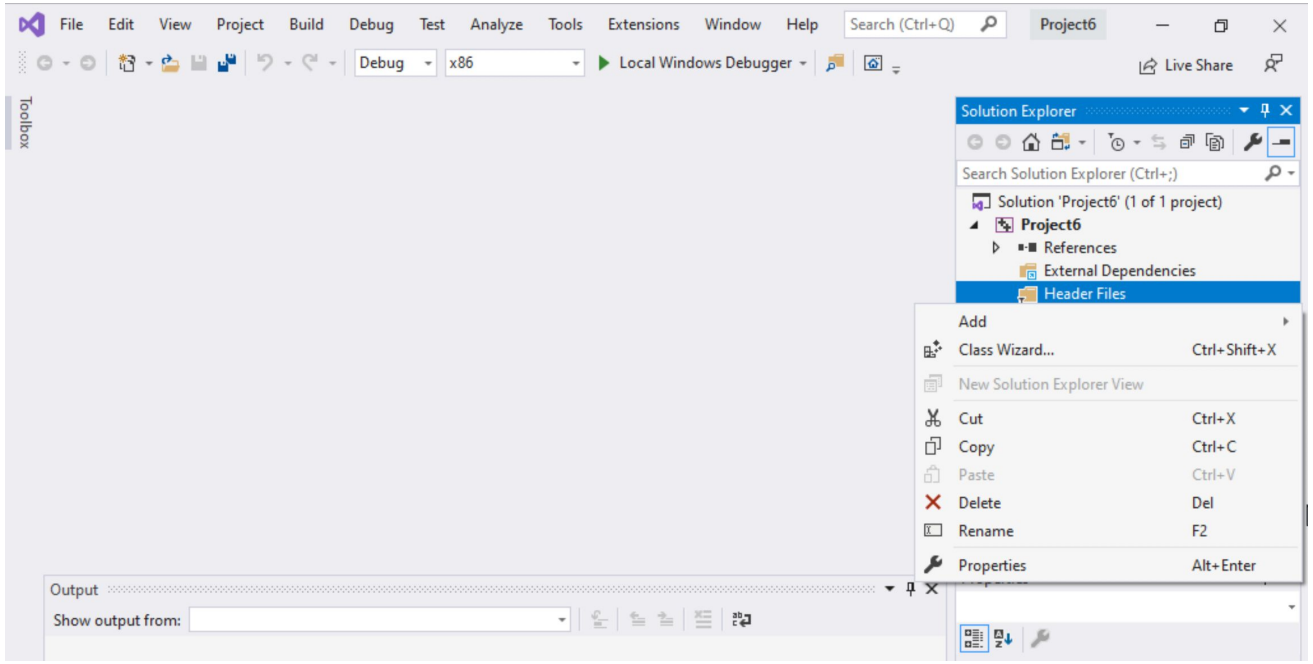
# Step 1: Create a project (5)

To delete :

Select the folders

Right click on it

Select delete



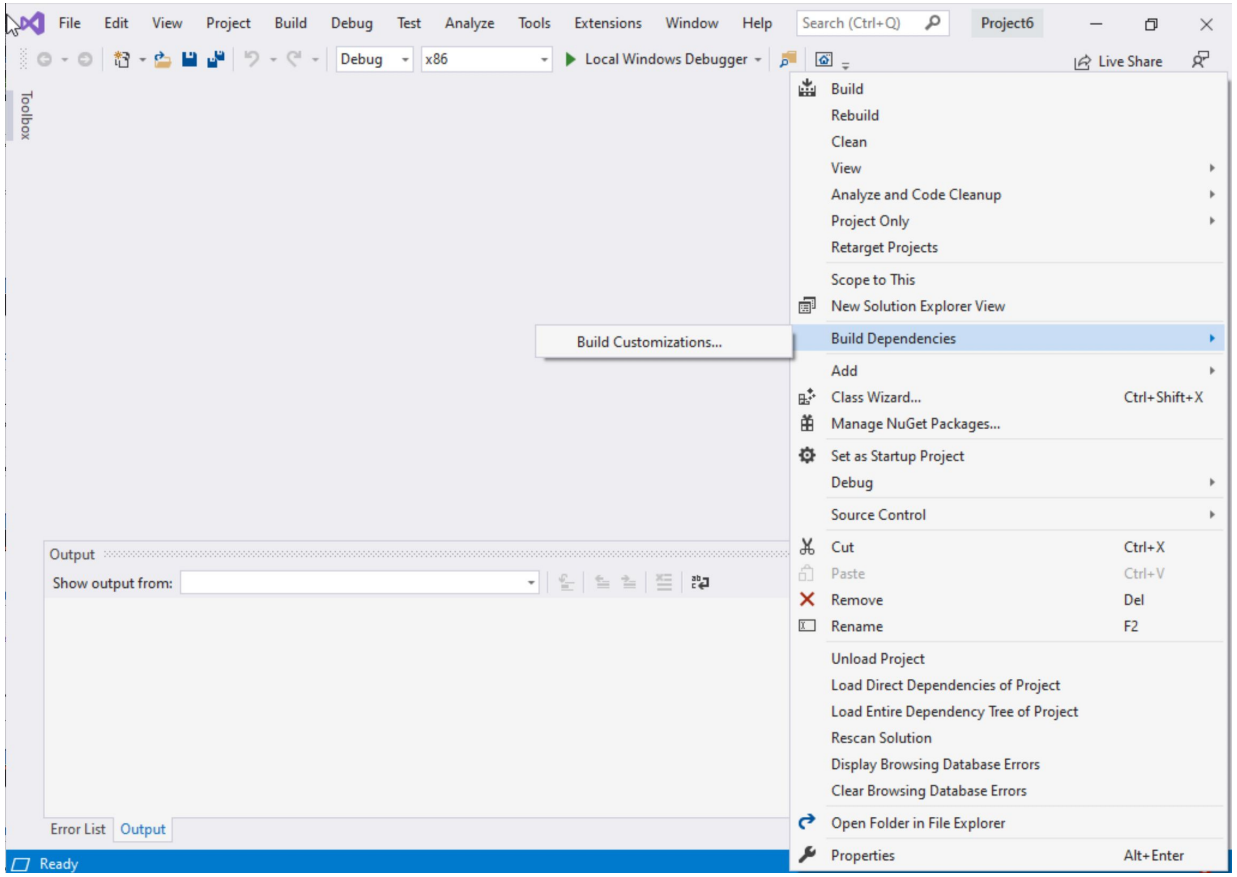
# Step 1: Create a project (6)

Select Project Name on solution explorer

Right click on it

Go to Build Dependencies

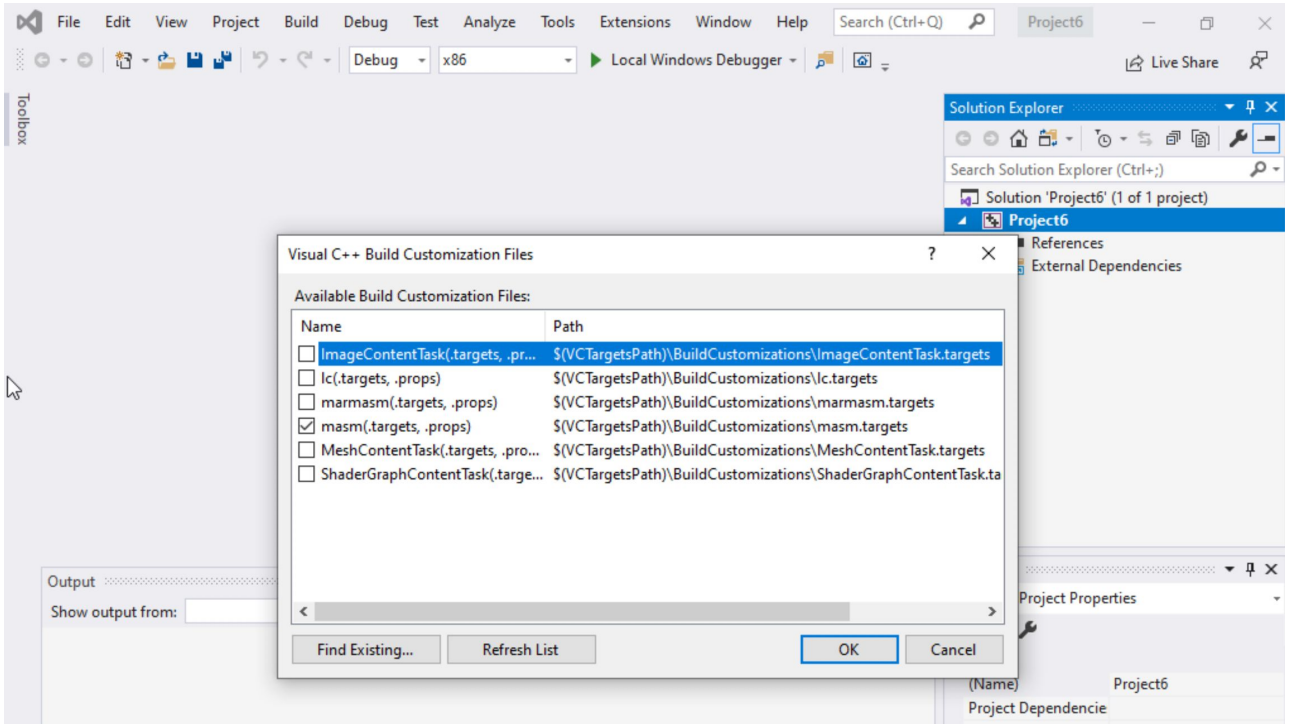
Click on Build Customizations



# Step 1: Create a project (7)

Select masm(.target, .props)

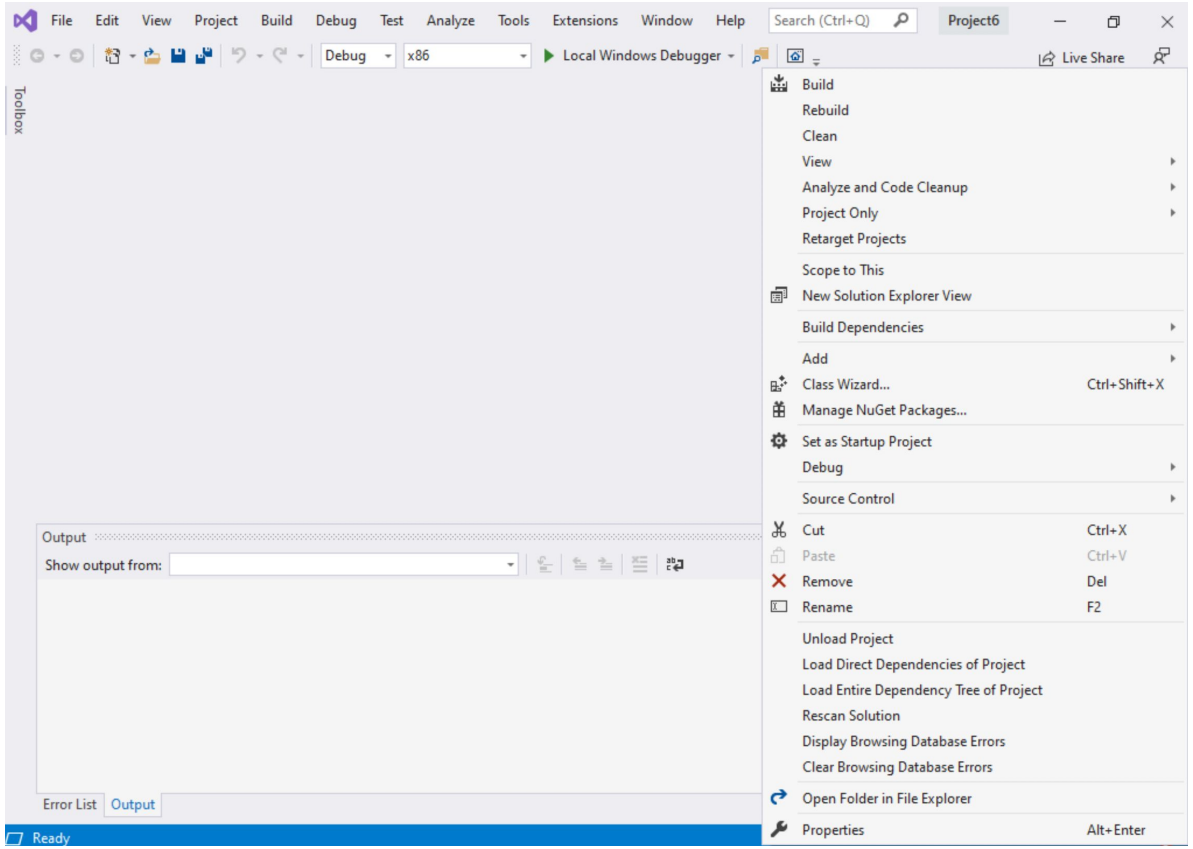
Click ok



# Step 1: Create a project (8)

Right click on the Project name in the solution explorer

Click properties



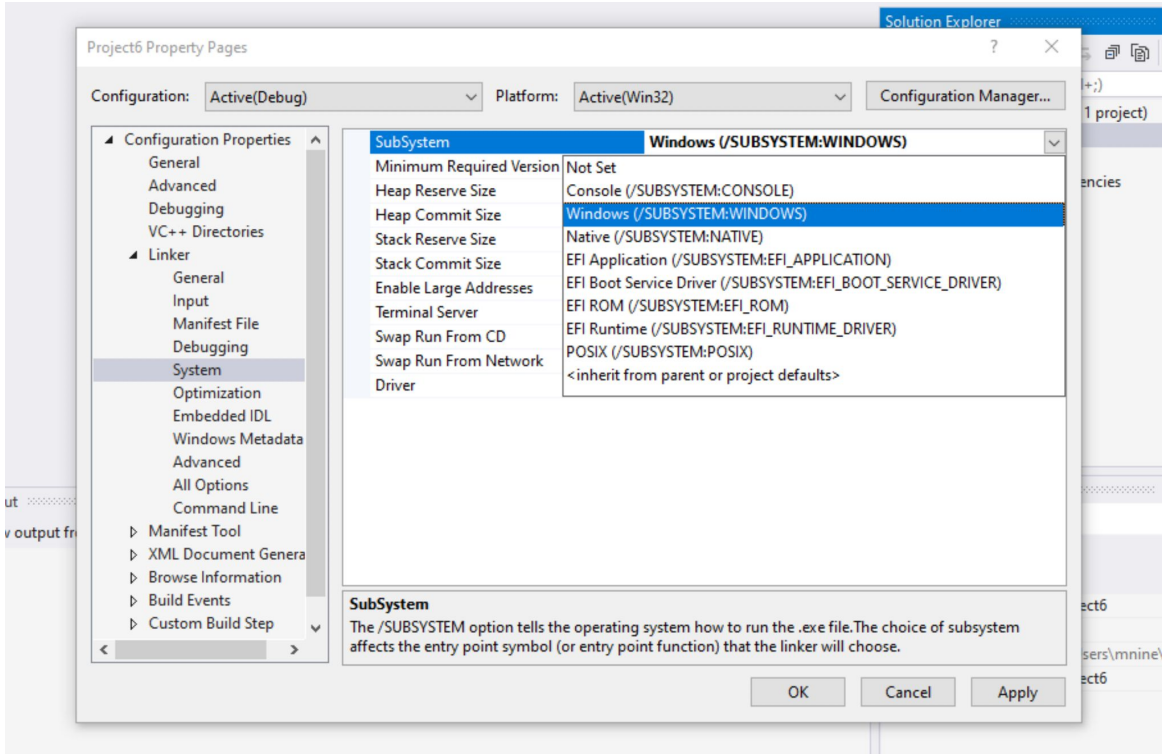
# Step 1: Create a project (9)

Expand the ‘Linker’

Select ’System’

Select Windows(/SUBSYSTEM:WINDOWS)

Click OK



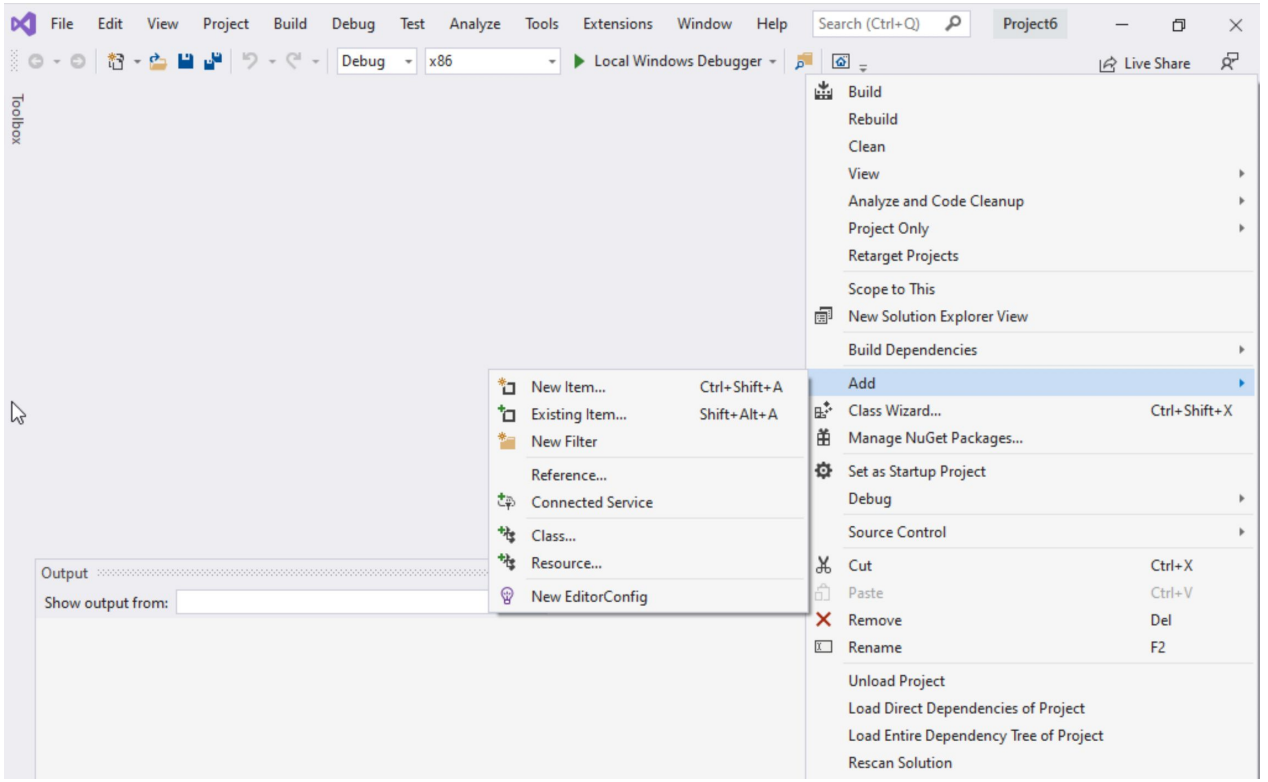
# Step 1: Create a project (10)

Select Project name on solution explorer

Right click on it

Expand Add

Choose New Item

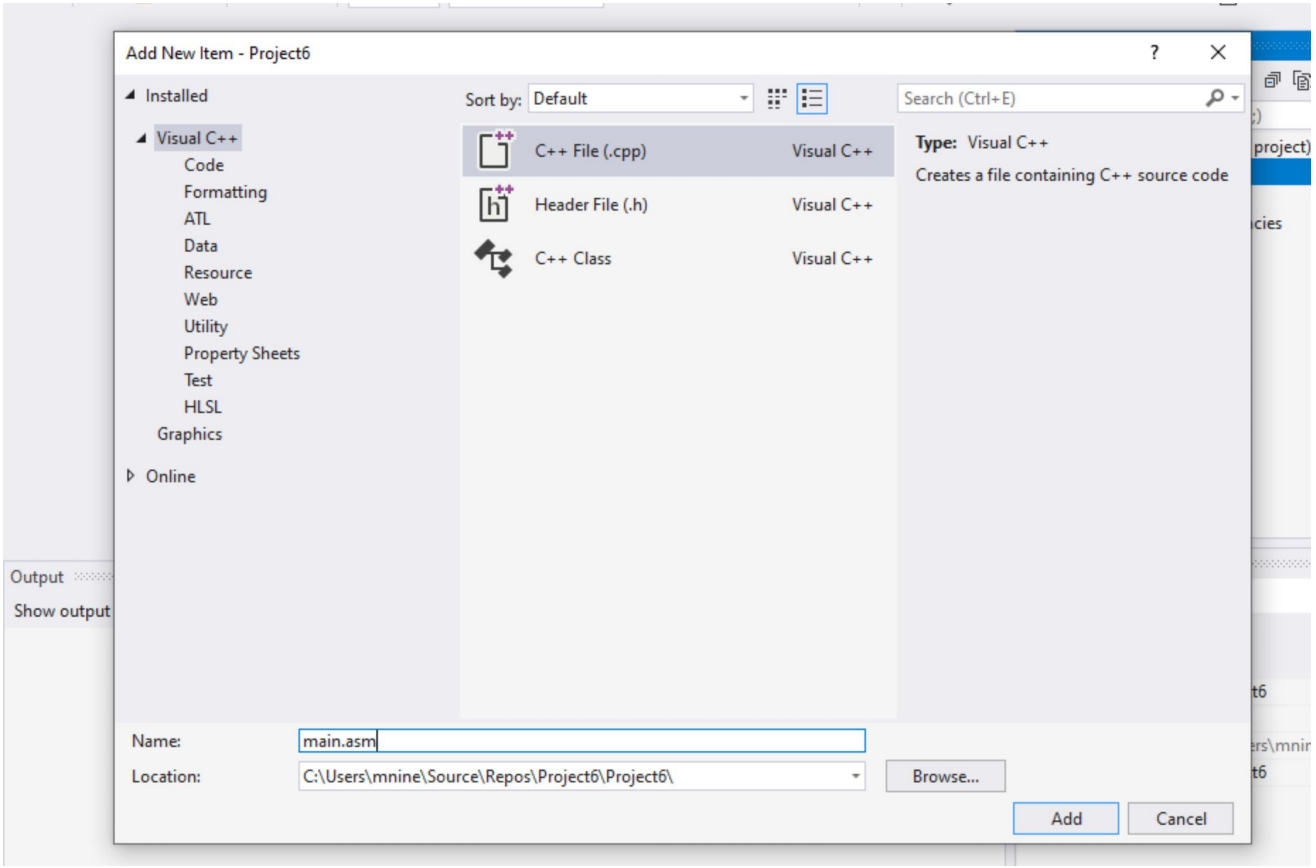


# Step 1: Create a project (11)

Select C++ File(.cpp)

Name: main.asm

Click Add



# Step 1: Create a project (12)

Select main.asm

Add your code

In the main.asm File.