Basic C Programming

Azlan Mukhtar
CYSECA Solutions Sdn Bhd

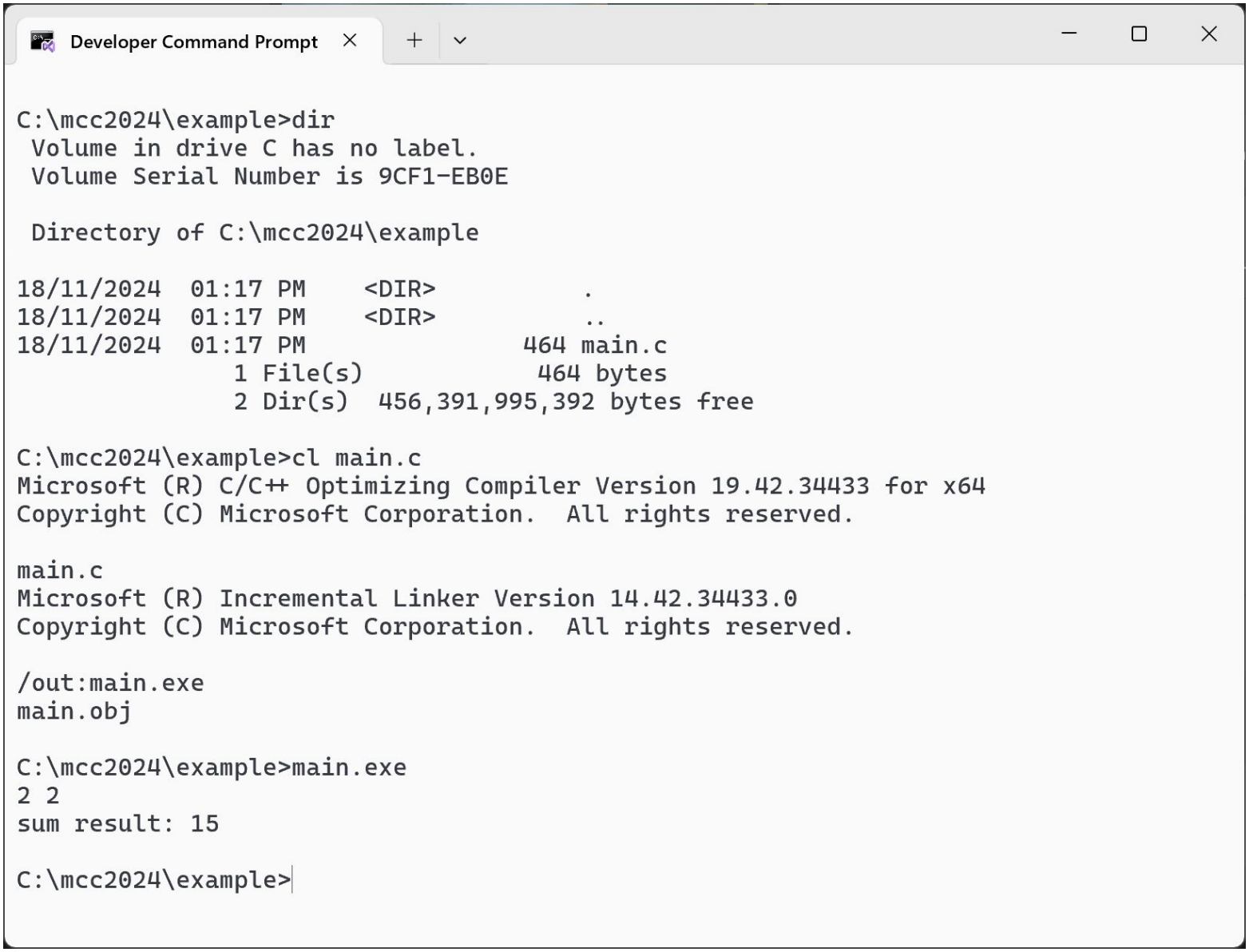
Topics

- Visual C++ command line compiler
- Visual Studio IDE for C/C++
- Useful compiler commands/switches
- Basic C
  - Variable
  - Procedure/Function
  - Array and string
  - Structure
  - Pointer
- Hands-on exercise

Visual C++ command line compiler

- Run **Developer Command Prompt** (64 bit) or **x86 Native Tools Command Prompt** (32 bit) from Windows start menu
- Basic compilation
  - cl.exe main.c
- Produce executable plus assembly listing
  - cl.exe /Famain.asm main.c
- With optimization, listing, debug info (pdb) and link options
  - cl.exe /O2 /Zi /FA source.c /link user32.lib /out:output.exe
- cl.exe options
  - https://msdn.microsoft.com/en-us/library/9s7c9wdw.aspx
- Debugging an exe
  - https://msdn.microsoft.com/en-us/library/0bxe8ytt.aspx

Basic compilation



With assembly listing

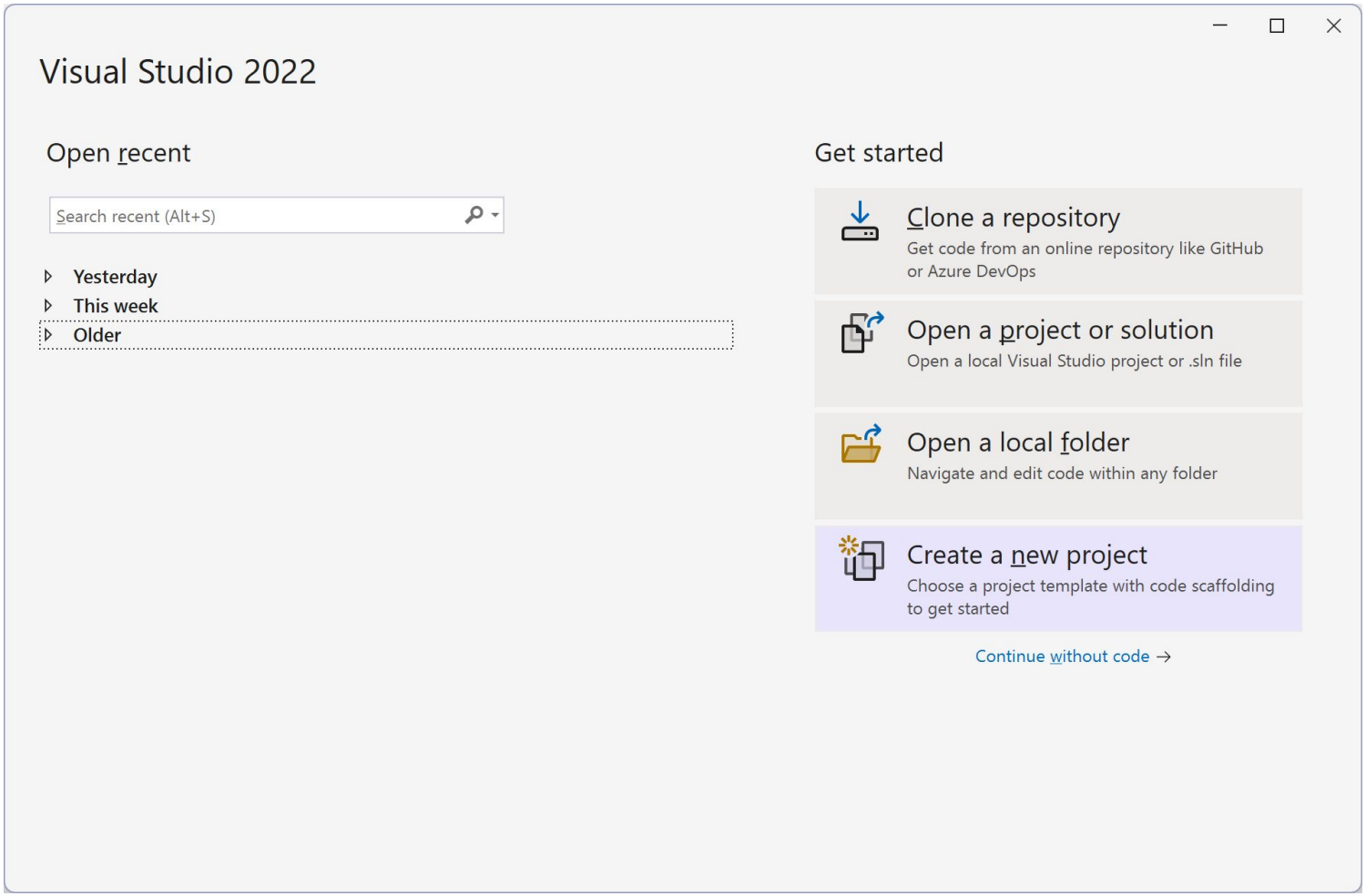
Developer Command Prompt

```
C:\mcc2024\example>cl.exe /Famain.asm main.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.42.34433 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

main.c
Microsoft (R) Incremental Linker Version 14.42.34433.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:main.exe
main.obj

C:\mcc2024\example>cat main.asm
```

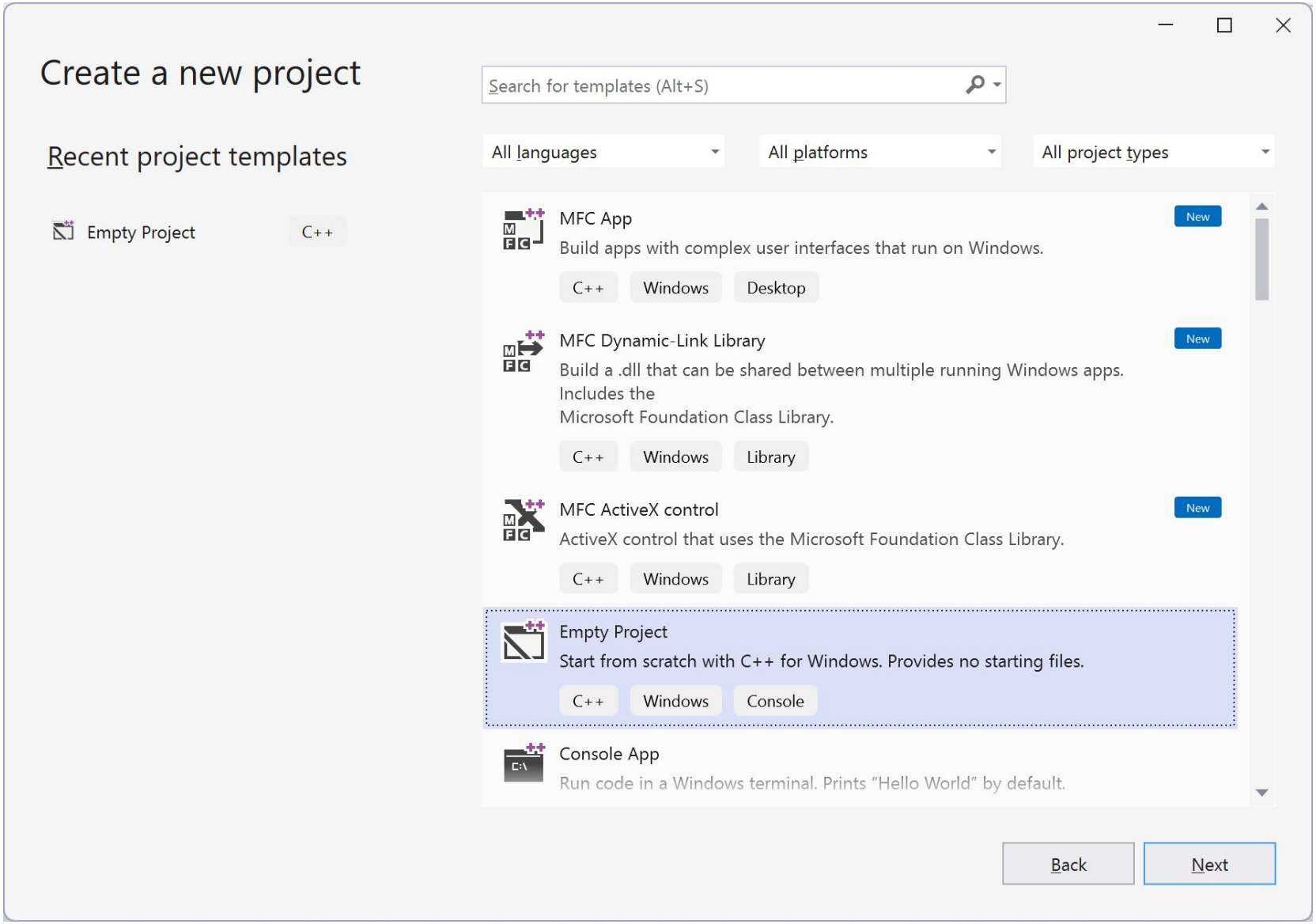
```
       File: main.asm
```

```
   1   ; Listing generated by Microsoft (R) Optimizing Compiler Version 19.42.34
       433.0
   2
   3   include listing.inc
   4
   5   INCLUDELIB LIBCMT
   6   INCLUDELIB OLDNAMES
   7
   8   PUBLIC  global_var1
   9   PUBLIC  global_result
  10   _BSS    SEGMENT
  11   global_result DD 01H DUP (?)
  12   _BSS    ENDS
  13   _DATA   SEGMENT
  14   global_var1 DD  05H
```

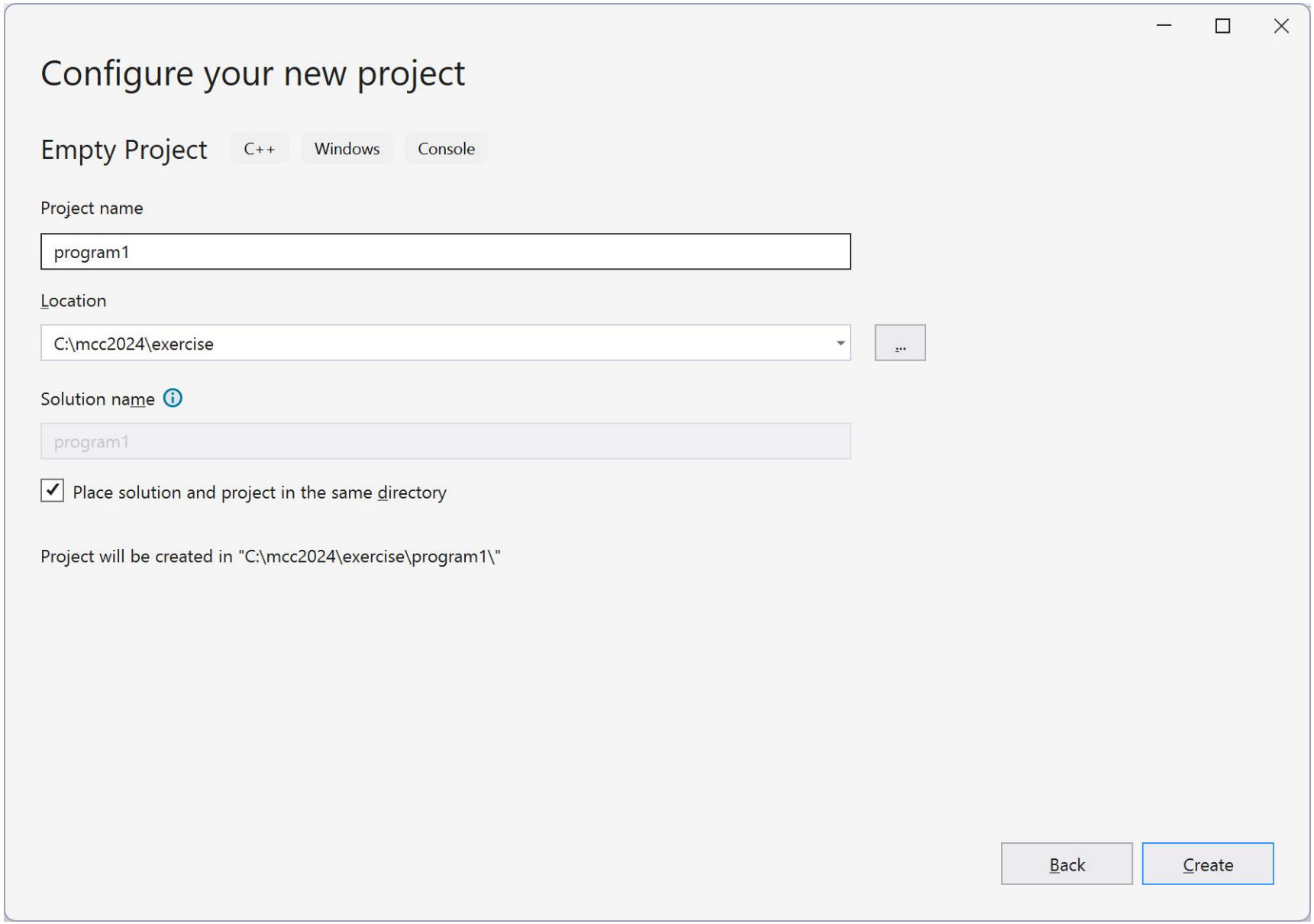
Visual Studio IDE for C/C++



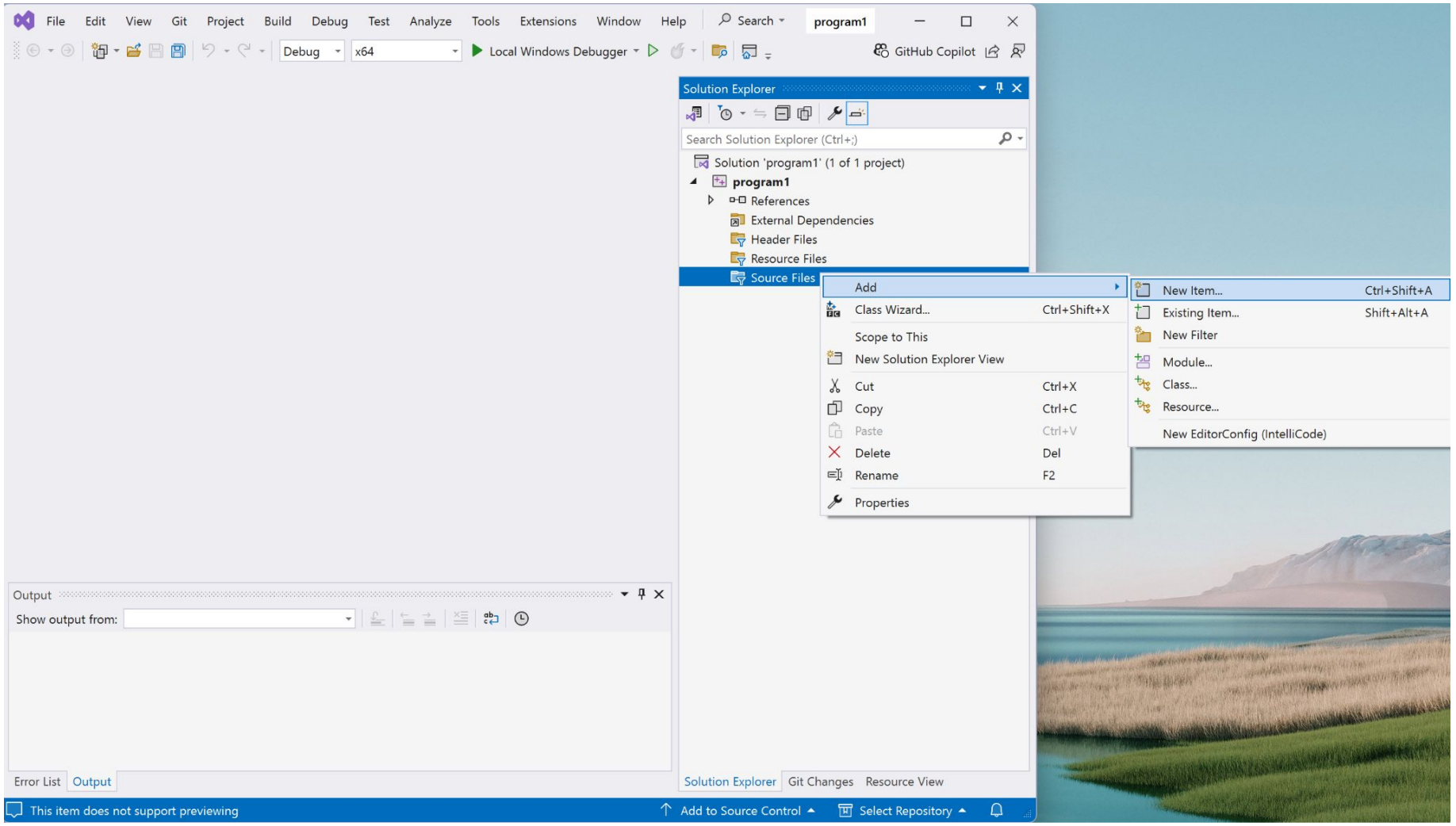
Create empty project



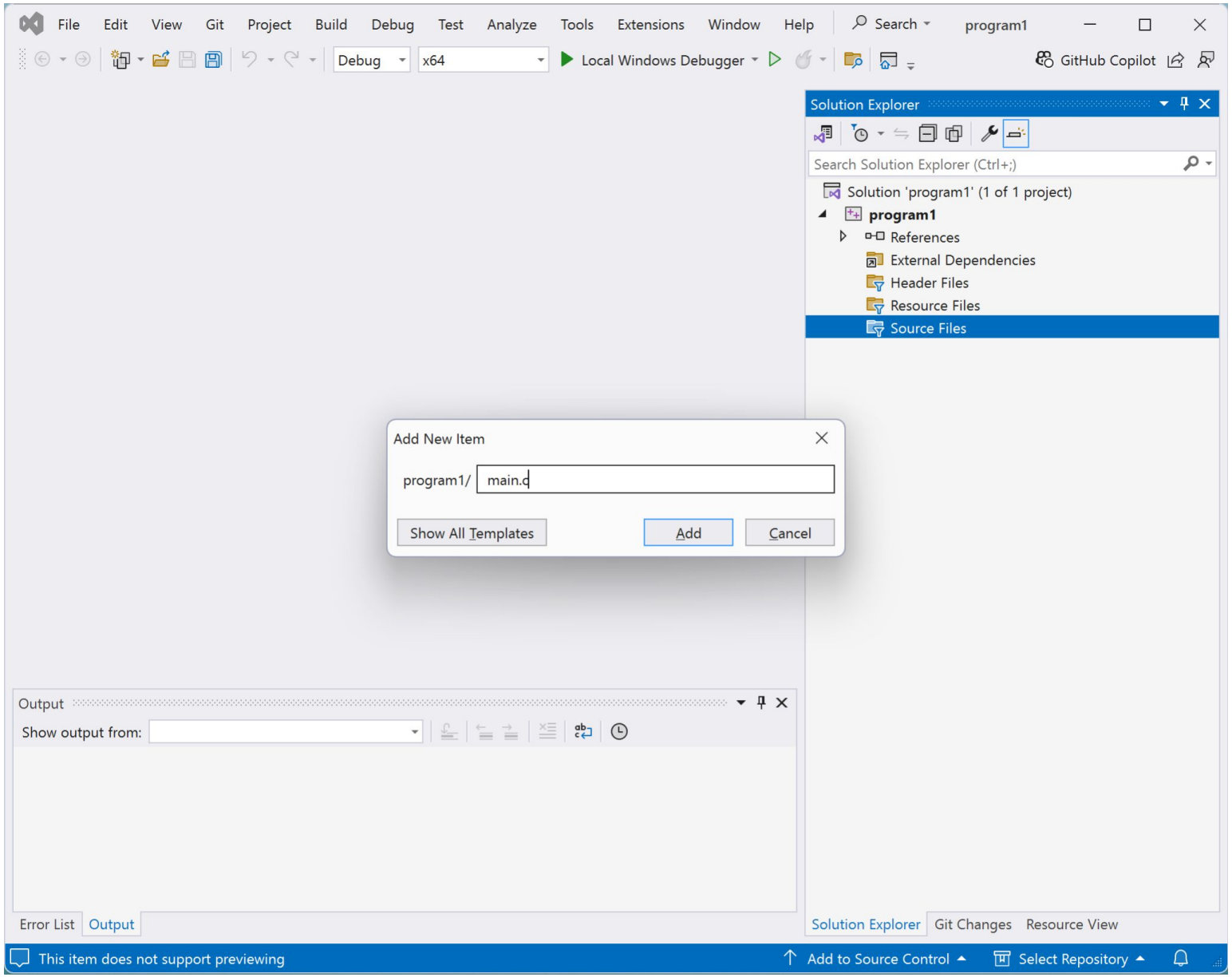
Set program name



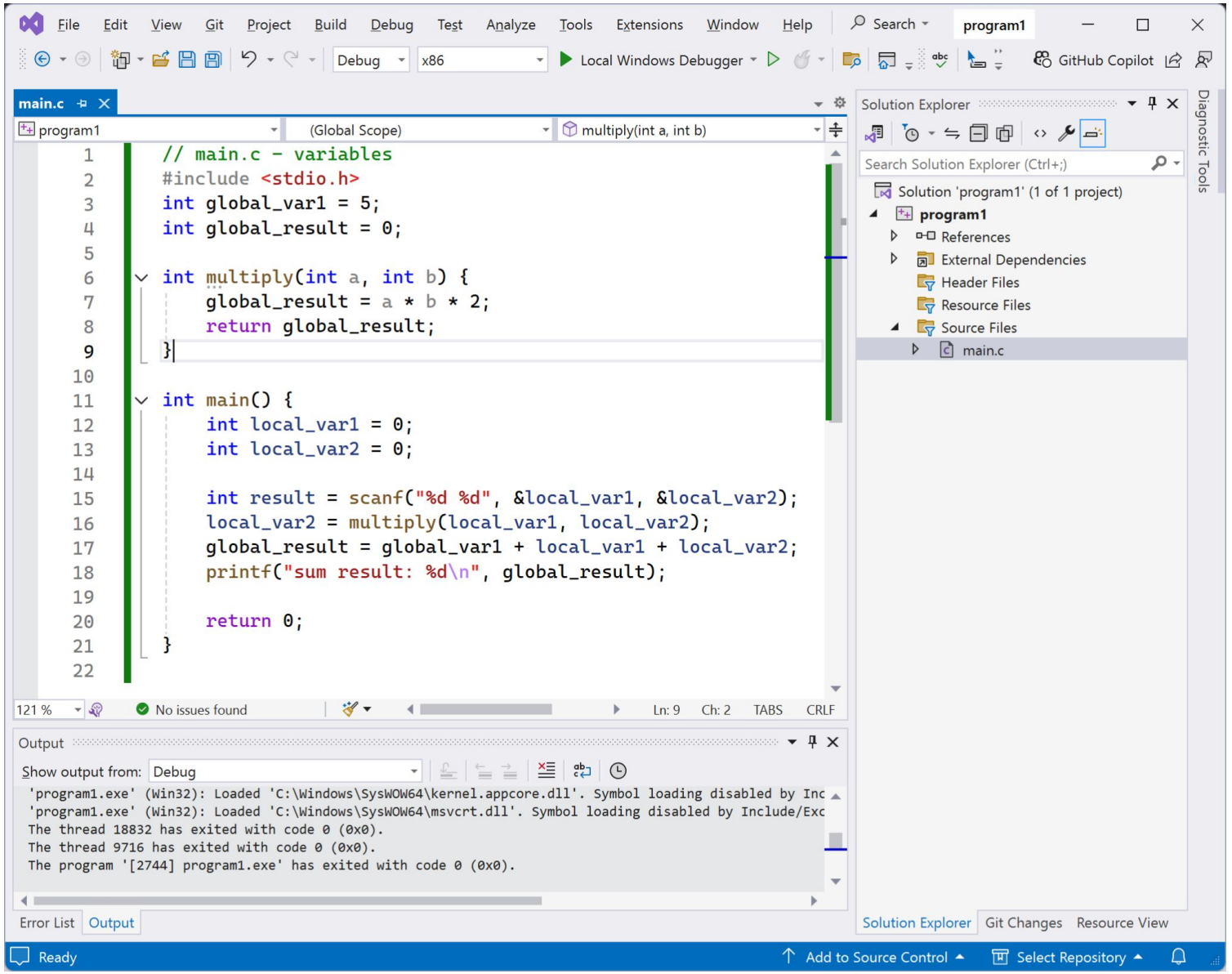
Add new source file



Input filename



Paste source code from the next slide



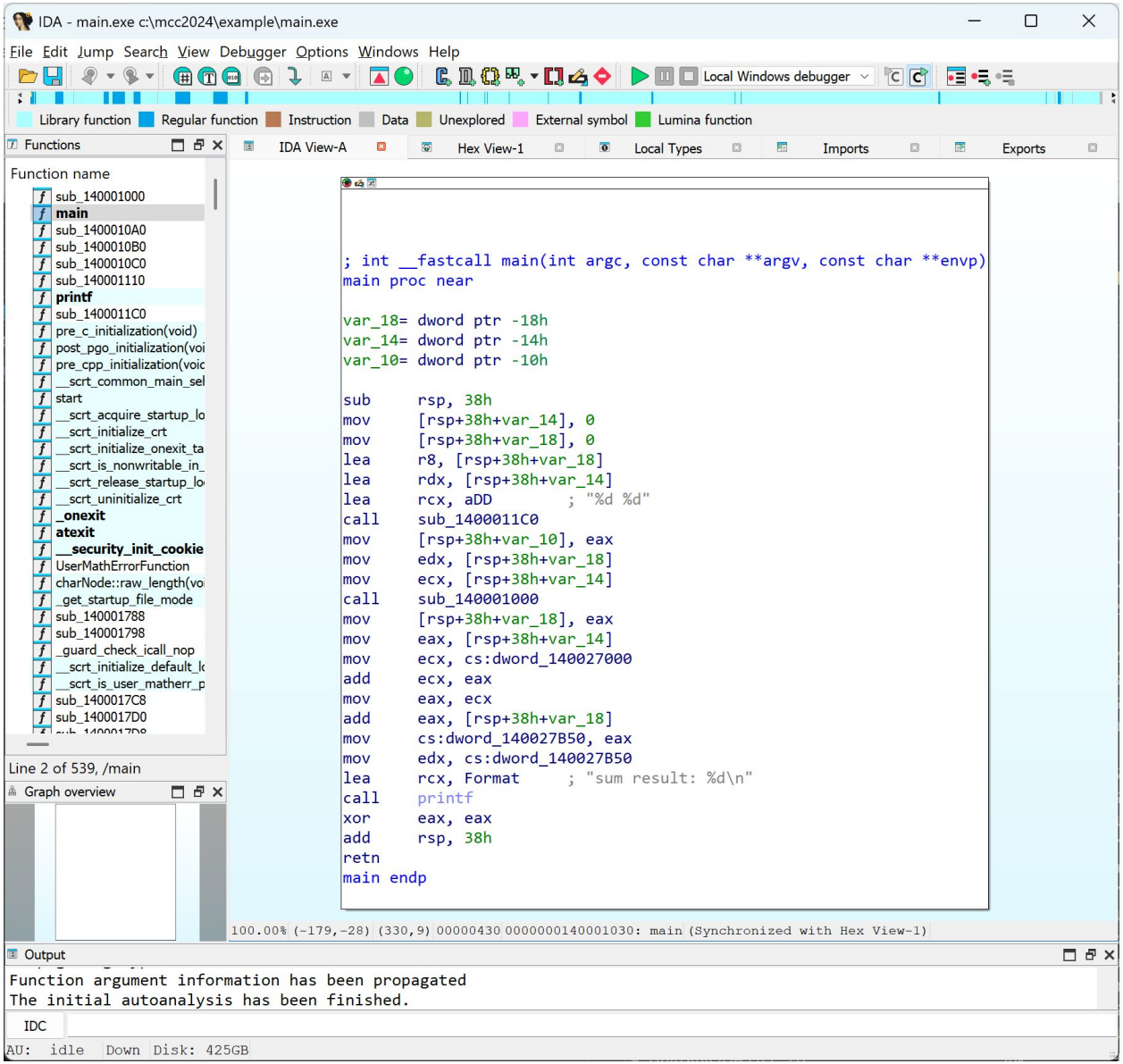
Variables: Example code

```
// main.c - variables
#include <stdio.h>
int global_var1 = 5;
int global_result = 0;

int multiply(int a, int b) {
    global_result = a * b * 2;
    return global_result;
}

int main() {
    int local_var1 = 0;
    int local_var2 = 0;

    int result = scanf("%d %d", &local_var1, &local_var2);
    local_var2 = multiply(local_var1, local_var2);
    global_result = global_var1 + local_var1 + local_var2;
    printf("sum result: %d\n", global_result);

    return 0;
}
```

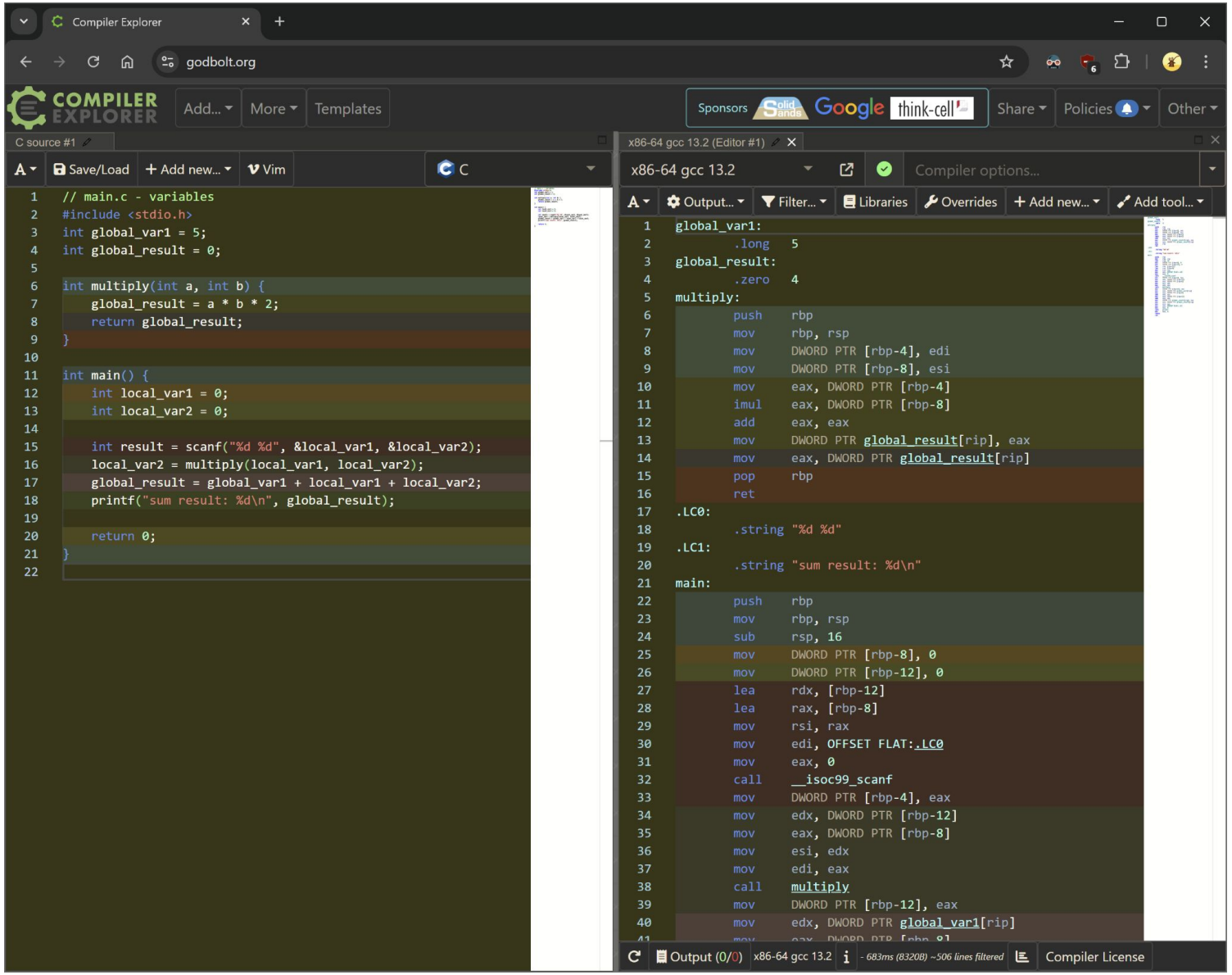
Variable Scopes

- Local variable
  - Dynamically allocated on stack memory
  - Temporarily available
- Static local variable
  - Usually located inside memory section
  - Initialization occurs once and then the variable retains its value
  - Only accessible from within the function
- Global variable
  - Usually located inside memory section
  - Static location, always accessible from anywhere

Inspecting compiler result



Alternative - godbolt.org



Procedure Call/Function

- Procedural programming is derived from structured programming, based upon the concept of the procedure call
- Procedures, also known as routines, subroutines, methods, or functions
- Most CPU architecture supports procedure call, if not all
- The x86 processor supports procedure using two instructions:
  - CALL
  - RET (return)
- The procedure stack, commonly referred to simply as “the stack”, will save the state of the calling procedure, **pass parameters** to the called procedure, and **store local variables** for the currently executing procedure

Procedure: Example code

```
// function.c
int calculate2(int a, int b) {
    int result = a + b * 2;
    return result;
}

int calculate(int a, int b)  {
    int result = a + b -2;
    result += calculate2(result, 10);
    return result;
}

int main() {
    int sum = 0;
    int result = calculate(5,6);
    sum += result;

    return 0;
}
```

Procedure: Parameter

- Parameters/arguments
- Argument passing:
  - Value
  - Reference
  - Address (pointer)
- Return value

Calling Convention: CDECL

```
// C/C++ codes

int __cdecl MyFunction1(int a, int b)
{
      return a + b;
}
int x = MyFunction2(2, 3);
```

```
;  x86 asm codes

 _MyFunction1:
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret

Start:
push 3
push 2
call _MyFunction1
add esp, 8 ; clean up
```

prologue

epilogue

- Parameters are passed from right to left
- The stack is cleaned up by the **caller**
- Return value in **eax**

Calling Convention: STDCALL

```
// C/C++ codes

int __stdcall MyFunction2(int a, int b)
{
   return a + b;
}

int y = MyFunction2(2, 3);
```

```
; x86 asm codes

:_MyFunction2@8
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret 8       ; clean up

Start:
push 3
push 2
call _MyFunction2@8
```

- Parameters are passed from right to left
- The stack is cleaned up by the **callee**
- Return value in **eax**

Calling Convention: FASTCALL

```
// C/C++ codes

int _fastcall MyFunction3(int a, int b)
{
   return a + b;
}

z = MyFunction3(2, 3);
```

```
; x86 asm codes

:@MyFunction3@8
push ebp
mov eax, ecx
add eax, edx
pop ebp
ret

mov edx, 3
mov ecx, 2
call @MyFunction3@8
```

- Parameters are passed using two registers, then push to stack
- The stack is cleaned up by the **callee**
- Return value in **eax**

Calling Convention: Microsoft x64

Parameter type	fifth and higher	fourth	third	second	leftmost
floating-point	stack	XMM3	XMM2	XMM1	XMM0
integer	stack	R9	R8	RDX	RCX
Aggregates (8, 16, 32, or 64 bits) and `__m64`	stack	R9	R8	RDX	RCX
Other aggregates, as pointers	stack	R9	R8	RDX	RCX
`__m128`, as a pointer	stack	R9	R8	RDX	RCX

```
func1(int a, int b, int c, int d, int e, int f);
// a in RCX, b in RDX, c in R8, d in R9, f then e pushed on stack
```

https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention

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

Strings

- Null terminated const strings
  - `char *str = "my strings";`
- Null terminated strings array
  - `char str[] = "my strings";`
- Null terminated byte array
  - `char str[] = {'m','y',' ','s','t', 'r','i','n','g','s', '\x0'};`

Array

- int arr[] = {2, 3, 5, 7 ,11};
- Accessing array
  - Value: arr[2] □ 5
  - Address: &arr[2]

Pointer

- A variable to hold a memory address/location
- How does it look
- Passing pointer around
- Pointer arithmetic
- Void pointer
- Dereferencing

Pointer: Example

```
// pointer.c - pointer use case
#include <stdio.h>
#include <stdlib.h>

int main() {
   int *ptr;
   int n, i;

   printf("Enter number of elements:");
   scanf_s("%d", &n);
   printf("Entered number of elements: %d\n", n);

   ptr = (int*)malloc(n * sizeof(int));
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else {
      printf("Memory successfully allocated using malloc.\n");
      for (i = 0; i < n; ++i) {
         ptr[i] = i + 1;
      }
      printf("The elements of the array are: ");
      for (i = 0; i < n; ++i) {
         printf("%d, ", ptr[i]);
      }
   }
   return 0;
}
```

https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Pointer: Handle

- Standard library
  - FILE*
- Windows API
  - HANDLE
  - HINSTANCE
  - HWND
  - HPROCESS

Pointer is just a variable to hold a memory address

Dynamic Memory

- malloc and free (C)
- new and delete (C++)
- HeapAlloc and HeapFree (Windows)
- VirtualAlloc and VirtualFree (Windows)
- Exercise:
  - Load a big file into memory, encrypt using XOR, then save the encrypted data to disk.

Structure

- When to use structure
- Passing structure to functions
- Exercise:
  - Parse PE File by using structs from winnt.h and print out AddressOfEntryPoint

Hands-on Exercise

- Write your own Crackme CTF

References

- https://www.google.com/search?q=c+programming+tutorial
- Procedural Programming
  - http://en.wikipedia.org/wiki/Procedural_programming
- Functions and Stack Frame
  - http://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames
- C Pointers
  - https://www.programiz.com/c-programming/c-pointers