# Table of Contents

**1. Lab 2: Configuration of the Visual Studio Community 2019 for dll libraries written in MASM.** ..... 2
***1.1. Creating the project with dll library written in assembler*** ..... 2
1.1.1. Creating of the new project ..... 2
1.1.2. Adding the assembly part to the solution ..... 4
1.1.3. Setting dependencies in the solution ..... 7
1.1.4. Changing the type from a C++ library to an assembler one ..... 9
1.1.5. Setting the compilation mode ..... 15
1.1.6. Final setup of the assembly library ..... 22
***1.2. Using a dll library written in assembler*** ..... 26
1.2.1. dll library written in assembler ..... 26
1.2.2. Calling functions in dll from C++ program ..... 27
1.2.3. Task to be completed in the laboratory ..... 28

# 1. Lab 2: Configuration of the Visual Studio Community 2019 for dll libraries written in MASM.

## 1.1. Creating the project with dll library written in assembler

In the beginning, we can create a project in which the main program will be written in HLL (High-Level Language) like C++, and the library written in assembler will be linked dynamically (dll).



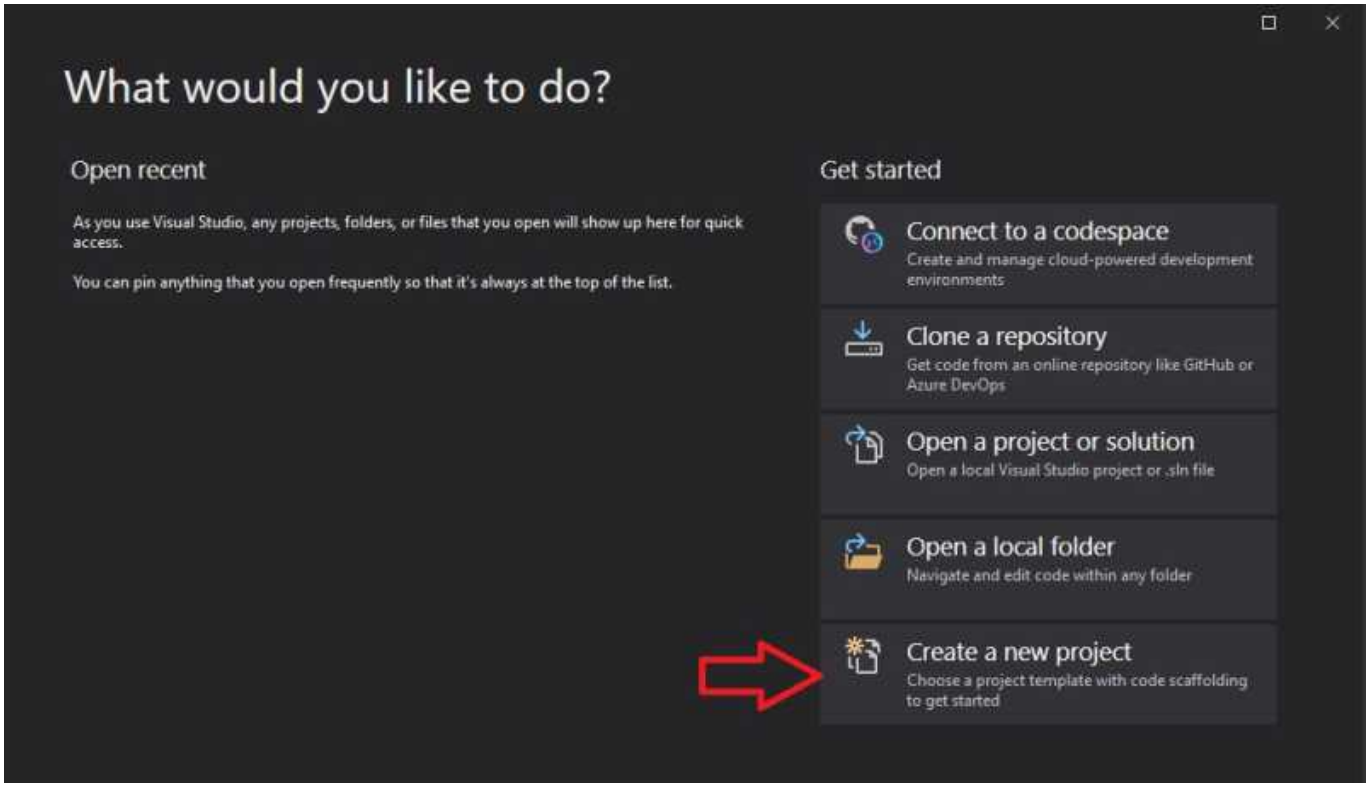
Creation of libraries written in assembly language, with functions called from the application written in a chosen high-level language is currently the most common assembler use.



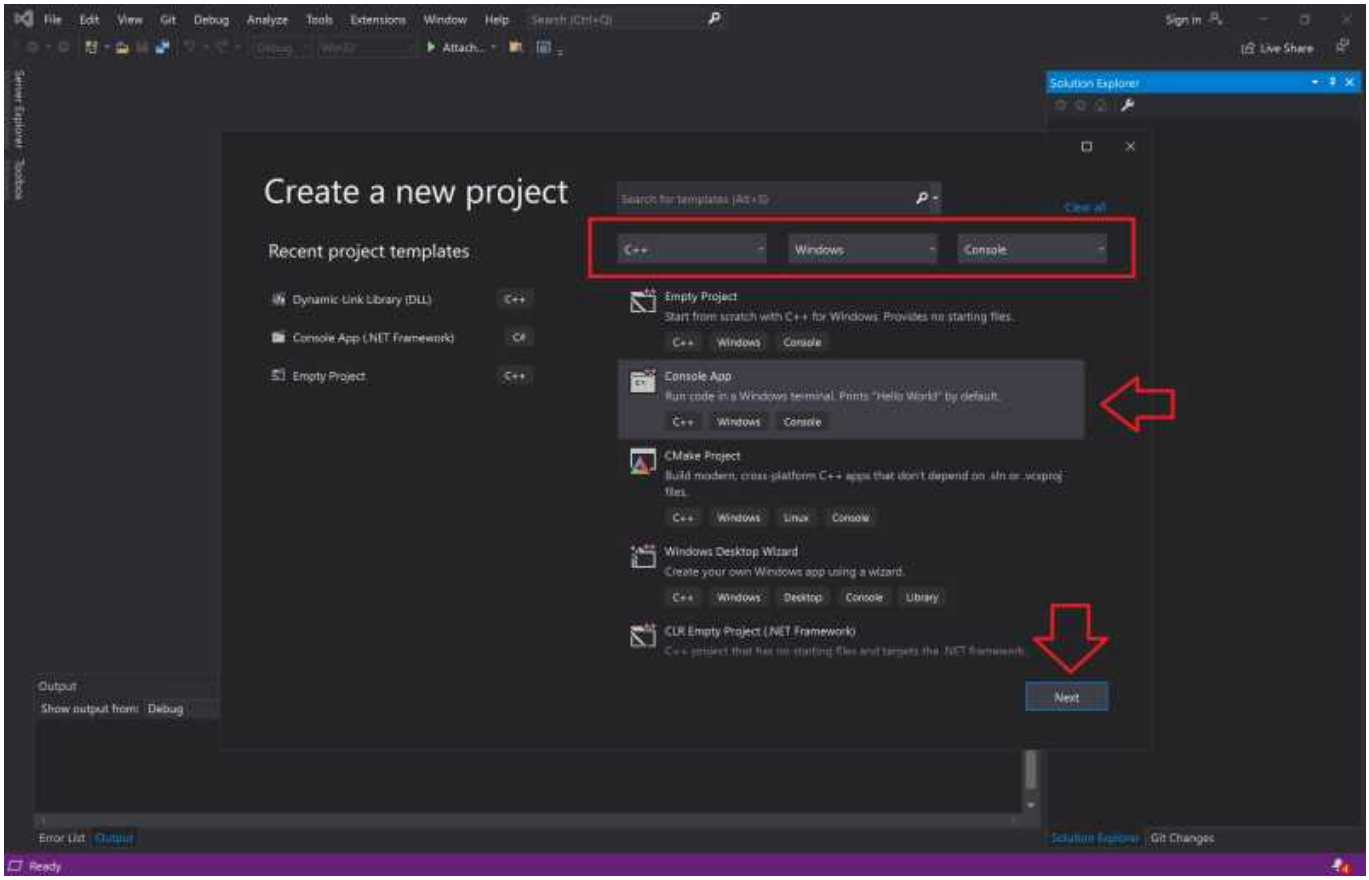
The configuration process presented in the following slides can slightly differ in other versions of Visual Studio and is identical to the main program written in C++ and C#. The screenshots used are from the DLL configuration for both programming languages, but the differences are usually secondary and do not affect the environment configuration process.

### 1.1.1. Creating of the new project

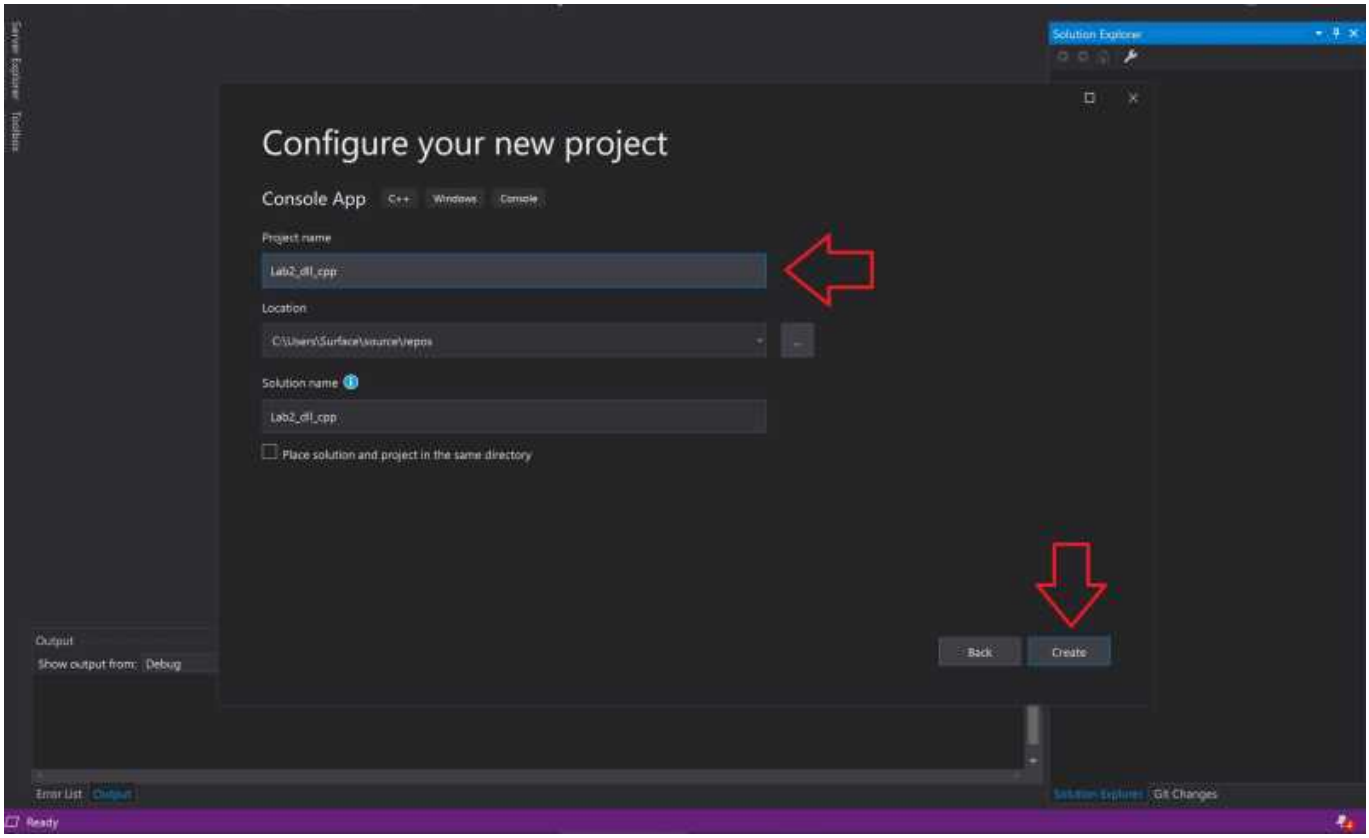
During this laboratory, we'll show how to create and use dll (dynamically linked library) written in assembler in that application written in C++ We start by creating a new C++ project.




We'll create the console application, written in C++, for execution in the Windows operating system.




We need to define that name for the project. We suggest the name “Lab2_dll_cpp”. All paths can be left with default values.

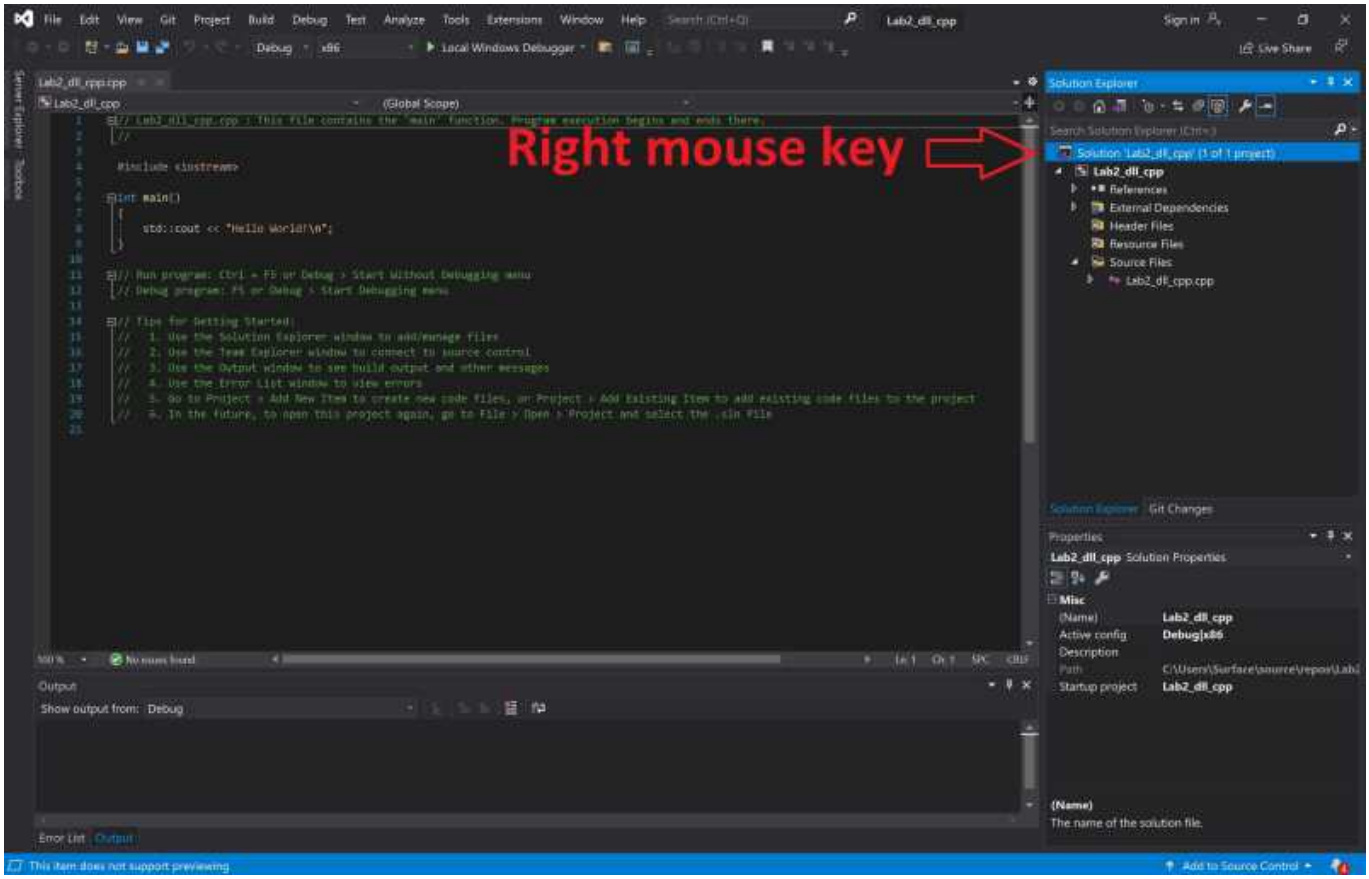


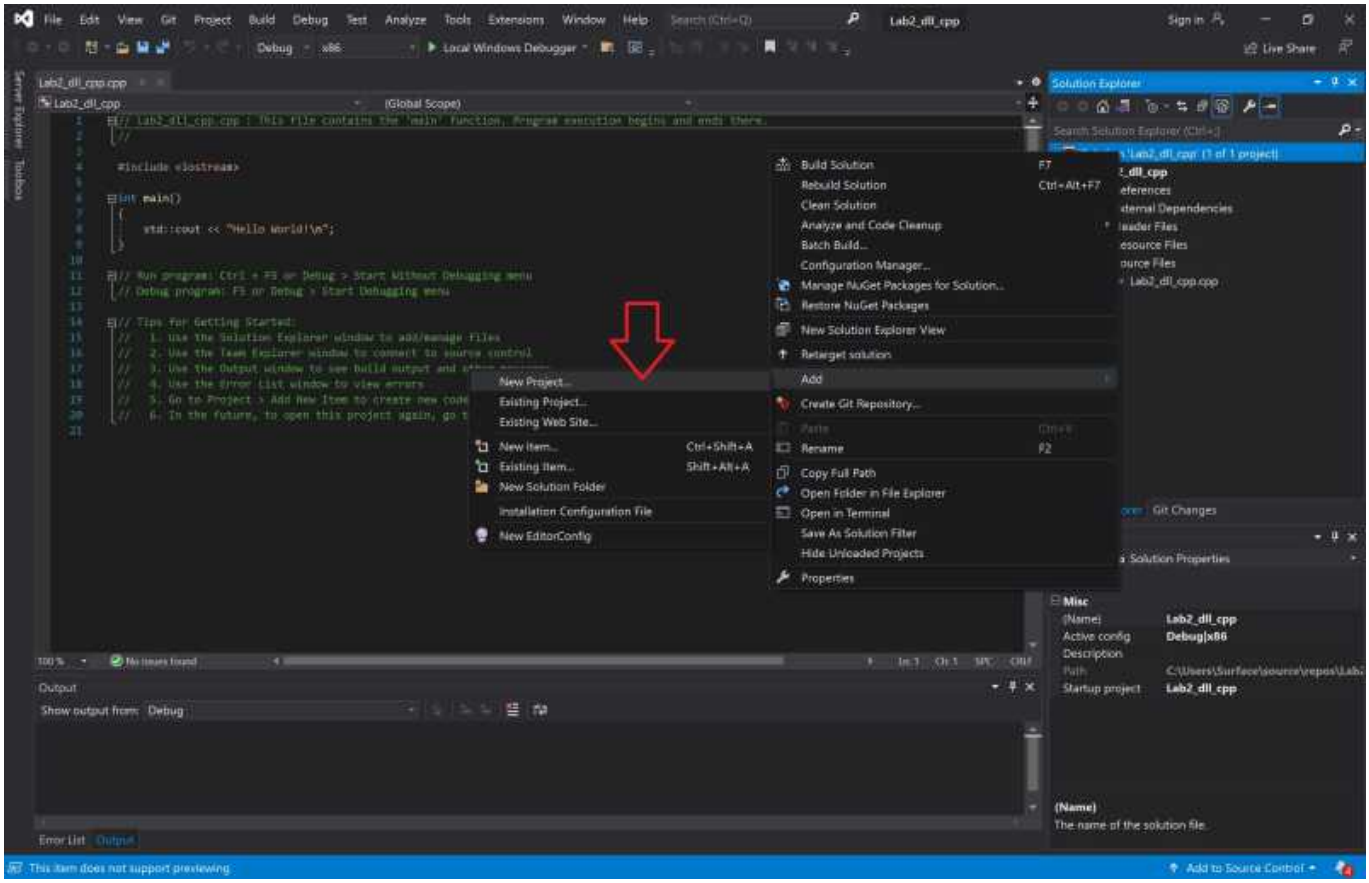



At this stage the solution will be created including the project with the C++ program template.

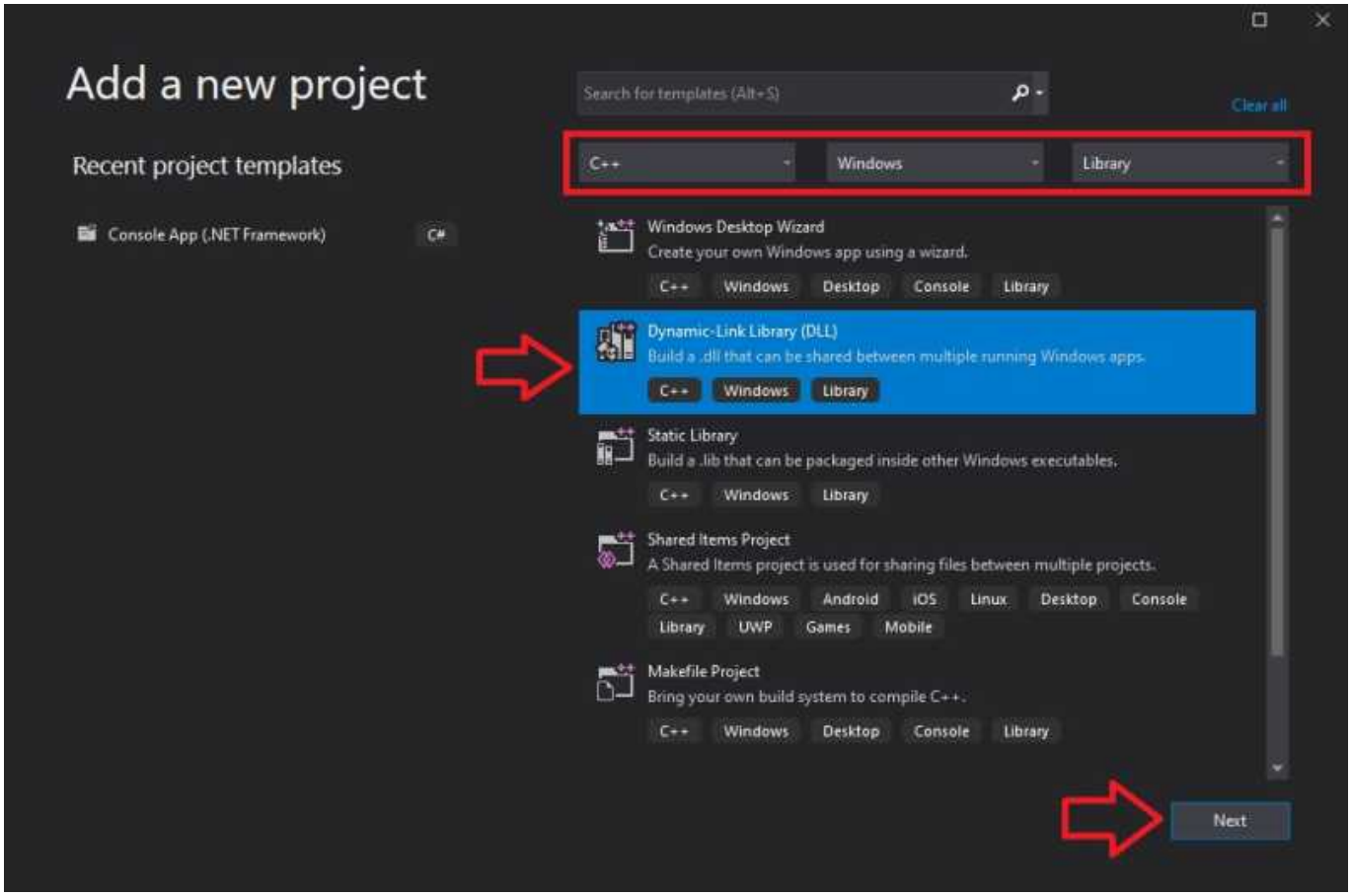
## 1.1.2. Adding the assembly part to the solution

To add a library written in assembler to our solution, we add a new project to the solution. Right-click on the solution and select add new project.

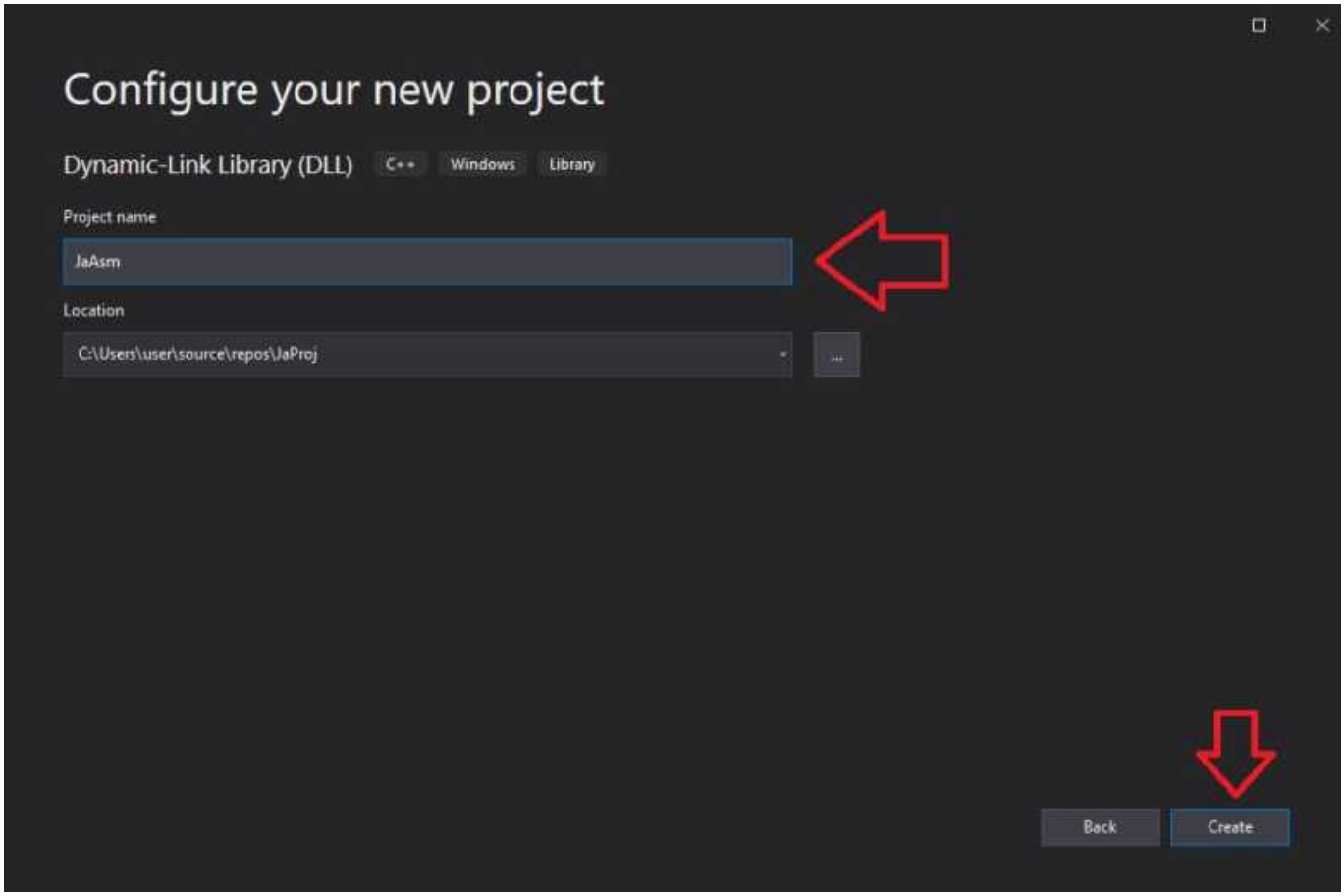


We create our new assembler project as a C++ compatible library project, running under the Windows operating system. From the list of available options, we select the Dynamic-Link Library (DLL).




We name the project e.g. “JaAsm” and create it leaving the default path.

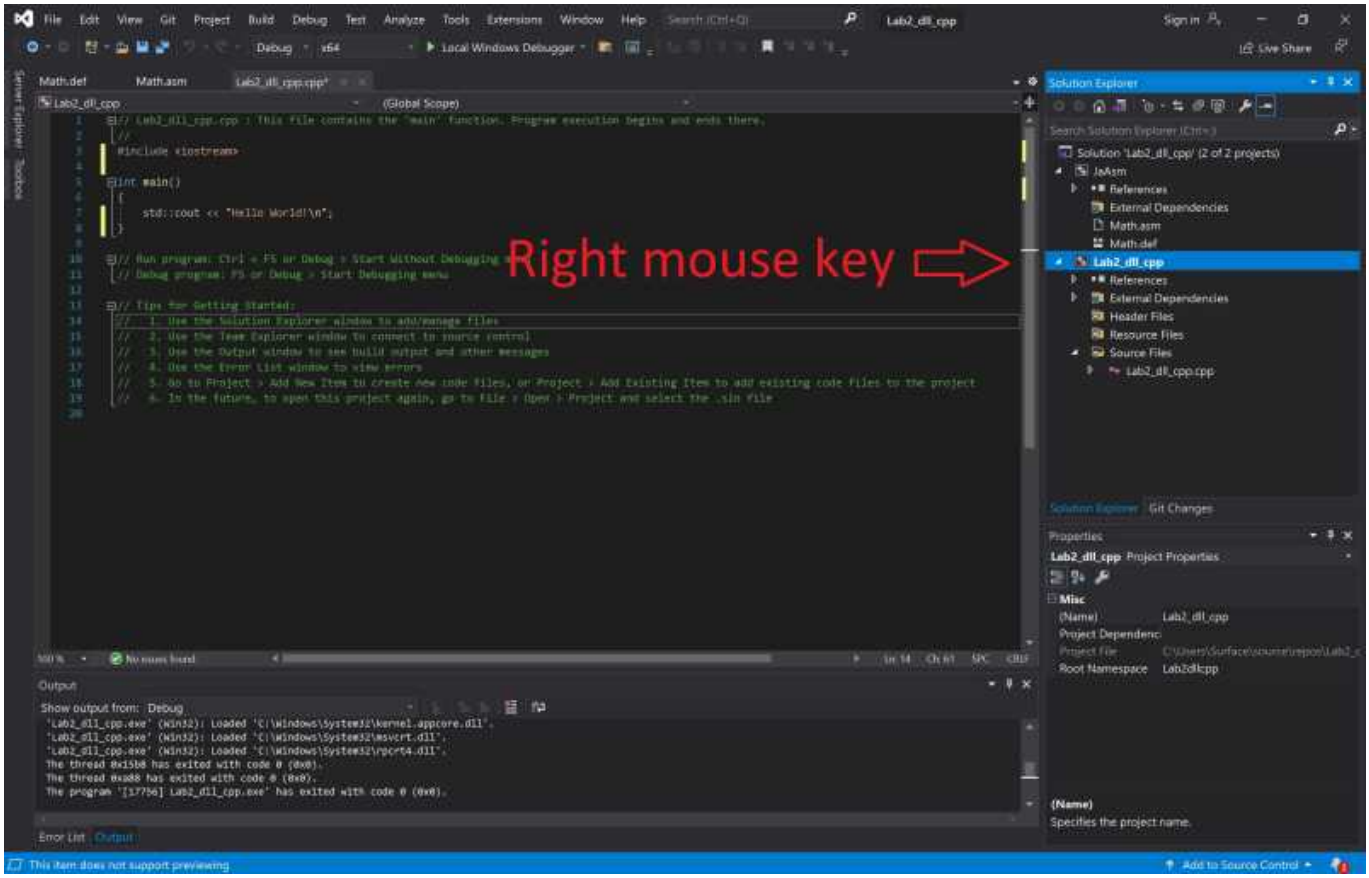


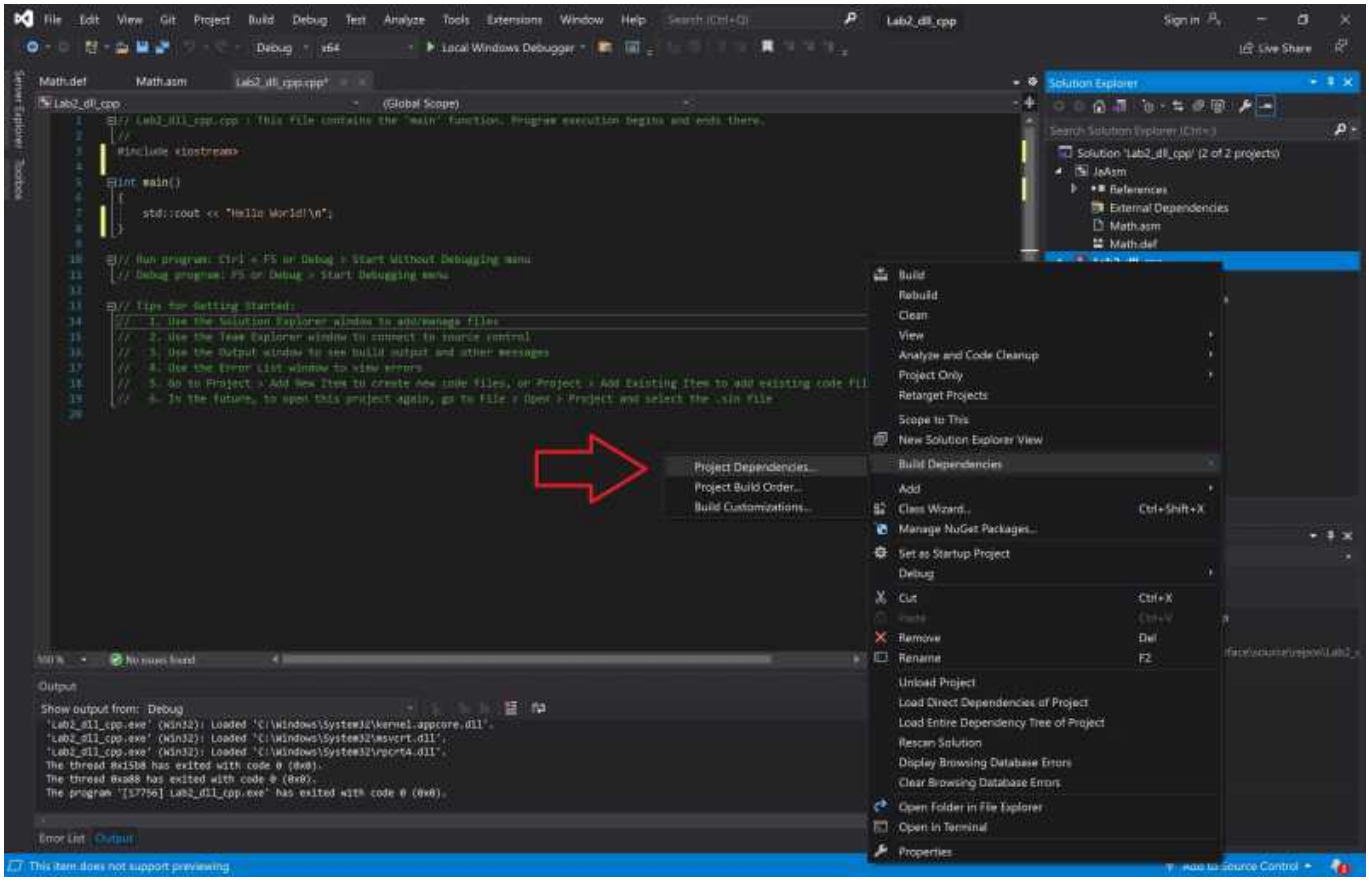


Both parts of our solution will be placed as subfolders in one folder named according to the name of the solution, in our case “Lab2_dll_cpp”. Each of these subfolders will take the name of the respective projects, i.e. “Lab2_dll_cpp” for the part written in C++ and “JaAsm” for the part written in assembler.

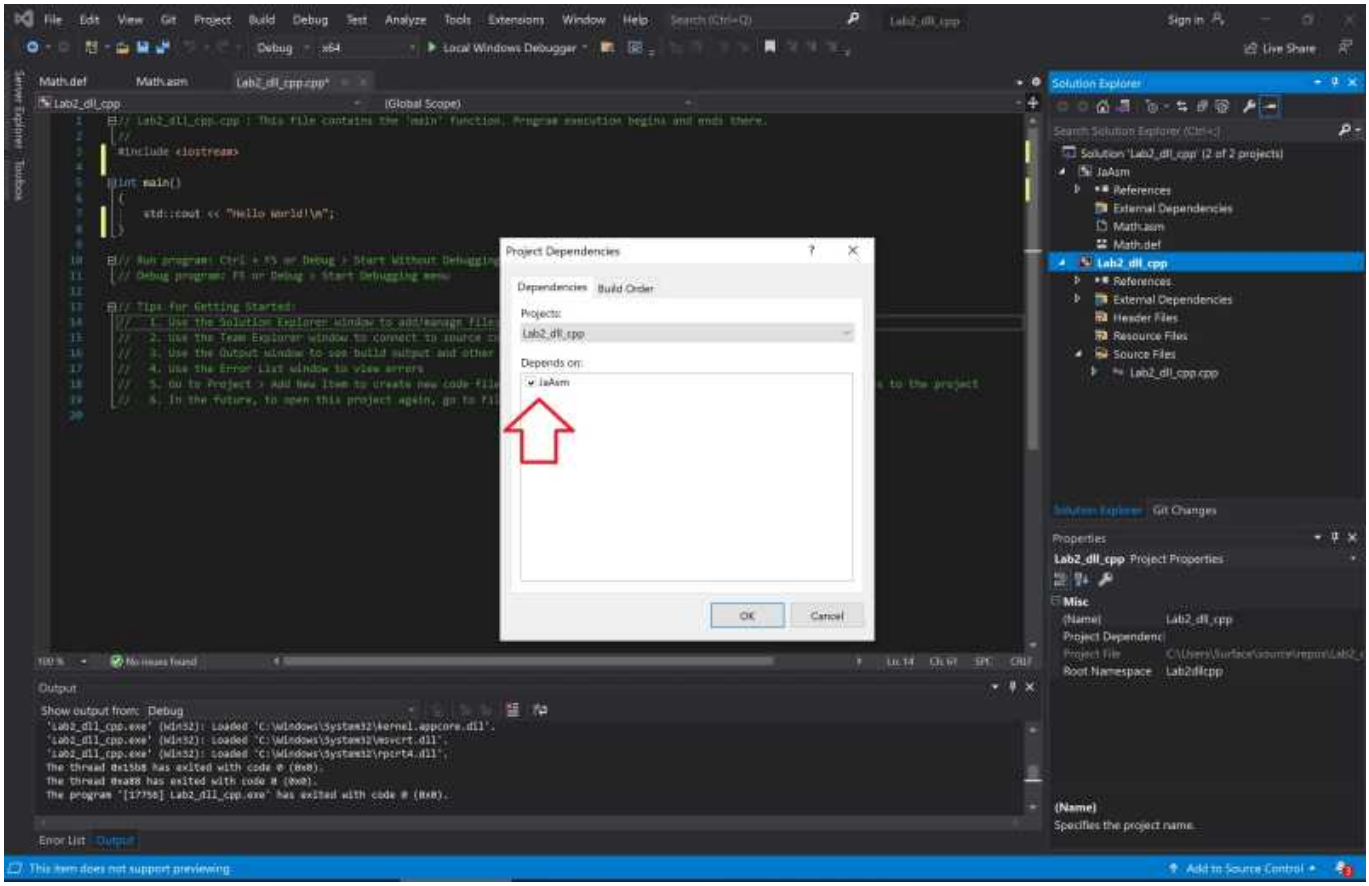
## 1.1.3. Setting dependencies in the solution

The next step is to set up the project dependencies so that compiling the C++ console application automatically forces the assembly library to be compiled. To do this, right-click on the C++ console project, select the “Build dependencies” menu, and then select “Project dependencies”.






In the newly opened window, select our “JaAsm” library and click “OK”.

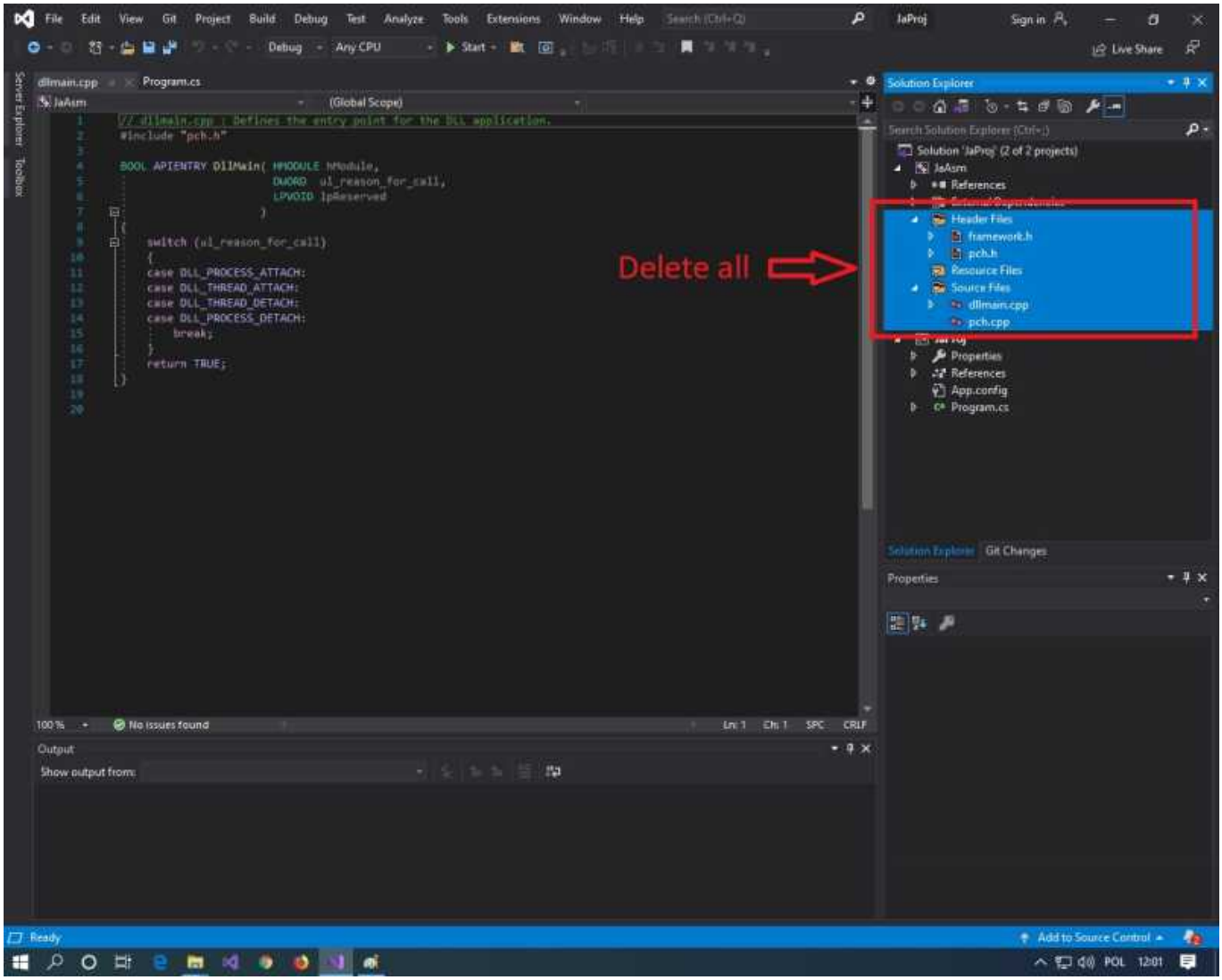



Once this dependency is set, when you compile a project or program written in C++,

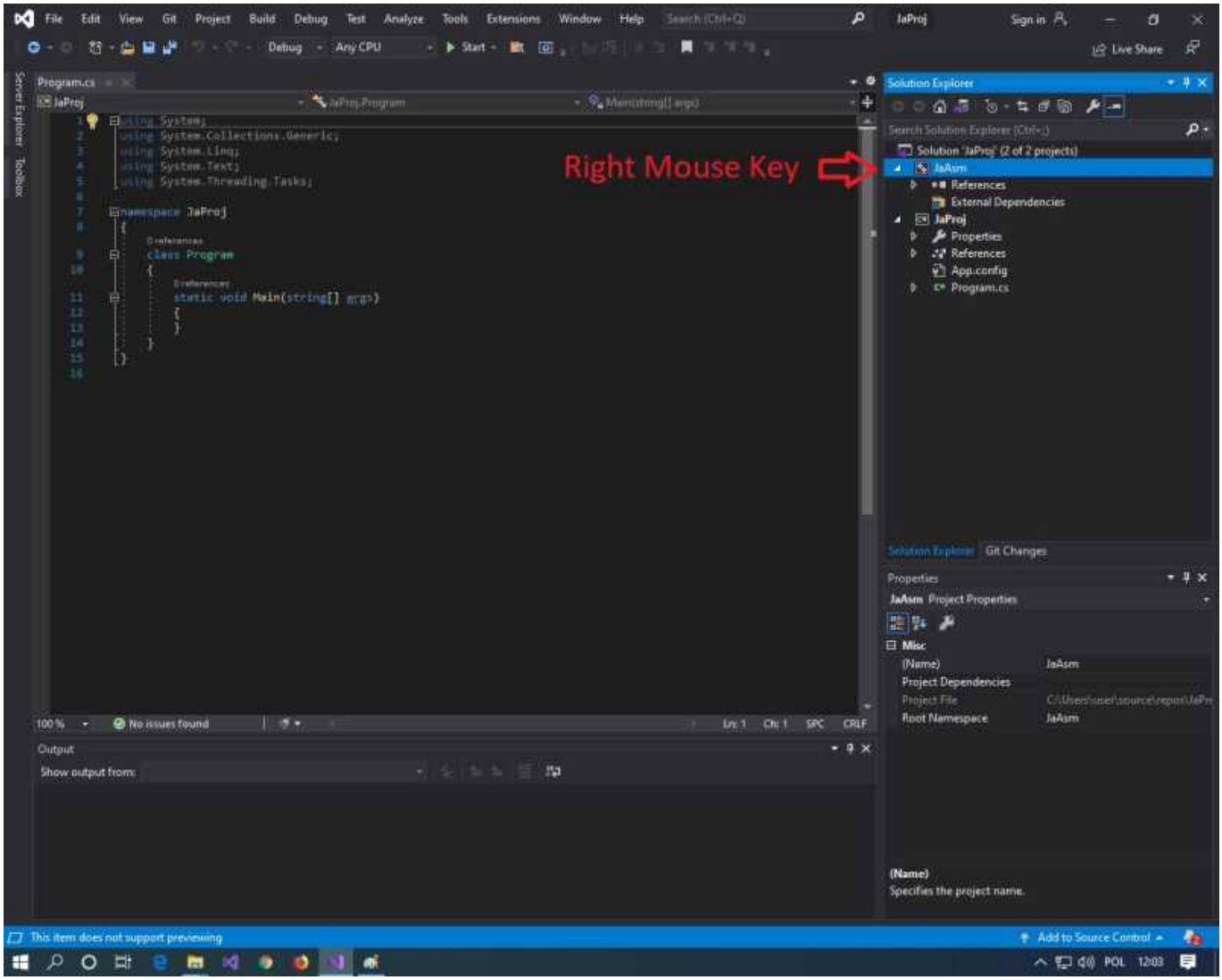
the assembler file will also be recompiled automatically.

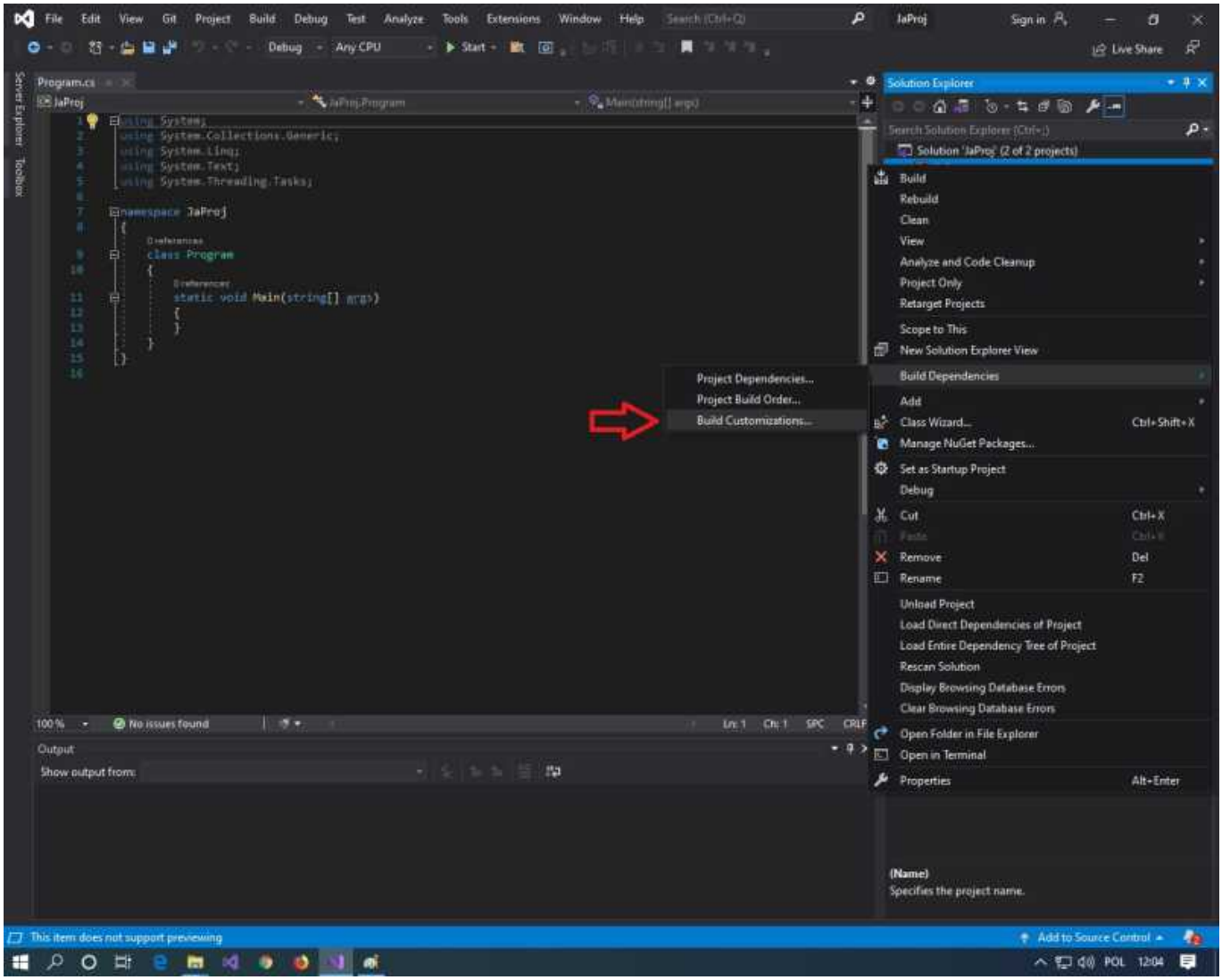
### 1.1.4. Changing the type from a C++ library to an assembler one

First, you need to remove the automatically created C++ project files. To do this, simply select the header and source files, then right-click and choose “Remove” or click Delete key.

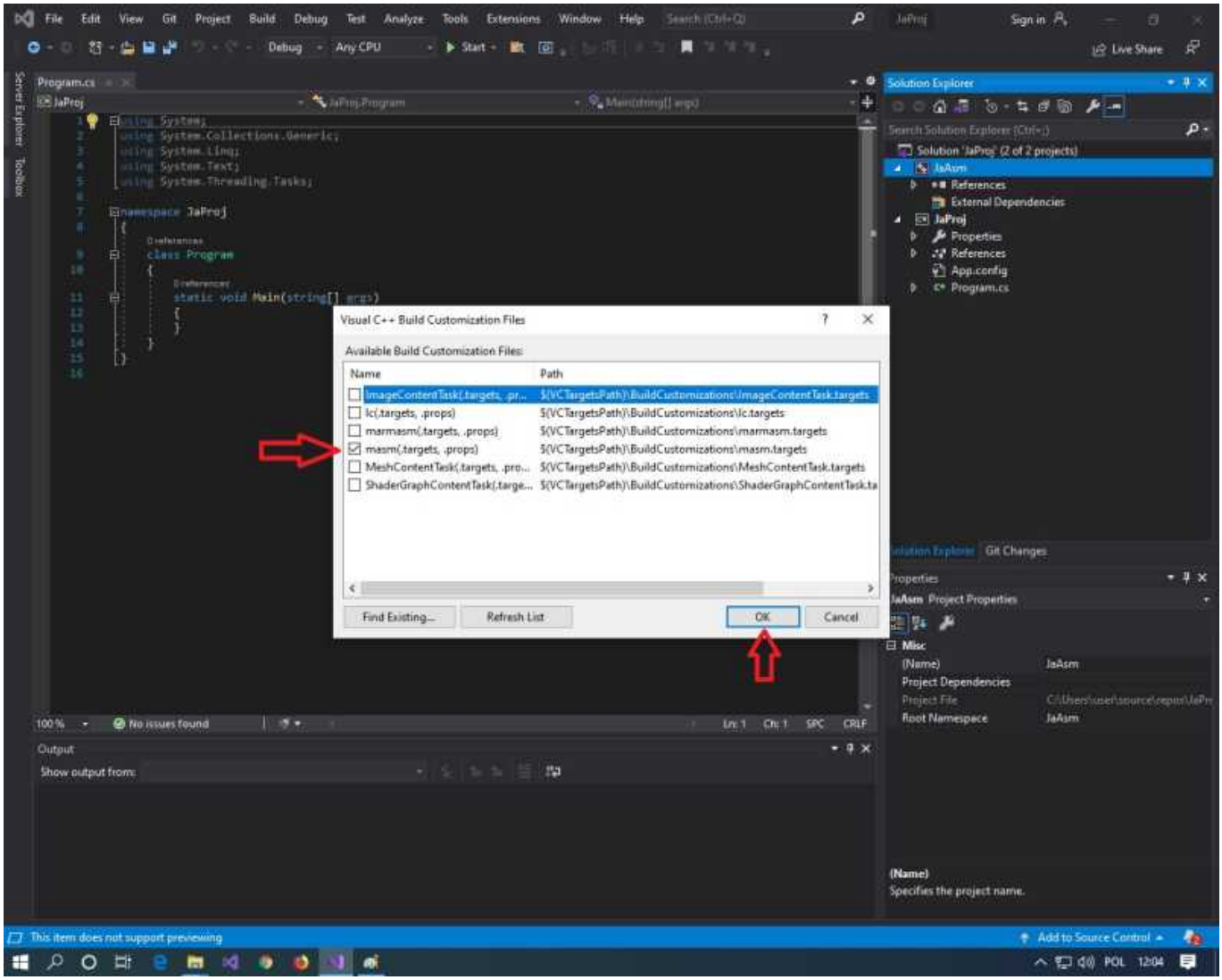



For the library, it is necessary to customize the compilation as an MASM assembler file. To do this, right-click the library project and select “Build dependencies” from the list, then “Build customization”.

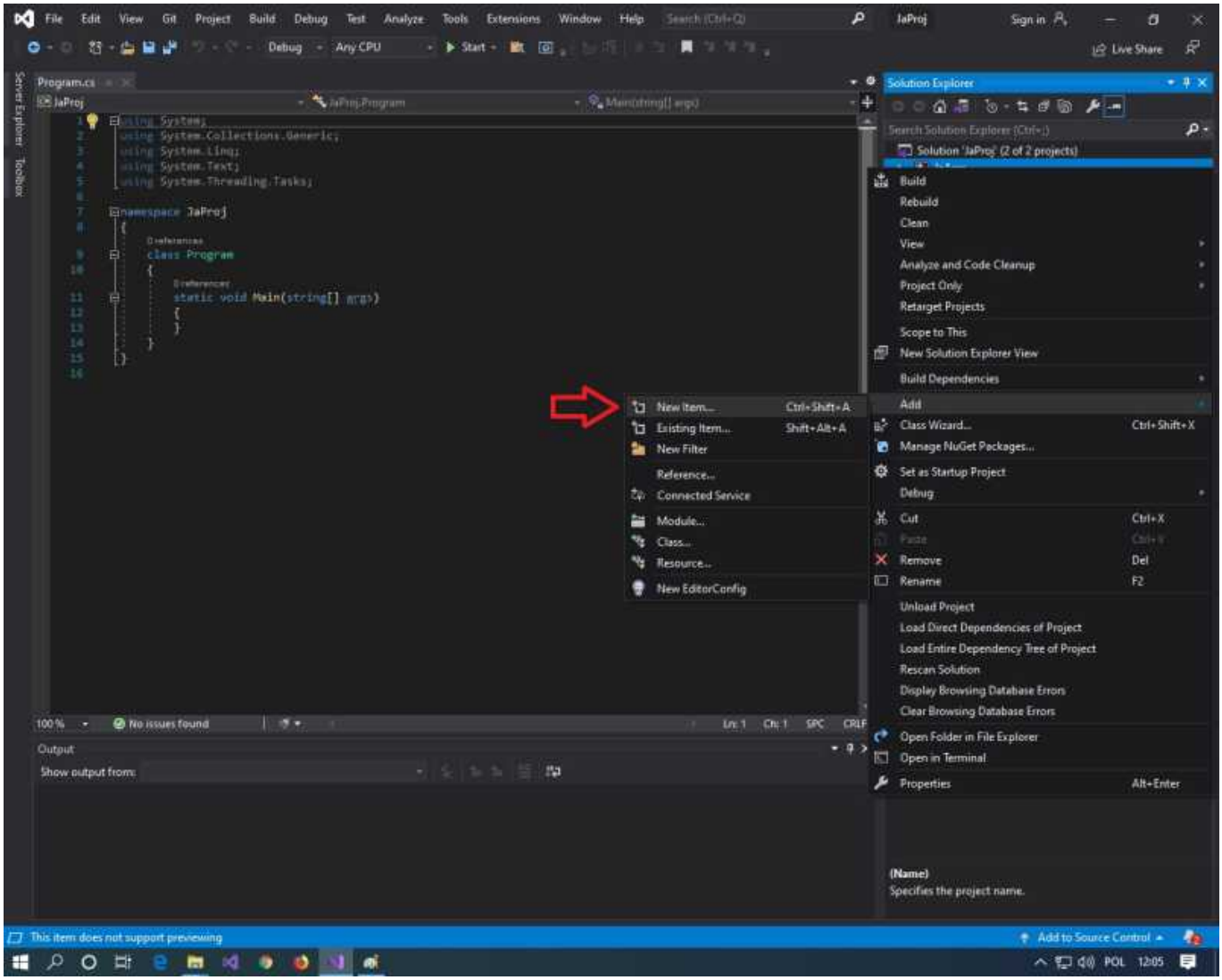

Right Mouse Key



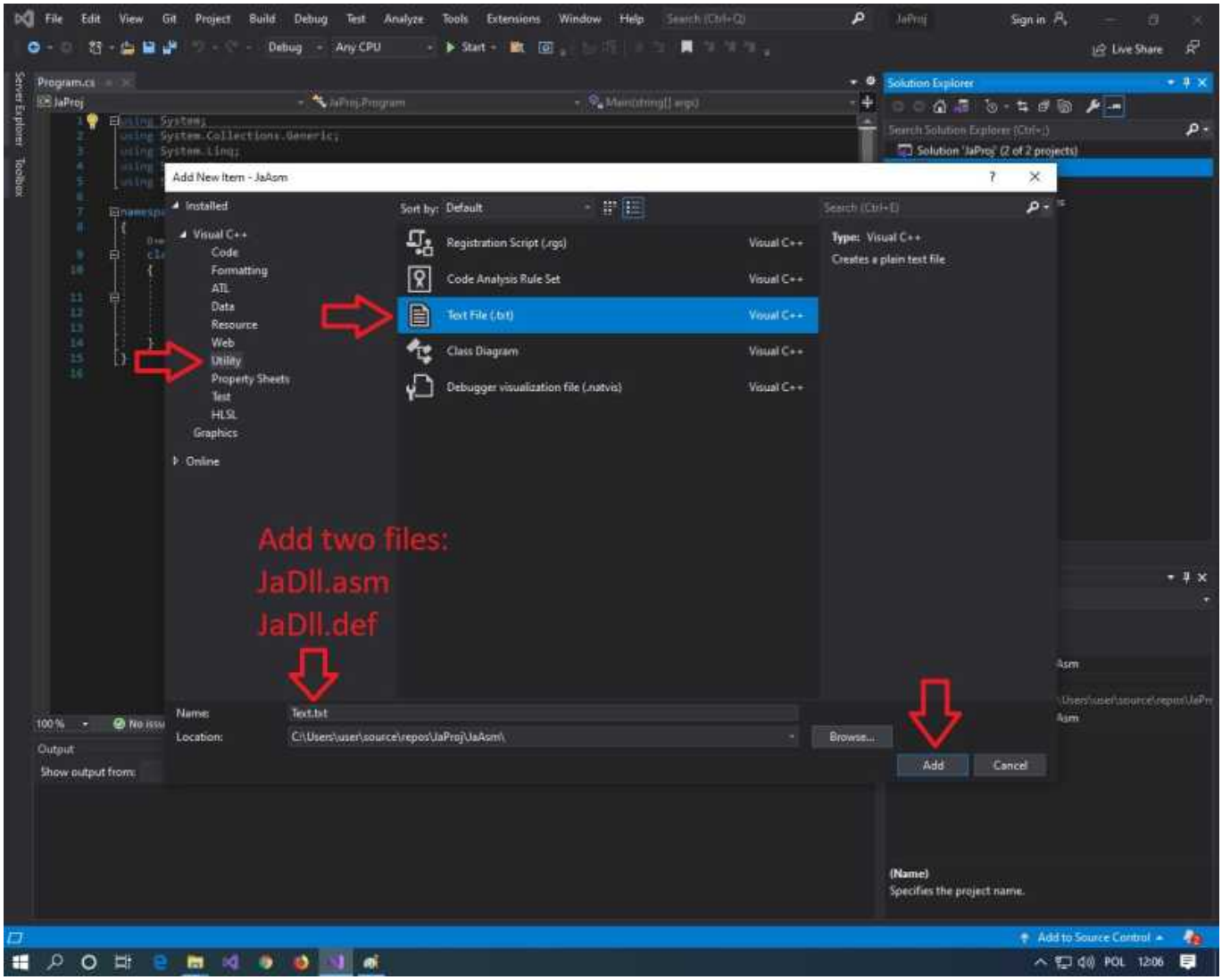

In the newly opened window, select “masm” and confirm our selection with “OK”.




The next step is to add the source files in the assembler and the definition file to the library. To do this, right-click on the “JaAsm” library again, then select “Add” and “New item”.



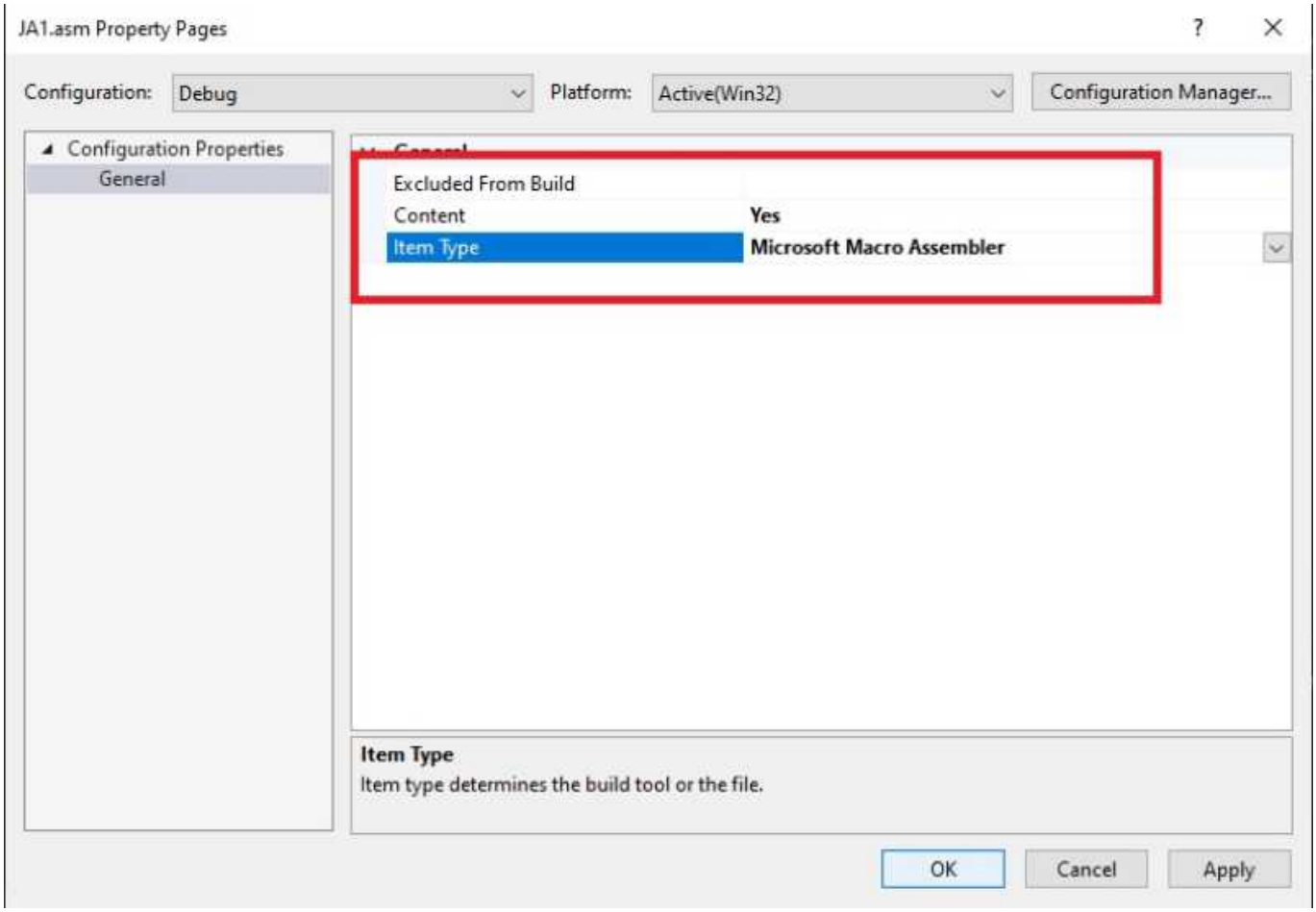

We repeat this operation twice, each time selecting “Utility” and the text file type, giving the names: “JaDll.asm” and “JaDll.def”.





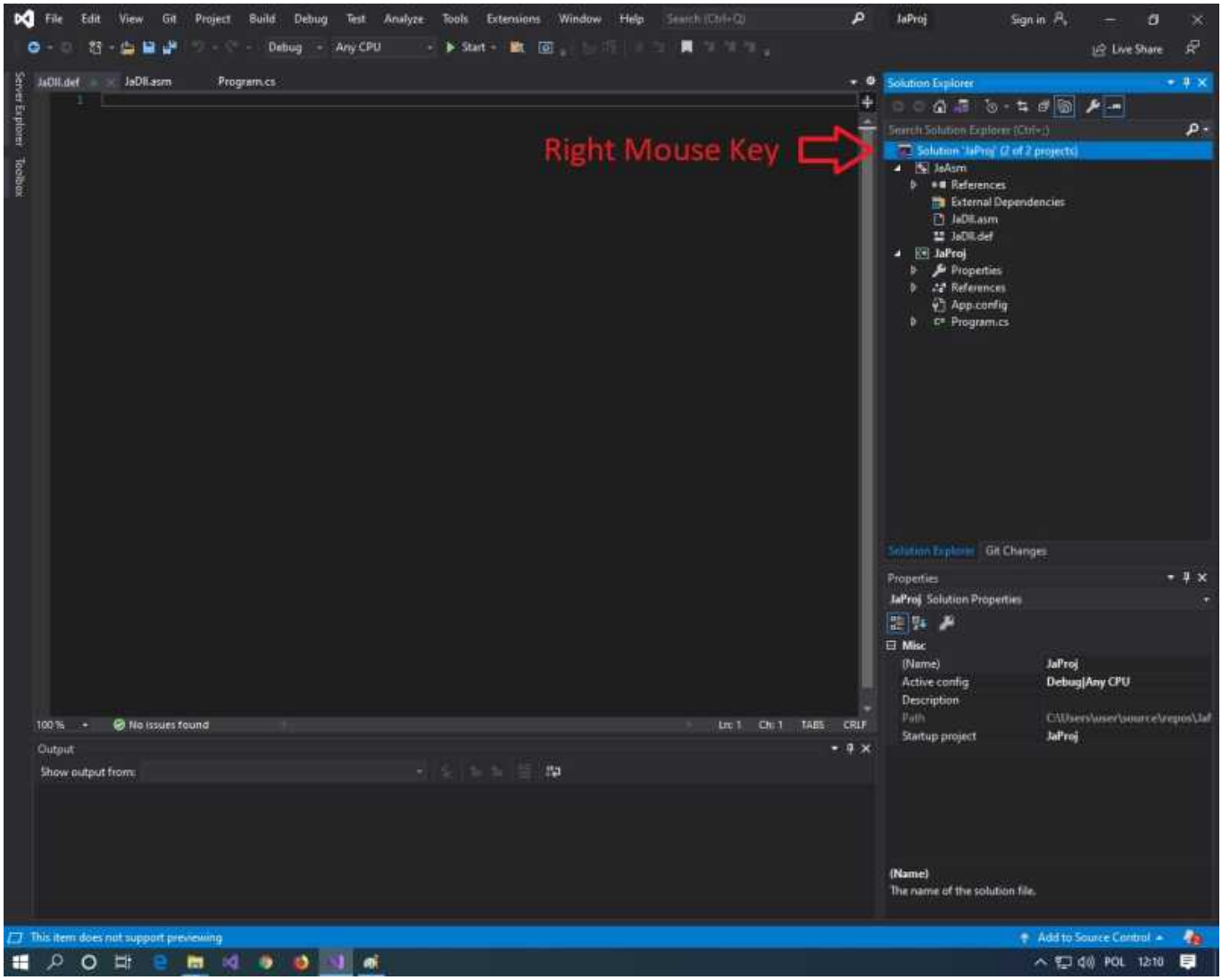

If you perform the activities in the order given above, it is highly likely that the step shown in the next figure will not be necessary.

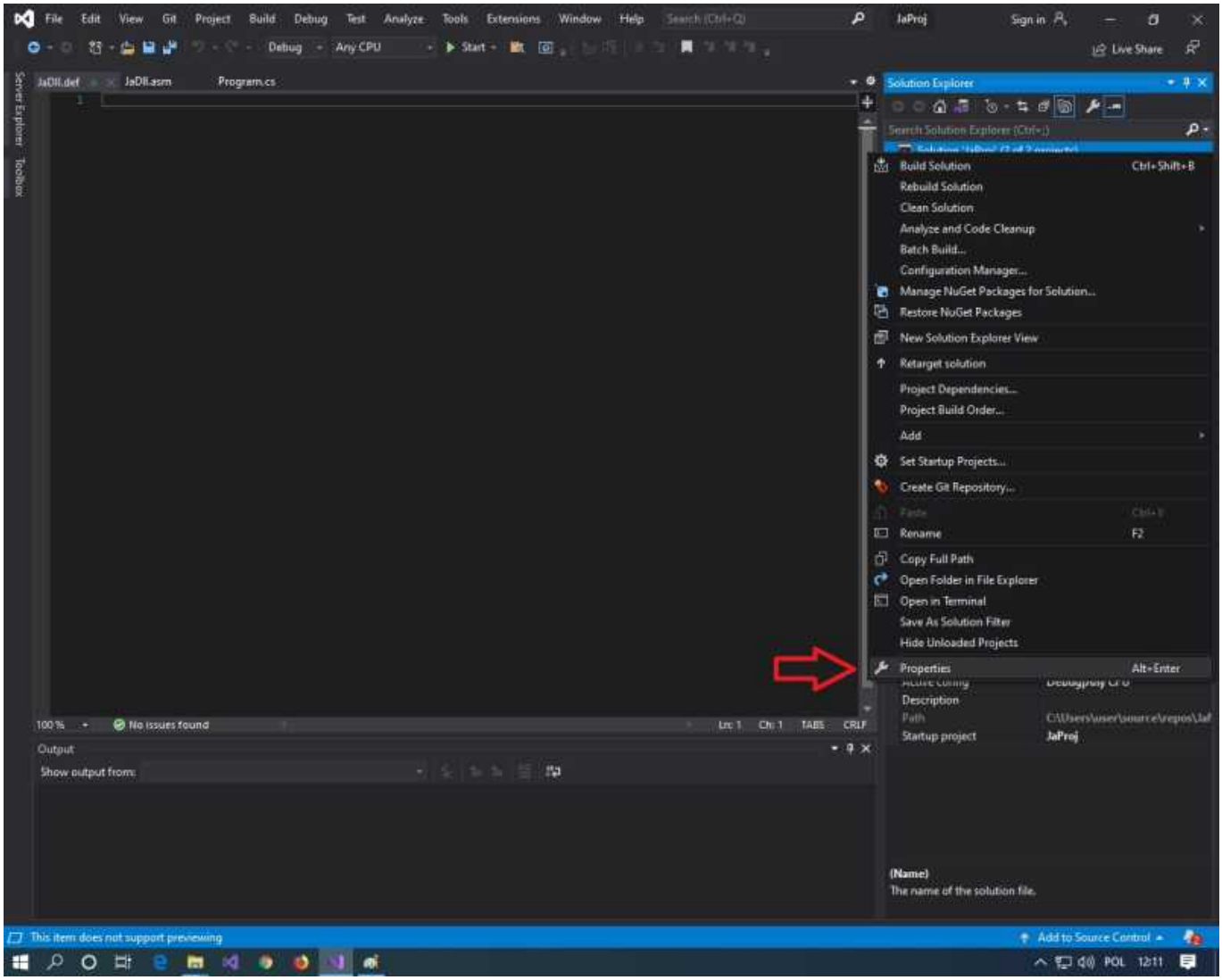
In order for the compiler to know that the created assembler file is associated with MASM, its type must be indicated. This step is necessary when we use a C++ source file and only change its extension to *.asm. By clicking the work key on the *.asm file, we select properties and set the file type to Microsoft Macro Assembler.



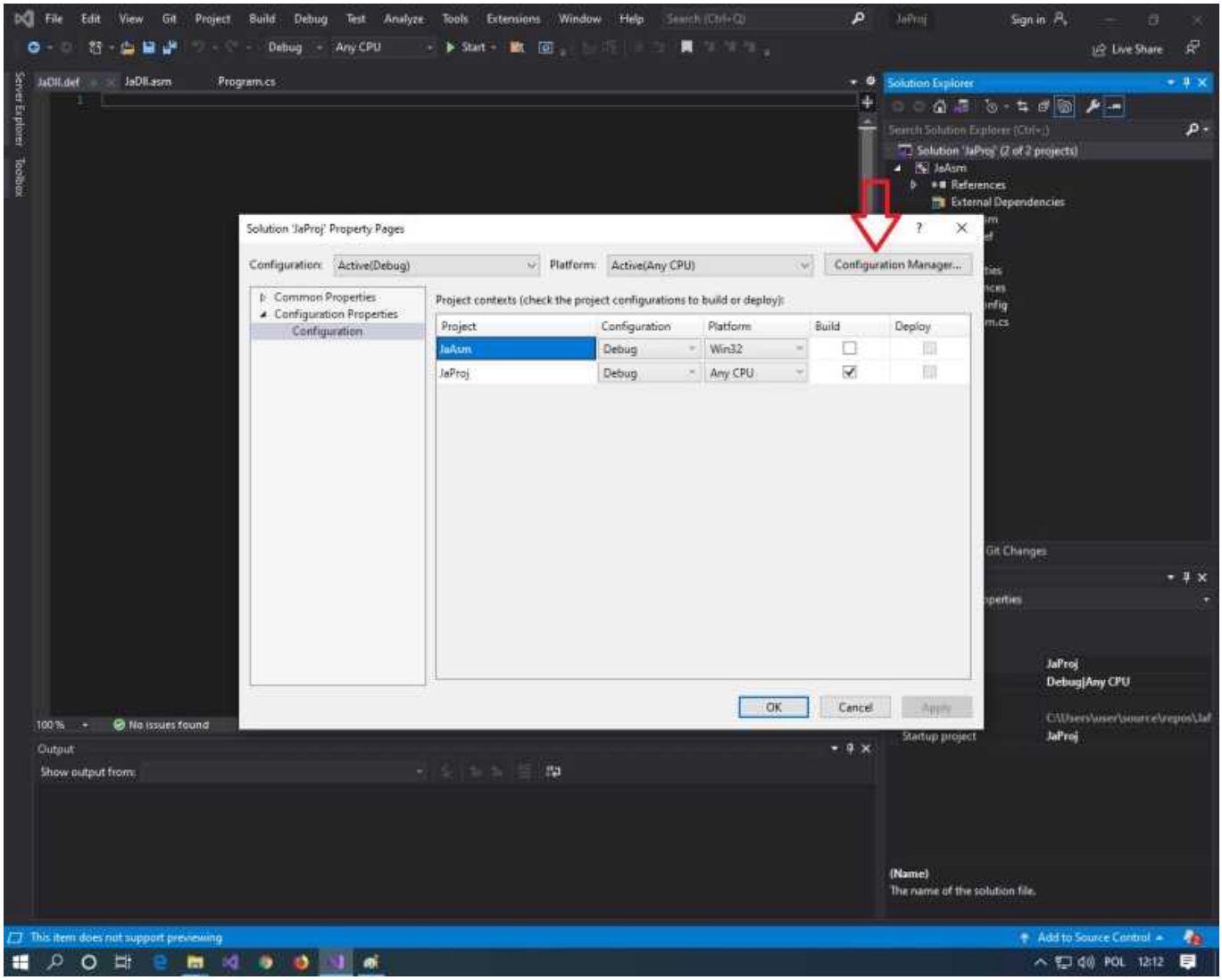

## 1.1.5. Setting the compilation mode

Compilation should be done in 64-bit mode (x64). To do this, set this mode in the solution options. Right-click on our solution and select “Properties”.

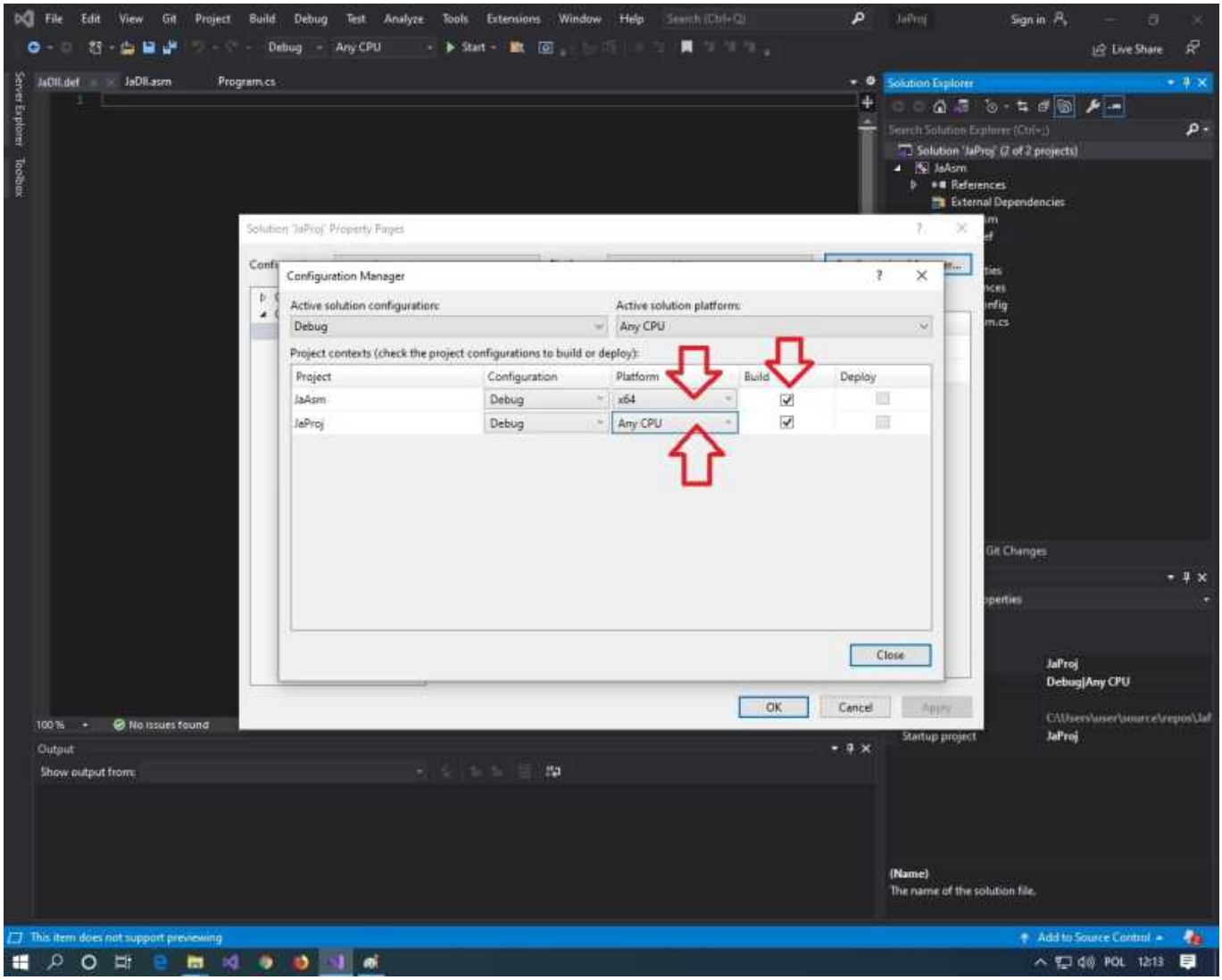

Right Mouse Key

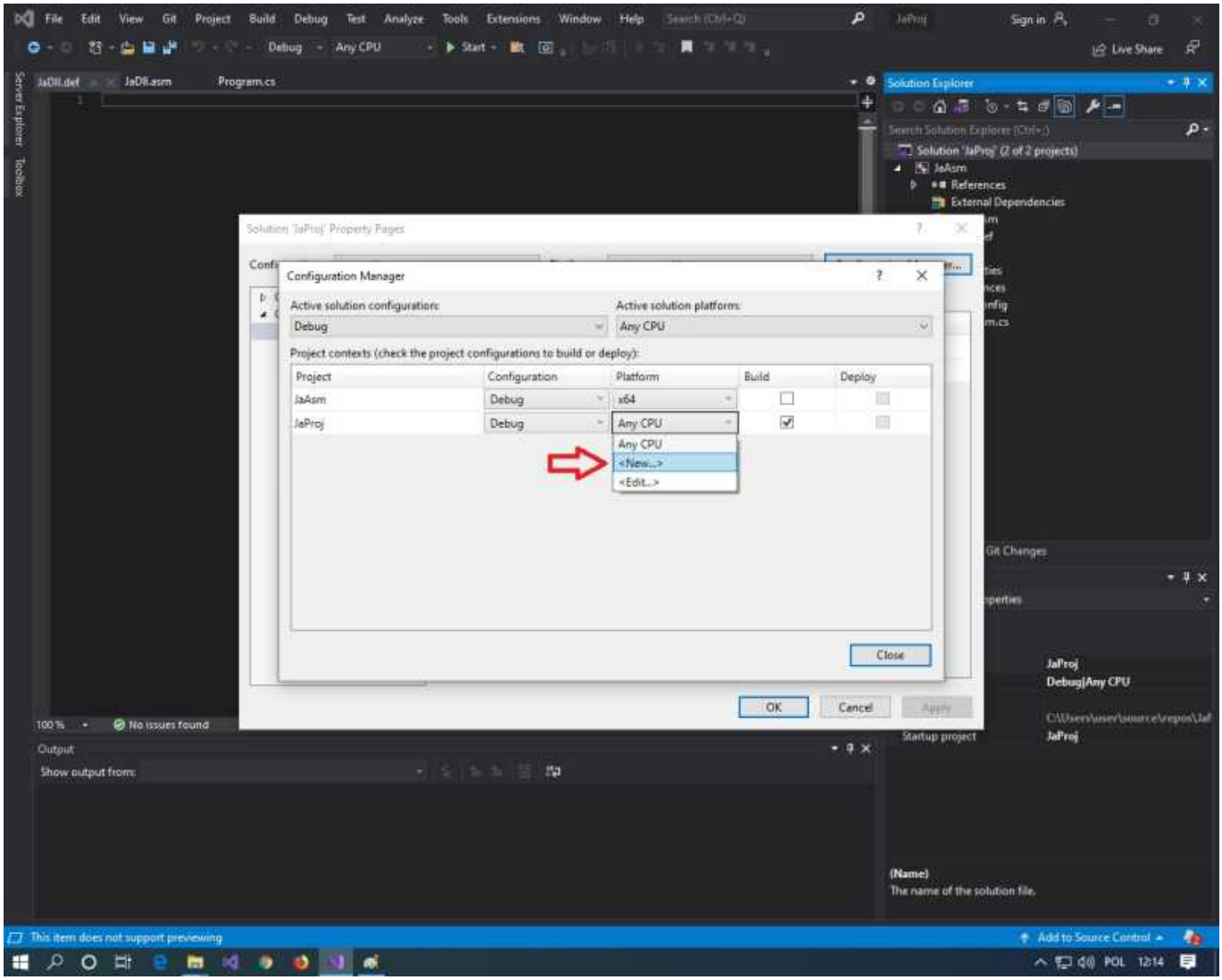


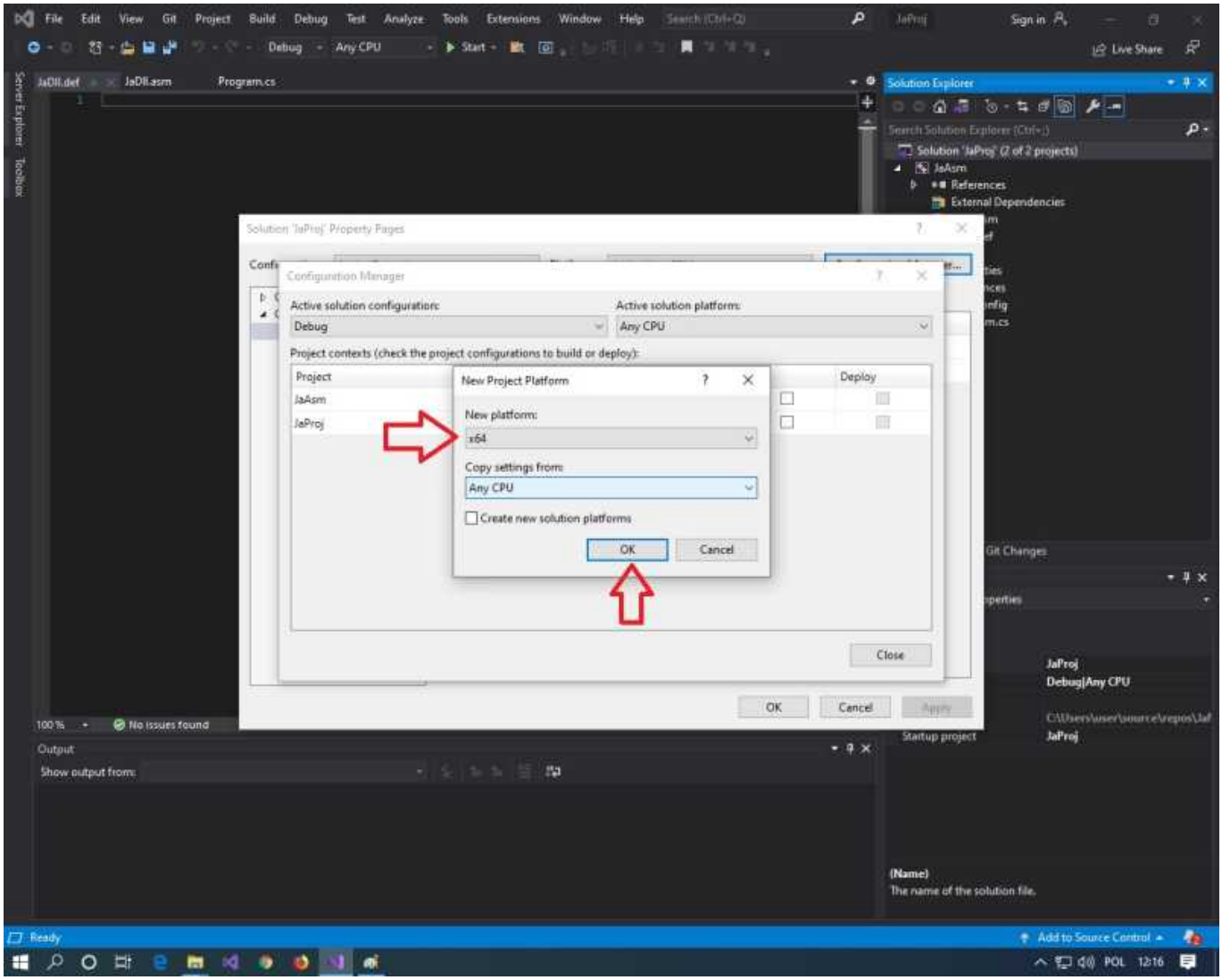

In the newly opened window, go to the “Configuration properties” → “Configuration” tab and then click “Configuration manager”.



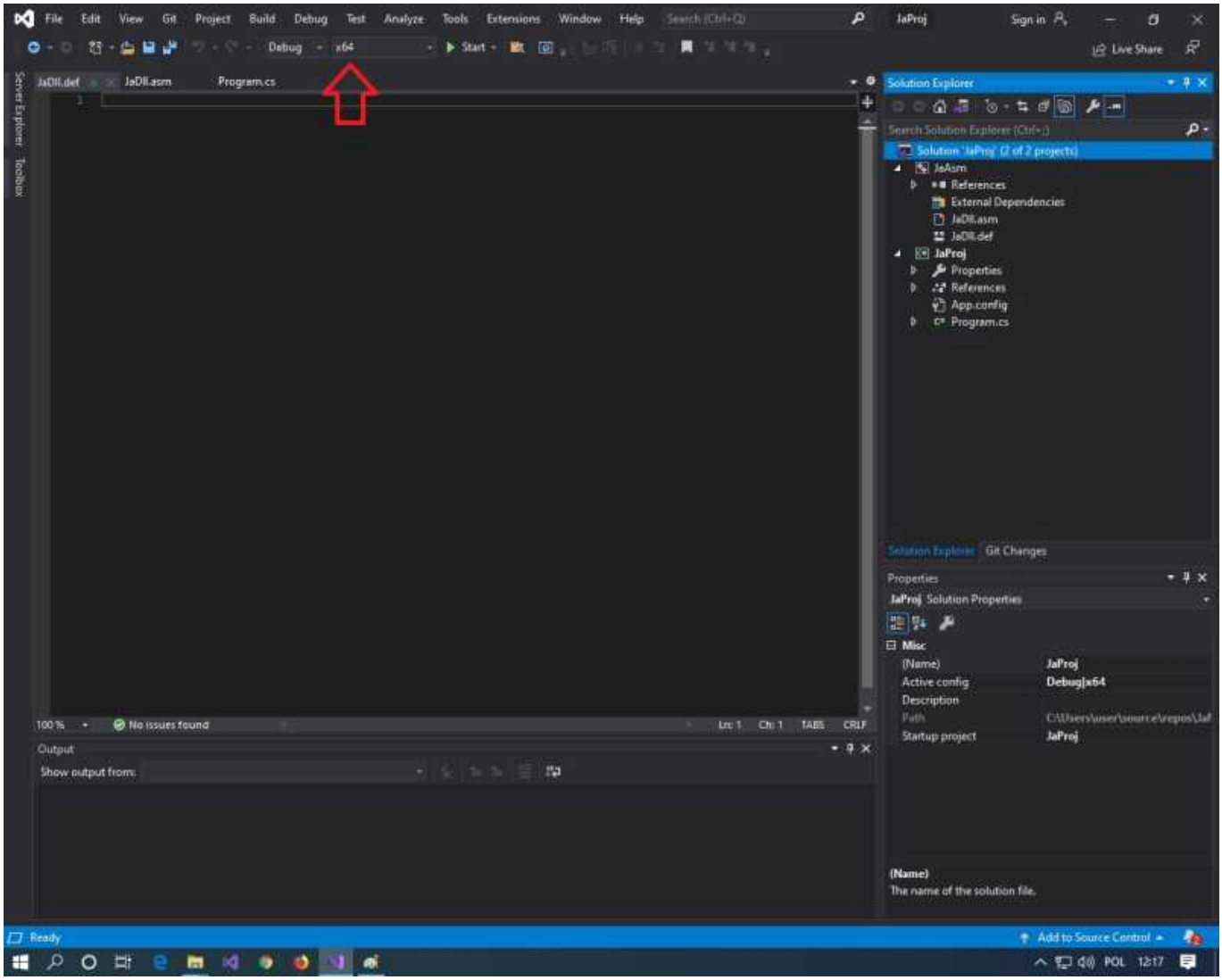

In the new window, we indicate the x64 platform everywhere (if a given option is unavailable, we select “New” from the available list, and then create the x64 platform by copying the settings from Any CPU). You should also enable compilation (Build) for both parts of the solution.


Configuration Manager
Active solution configuration:
Debug
Active solution platform:
Any CPU
Project contexts (check the project configurations to build or deploy):
Project
Configuration
Platform
Build
Deploy
JaAsm
Debug
x64
JaProj
Debug
Any CPU
Close
OK
Cancel
Apply


Configuration Manager
Active solution configuration:
Debug
Active solution platform:
Any CPU
Project contexts (check the project configurations to build or deploy):
Project
Configuration
Platform
Build
Deploy
JaAsm
Debug
x64
JaProj
Debug
Any CPU
Any CPU
<New...>
<Edit...>
Close
OK
Cancel
Apply

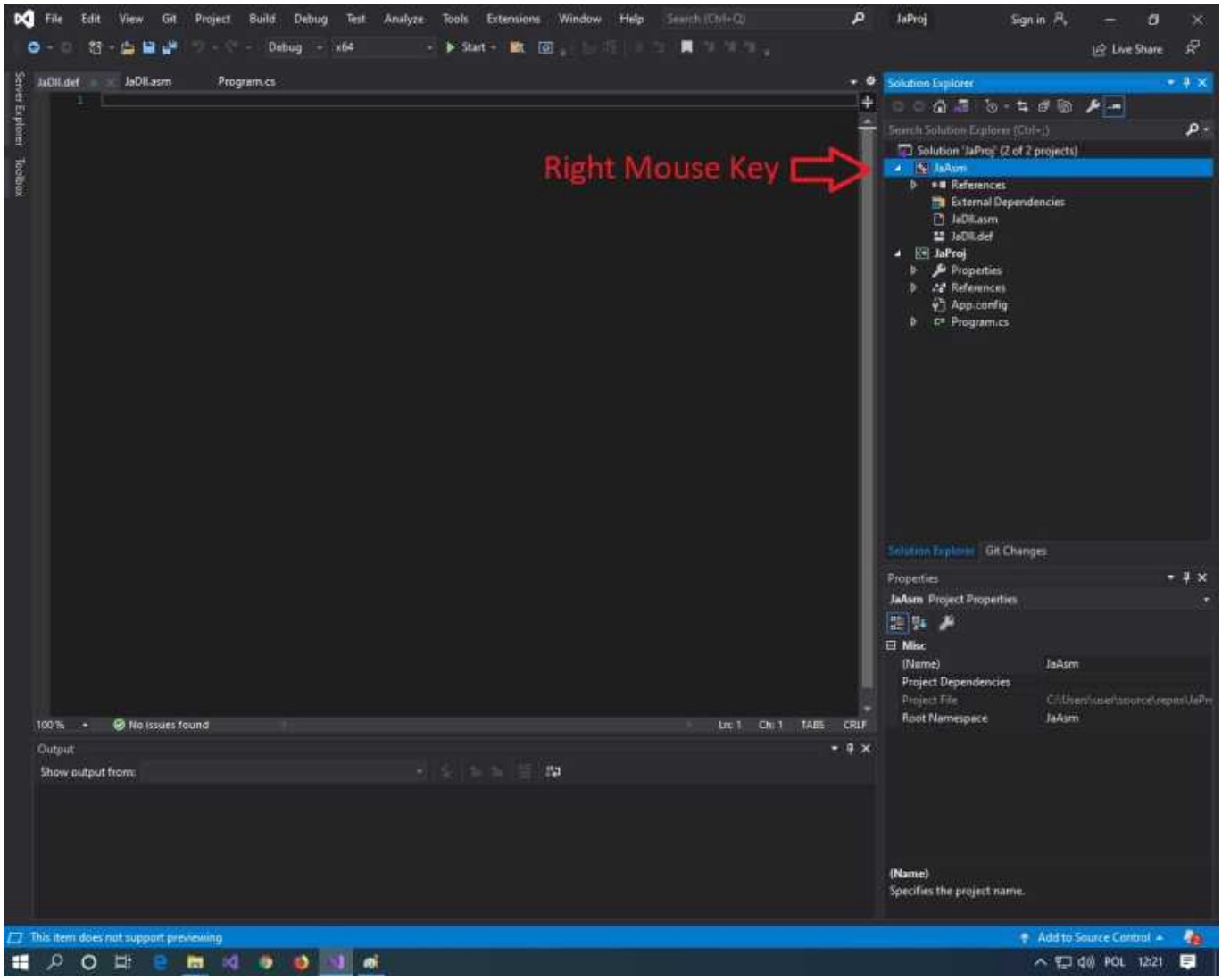


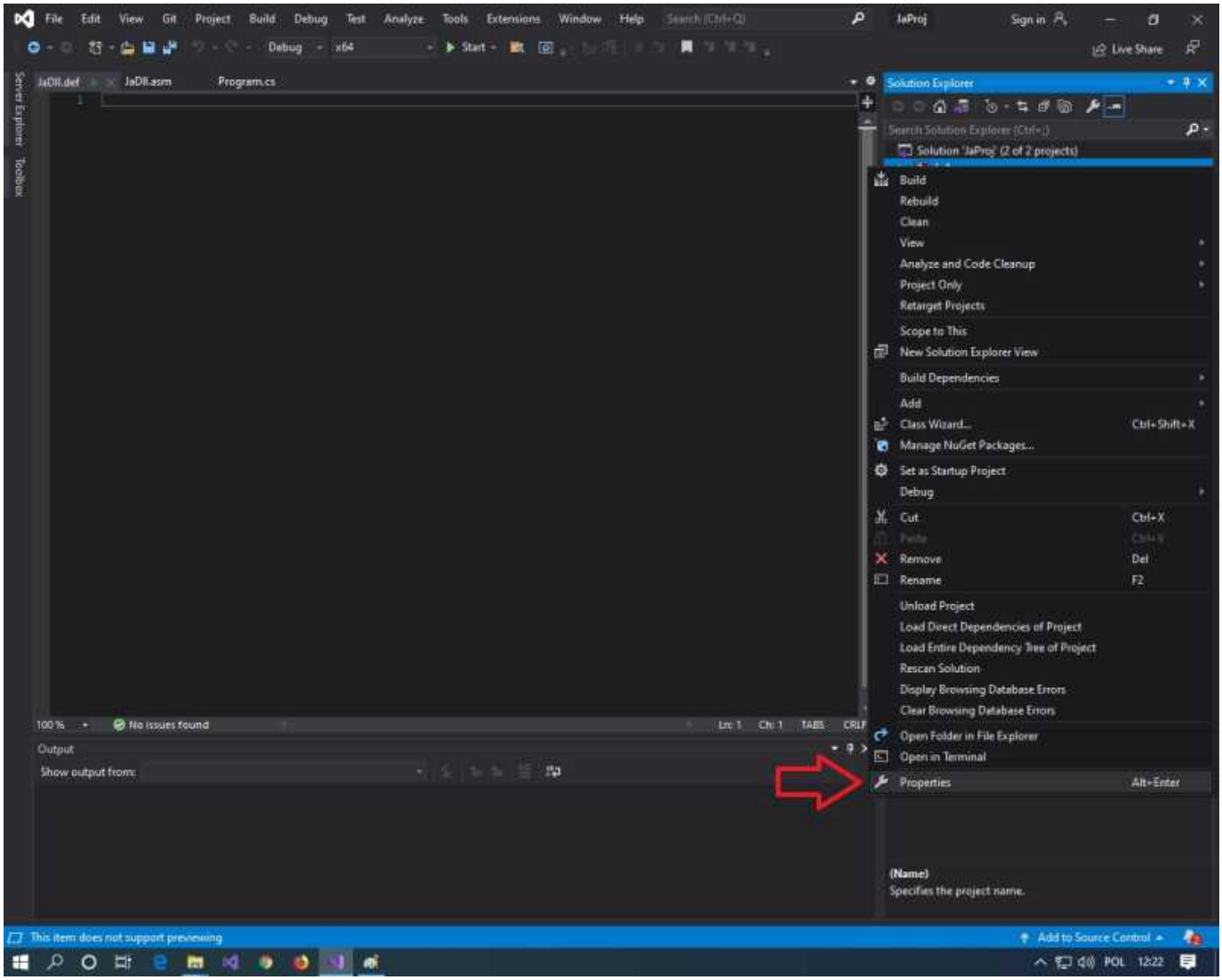

After that, we make sure once again that the platform indicated everywhere is x64 and close the window. The “x64” option should automatically be set in the main menu bar of Visual Studio, but it is worth checking and modifying it if necessary.



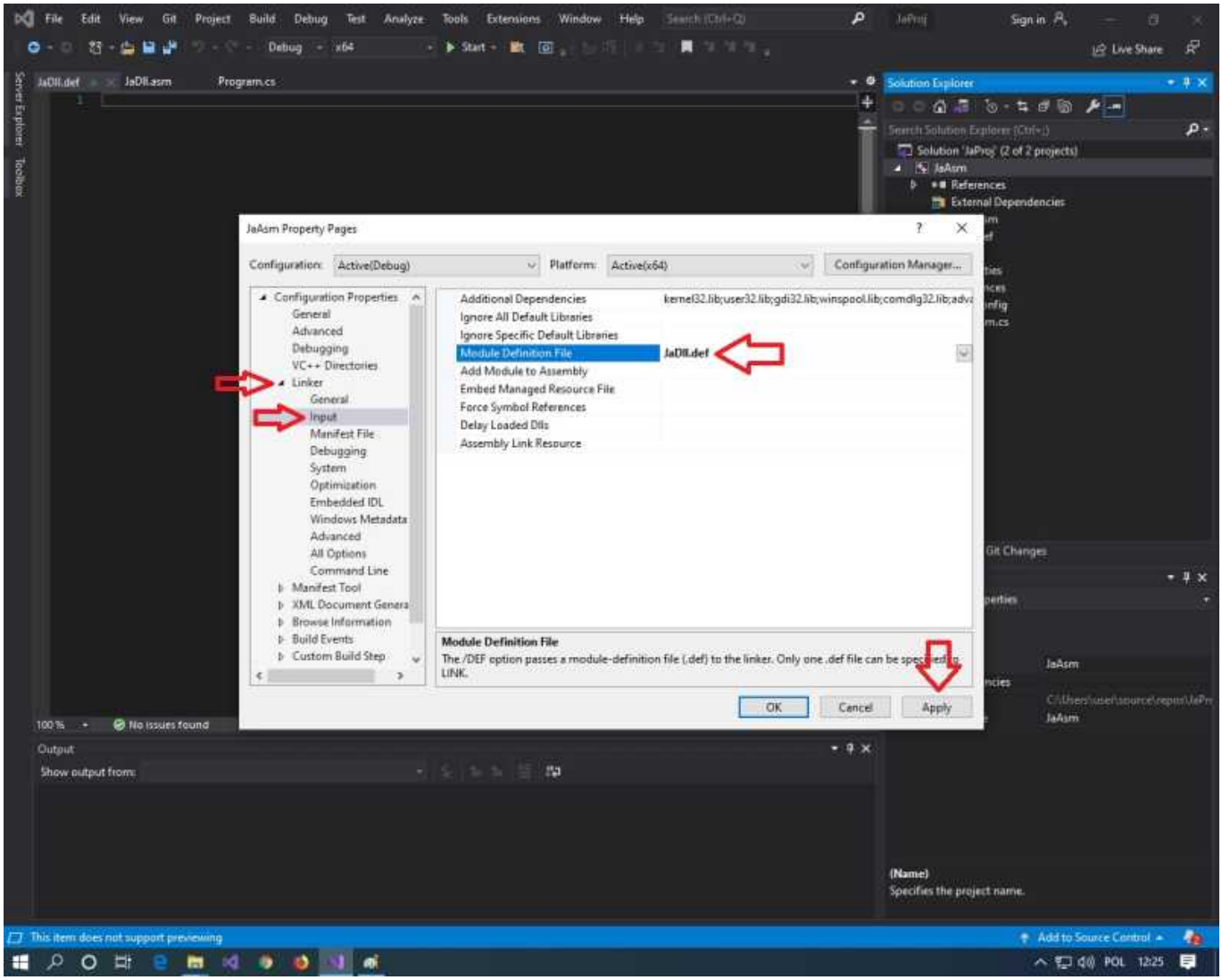

## 1.1.6. Final setup of the assembly library

We proceed to configure the library. To do this, right-click on the library and select “Properties”.

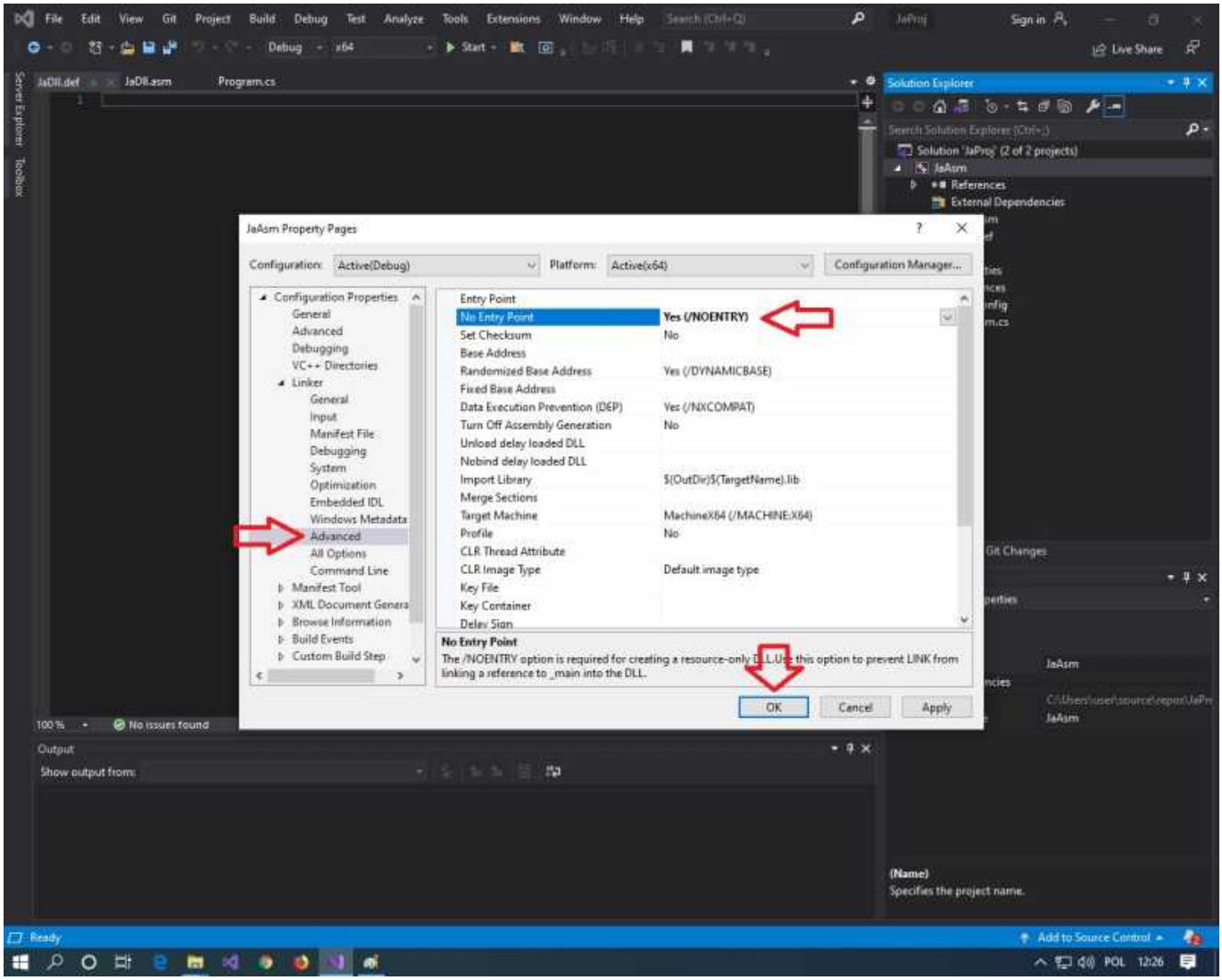

Right Mouse Key




In the newly opened window, go to “Linker” → “Input” and in the “Module definition file” field, enter the name of our file “*.def”. In our case, “JaDll.def”. Confirm the entry.




The final step in the configuration is to disable the library entry point. To do this, go to the “Linker” tab and select “Advanced” and select “Yes (/NOENTRY)” in the “No entry point” field. Save the settings and close the window.






A dll library, unlike a main program, does not have a starting point (address) that the operating system uses to start the program. It only provides a set of functions for other programs.

# 1.2. Using a dll library written in assembler

## 1.2.1. dll library written in assembler

The dll library contains functions that can be used in other programs. The first version of our library will provide one function that returns a constant value. We will then read this value in a C++ file. An example of the assembly file (JaDll.asm) may look like this.

```
.code
MyProc proc
mov RAX, 2021
ret
MyProc endp
```

```
end
```

The program code file is inextricably linked to the definition file (JaDll.def). Such a file must contain the name of the DLL library as an argument of the LIBRARY directive, and a list of function names available to other modules preceded by the EXPORTS directive. In order for our MyProc function to be seen by the C++ program, we must declare it as exported. The JaDll.def file in our project may look like this.

```
LIBRARY JaAsm
EXPORTS MyProc
```

We often want to use the parameters passed to the function. Recall that in the 64-bit calling convention, the first four parameters are placed directly in the processor registers. The details were provided during lectures and are described in the instructions for the first lab exercise. Thus, two integer-type parameters are placed in the RCX and RDX registers. A function that returns the sum of two integer arguments would look like this.

```
...
MyProc proc
mov RAX, RCX
add RAX, RDX
ret
MyProc endp
...
```

Floating point arguments should be passed through registers XMM0 and XMM1, and the result should be returned through XMM0. Of course, the MMX/SSE extension instructions should be used.

```
...
FloatProc PROC
addsd xmm0, xmm1
ret
FloatProc ENDP
...
```

## 1.2.2. Calling functions in dll from C++ program

In order for a program written in C++ to use a function from a library, several steps must be performed. - Prepare the type of the function that is compatible with its later call, - Load the library using the LoadLibrary function while obtaining its handle, - Read the current address of the function of interest to us by calling GetProcAddress, - Use the function by referring to its address. The sample code could look like this:

```
#include <windows.h>
#include <iostream>
typedef int(_stdcall* MyProc)();
HINSTANCE dllHandle = NULL;

int main()
```

```
{
    dllHandle = LoadLibrary(TEXT("JaAsm.dll"));
    MyProc procedura = (MyProc)GetProcAddress(dllHandle, "MyProc");
    std::cout << procedura();
}
```

Using a function with parameters requires slightly modified type definitions. You should also pay attention to the type of the value returned by the function. Remember that in the case of floating-point numbers, the size of the XMM0, XMM1 registers is 64 bits. This corresponds to a double-precision floating-point value, or in C++ to the type “double”.

```
...
typedef int(_stdcall* MyProc)(int, int);
typedef double(_stdcall* FloatProc)(double, double);
...
```

### 1.2.3. Task to be completed in the laboratory

You should write a DLL in assembler and a C++ program that demonstrates how the DLL works.
**The DLL** should provide functions that perform and return the result of basic arithmetic calculations (addition, subtraction, multiplication, and division) on pairs of numbers passed as parameters to their call. The functions should have fixed- and floating-point versions (eight functions in total).
**The C++ program** should call all functions, pass sample parameter values, and display the results on the screen. No user interface is required - the values for calculations can be placed in the program code.

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
**https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s1:apl:lab:lab2?rev=1732532959**

Last update: **2024/11/25 12:09**