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. Dodanie do rozwiązania części asemblerowej 4
1.1.3. Ustawienie zależności w rozwiązaniu 7
1.1.4. Zamiana biblioteki z C++ na asemblerową 9
1.1.5. Ustawienie trybu kompilacji 15
1.1.6. Końcowa konfiguracja biblioteki asemblerowej 22
1.2. Wykorzystanie biblioteki dll napisanej w asemblerze 26
1.2.1. Biblioteka asemblerowa dll 26
1.2.2. Odwołanie do biblioteki dll z programu C++ 27
1.2.3. Zadanie do zrealizowania w ramach laboratorium 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).



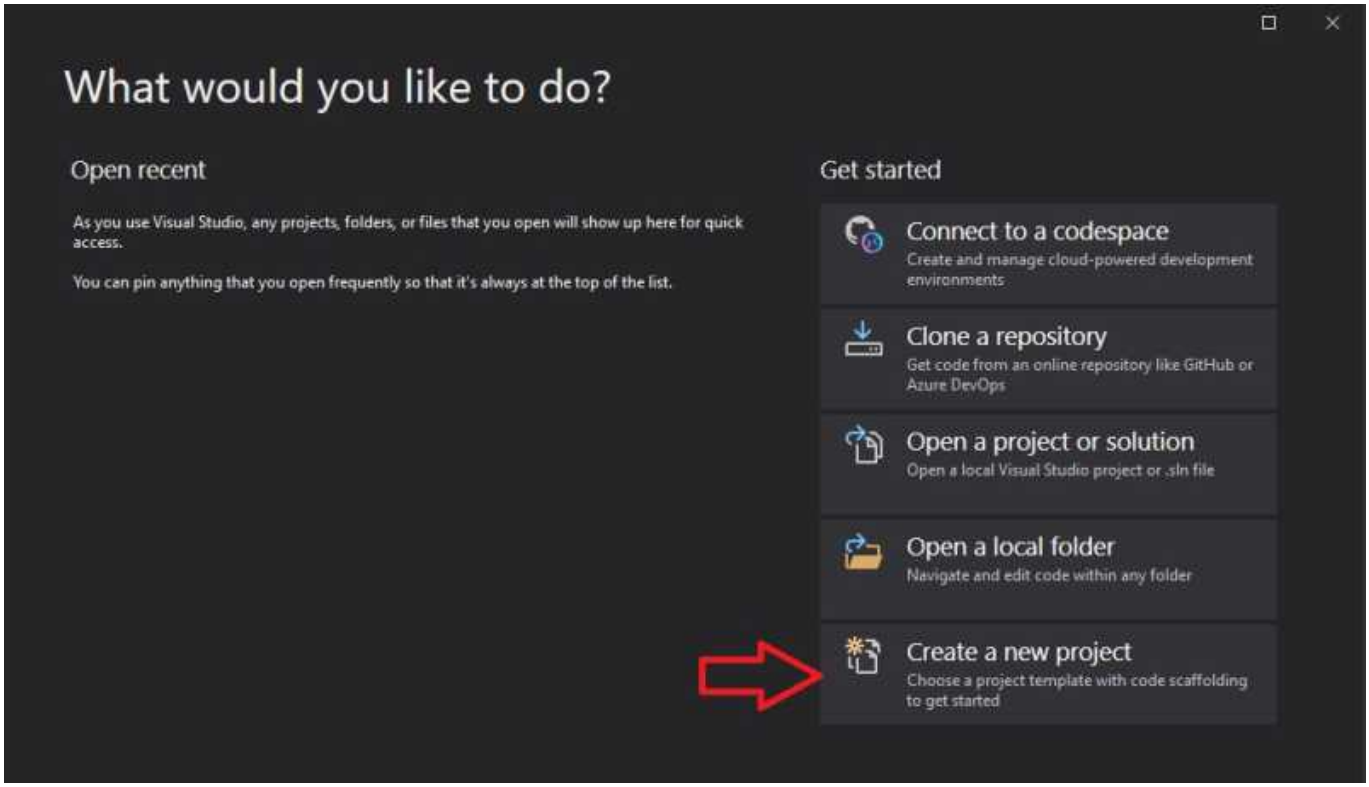
Creation of libraries written in assembly language, with functions called from the application written in a chosen high-level language is currently the most common assembler use.



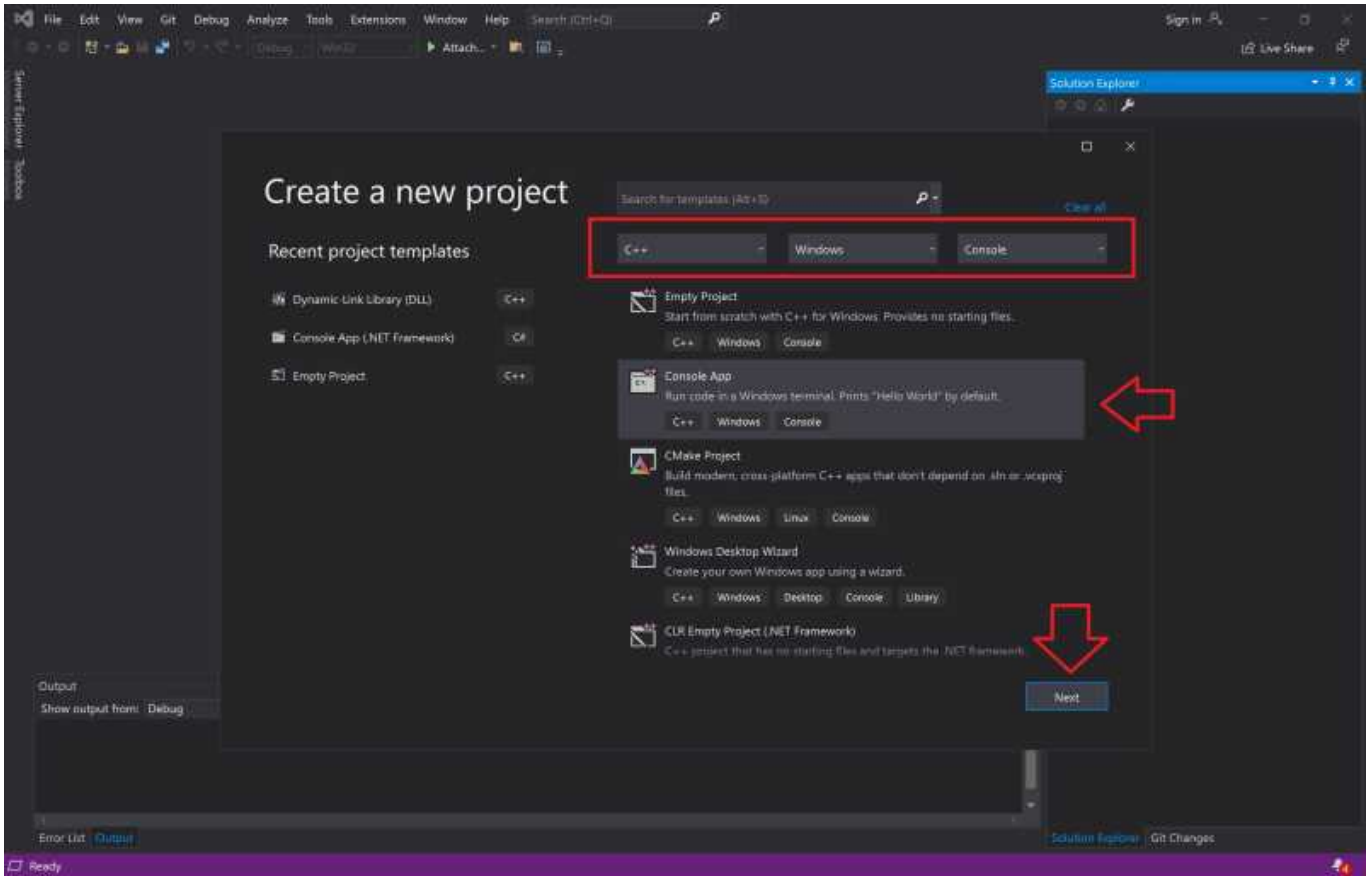
The configuration process presented in the following part is identical to the main program written in C++ and C#. The screenshots used are from the DLL configuration for both programming languages, but the differences are usually secondary and do not affect the environment configuration process.

1.1.1. Creating of the new project

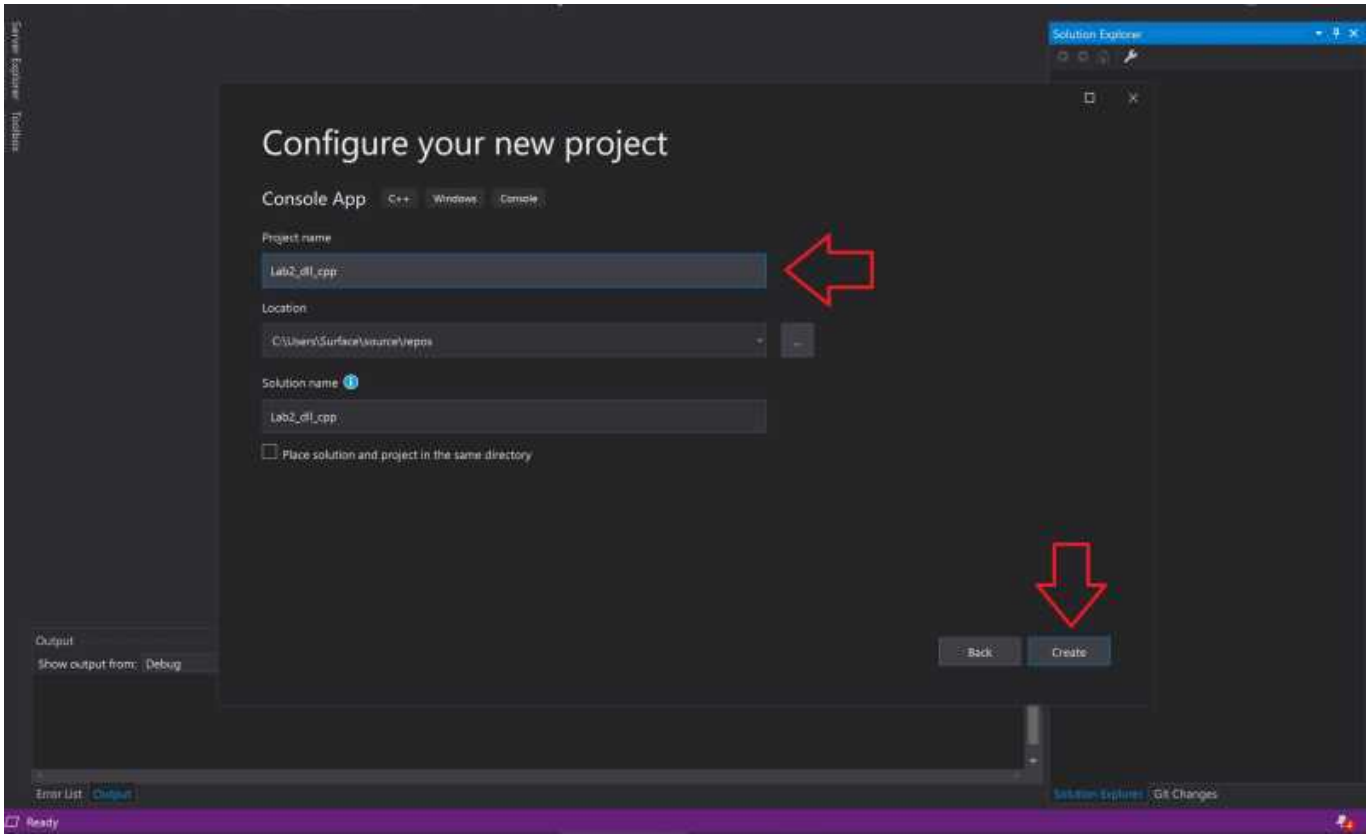
During this laboratory, we'll show how to create and use dll (dynamically linked library) written in assembler in that application written in C++ We start by creating a new C++ project.



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.

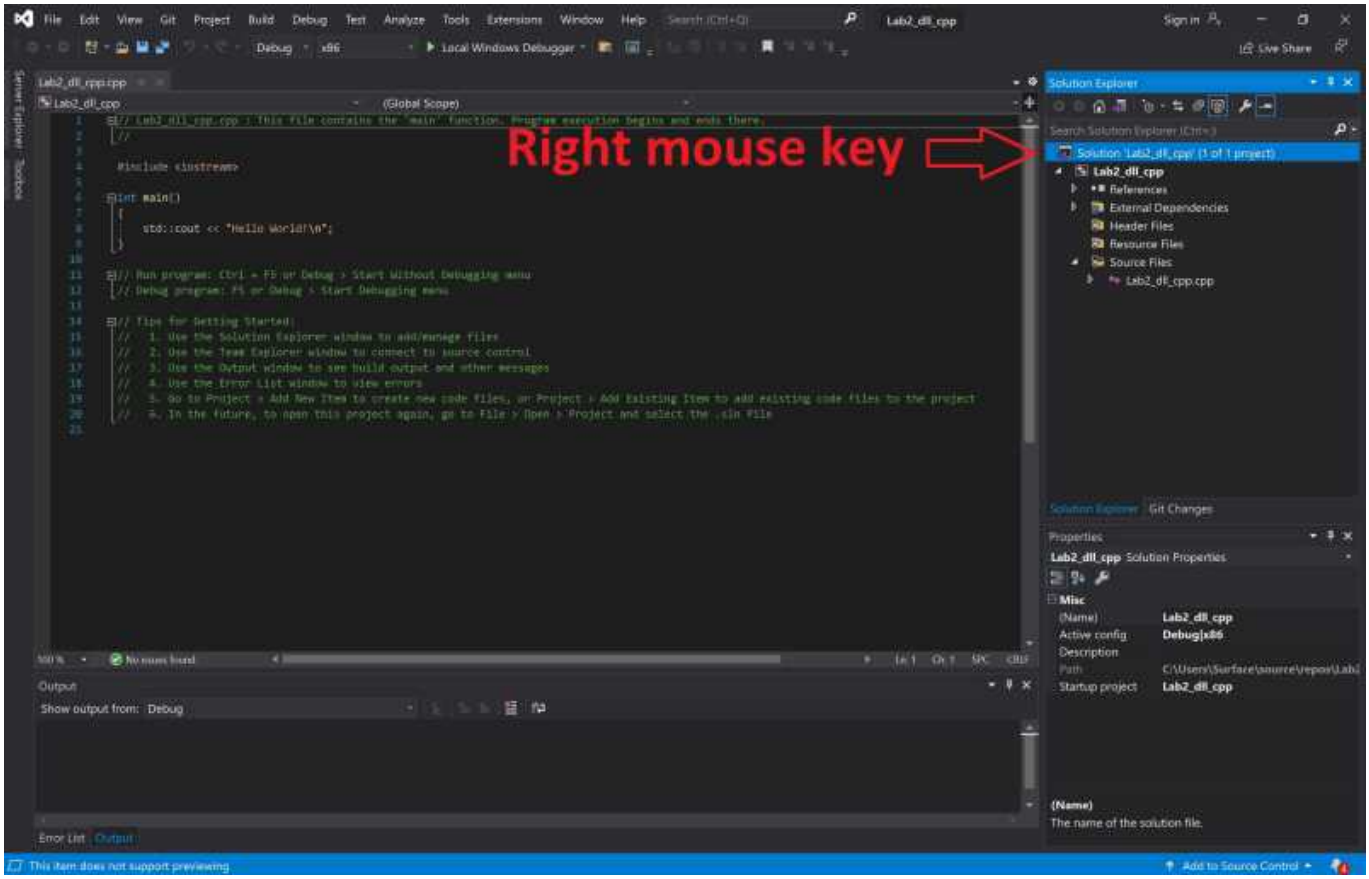


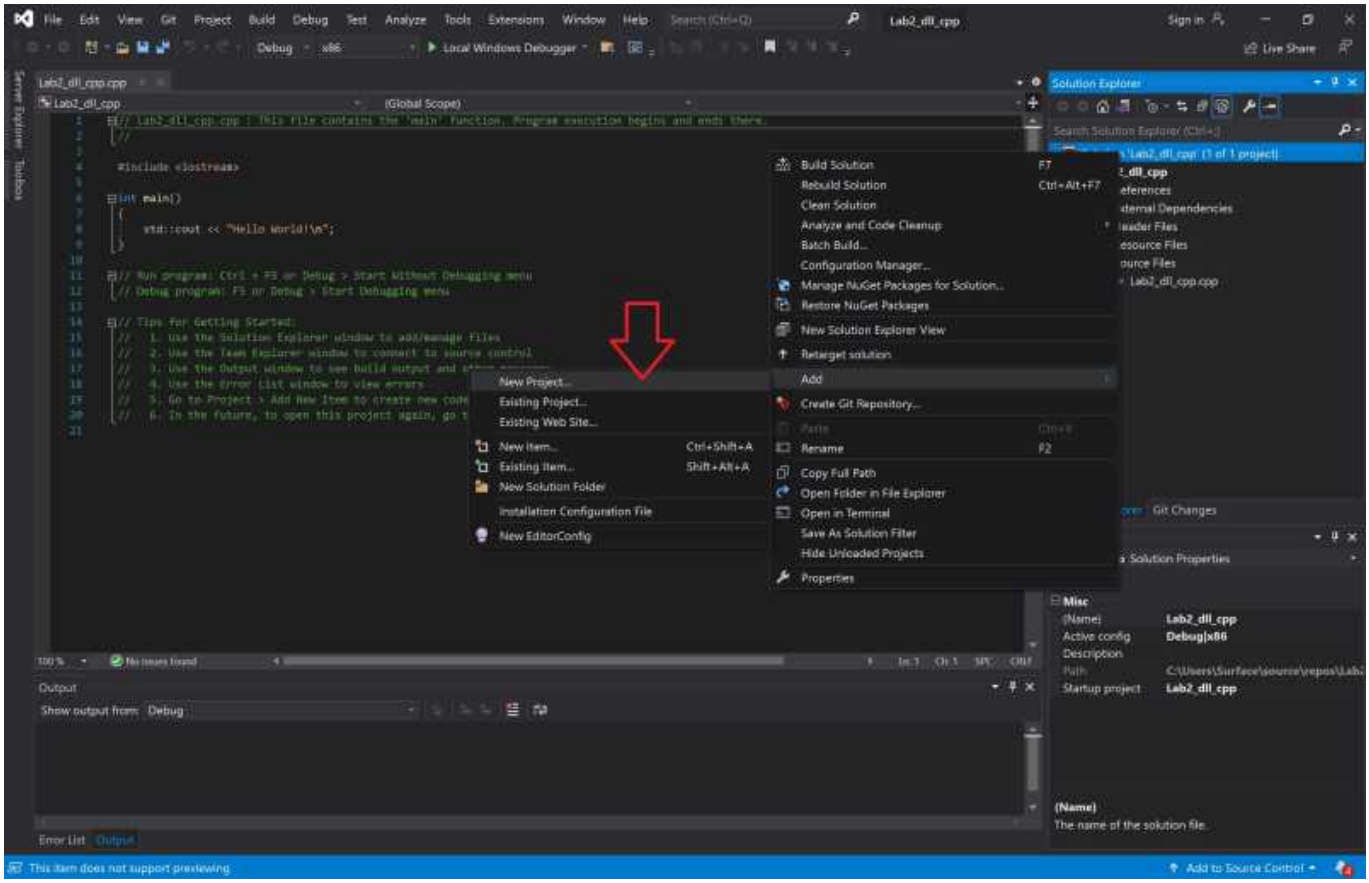


At this stage the solution will be created including the project with the C++ program template.

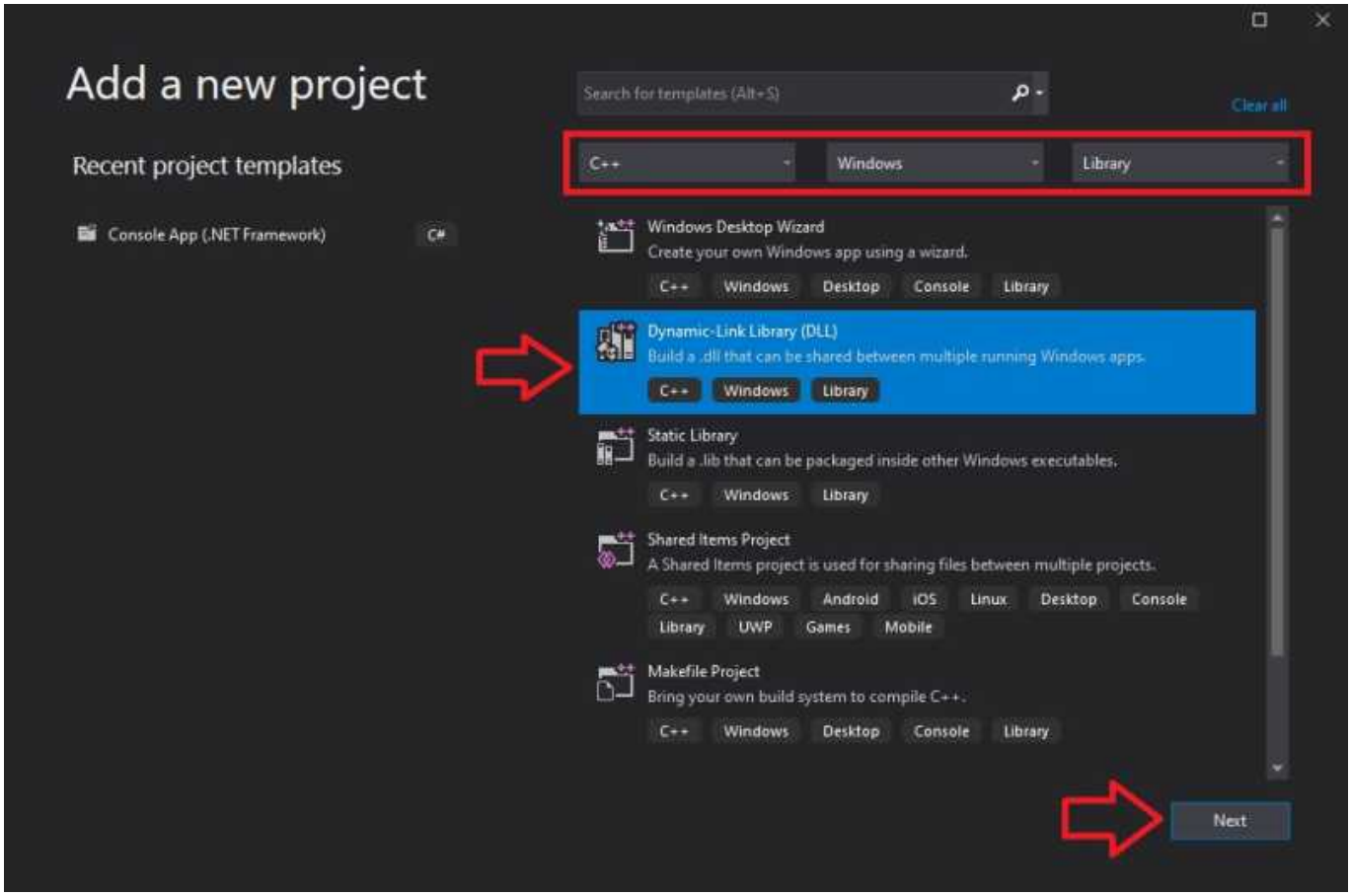
1.1.2. Dodanie do rozwiązania części asemblerowej

Aby w naszym rozwiązaniu dodać bibliotekę napisaną w asemblerze, dodajemy do solucji nowy projekt. Klikając prawym klawiszem myszy na rozwiązaniu wybieramy dodaj (add) nowy projekt (new project).

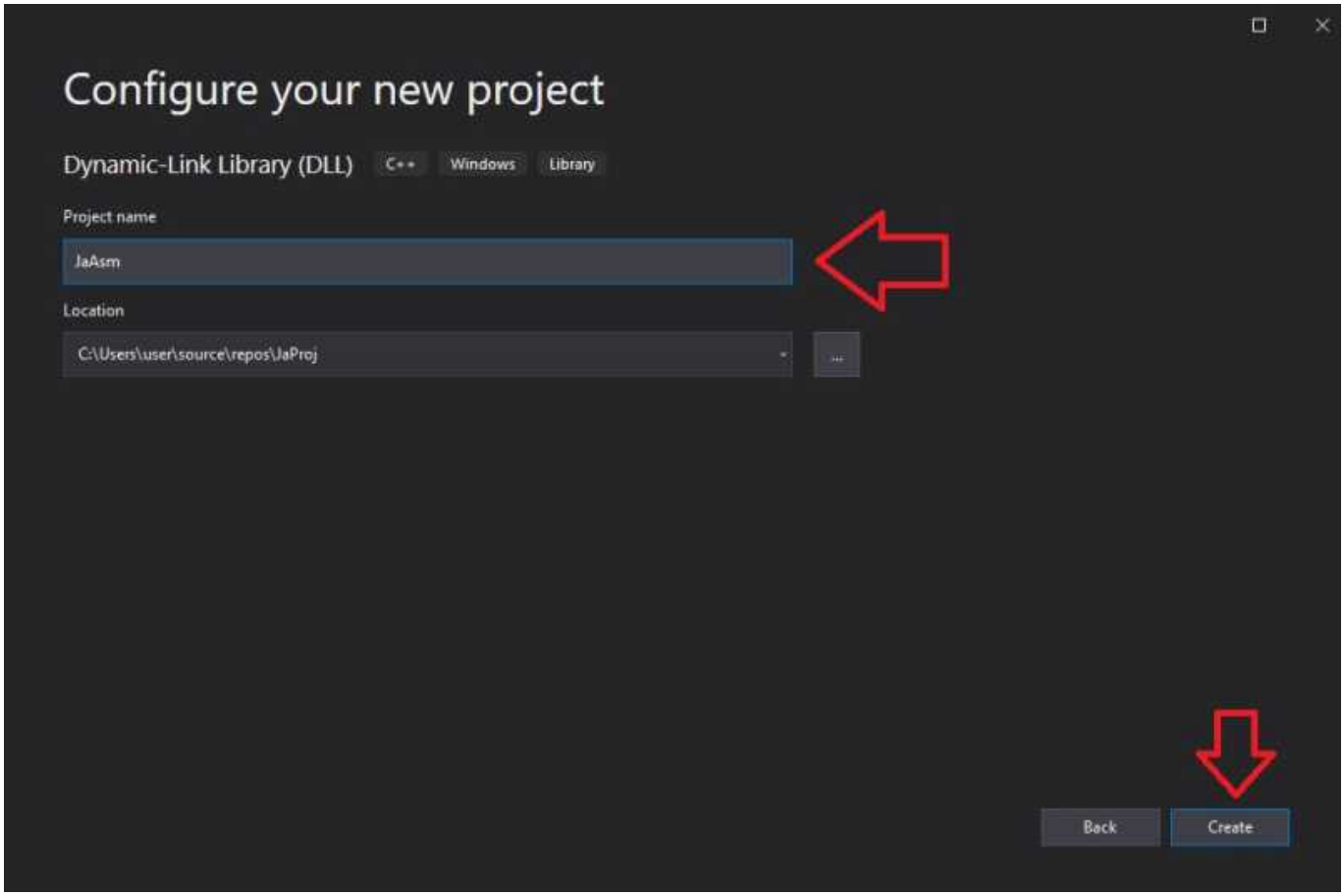




Nasz nowy projekt asemblerowy tworzymy jako projekt biblioteki (library) zgodny z C++, pracujący pod kontrolą systemu operacyjnego Windows. Z listy dostępnych opcji wybieramy bibliotekę dołączaną dynamicznie Dynamic-Link Library (DLL).



Nazywamy pojekt np. “JaAsm” i tworzymy go pozostawiając domyślną ścieżkę.

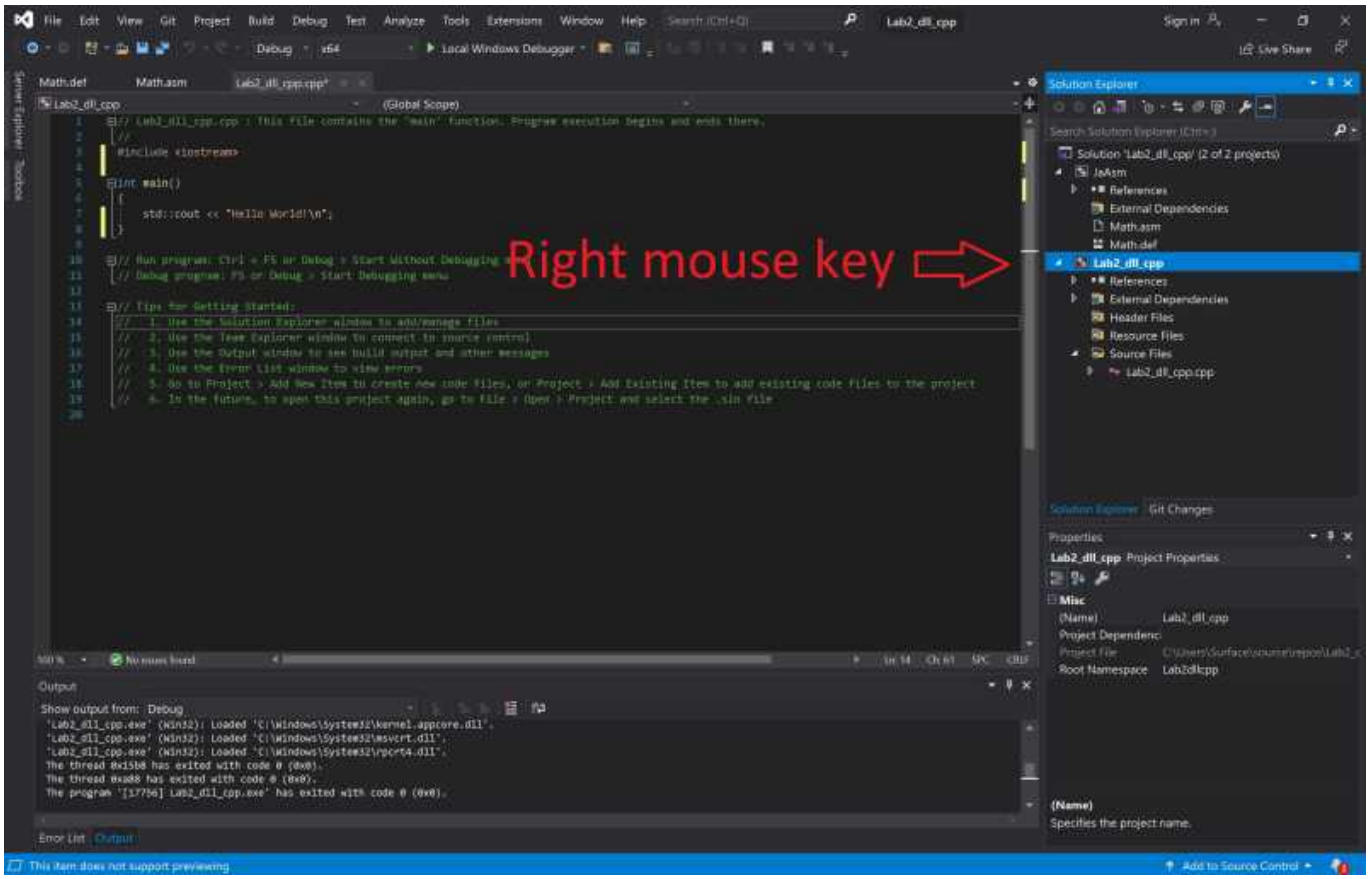


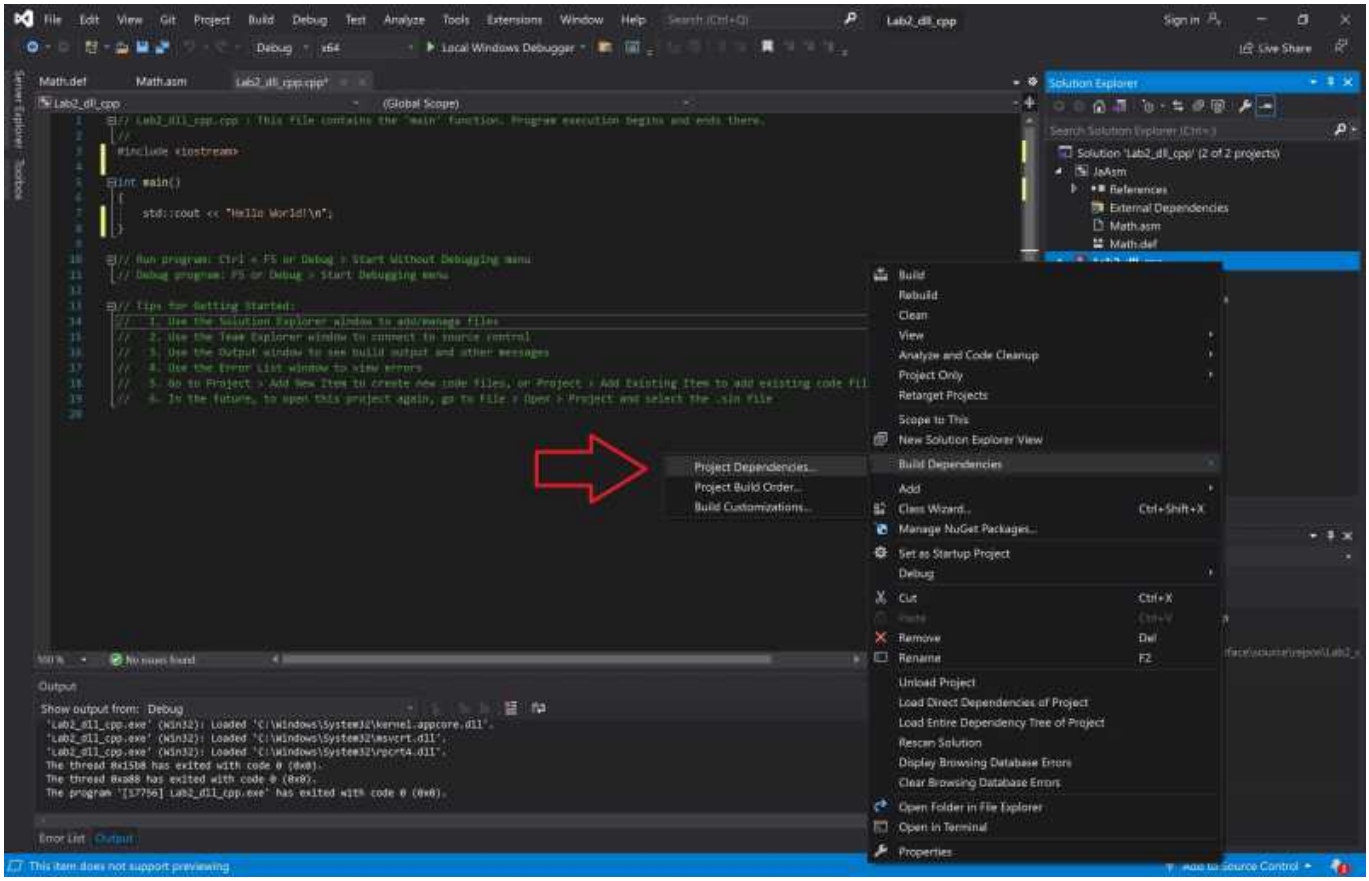


Obie części naszego rozwiązania będą umieszczone jako podfoldery w jednym folderze o nazwie zgodniej z nazwą rozwiązania, w naszym przypadku “Lab2_dll_cpp”. Każdy z tych podfolderów przyjmie nazwę odpowiednich projektów, czyli “Lab2_dll_cpp” dla części napisanej w C++ i “JaAsm” dla części napisanej w asemblerze.

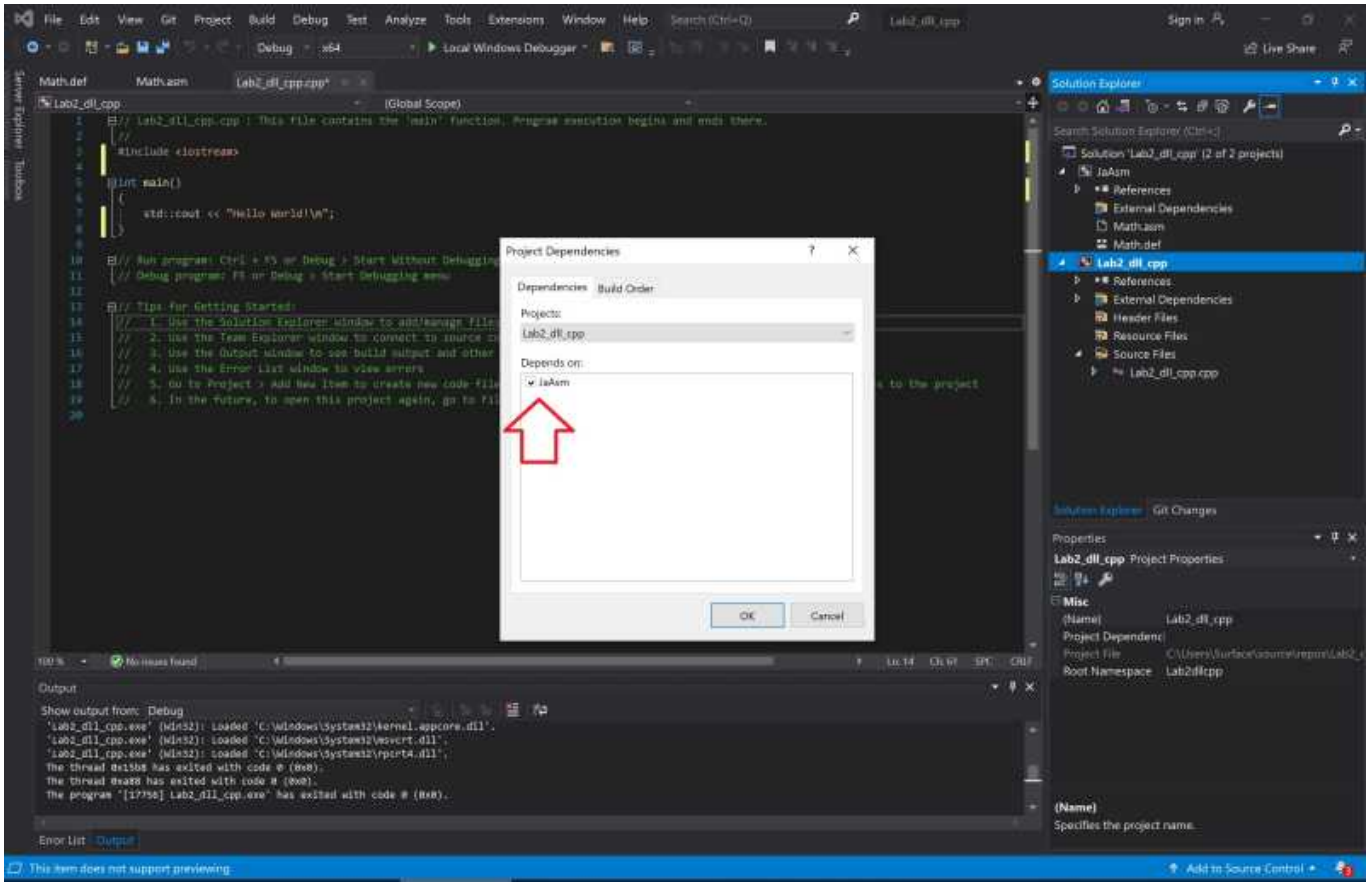
1.1.3. Ustawienie zależności w rozwiązaniu

Następnym krokiem jest ustawienie zależności projektu w taki sposób, aby kompilacja aplikacji konsolowej w C++ wymuszała automatycznie kompilację biblioteki asemblerowej. W tym celu należy kliknąć prawym klawiszem myszy na projekt konsolowy C++, wybrać menu „Zależności kompilacji” (Build dependencies), a następnie „Zależności projektu” (Project dependencies).





W nowo otwartym oknie zaznaczamy naszą bibliotekę “JaAsm” i klikamy „OK”

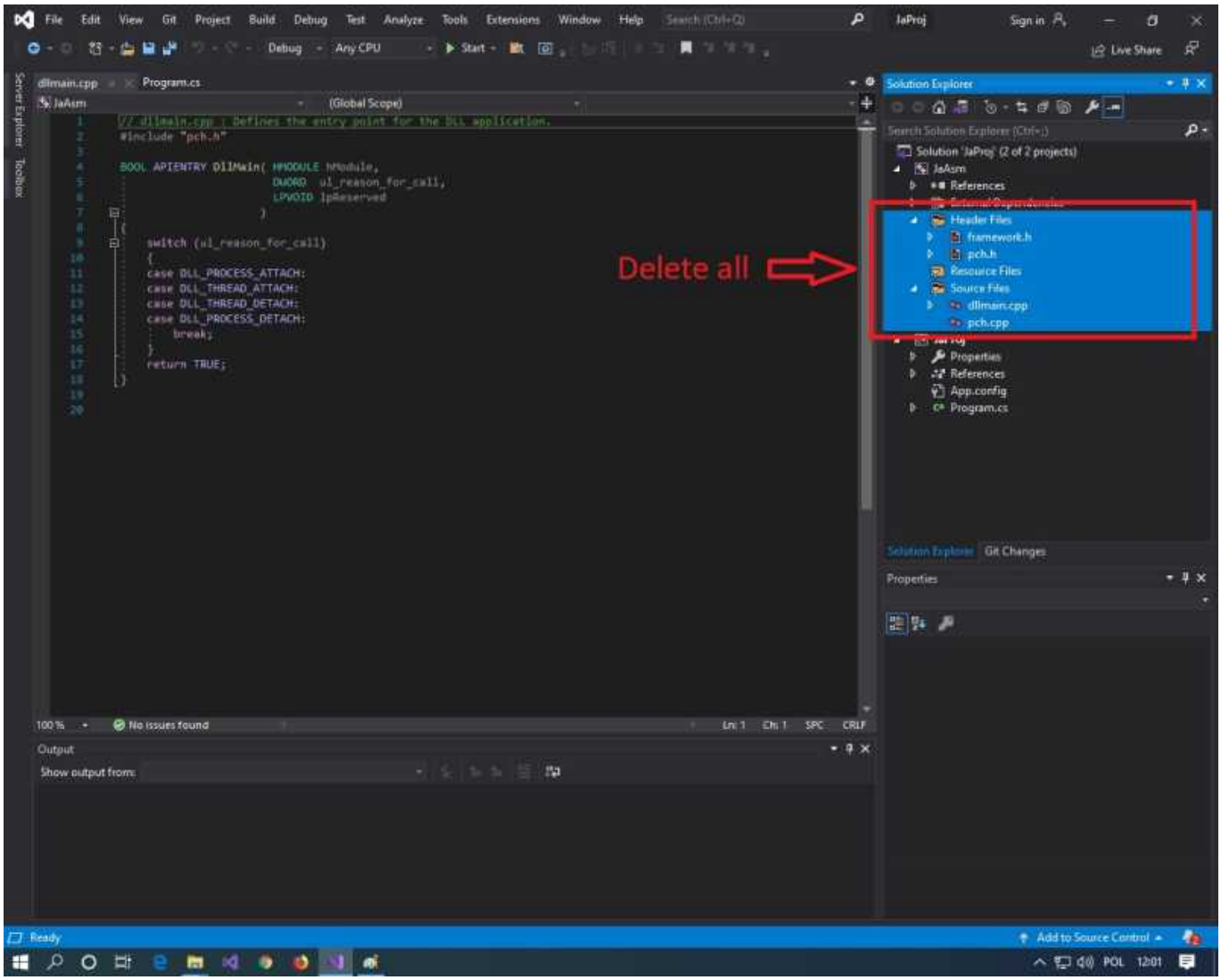


Po ustawieniu tej zależności, przy kompialcji projektu lub programu napisanego w C++

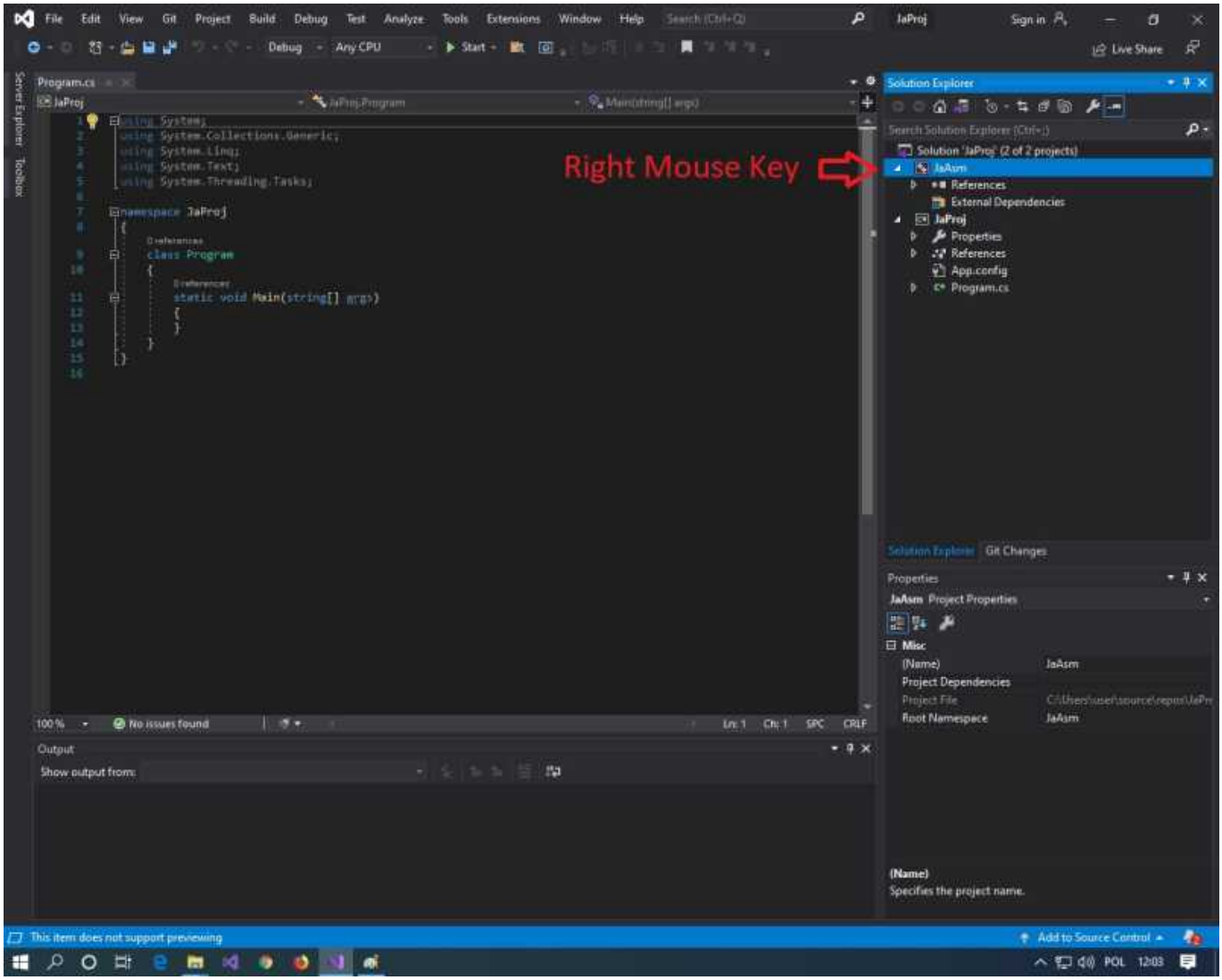
automatycznie zostanie także przekompilowany plik asemblerowy.

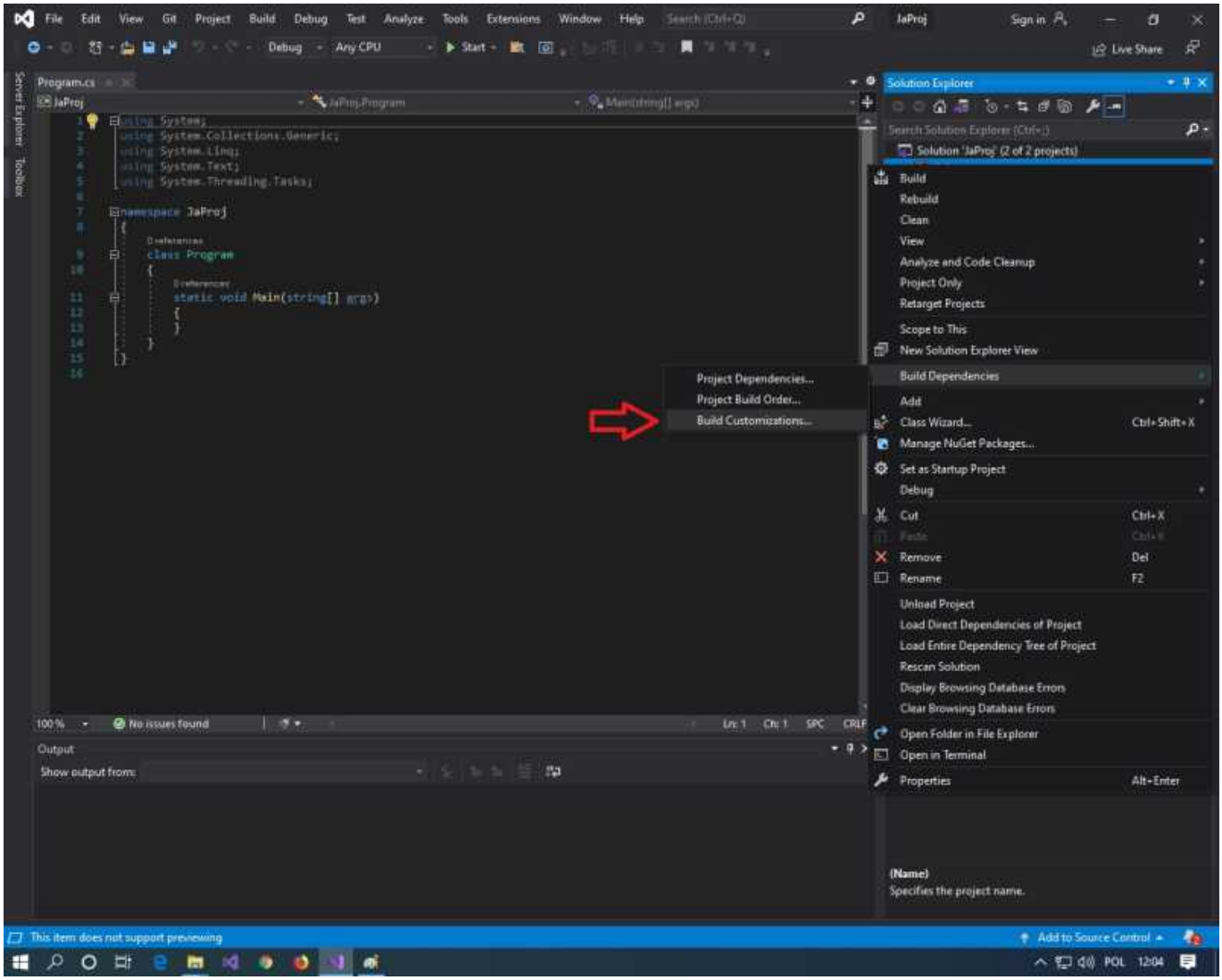
1.1.4. Zamiana biblioteki z C++ na asemblerową

Najpierw należy usunąć automatycznie utworzone pliki projektu zgodnego z C++. W tym celu wystarczy zaznaczyć wskazane pliki nagłówkowe i źródłowe, a następnie prawy klawisz myszy i wybrać „Usuń” bądź też kliknąć Delete.

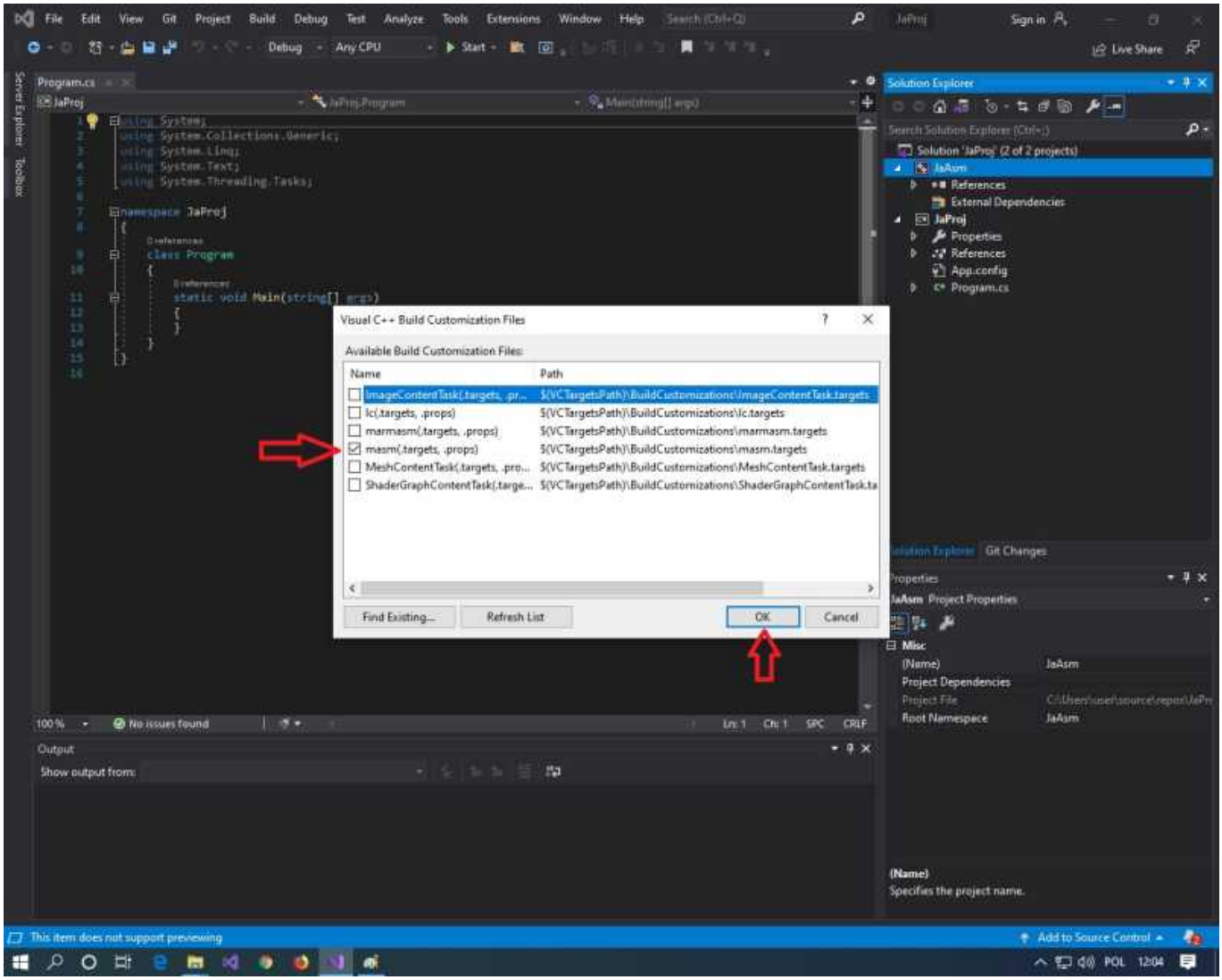


Dla biblioteki konieczne jest dostosowanie kompilacji jako plik asemblerowy MASM. W tym celu klikamy projekt biblioteki prawym klawiszem myszy i z listy wybieramy „Zależności kompilacji” (Build dependencies), a następnie „Dostosowanie kompilacji” (Build customization).

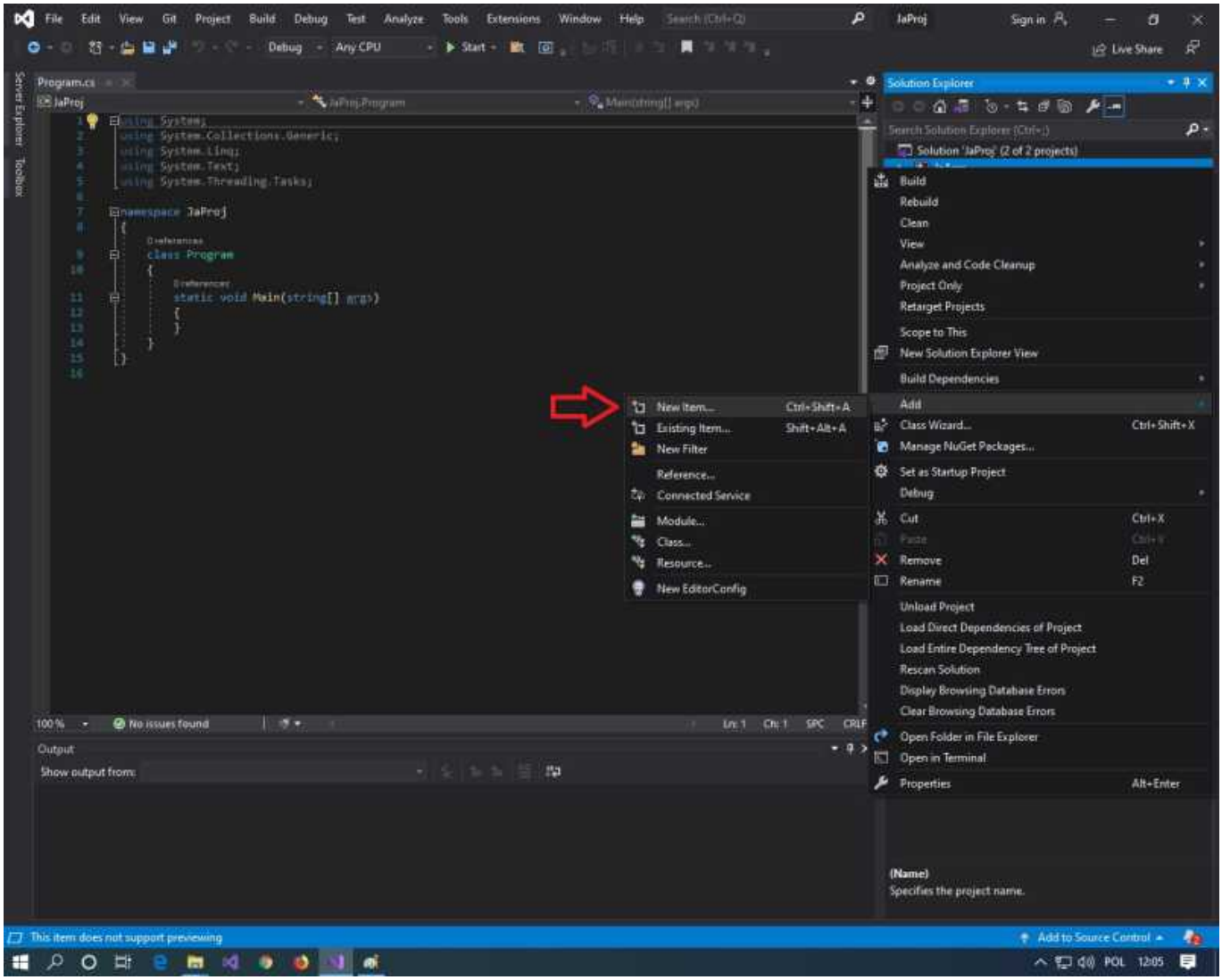

Right Mouse Key



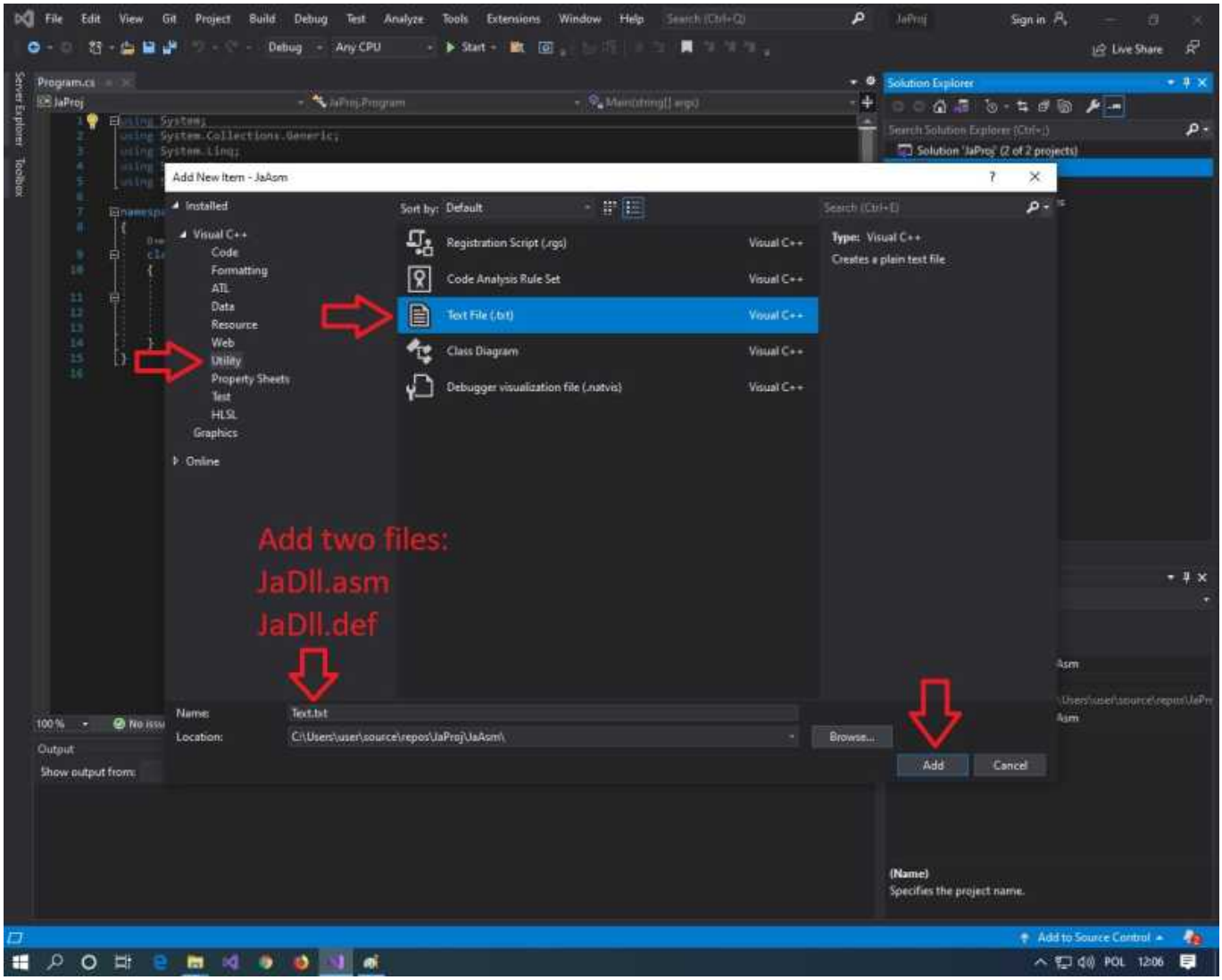
W nowo otwartym oknie zaznaczamy „masm” i potwierdzamy nasz wybór „OK”.



Kolejnym krokiem jest dodanie do biblioteki plików źródłowego w asemblerze oraz pliku definicji. Aby to zrobić ponownie klikamy prawym przyciskiem myszy na bibliotekę “JaAsm”, a następnie wybieramy „Dodaj” (Add) oraz „Nowy element” (New item).



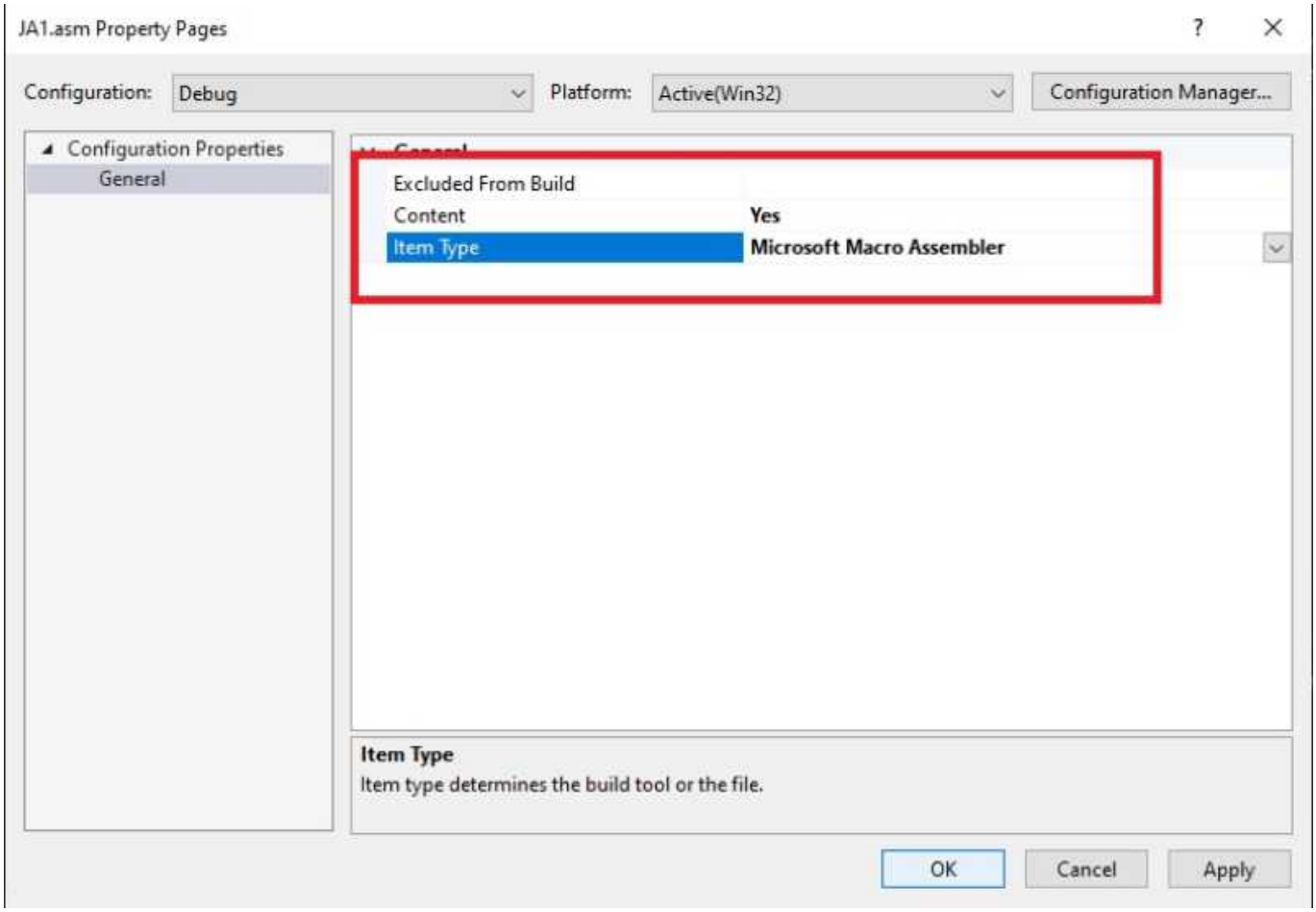
Czynność tę powtarzamy dwukrotnie wybierając za każdym razem “Narzędzie” (Utility) i typ pliku tekstowy, podając nazwy: “JaDll.asm” oraz “JaDll.def”.





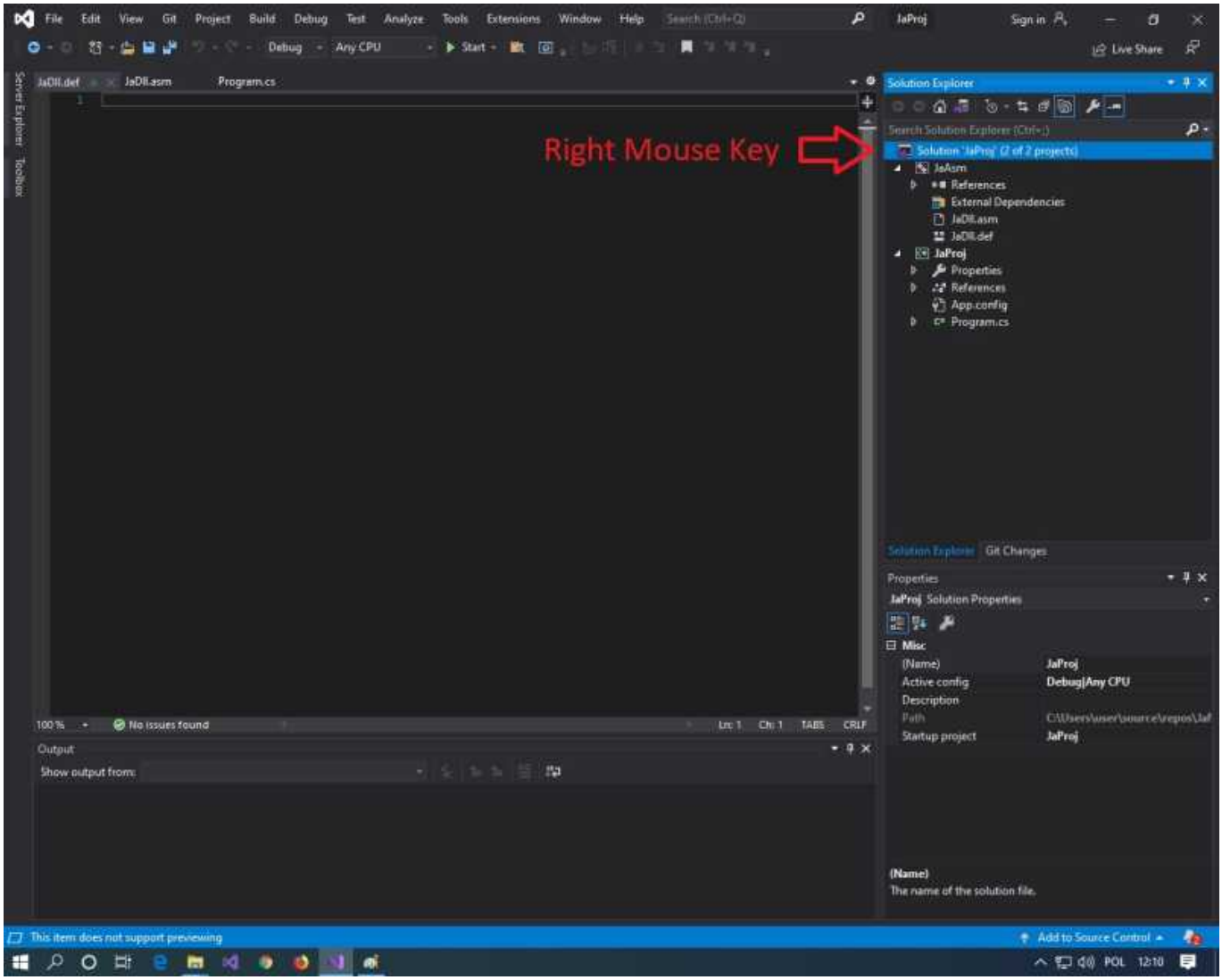
Przy wykonaniu czynności zgodnie z podaną uprzednio kolejnością, z dużym prawdopodobieństwem krok przedstawiony na kolejnym rysunku nie będzie konieczny.

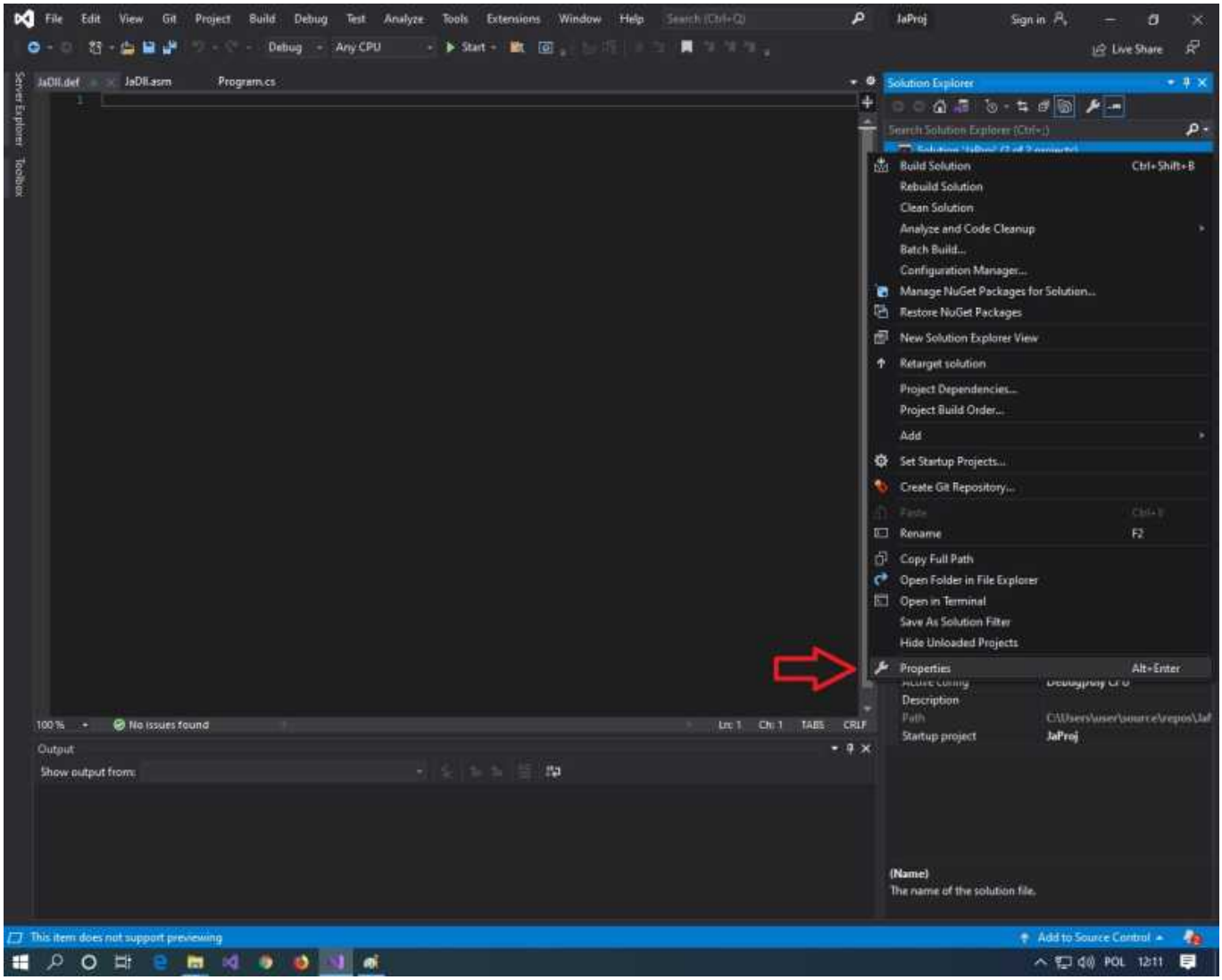
Aby kompilator wiedział, że stworzony plik asemblerowy jest skojarzony z MASM, należy wskazać jego typ. Krok ten jest niezbędny gdy wykorzystujemy plik źródłowy C++ i zmieniamy jedynie jego rozszerzenie na *.asm. Klikając pracym klawiszem na pliku *.asm wybieramy properties i ustawiamy typ pliku na Microsoft Macro Assembler.



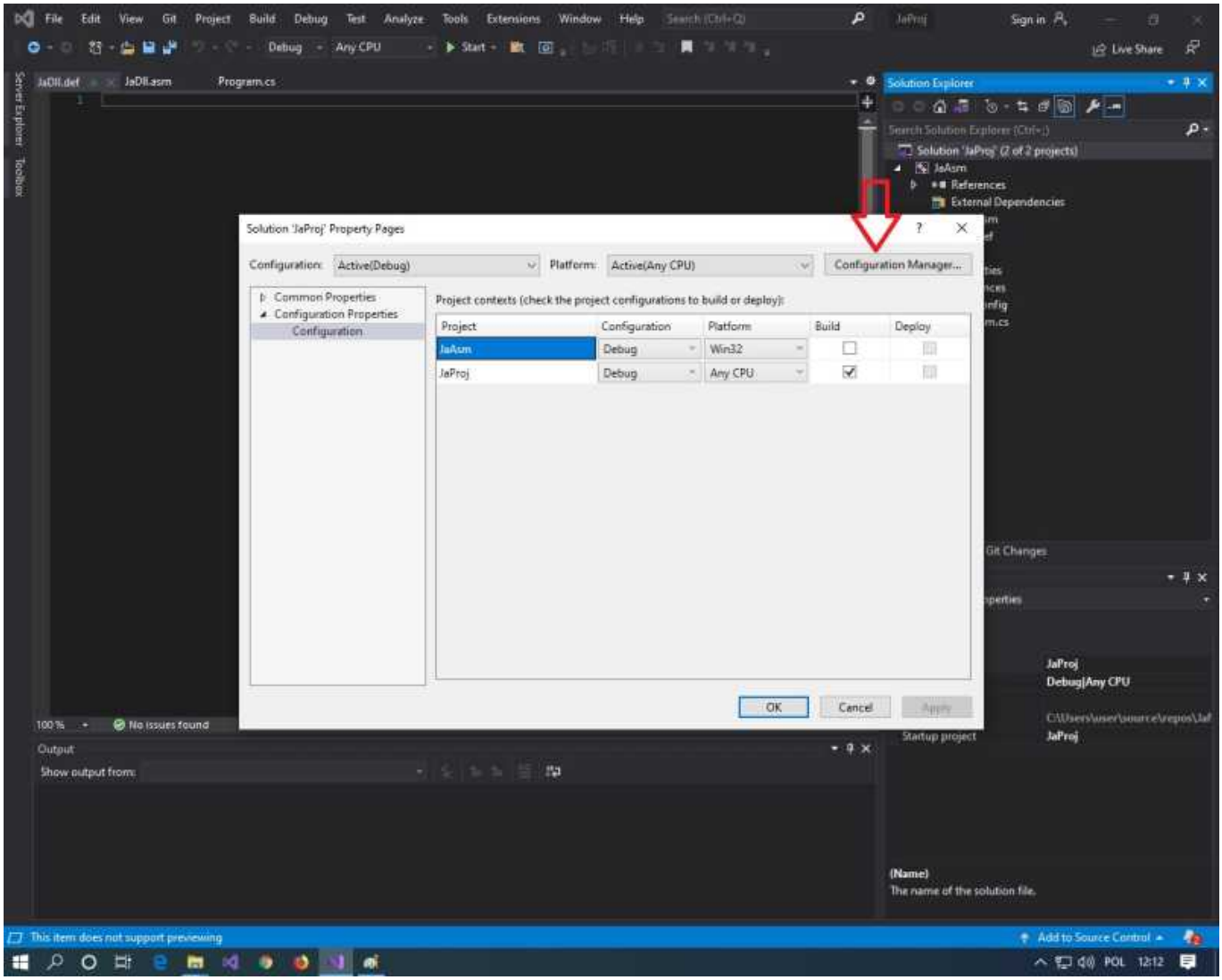
1.1.5. Ustawienie trybu kompilacji

Kompilacja powinna przebiegać w trybie 64-bitowym (X64). W tym celu należy ustawić taki tryb w opcjach rozwiązania. Klikamy prawym przyciskiem myszy na nasze rozwiązanie i wybieramy „Właściwości” (Properties).

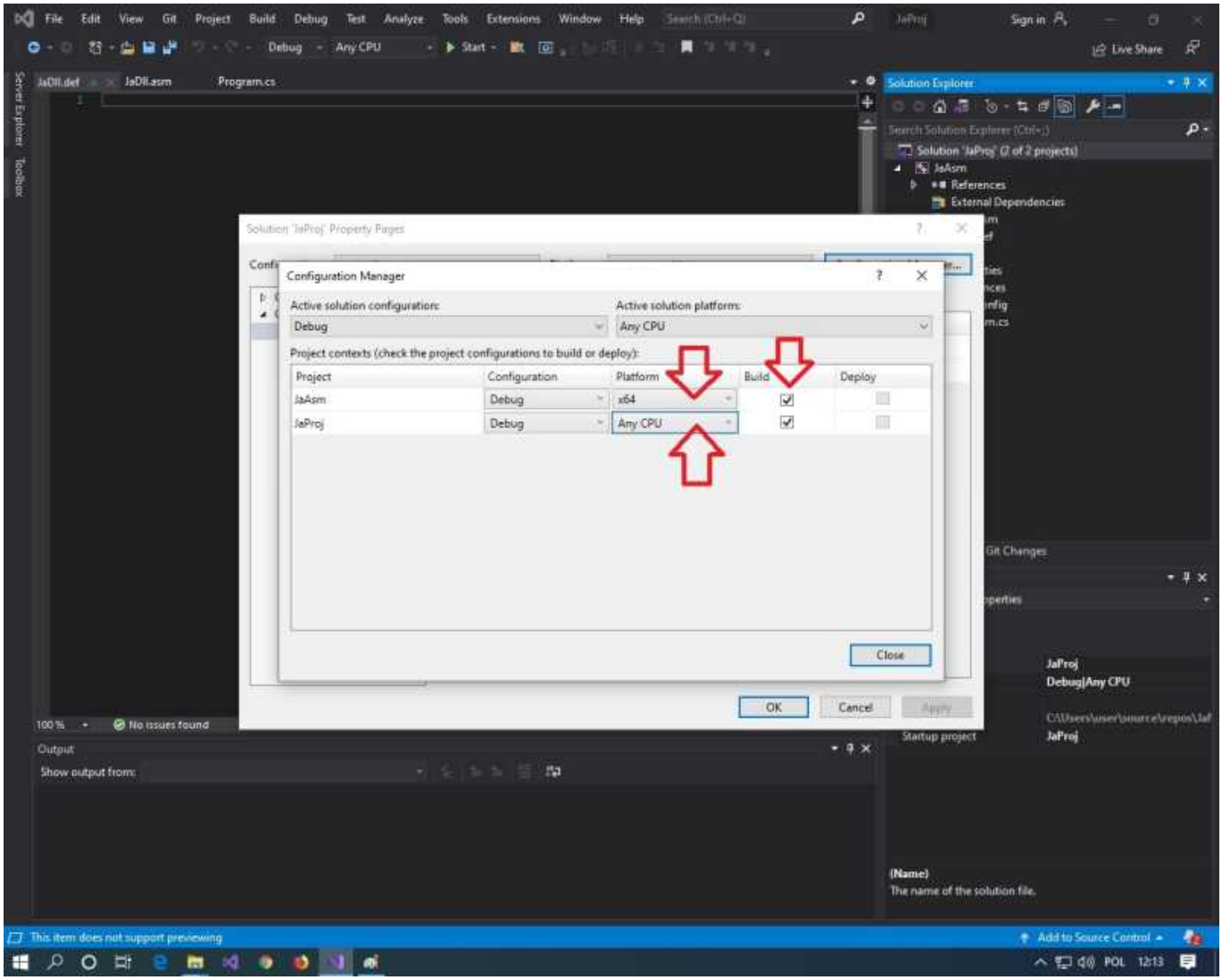

Right Mouse Key

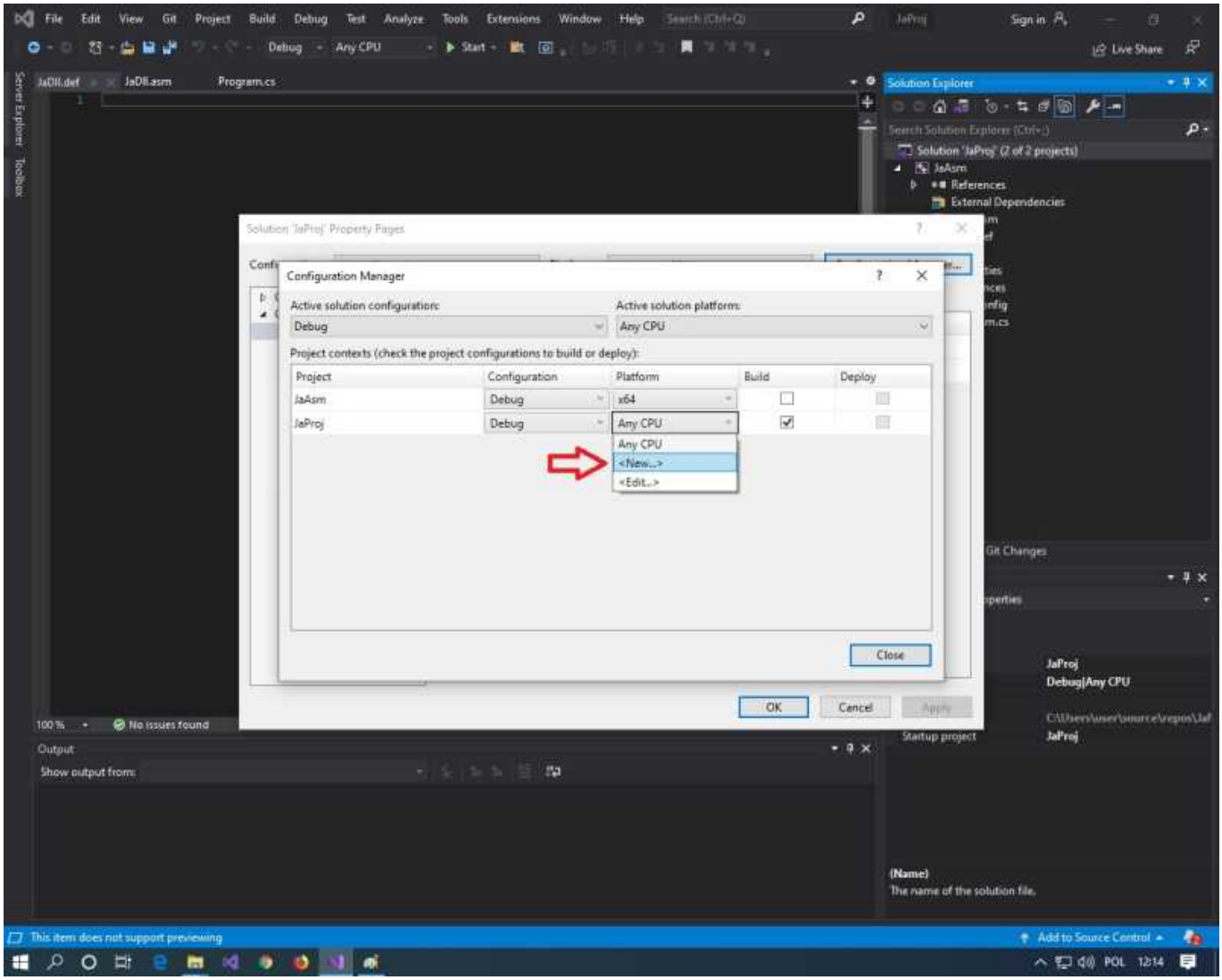


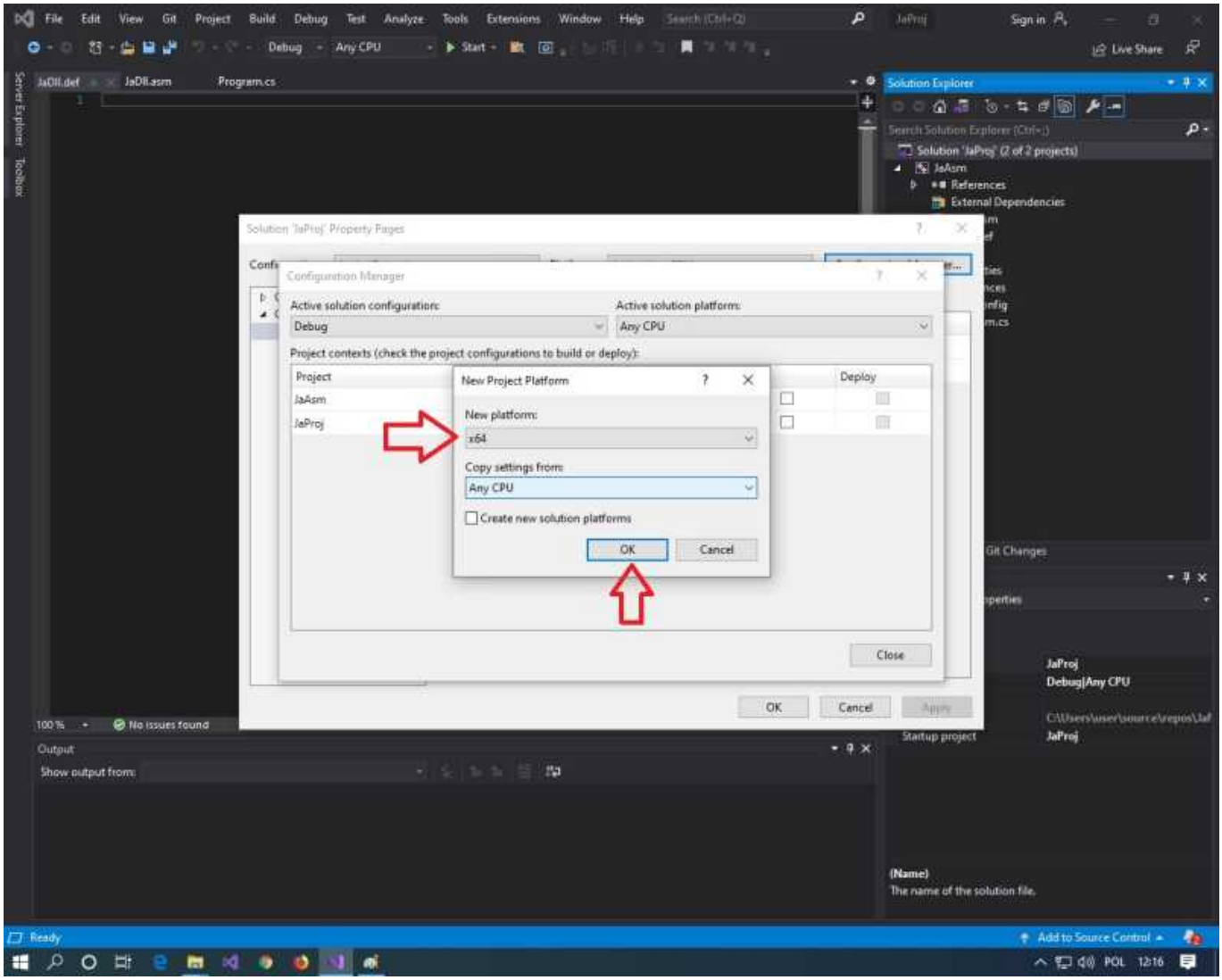
W nowo otwartym oknie przechodzimy na zakładkę „Właściwości konfiguracji” → „Konfiguracja” (Configuration properties → Configuration) na następnie klikamy w „Menedżer konfiguracji” (Configuration manager).



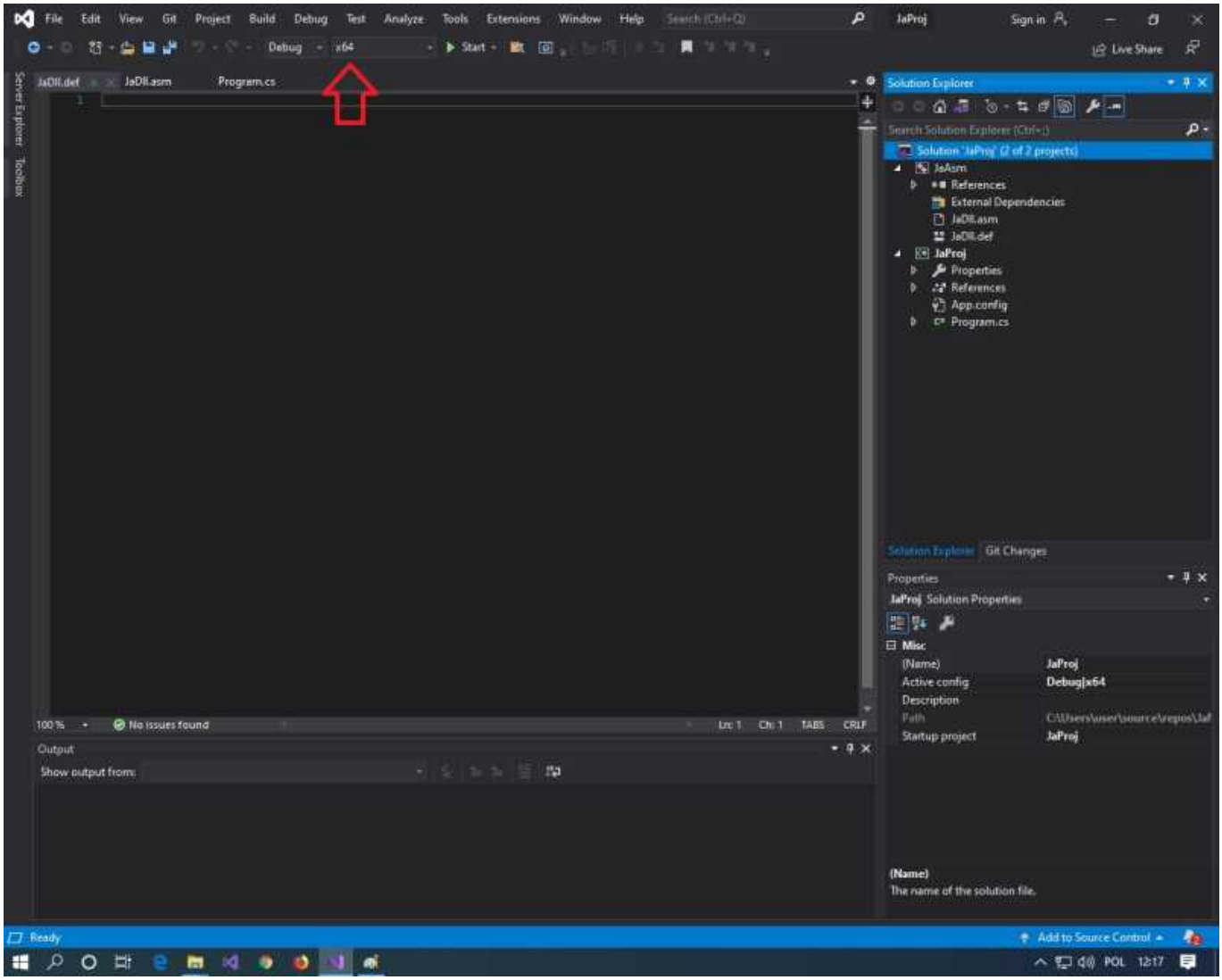
W nowym oknie wskazujemy wszędzie platformę x64 (gdyby dana opcja była niedostępna z dostępnej listy wybieramy pozycję „Nowy” (New), a następnie tworzymy platformę x64 poprzez kopiowanie ustawień z Any CPU). Należy także przy obu częściach solucji włączyć kompilację (Build).


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

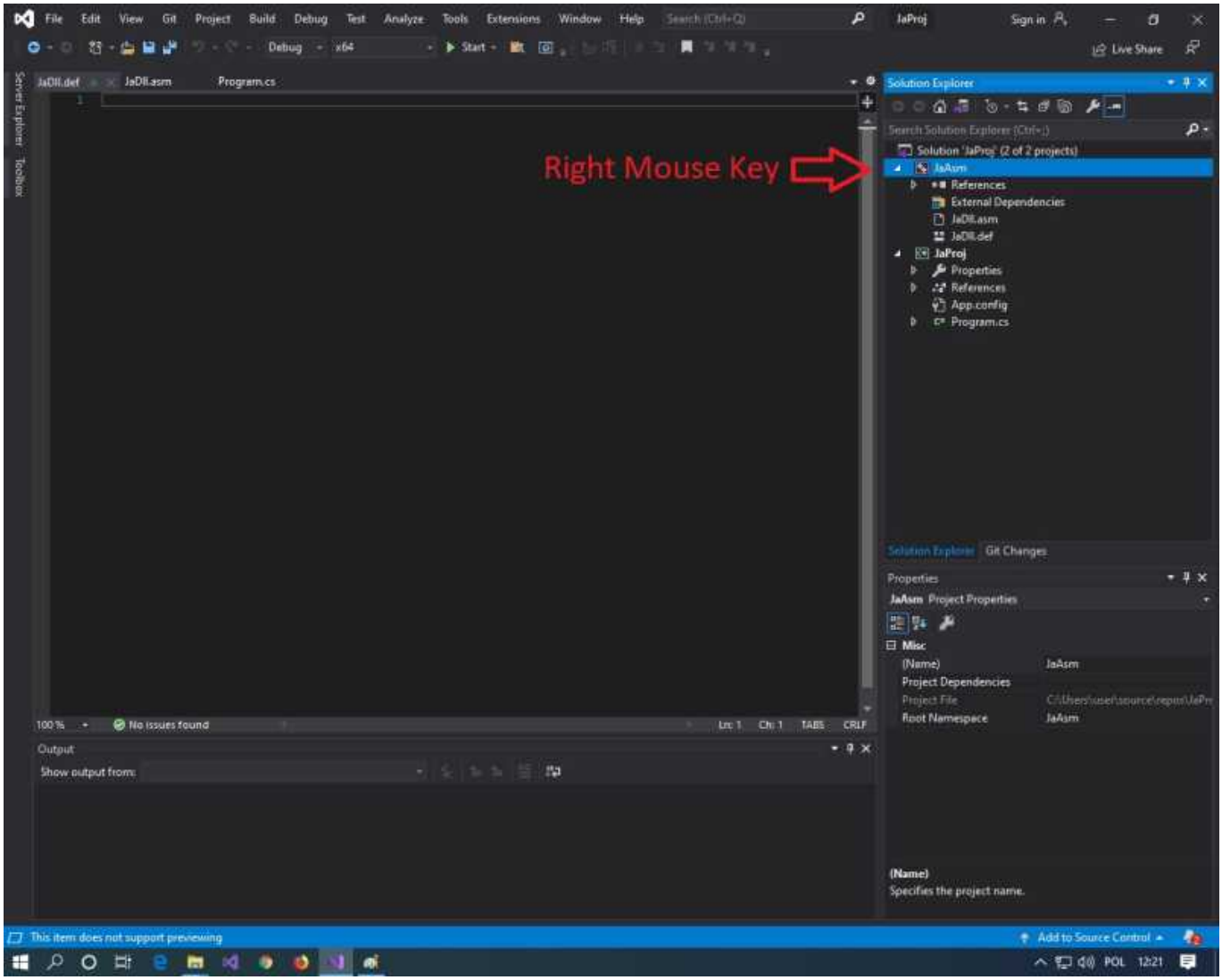


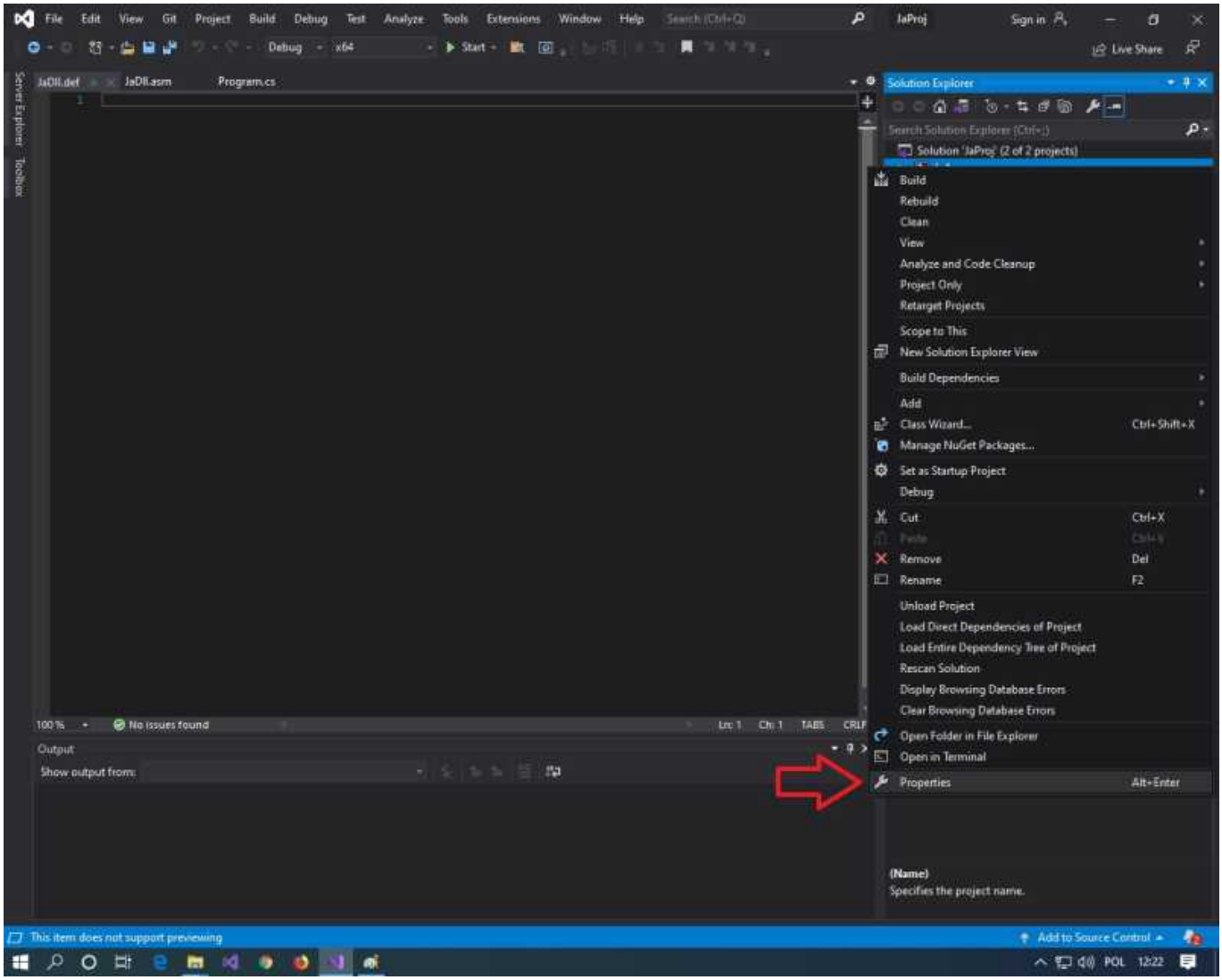
Po wszystkim upewniamy się raz jeszcze, że wszędzie wskazana platforma to x64 i zamykamy okno. W głównym pasku menu Visual Studio powinna automatycznie ustawić się opcja “x64”, niemniej warto to sprawdzić i w razie potrzeby zmodyfikować.



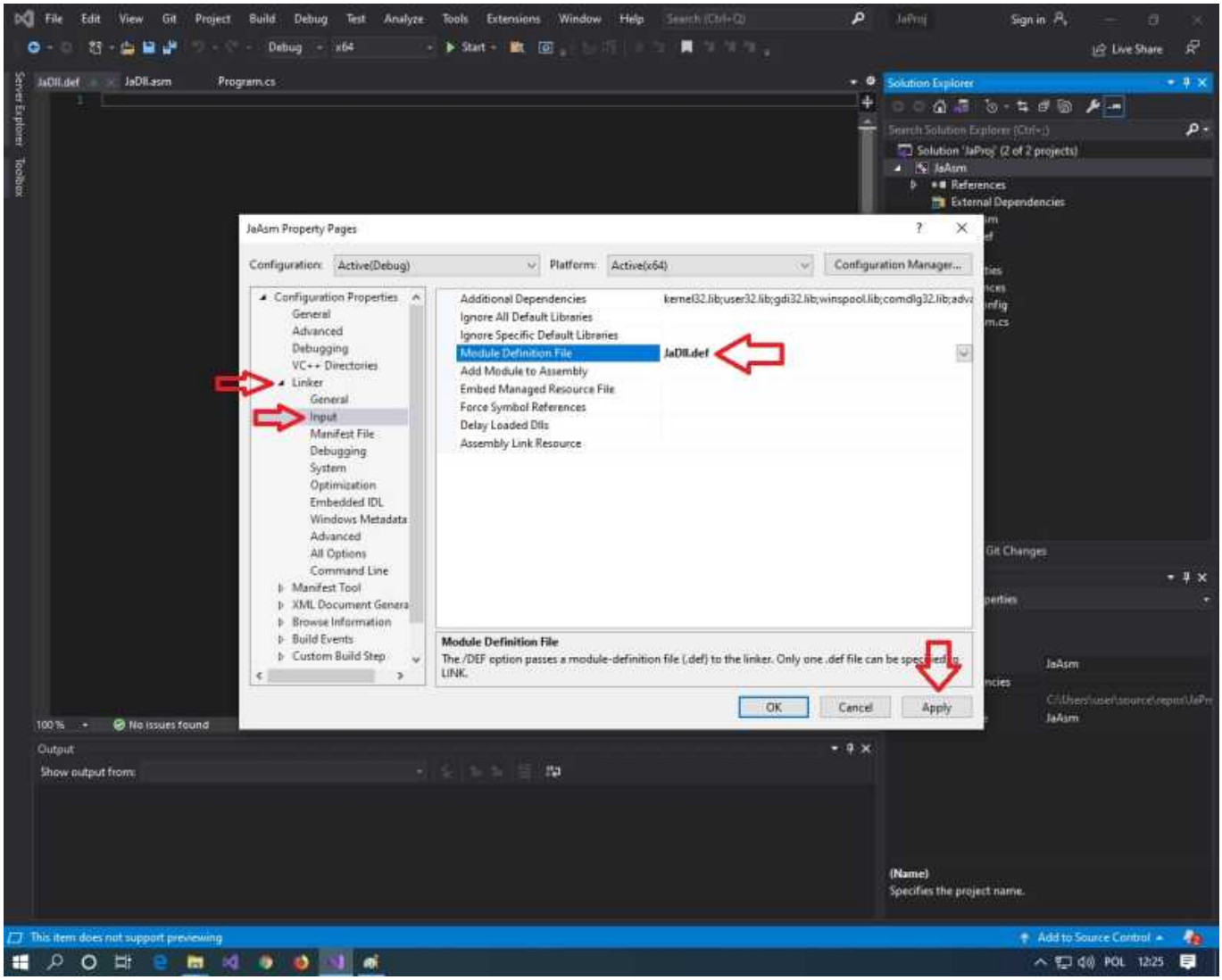
1.1.6. Końcowa konfiguracja biblioteki asemblerowej

Przystępujemy do konfiguracji biblioteki. W tym celu klikamy prawym przycskiem myszy na bibliotece i wybieramy „Właściwości” (Properties).

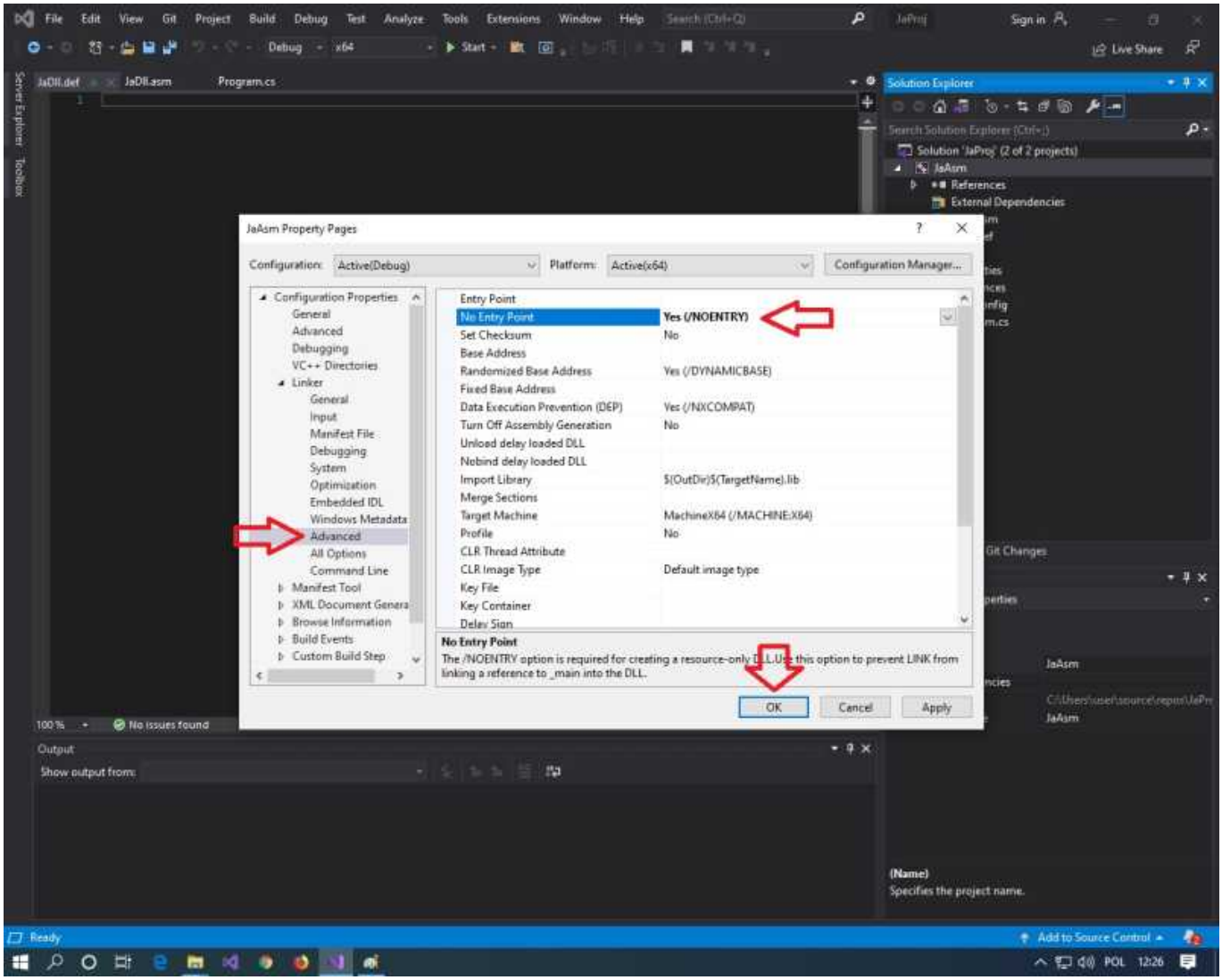

Right Mouse Key



W otwartym nowo oknie przechodzimy do „Konsolidator” → „Dane wejściowe” (Linker → Input) i w polu „Plik definicji modułu” (Module definition file) wpisujemy nazwę naszego pliku „*.def”. W naszym przypadku “JaDll.def”. Wpis potwierdzamy.



Końcowym krokiem konfiguracji jest wyłączenie punktu wejścia do biblioteki. W tym celu w zakładce “Konsolidator” (Linker) przechodzimy do opcji “Zaawansowane” (Advanced) i w polu „Brak punktu wejścia” (No entry point) wybieramy opcję „Tak (/NOENTRY)”. Zapisujemy wprowadzone ustawienia i zamykamy okno.





Biblioteka dll, w przeciwieństwie do programu głównego, nie ma punktu (adresu) startowego, który wykorzystuje system operacyjny do uruchomienia programu. Udostępnia jedynie zestaw funkcji dla innych programów.

1.2. Wykorzystanie biblioteki dll napisanej w asemblerze

1.2.1. Biblioteka asemblerowa dll

Biblioteka dll zawiera funkcje, które można użyć w innych programach. Pierwsza wersja naszej biblioteki będzie udostępniała jedną funkcję, zwracającą stałą wartość. Wartość tę następnie odczytamy w pliku C++. Przykładowa zawartość pliku asemblerowego (JaDll.asm) może wyglądać następująco.

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

```
end
```

Z plikiem kodu programu nierozerwalnie związany jest plik definicji (JaDll.def). Plik taki musi zawierać nazwę biblioteki DLL jako agrument dyrektywy LIBRARY, oraz listę nazw funkcji dostępnych dla innych modułów poprzedzonych dyrektywą EXPORTS. Aby nasza funkcja MyProc była widziana przez program C++ musimy ją zadeklarować jako eksportowaną. Plik JaDll.def w naszym projekcie może wyglądać następująco.

```
LIBRARY JaAsm
EXPORTS MyProc
```

Często chcemy wykorzystać przekazane do funkcji parametry. Przypomnijmy, że w 64-bitowej konwencji wywołań pierwsze cztery parametry są umieszczane bezpośrednio w rejestrach procesora. Szczegóły są opisane w instrukcji do pierwszego ćwiczenia. I tak, dwa parametry typu całkowitego są umieszczone w rejestrach RCX i RDX. Funkcja, któa zwraca sumę dwóch argmentów całkowitych wyglądałaby w następujący sposób.

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

Argumenty typu zmiennoprzecinkowego należy przekazać przez rejestry XMM0 i XMM1, a wynik zwrócić przez XMM0. Oczywiście należy użyć instrukcji rozszerzeń MMX/SSE.

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

1.2.2. Odwołanie do biblioteki dll z programu C++

Aby program napisany w C++ mógł wykorzystać funkcję z biblioteki należy wykonać kilka kroków.

1. Przygotować typ funkcji zgodny z jej późniejszym wywołaniem,
2. Załadować bibliotekę funkcją LoadLibrary uzyskując jednocześnie jej uchwyt,
3. Odczytać aktualny adres interesującej nas funkcji wywołując GetProcAddress,
4. Użyć funkcji odwołując się do jej adresu.

Przykładowy kod mógłby wyglądać tak:

```
#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();
}
```

Użycie funkcji z parametrami wymaga nieco zmodyfikowanych definicji typu. Należy także zwrócić uwagę na typ wartości zwracanej przez funkcję. Pamiętać należy, że w przypadku liczb zmiennoprzecinkowych, rozmiar rejestrów XMM0, XMM1 to 64 bity. Odpowiada to wartości zmiennoprzecinkowej podwójnej precyzji, czyli w języku C++ typowi “double”.

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

1.2.3. Zadanie do zrealizowania w ramach laboratorium

Należy napisać bibliotekę DLL w asemblerze i program w C++ demonstrujący działanie tej biblioteki.
Biblioteka DLL ma udostępniać funkcje wykonujące i zwracające wynik podstawowych obliczeń arytmetycznych (dodawanie, odejmowanie, mnożenie i dzielenie) na parach liczb przekazanych jako parametry ich wywołania. Funkcje powinny mieć wersję stało- i zmiennoprzecinkową (w sumie osiem funkcji).
Program w C++ ma wywoływać wszystkie funkcje przekazując przykładowe wartości parametrów i wyświetlić wyniki na ekranie. Nie jest wymagany interfejs użytkownika - wartości do obliczeń mogą być umieszczone w kodzie programu.

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=1732531577

Last update: **2024/11/25 11:46**