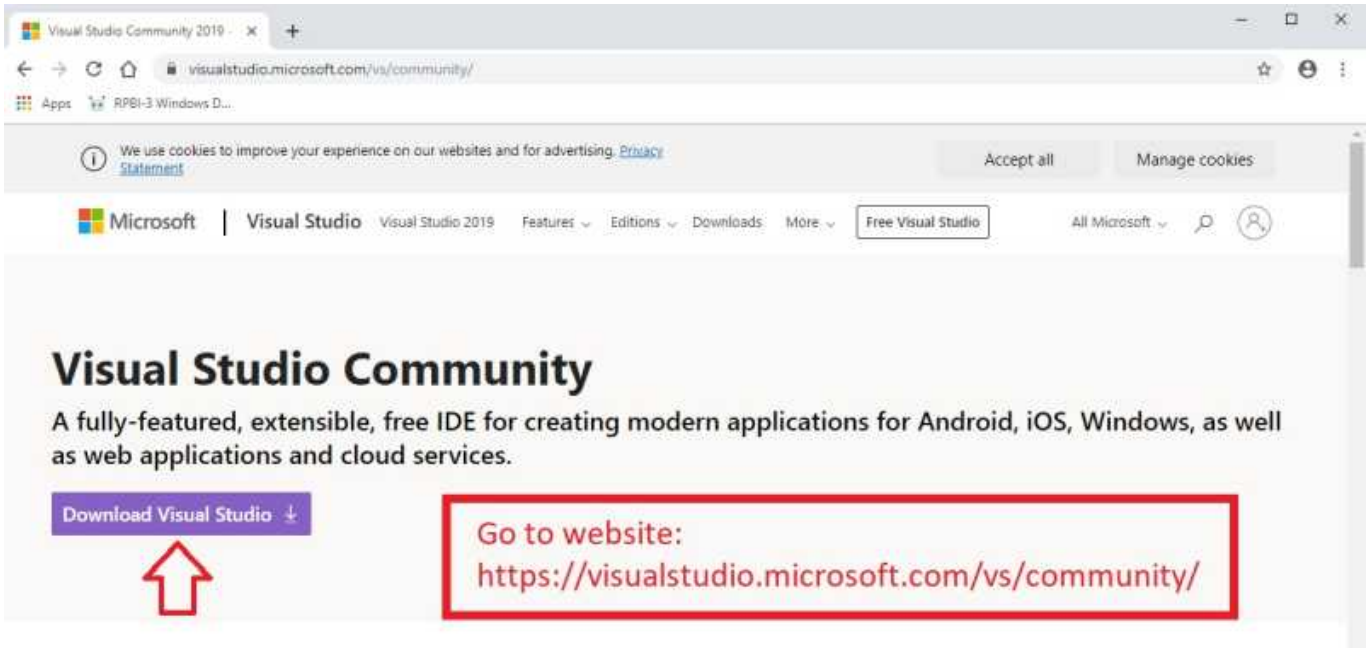
Table of Contents

1. Instalacja środowiska Visual Studio Community 2019 i konfiguracja do tworzenia bibliotek dll napisanych w MASM. 2
1.1. Instalacja 2
1.2. Utworzenie projektu z asemblerową biblioteką DLL 4
1.2.1. Utworzenie nowego projektu 5
1.2.2. Dodanie do rozwiązania części asemblerowej 7
1.2.3. Ustawienie zależności w rozwiązaniu 10
1.2.4. Zamiana biblioteki z C++ na asemblerową 12
1.2.5. Ustawienie trybu kompilacji 18
1.2.6. Końcowa konfiguracja biblioteki asemblerowej 25
1.3. Wykorzystanie biblioteki dll napisanej w asemblerze 29
1.3.1. Biblioteka asemblerowa dll 29
1.3.2. Odwołanie do biblioteki dll z programu C++ 30

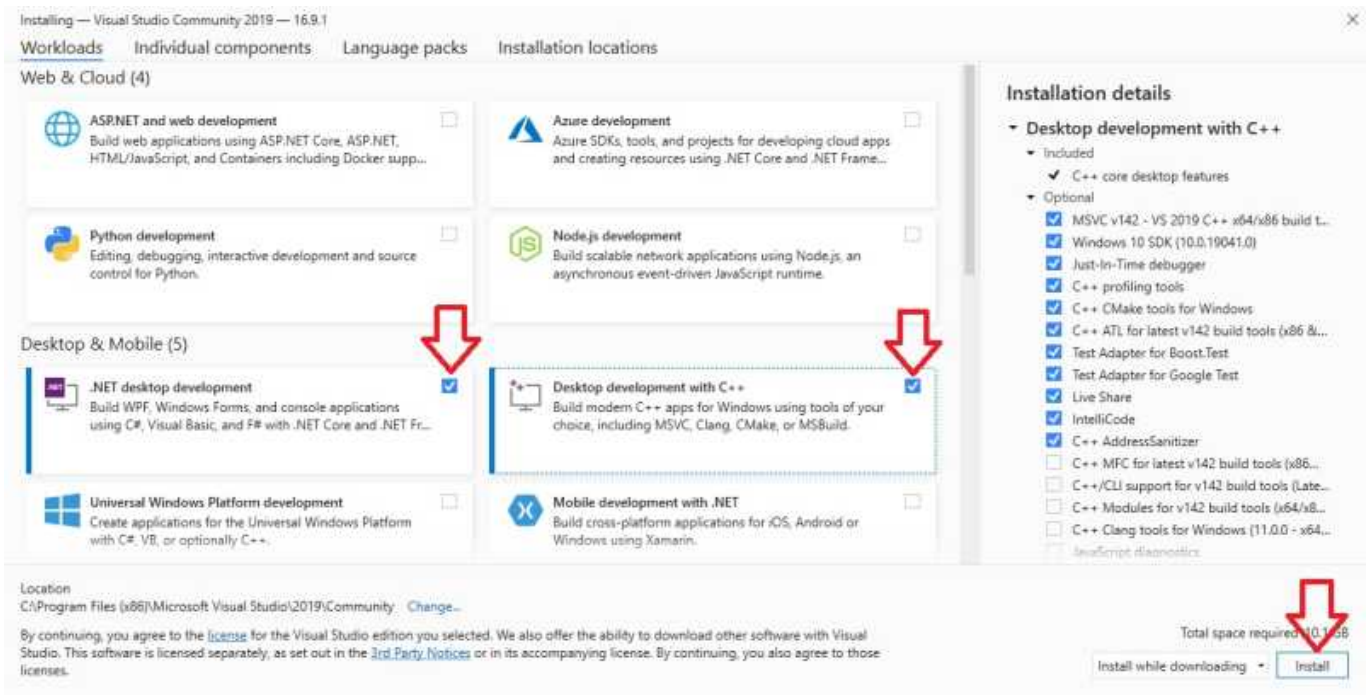
1. Instalacja środowiska Visual Studio Community 2019 i konfiguracja do tworzenia bibliotek dll napisanych w MASM.

1.1. Instalacja

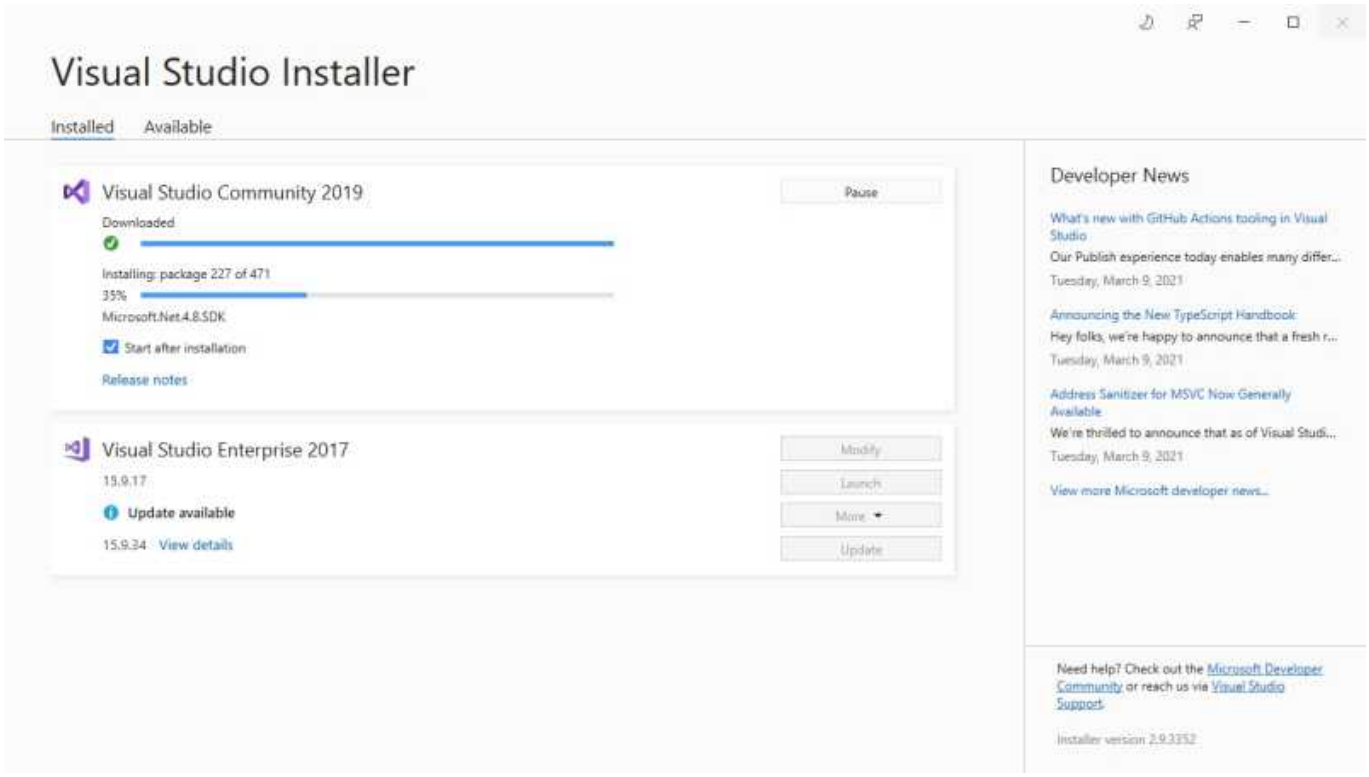
Rozpoczynamy od ściągnięcia instalatora ze strony Microsoft.



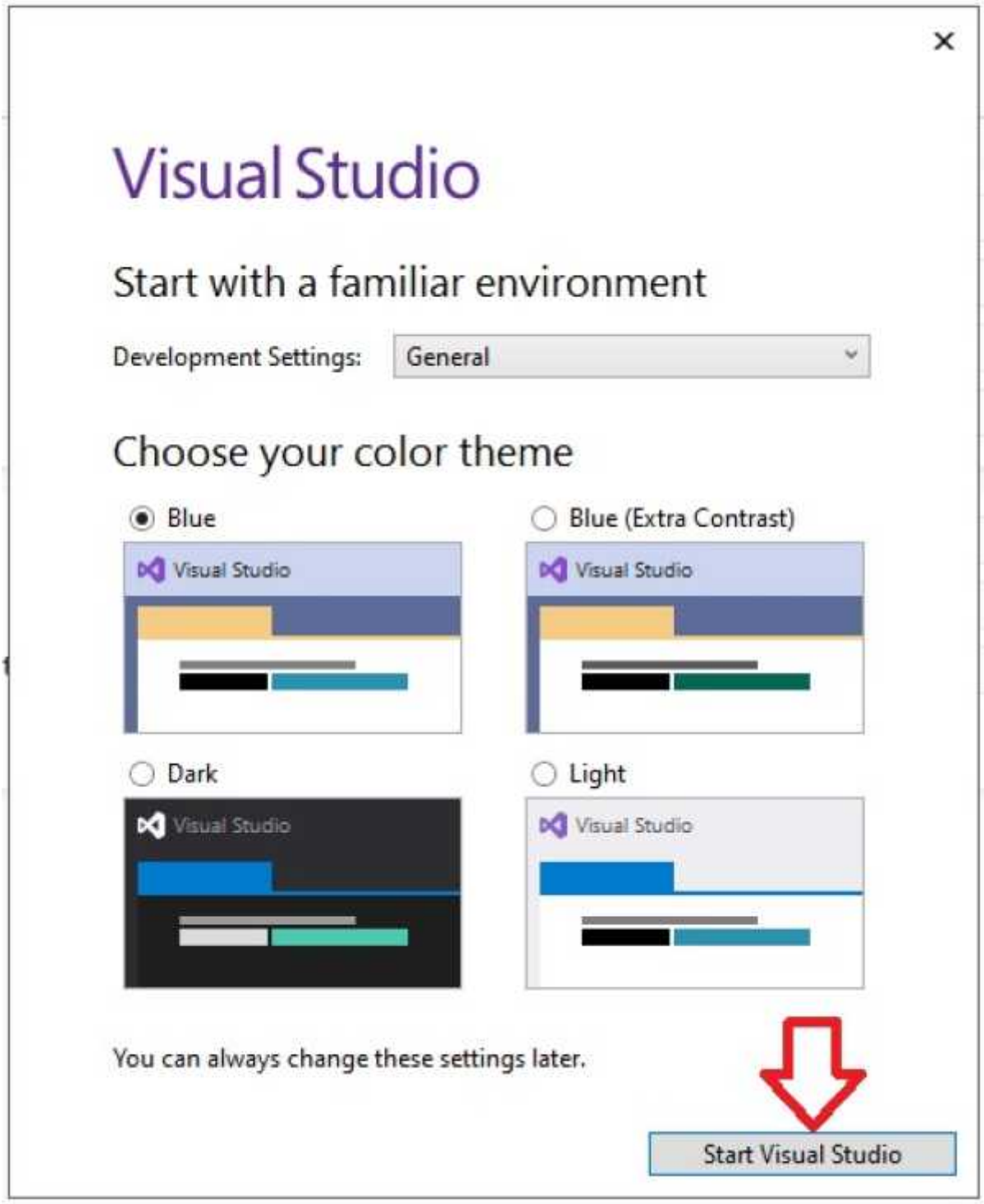
Po uruchomieniu instalatora wybieramy opcje .NET desktop development oraz Desktop development with C++ i rozpoczynamy instalację.



Instalacja przebiegnie automatycznie.



Po zainstalowaniu, przy pierwszym uruchomieniu środowiska, można ustawić jego wygląd.

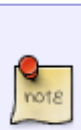


1.2. Utworzenie projektu z asemblerową biblioteką DLL

Po zainstalowaniu możemy utworzyć projekt, w którym główny program będzie napisany w języku wysokiego poziomu (np. C#, C++), a biblioteka dołączana dynamicznie (dll) będzie napisana w asemblerze.



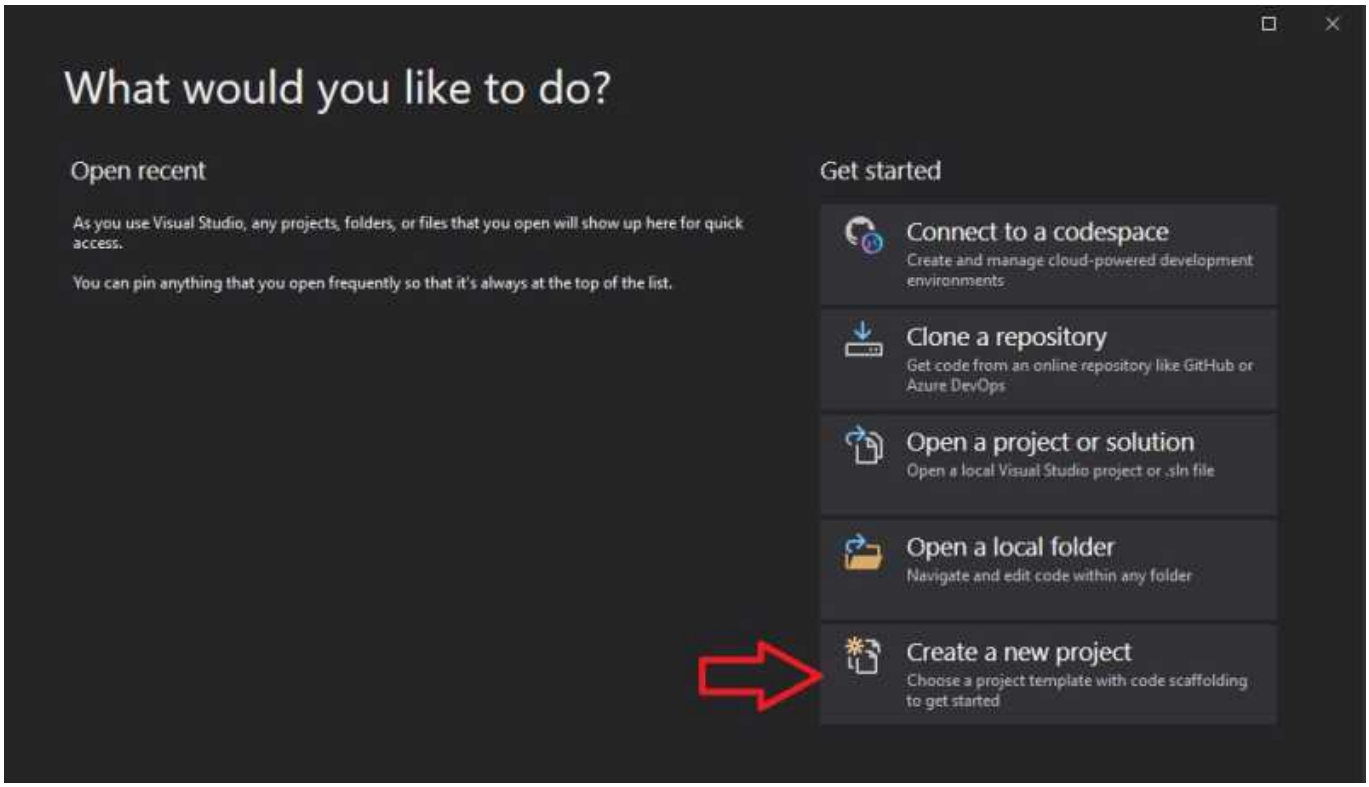
Tworzenie bibliotek w asemblerze, wywoływanych z programów stworzonych w wybranym języku wysokiego poziomu (HLL) jest obecnie najczęstszym wykorzystaniem języka asemblerowego.



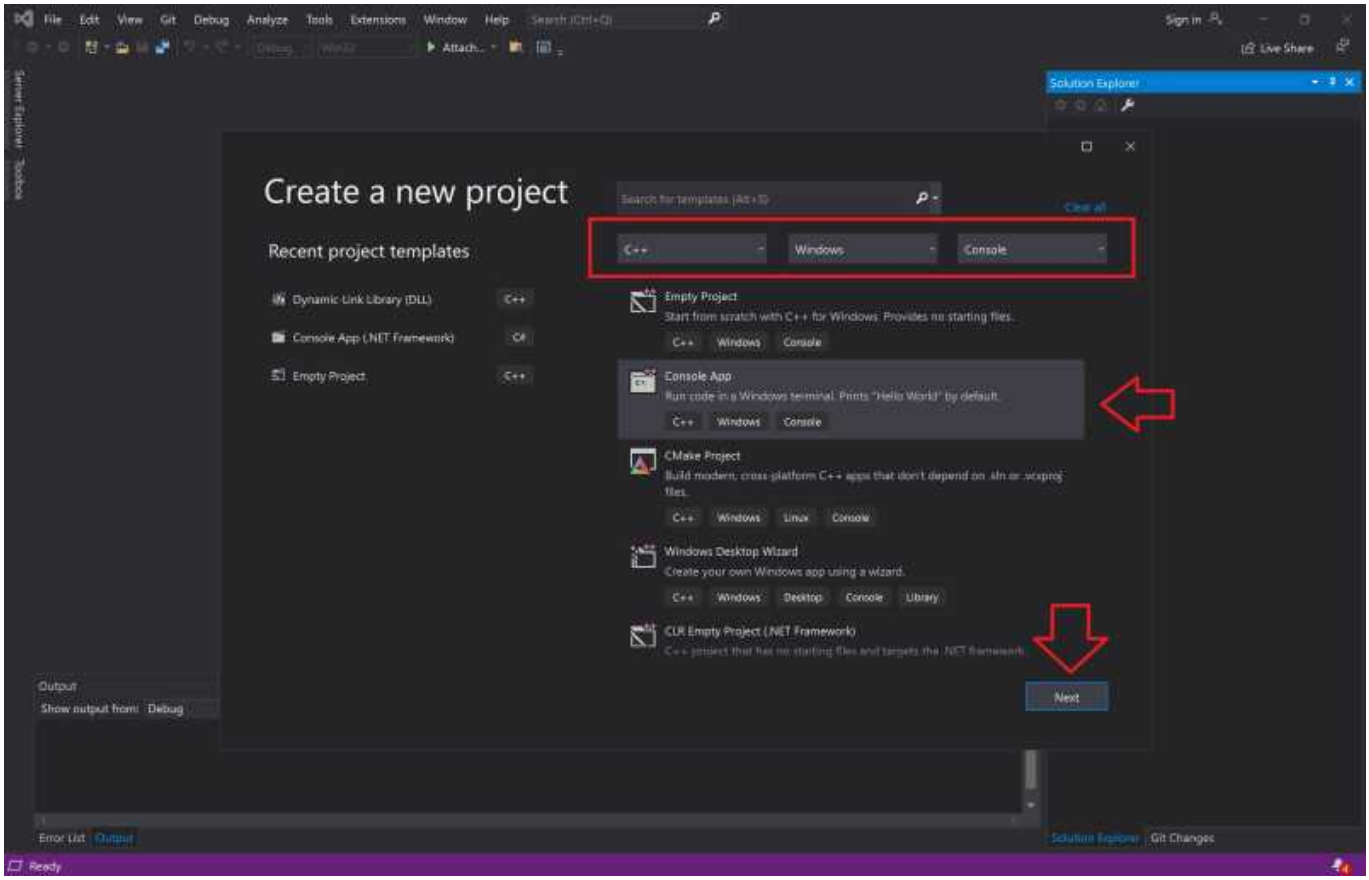
Przedstawiony w dalszej części proces konfiguracji przebiega identycznie dla programu głównego napisanego w C++ jak i C#. Wykorzystane zrzuty ekranu pochodzą z konfiguracji bibliotek DLL dla obu języków programowania, jednak zwykle różnice są drugoplanowe i nie mają wpływu na proces konfiguracji środowiska.

1.2.1. Utworzenie nowego projektu

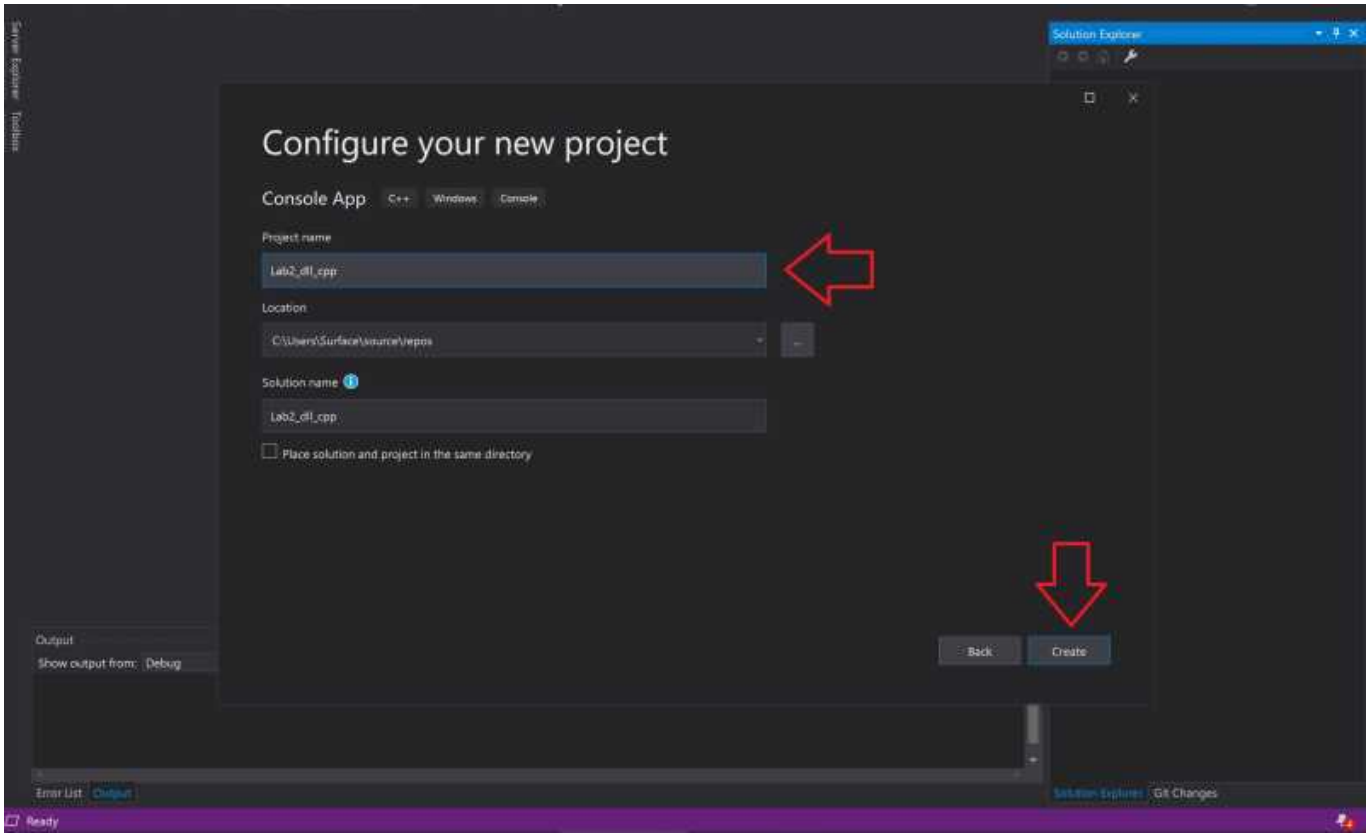
W tym ćwiczeniu zajmiemy się dołączeniem biblioteki dll do programu napisanego w C++. Konfigurację rozpoczynamy od utworzenia nowego projektu w języku C++.



Utworzymy aplikację konsolową, napisaną w języku C++, przeznaczoną do uruchomienia pod kontrolą systemu operacyjnego Windows.



Aby utworzyć projekt należy go nazwać. Dla tego projektu proponujemy nazwę “Lab2_dll_cpp”. Ścieżki pozostawiamy domyślne.

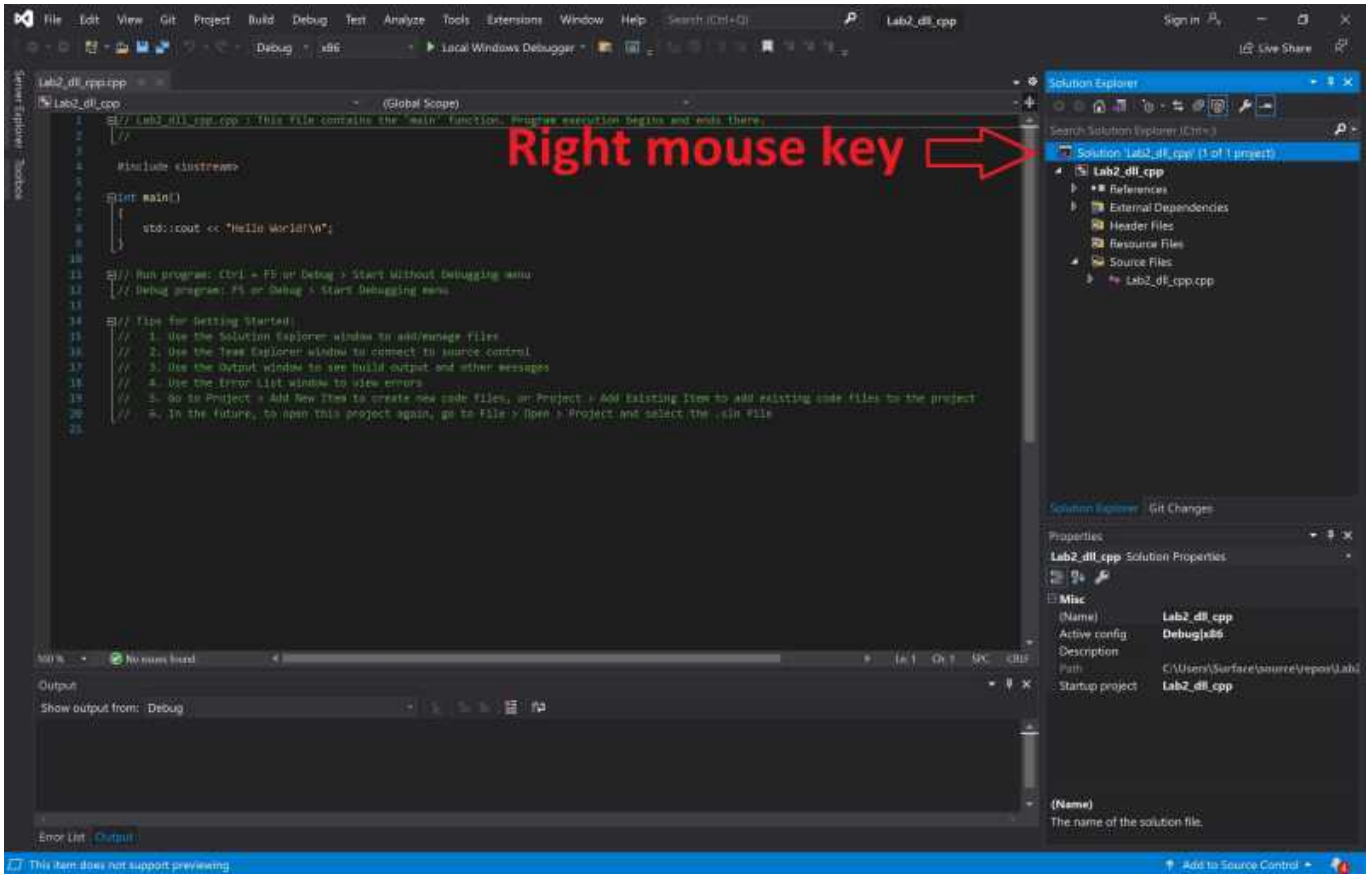


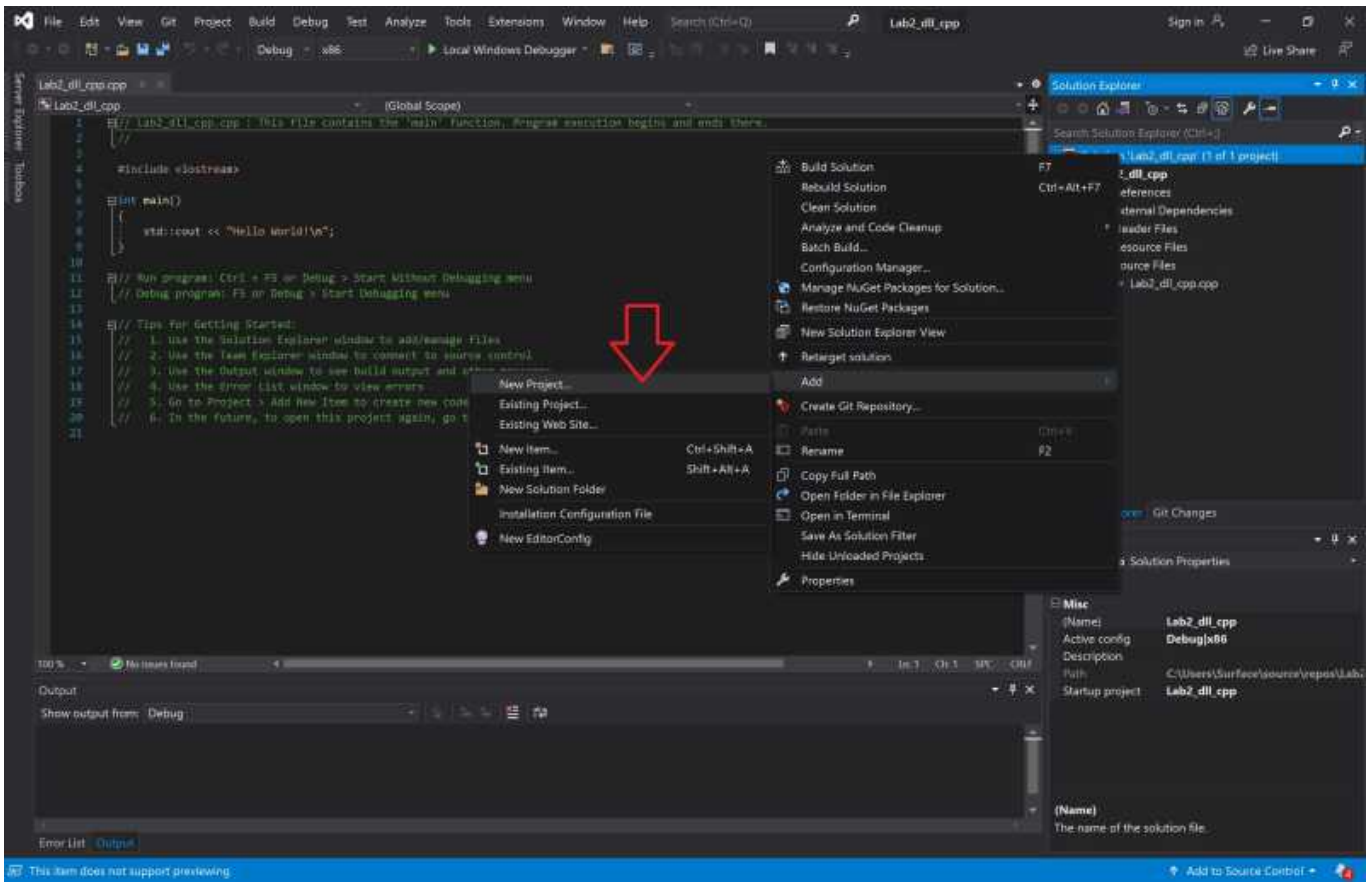


Na tym etapie zostanie utworzone rozwiązanie zawierające projekt z szablonem programu C++. Rozwiązanie jest często zwane solucją (z ang. solution).

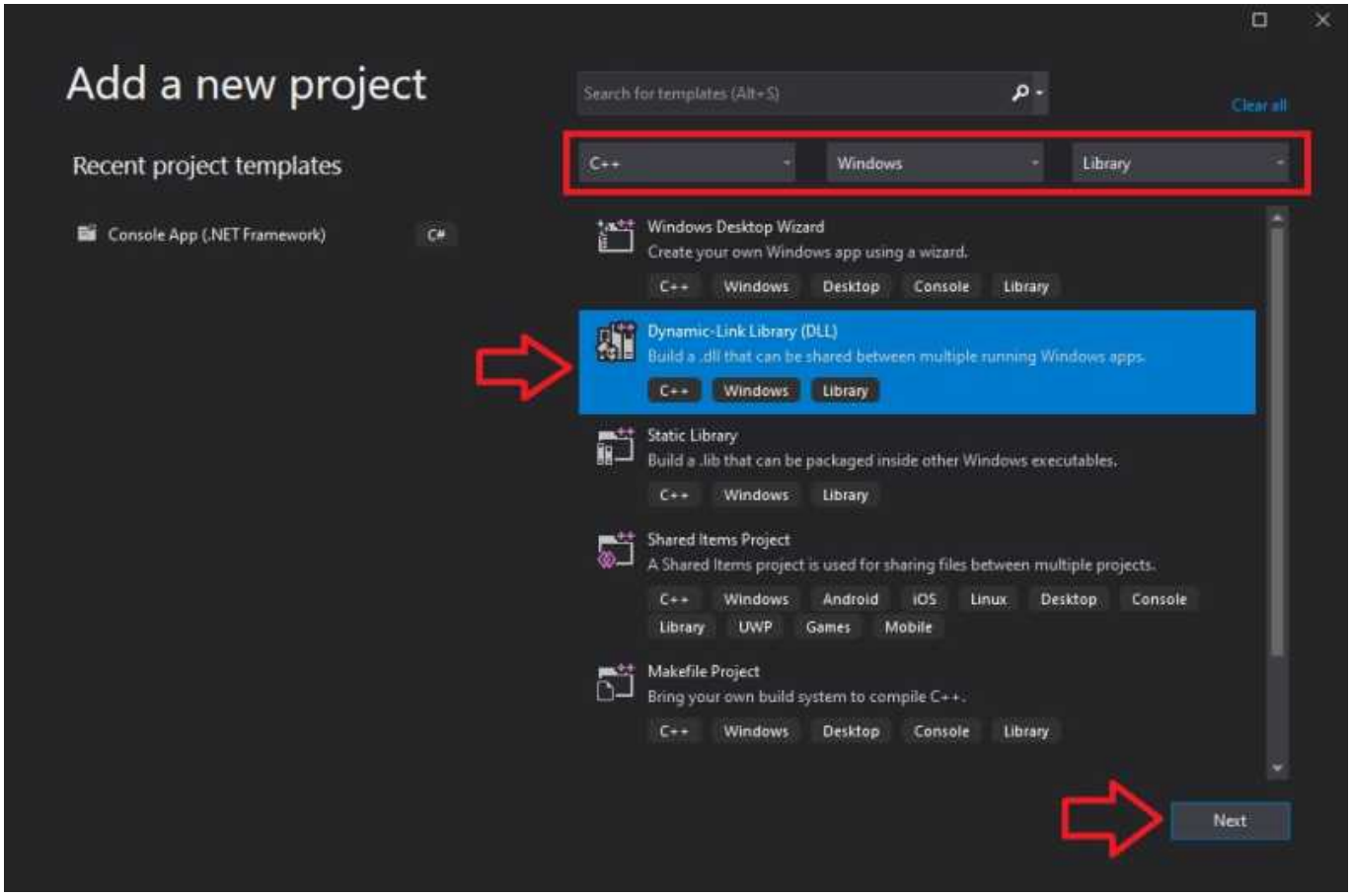
1.2.2. Dodanie do rozwiązania części asemblerowej

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).

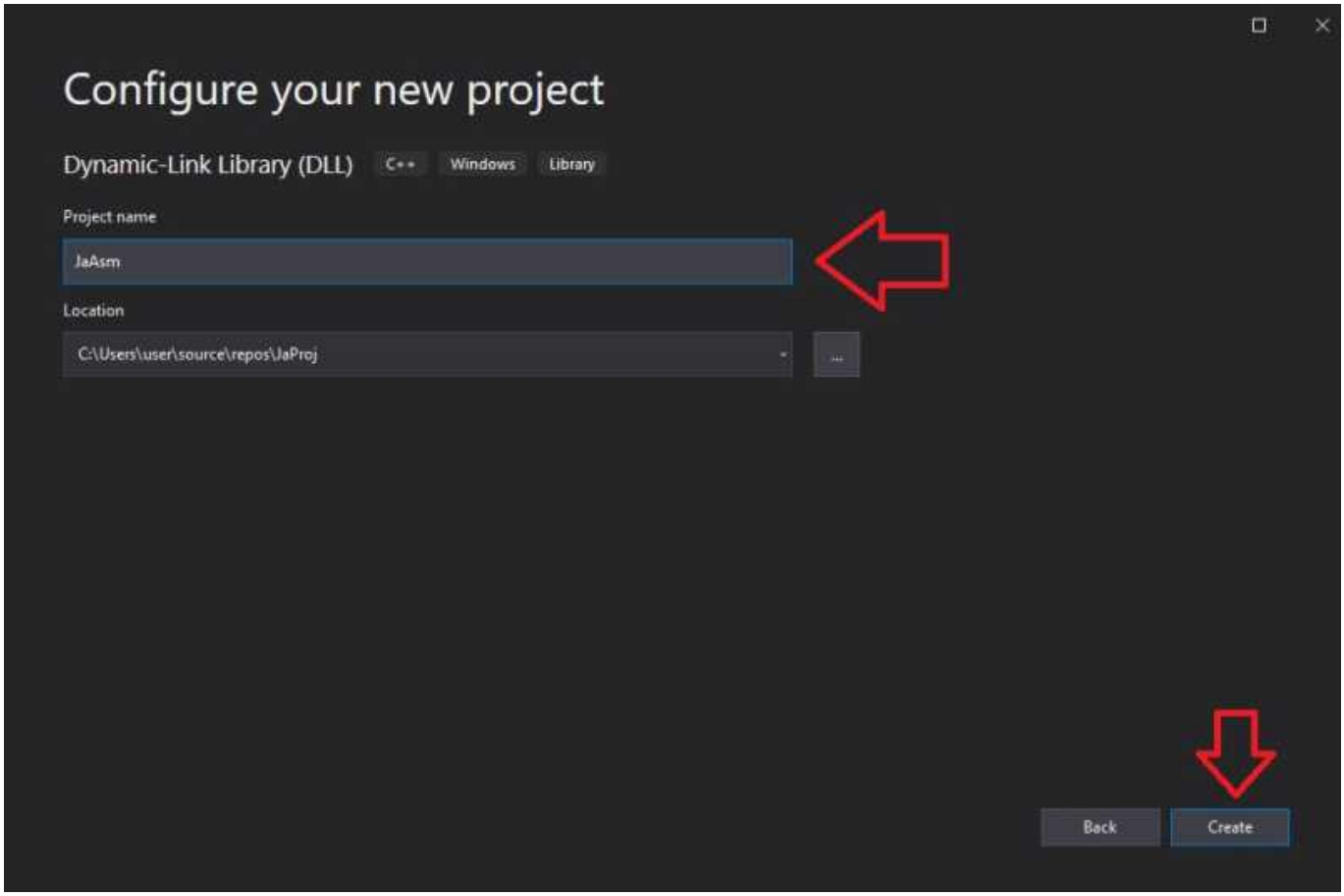




Nasz nowy projekt asemblerowy tworzymy jako projekt biblioteki (library) zgodny z C++, pracujący pod kontrolą systemu operacyjnego Windows. Z listy dostępnych opcji wybieramy bibliotekę dołączaną dynamicznie Dynamic-Link Library (DLL).



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

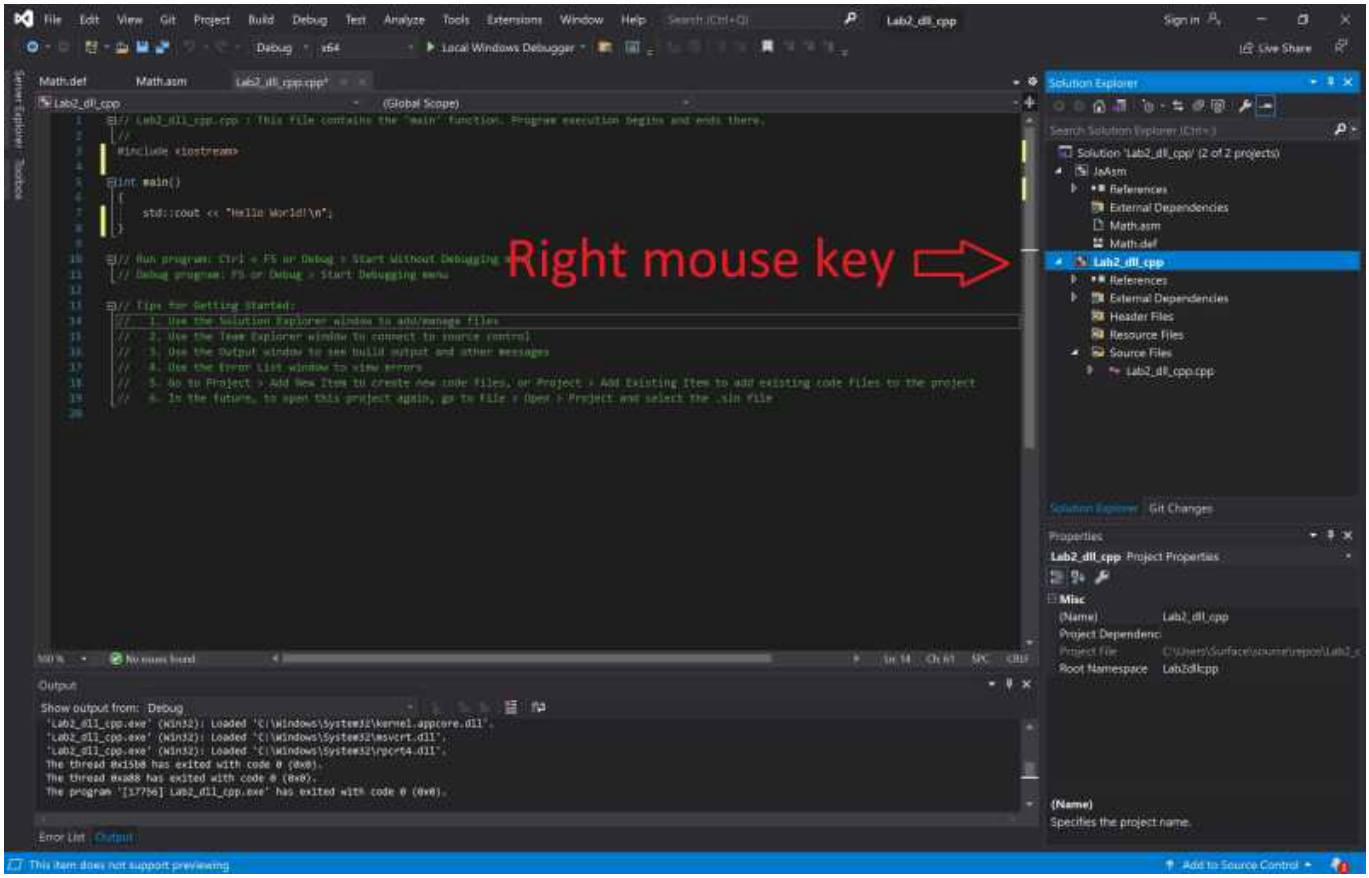


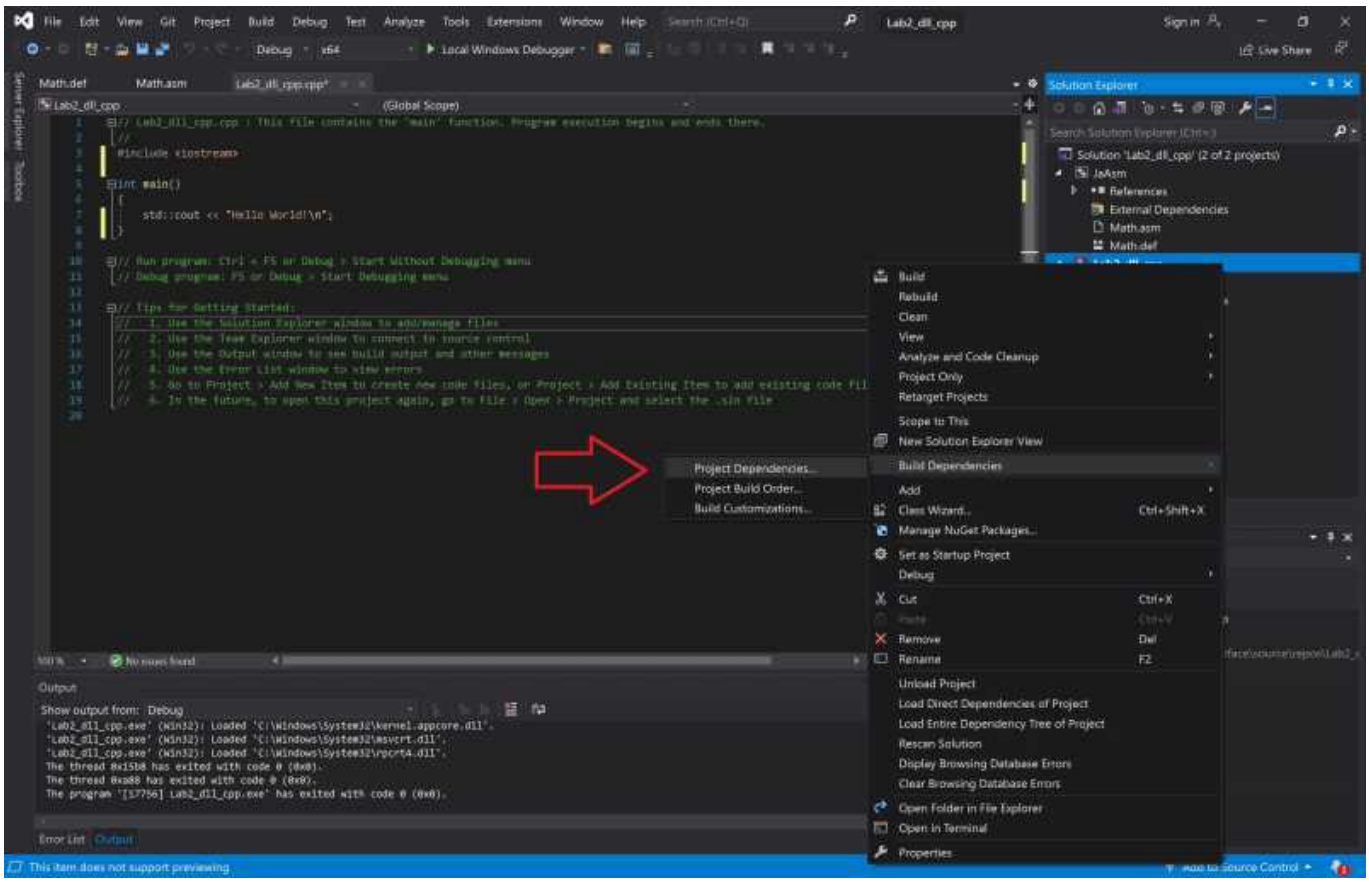


Obie części naszego rozwiązania będą umieszczone jako podfoldery w jednym folderze o nazwie zgodniej z nazwą rozwiązania, w naszym przypadku “Lab2_dll_cpp”. Każdy z tych podfolderów przyjmie nazwę odpowiednich projektów, czyli “Lab2_dll_cpp” dla części napisanej w C++ i “JaAsm” dla części napisanej w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

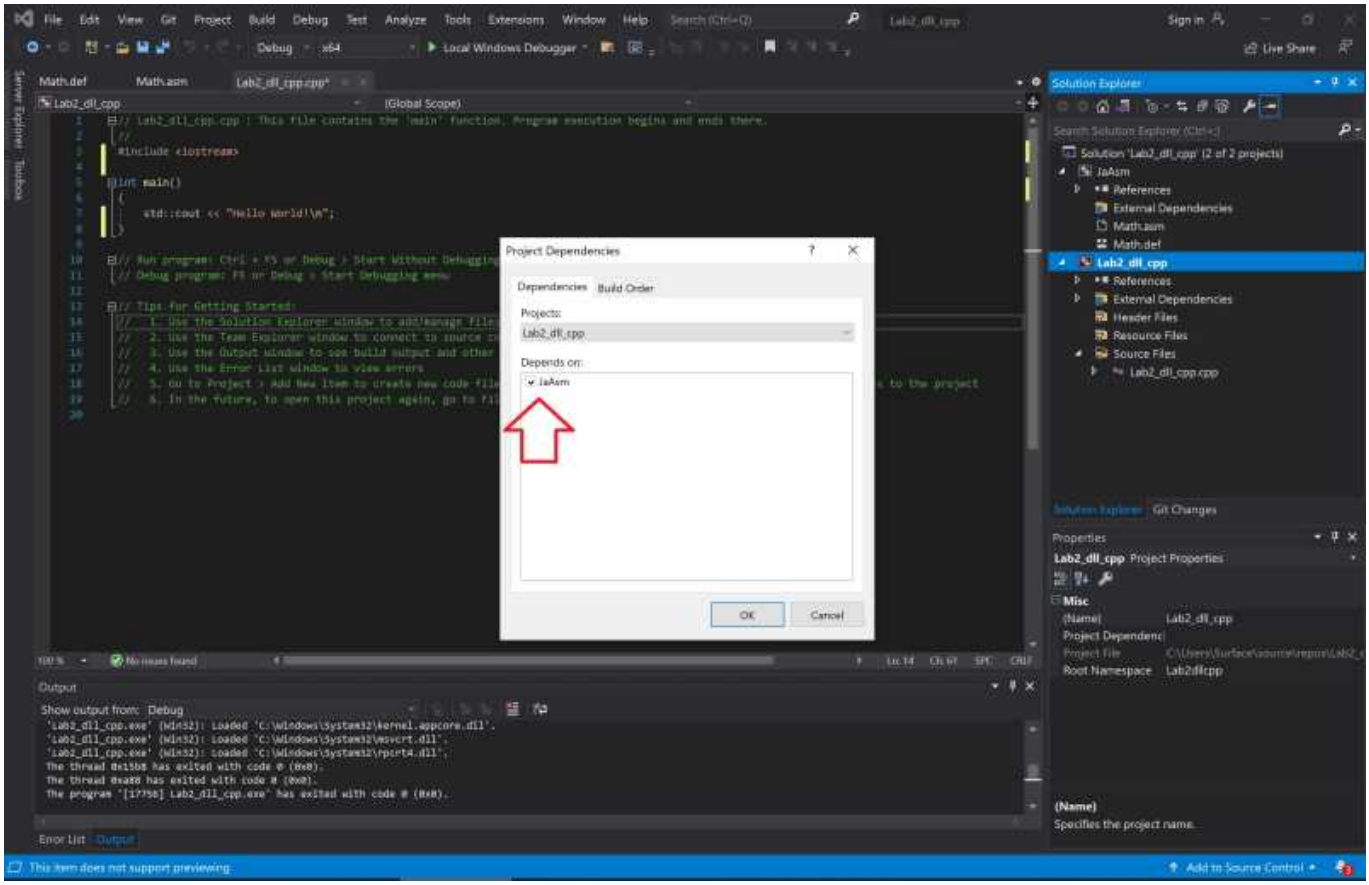
1.2.3. Ustawienie zależności w rozwiązaniu

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”

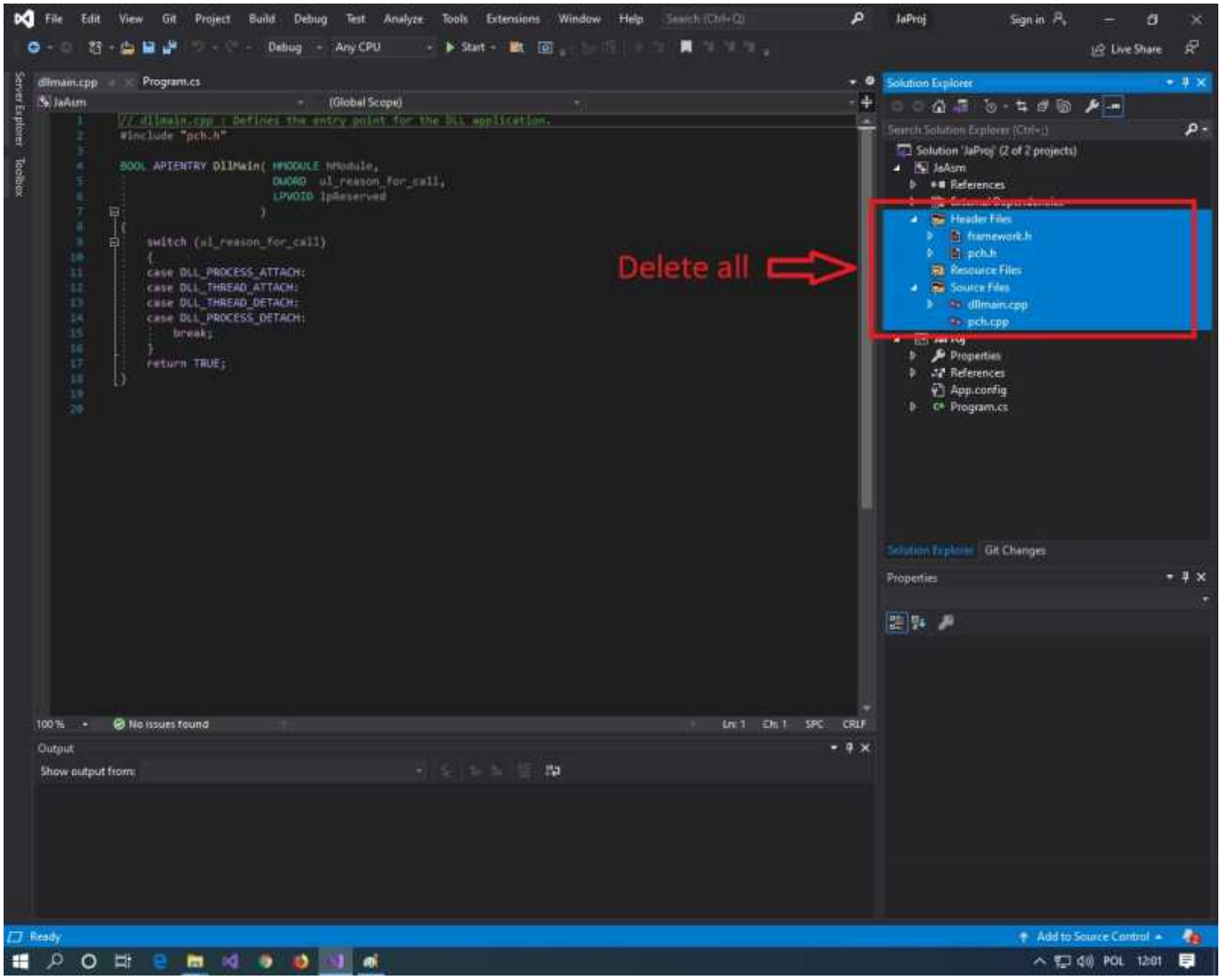




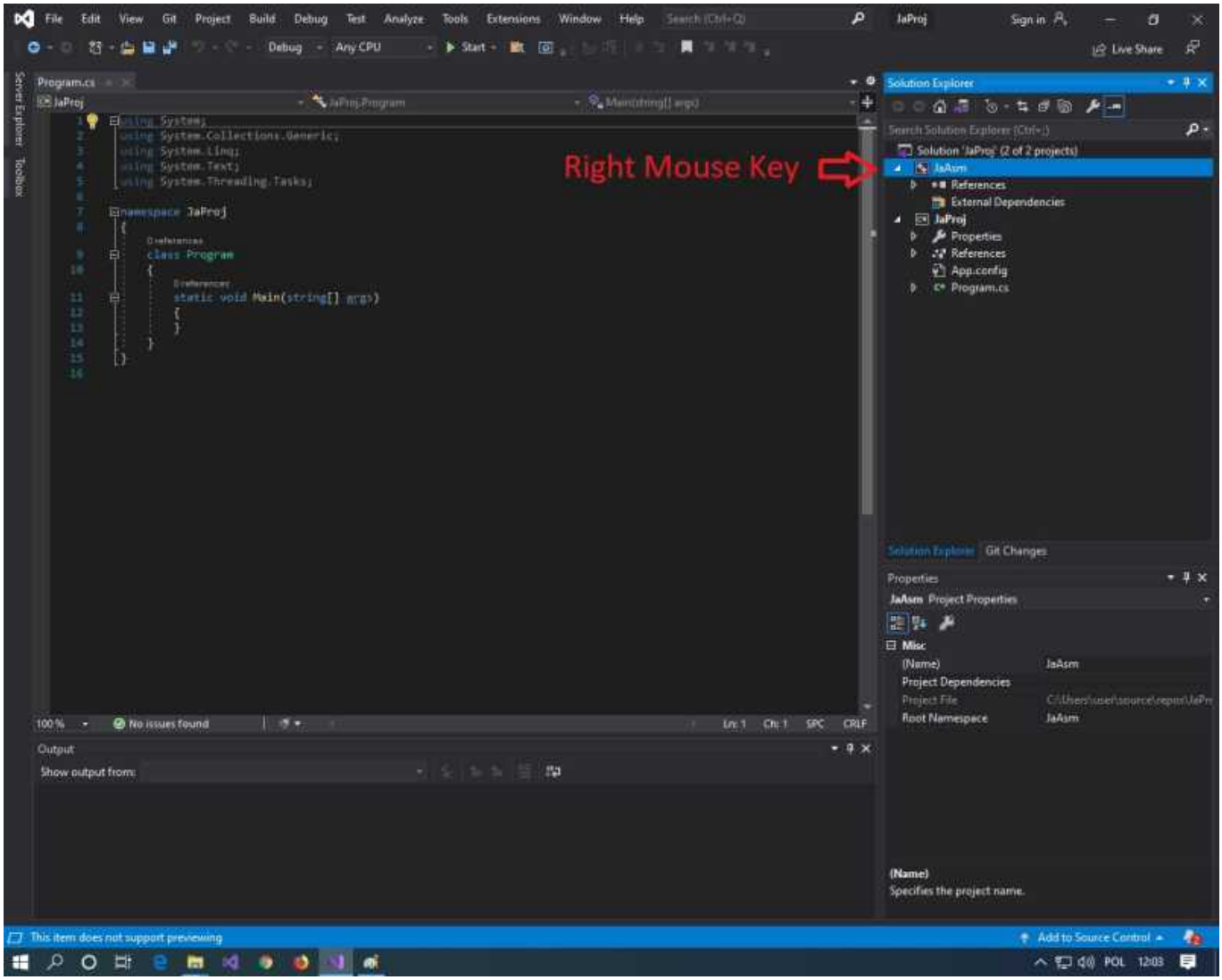
Po ustawieniu tej zależności, przy kompialcji projektu lub programu napisanego w C++ automatycznie zostanie także przekompilowany plik asemblerowy.

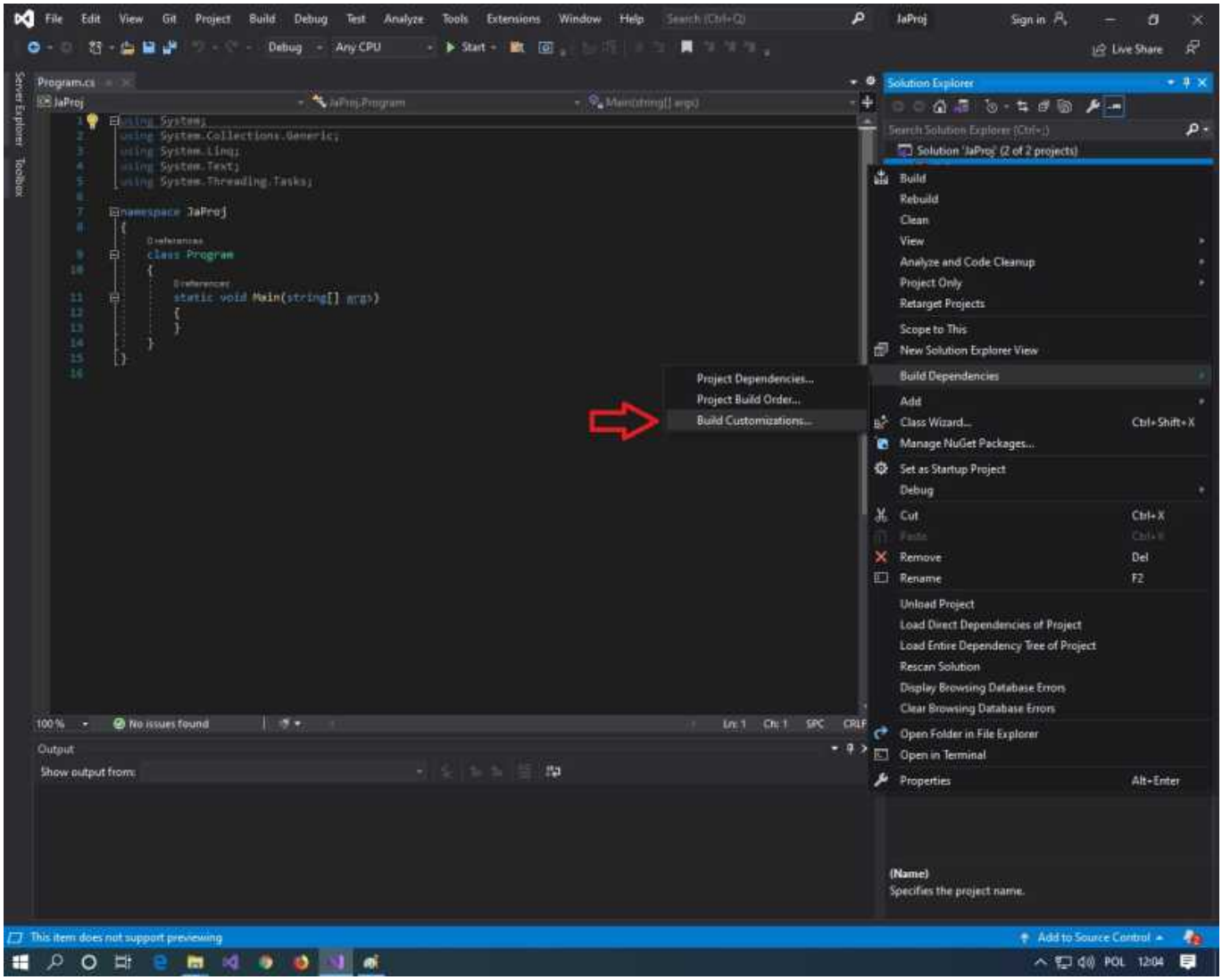
1.2.4. Zamiana biblioteki z C++ na asemblerową

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.

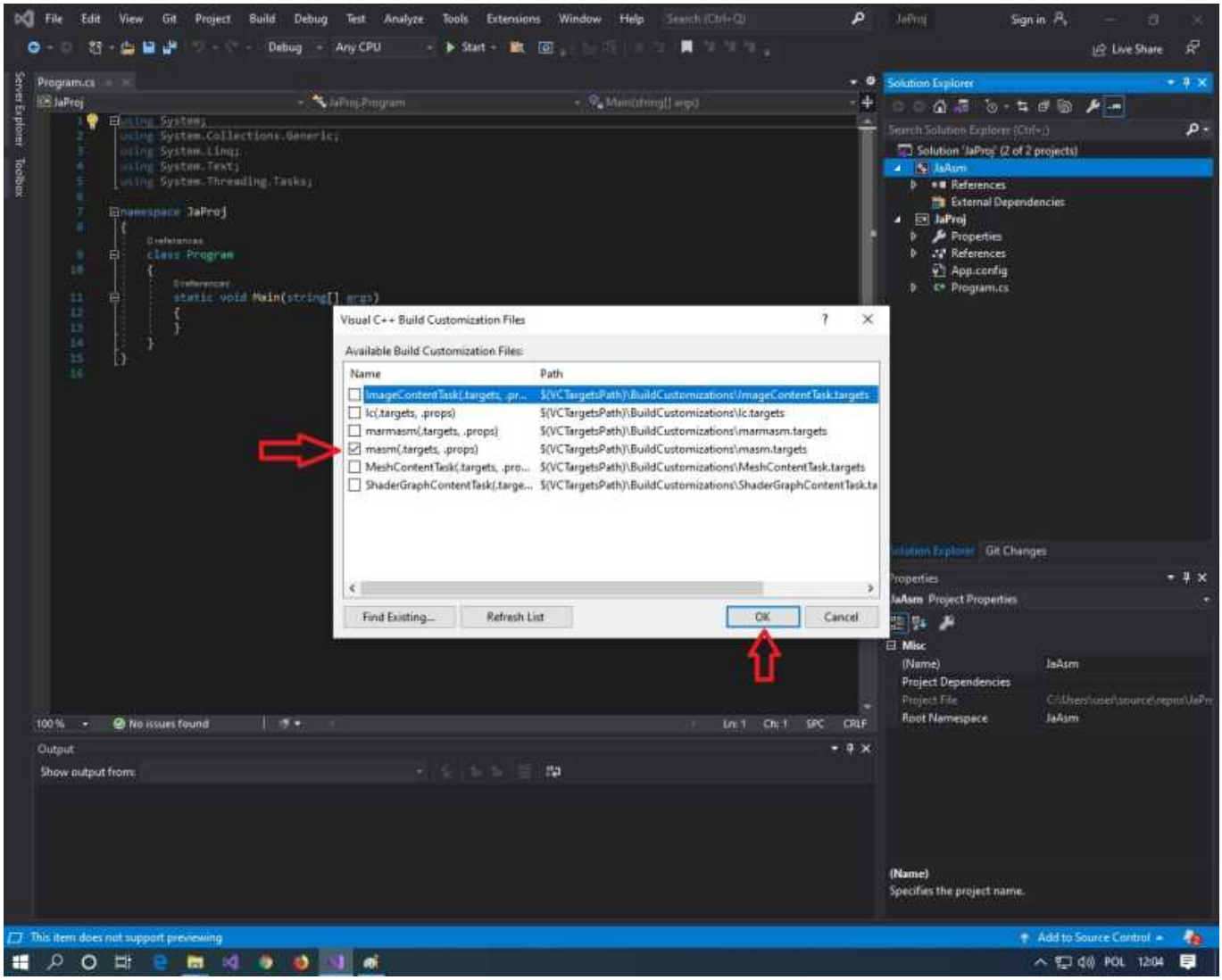


Dla biblioteki konieczne jest dostosowanie kompilacji jako plik asemblerowy MASM. W tym celu klikamy projekt biblioteki prawym klawiszem myszy i z listy wybieramy „Zależności kompilacji” (Build dependencies), a następnie „Dostosowanie kompilacji” (Build customization).

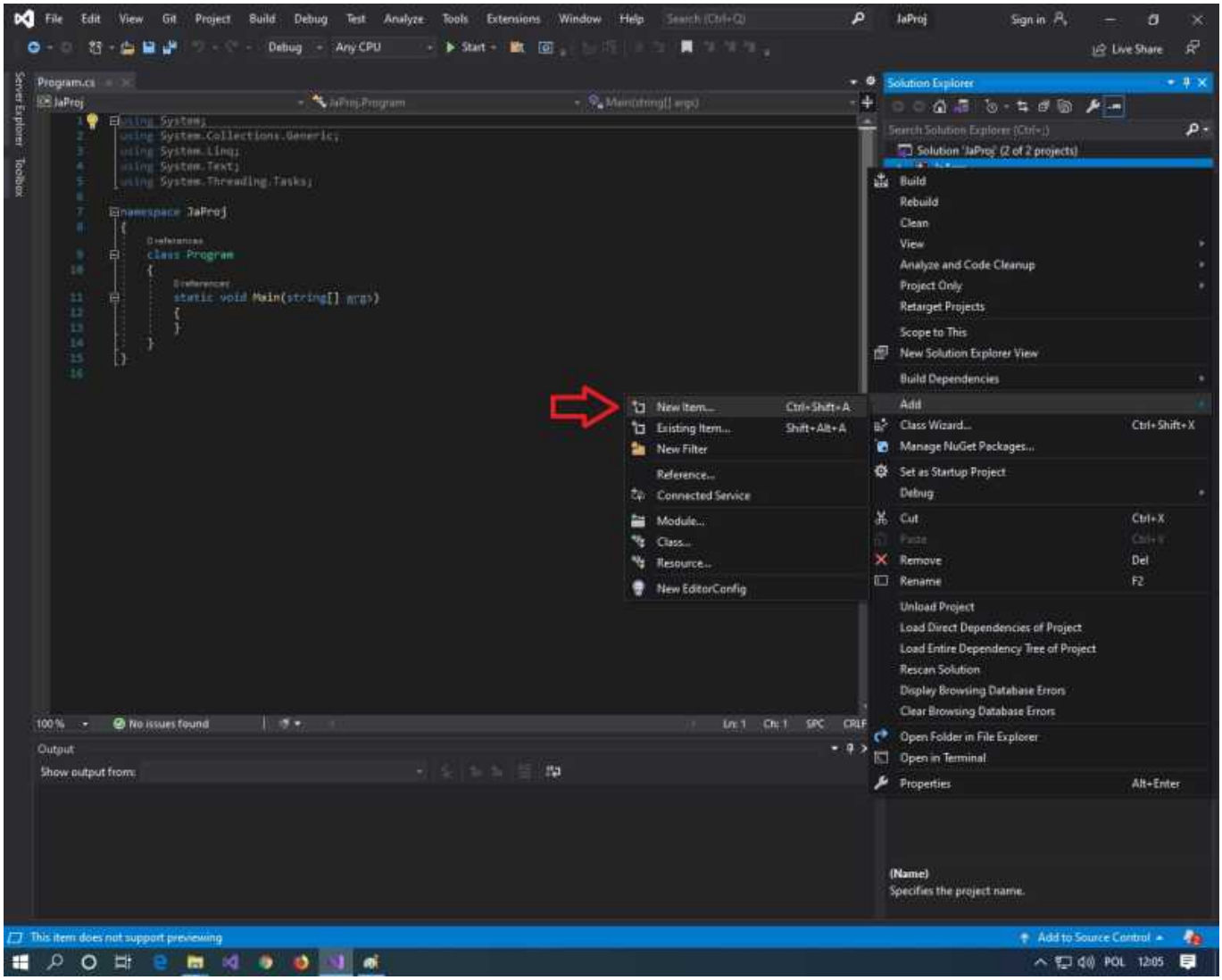

Right Mouse Key



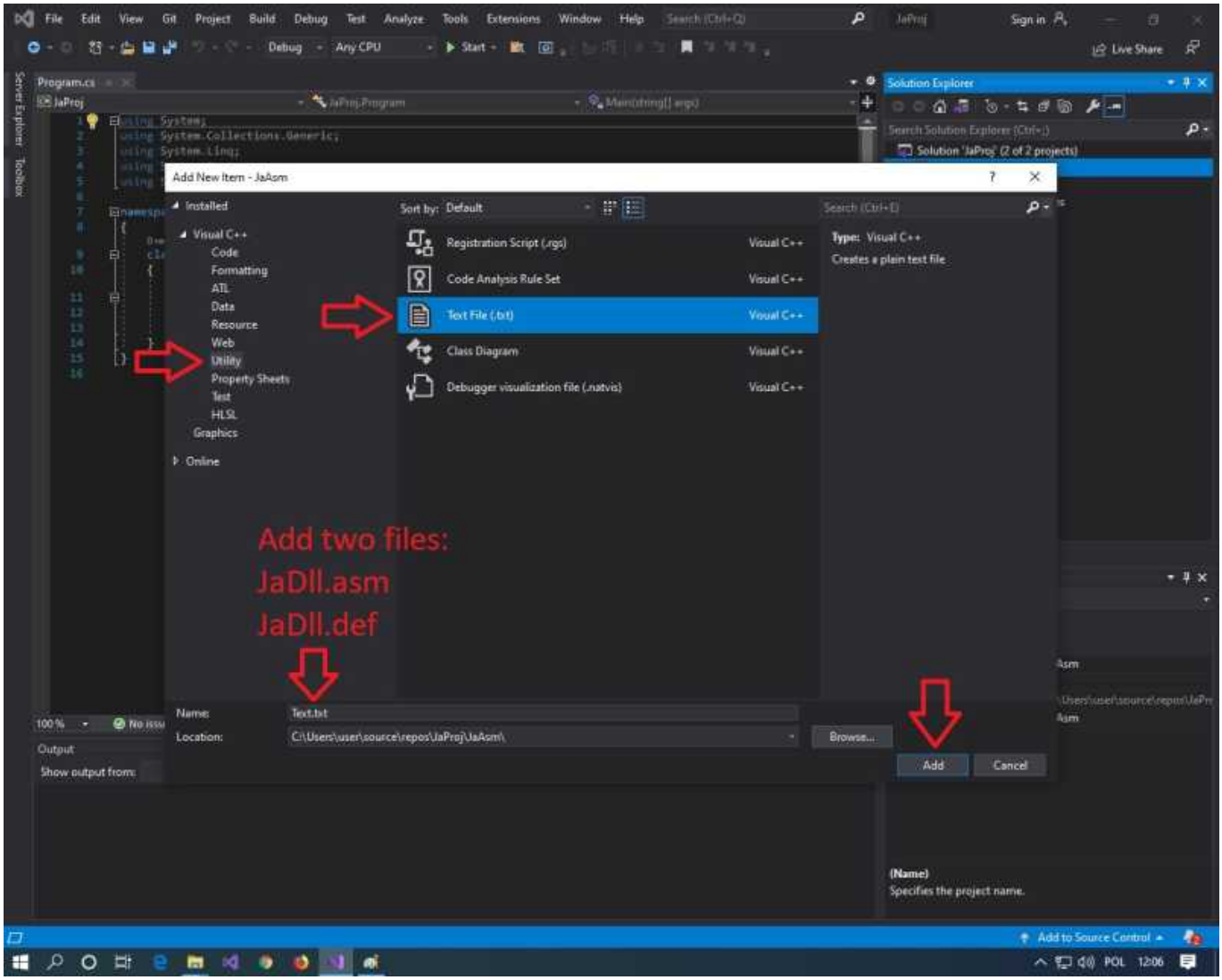
W nowo otwartym oknie zaznaczamy „masm” i potwierdzamy nasz wybór „OK”.



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).



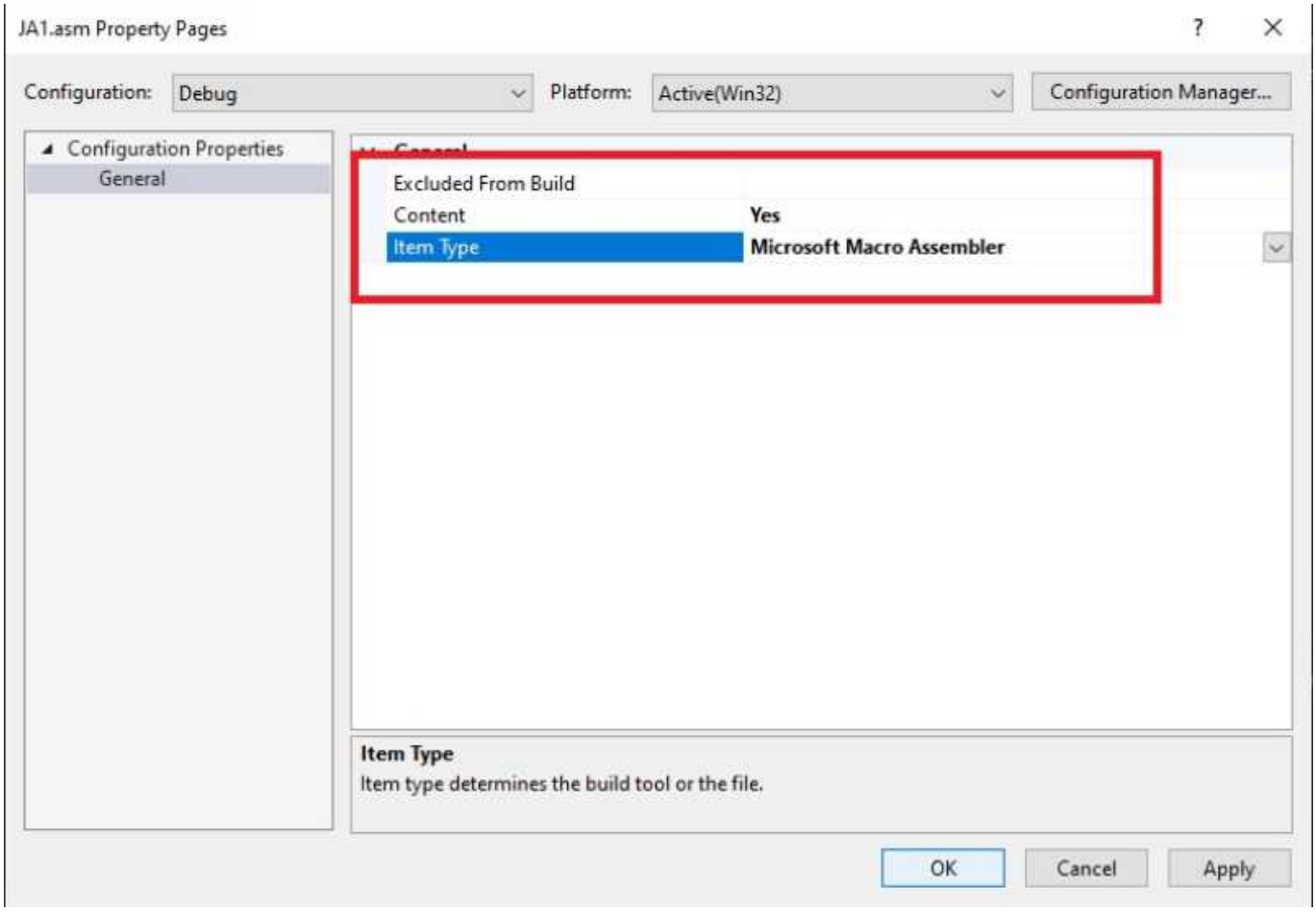
Czynność tę powtarzamy dwukrotnie wybierając za każdym razem “Narzędzie” (Utility) i typ pliku tekstowy, podając nazwy: “JaDll.asm” oraz “JaDll.def”.





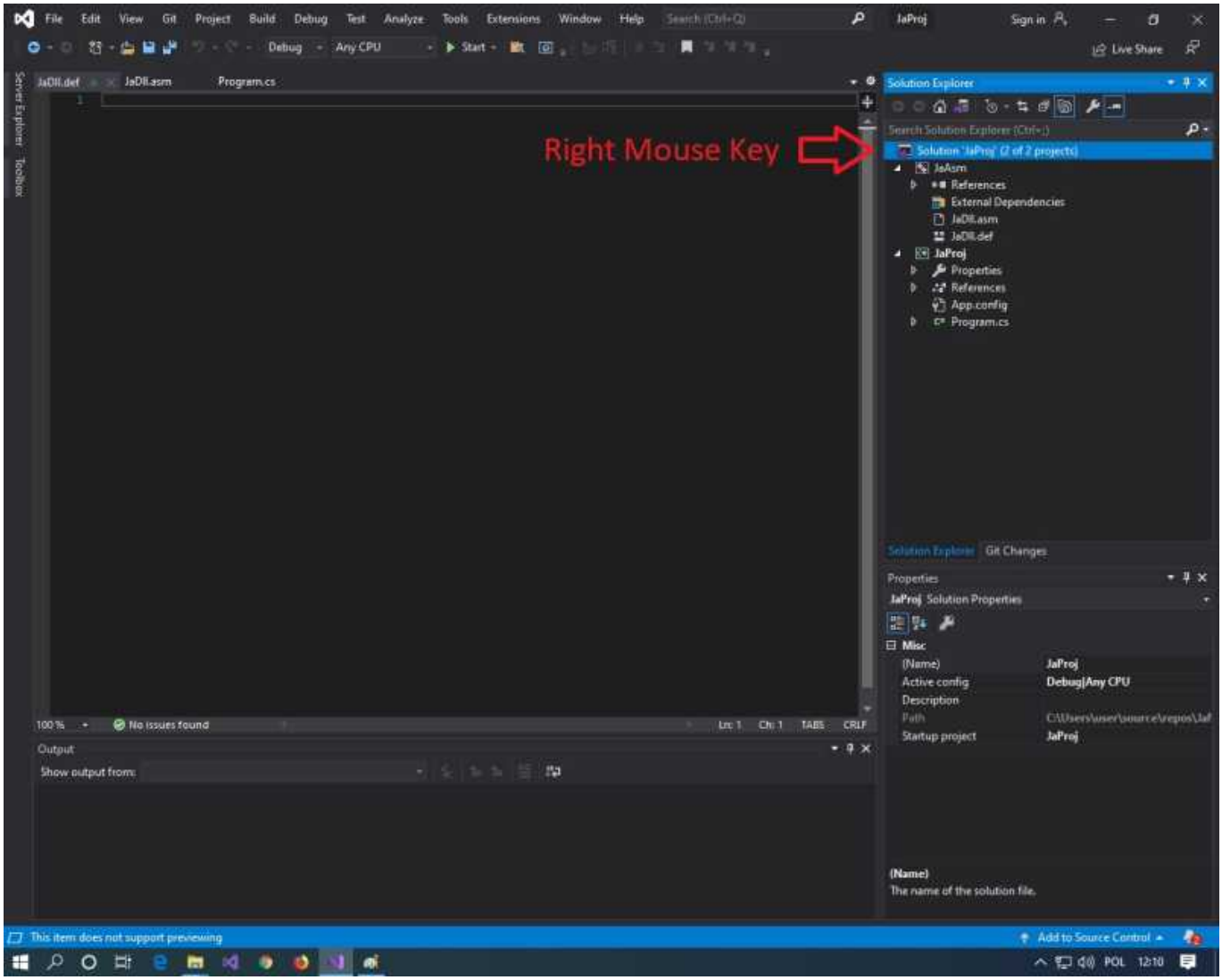
Przy wykonaniu czynności zgodnie z podaną uprzednio kolejnością, z dużym prawdopodobieństwem krok przedstawiony na kolejnym rysunku nie będzie konieczny.

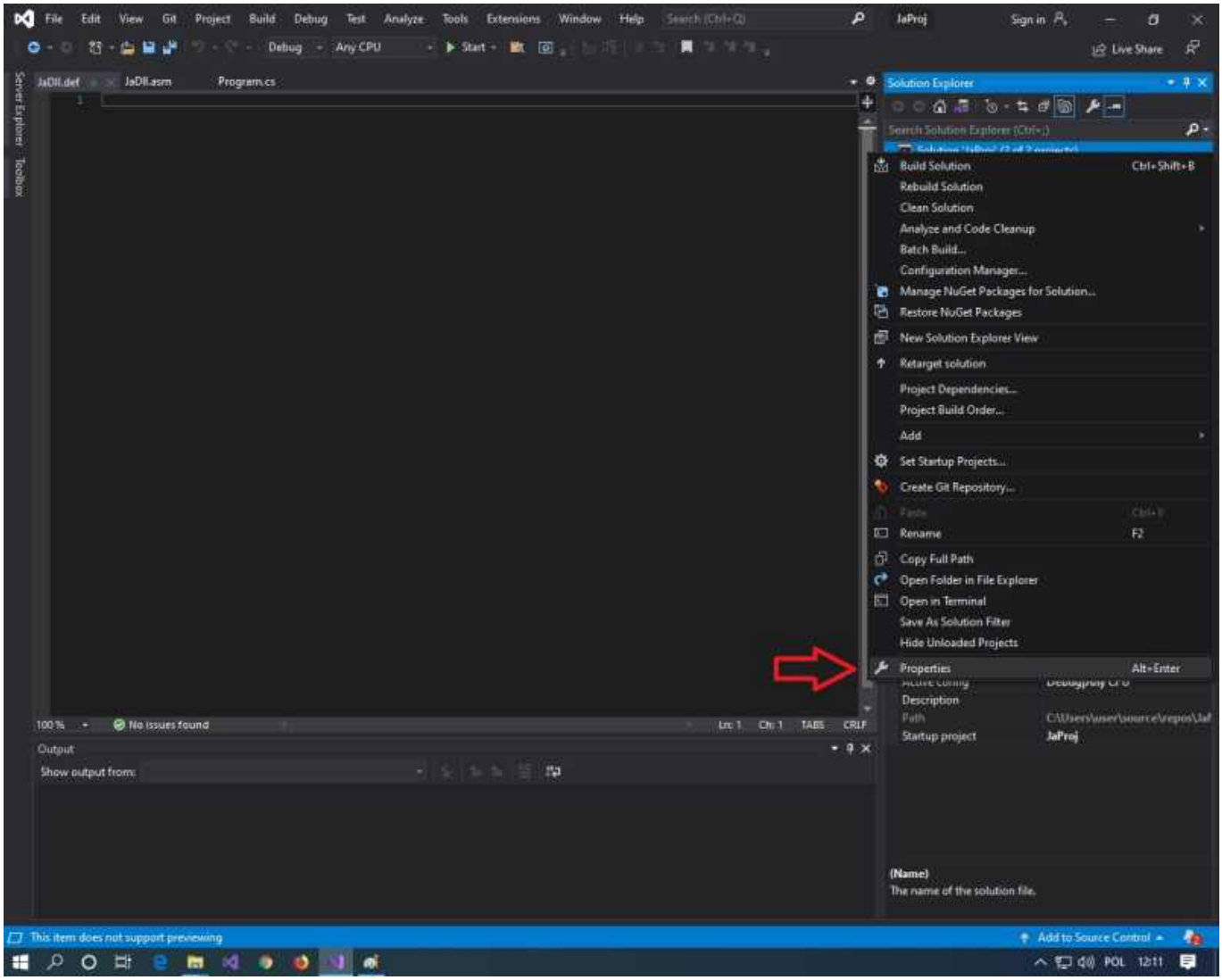
Aby kompilator wiedział, że stworzony plik asemblerowy jest skojarzony z MASM, należy wskazać jego typ. Krok ten jest niezbędny gdy wykorzystujemy plik źródłowy C++ i zmieniamy jedynie jego rozszerzenie na *.asm. Klikając pracym klawiszem na pliku *.asm wybieramy properties i ustawiamy typ pliku na Microsoft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



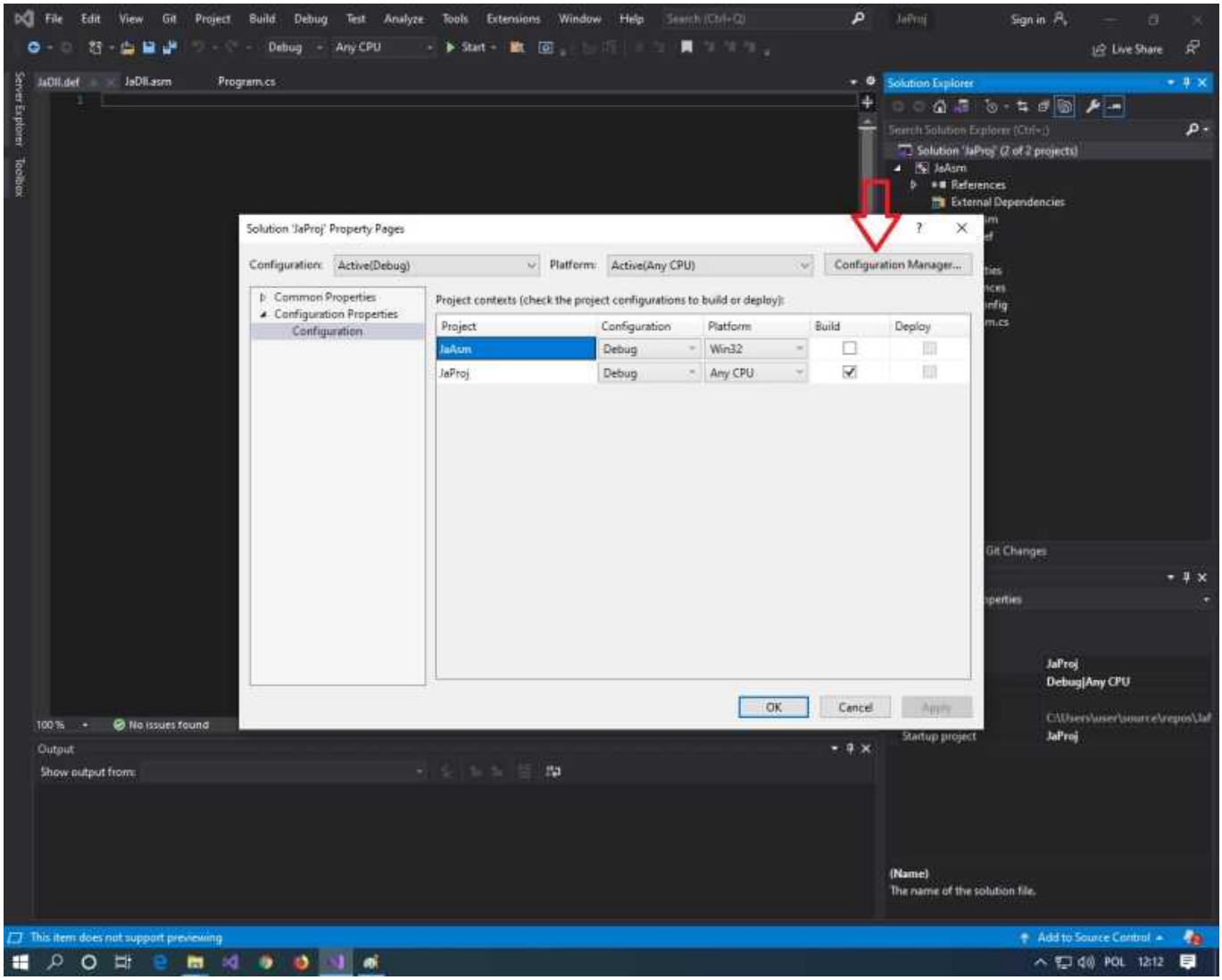
1.2.5. Ustawienie trybu kompilacji

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).

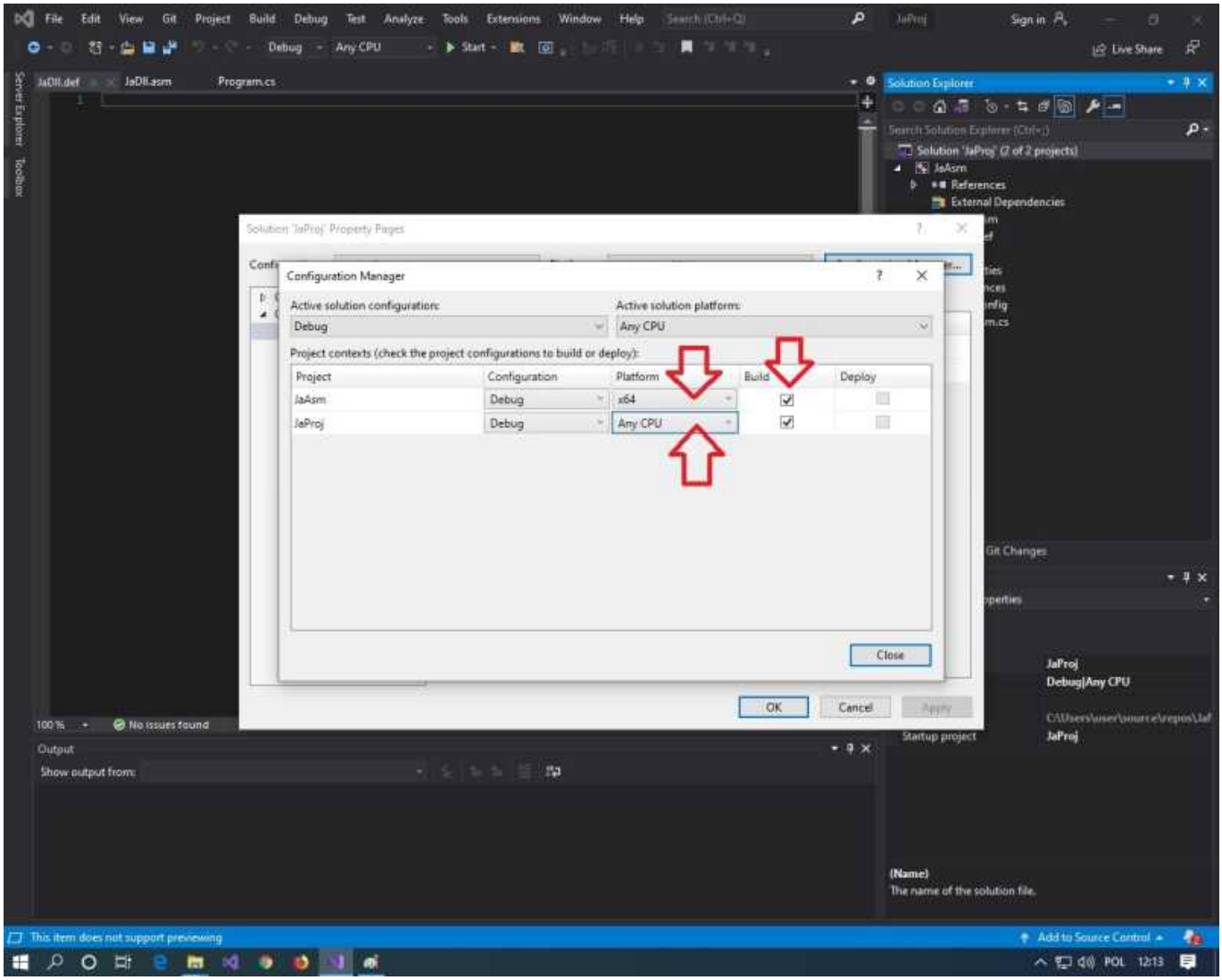

Right Mouse Key

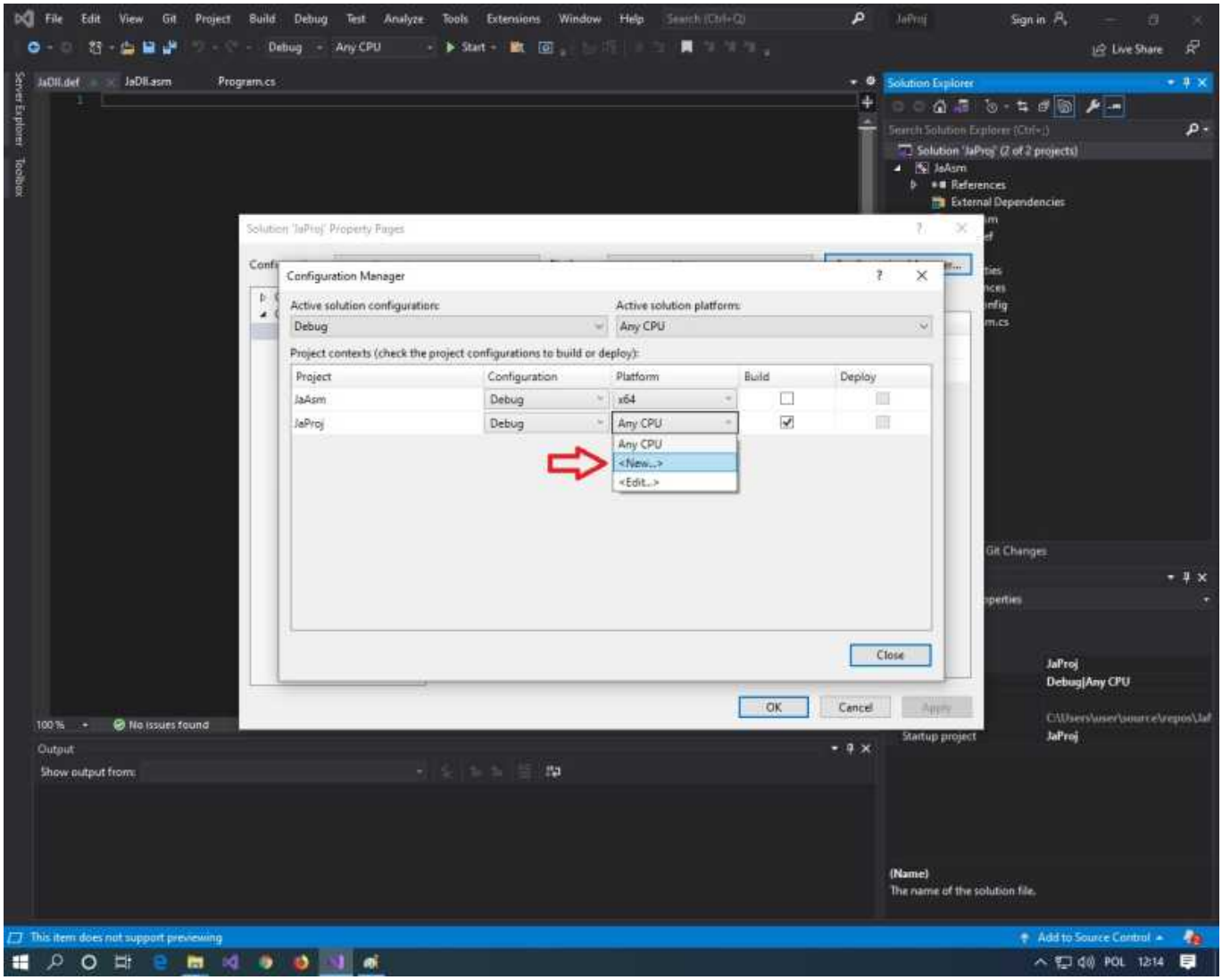


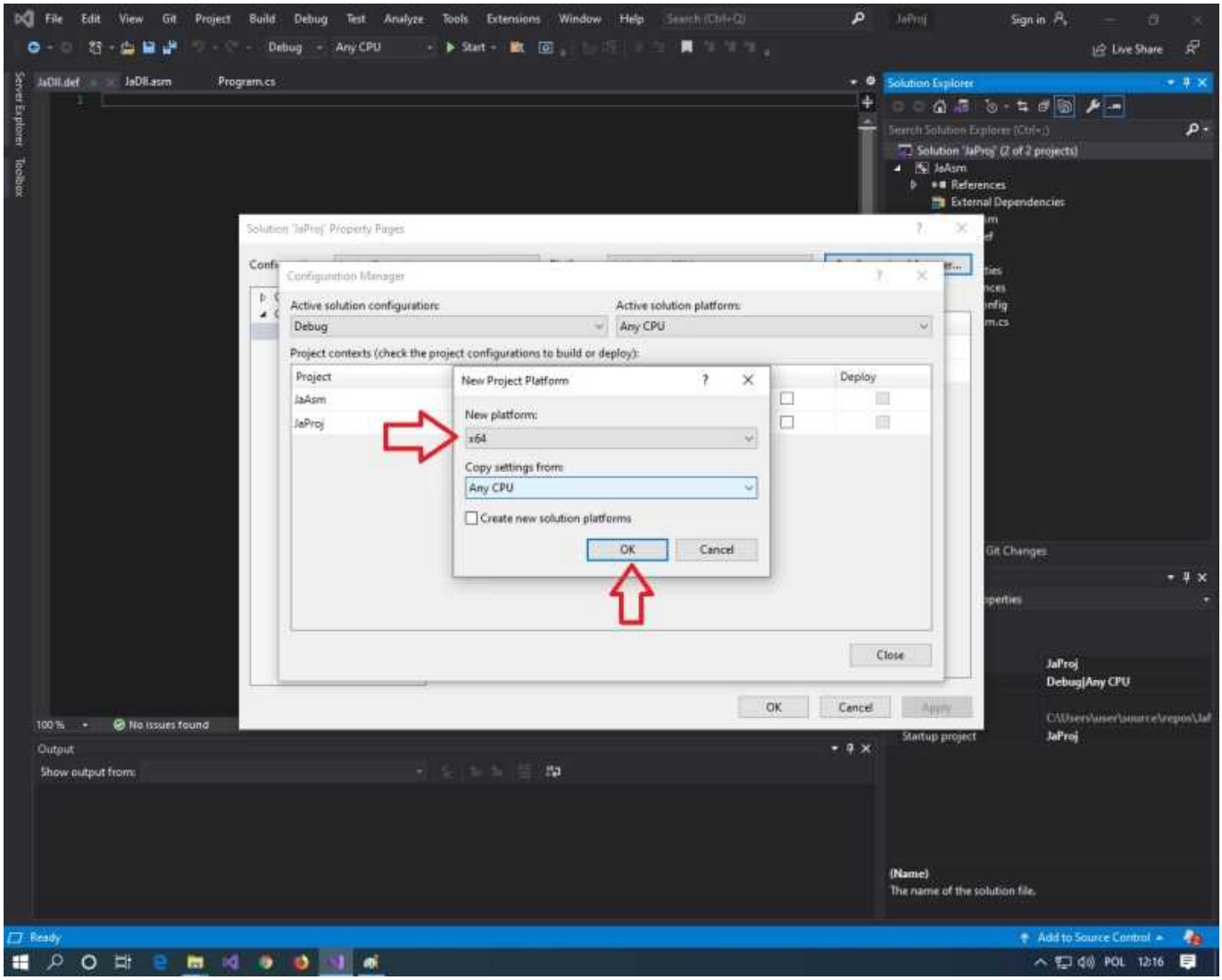
W nowo otwartym oknie przechodzimy na zakładkę „Właściwości konfiguracji” → „Konfiguracja” (Configuration properties → Configuration) na następnie klikamy w „Menedżer konfiguracji” (Configuration manager).



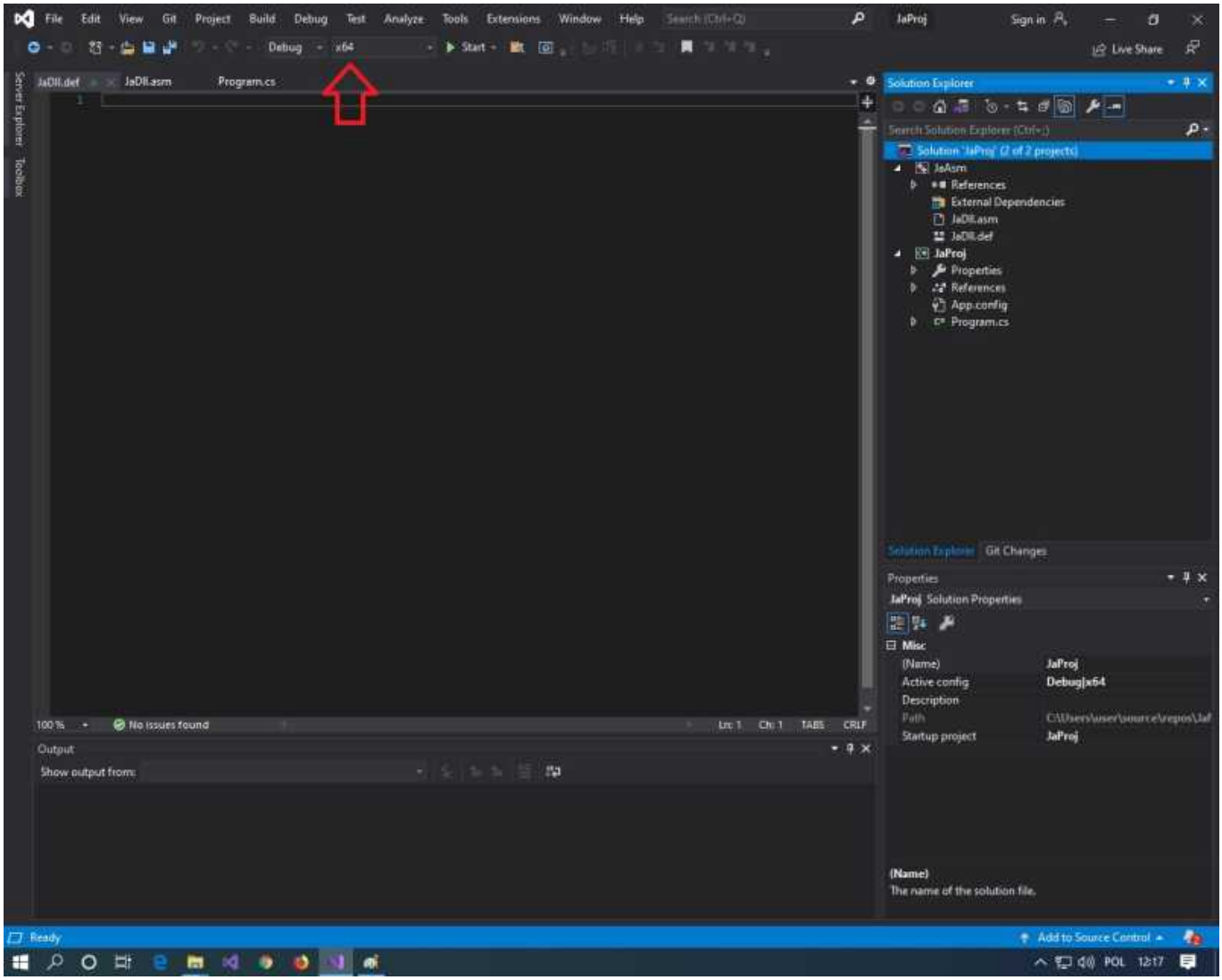
W nowym oknie wskazujemy wszędzie platformę x64 (gdyby dana opcja była niedostępna z dostępnej listy wybieramy pozycję „Nowy” (New), a następnie tworzymy platformę x64 poprzez kopiowanie ustawień z Any CPU). Należy także przy obu częściach solucji włączyć kompilację (Build).


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
Solution Explorer
Solution 'JaProj' (2 of 2 projects)
JaAsm
References
External Dependencies
Git Changes
JaProj
Debug|Any CPU
JaProj
(Name)
The name of the solution file.
Output
Show output from:
No issues found
This item does not support previewing
Add to Source Control

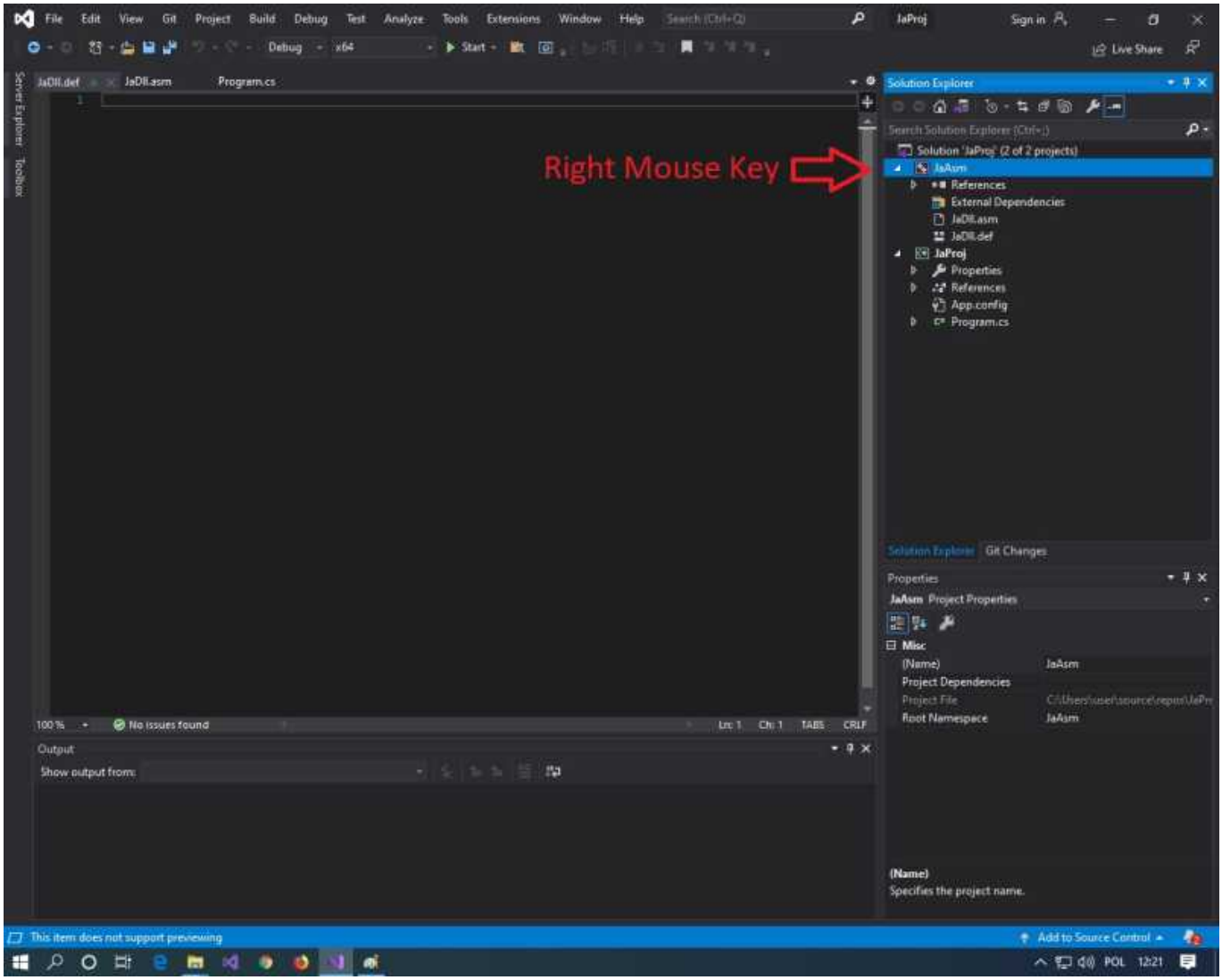


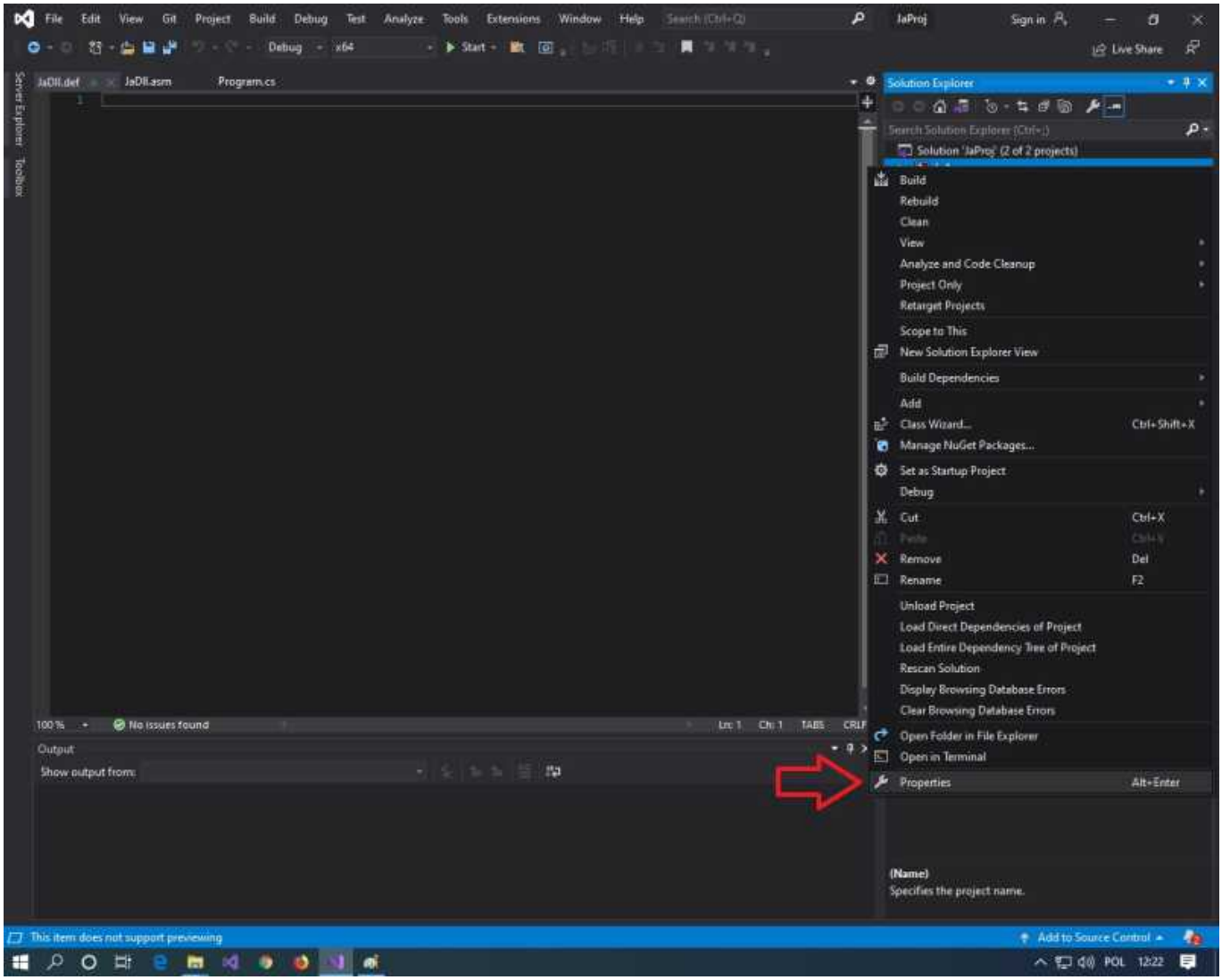
Po wszystkim upewniamy się raz jeszcze, że wszędzie wskazana platforma to x64 i zamykamy okno. W głównym pasku menu Visual Studio powinna automatycznie ustawić się opcja “x64”, niemniej warto to sprawdzić i w razie potrzeby zmodyfikować.



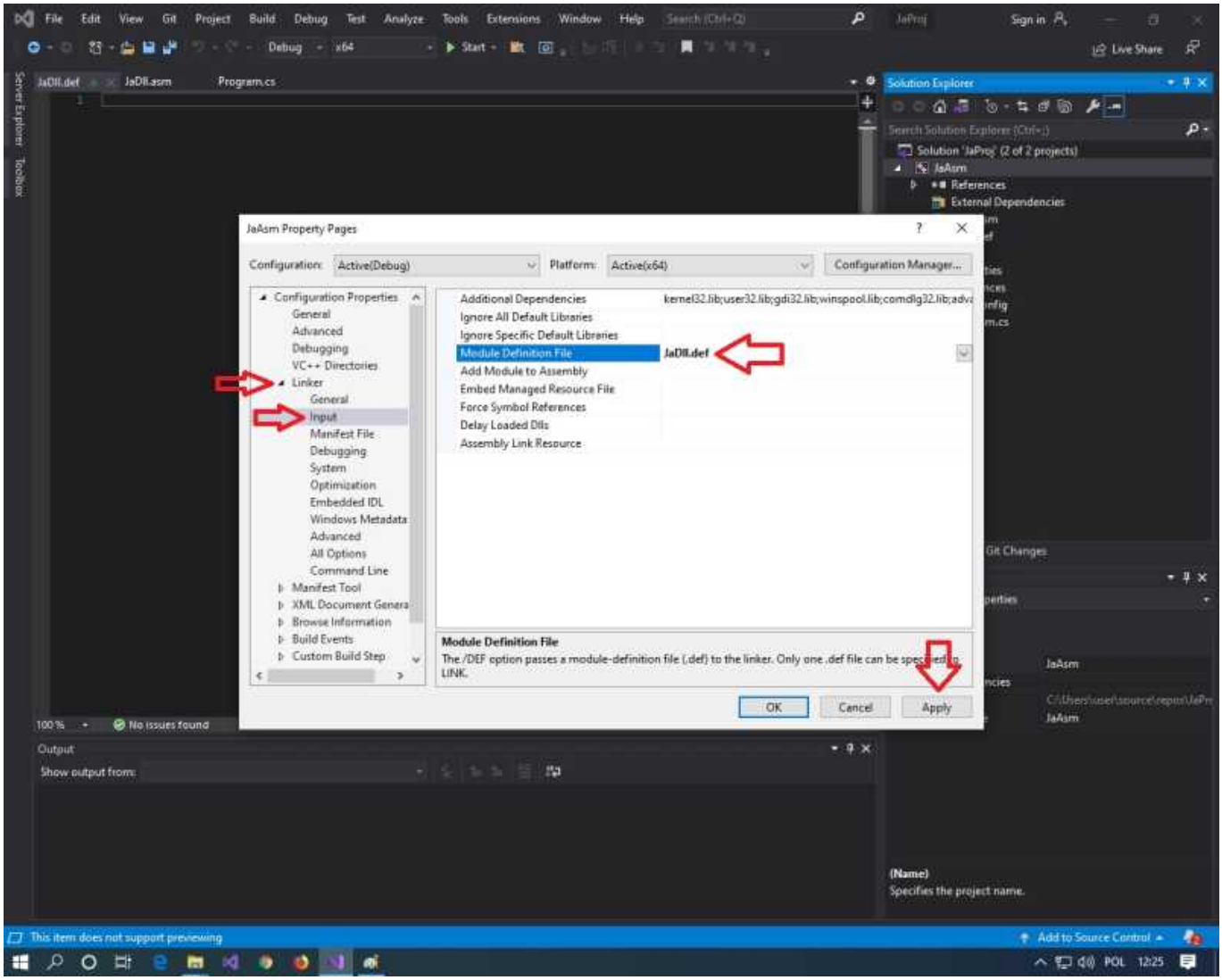
1.2.6. Końcowa konfiguracja biblioteki asemblerowej

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

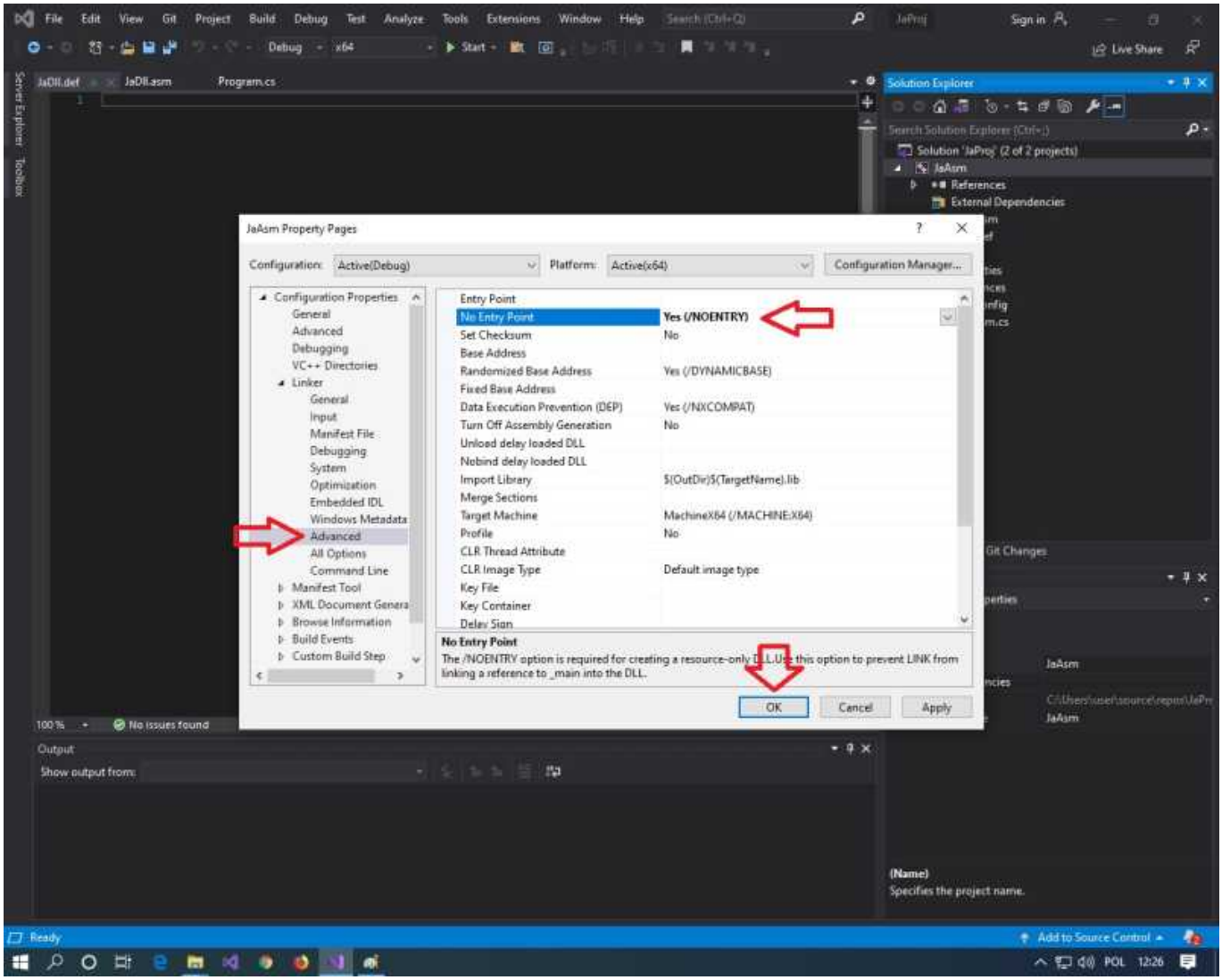

Right Mouse Key



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.

1.3. Wykorzystanie biblioteki dll napisanej w asemblerze

1.3.1. 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
...
```

1.3.2. 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);
...
```

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

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/pl:edu:zmitac:n1z:ja:lab2?rev=1617470759

Last update: **2021/04/03 19:25**