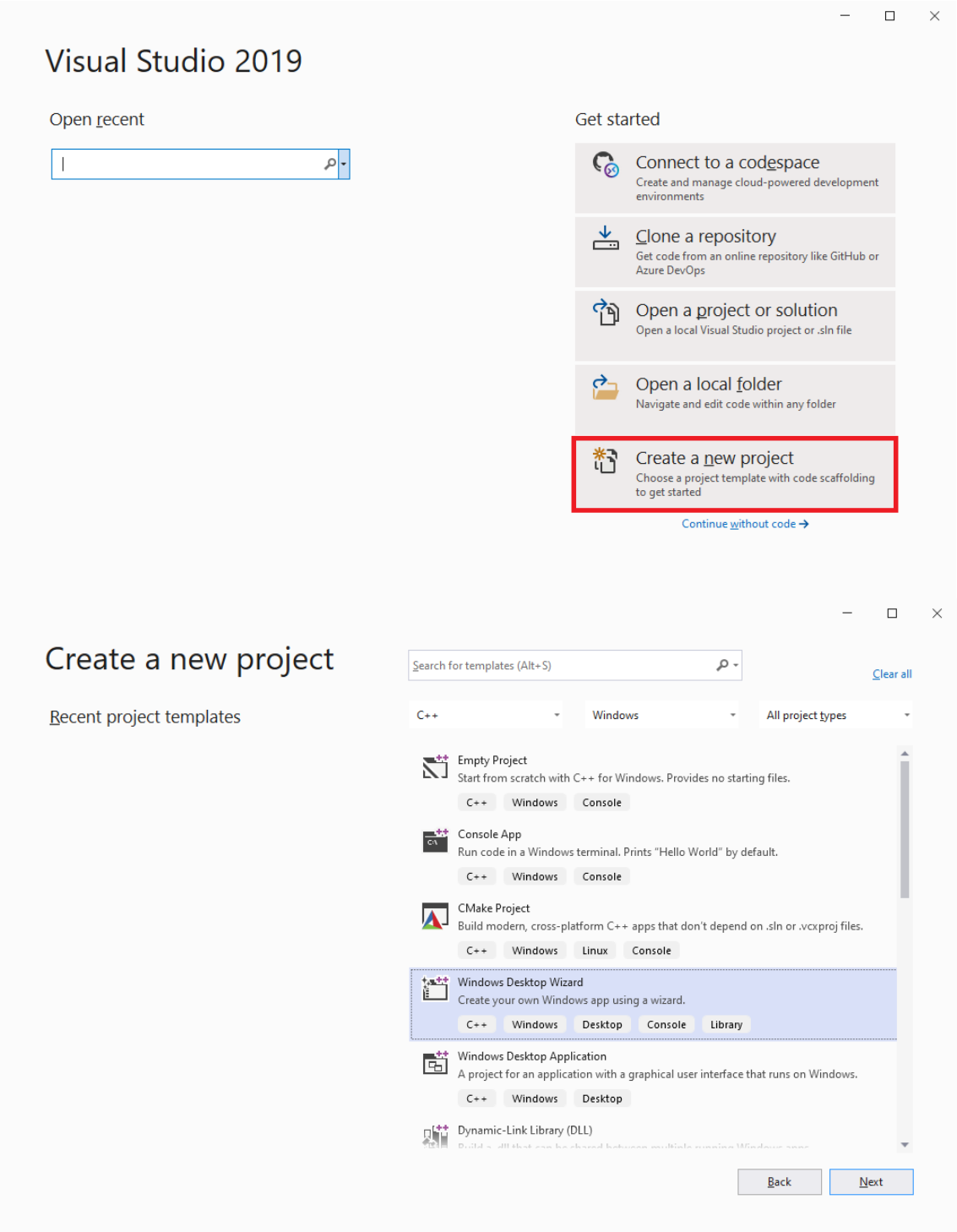
Laboratorium programowania niskopoziomowego

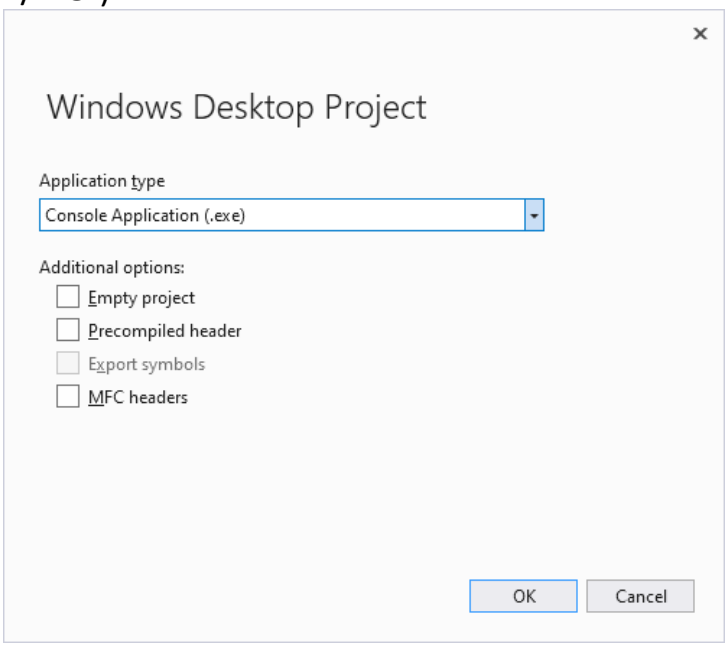
LAB 2 x64 - Proste podprogramy. Uruchamianie krokowe.

Celem ćwiczenia jest zapoznanie studenta z podstawową obsługą Visual Studio w wersji kompilacji z asemblerem 64 bitowym.
Na początku wybieramy projekt Windows Desktop Wizard (w poprzednich wersjach Visual Studio wybieramy Win32 Console Application) dla języka C++ (jak na załączonym obrazku) (**PL:** Kreator aplikacji klasycznej systemu Windows / **UA:** / **RU:**)

Następnie wybieramy odpowiednią nazwę i lokalizację.

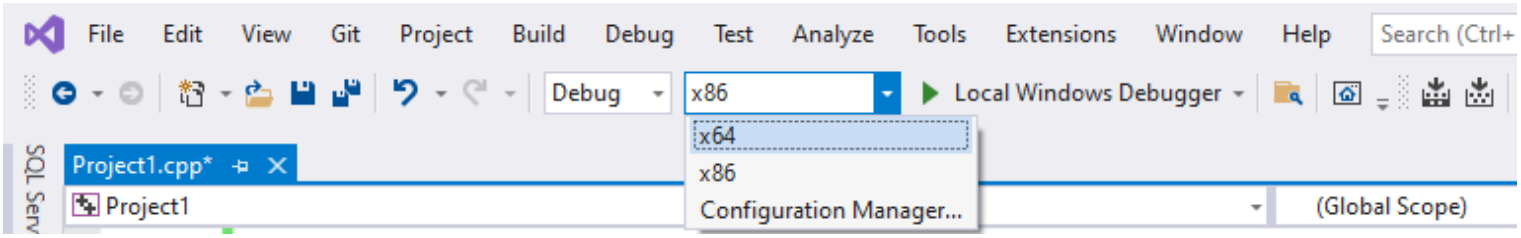


Następnym krokiem są dodatkowe ustawienia jak na poniższych obrazkach (klikamy Create) (**PL:** Aplikacja konsolowa / **UA:** / **RU:**):

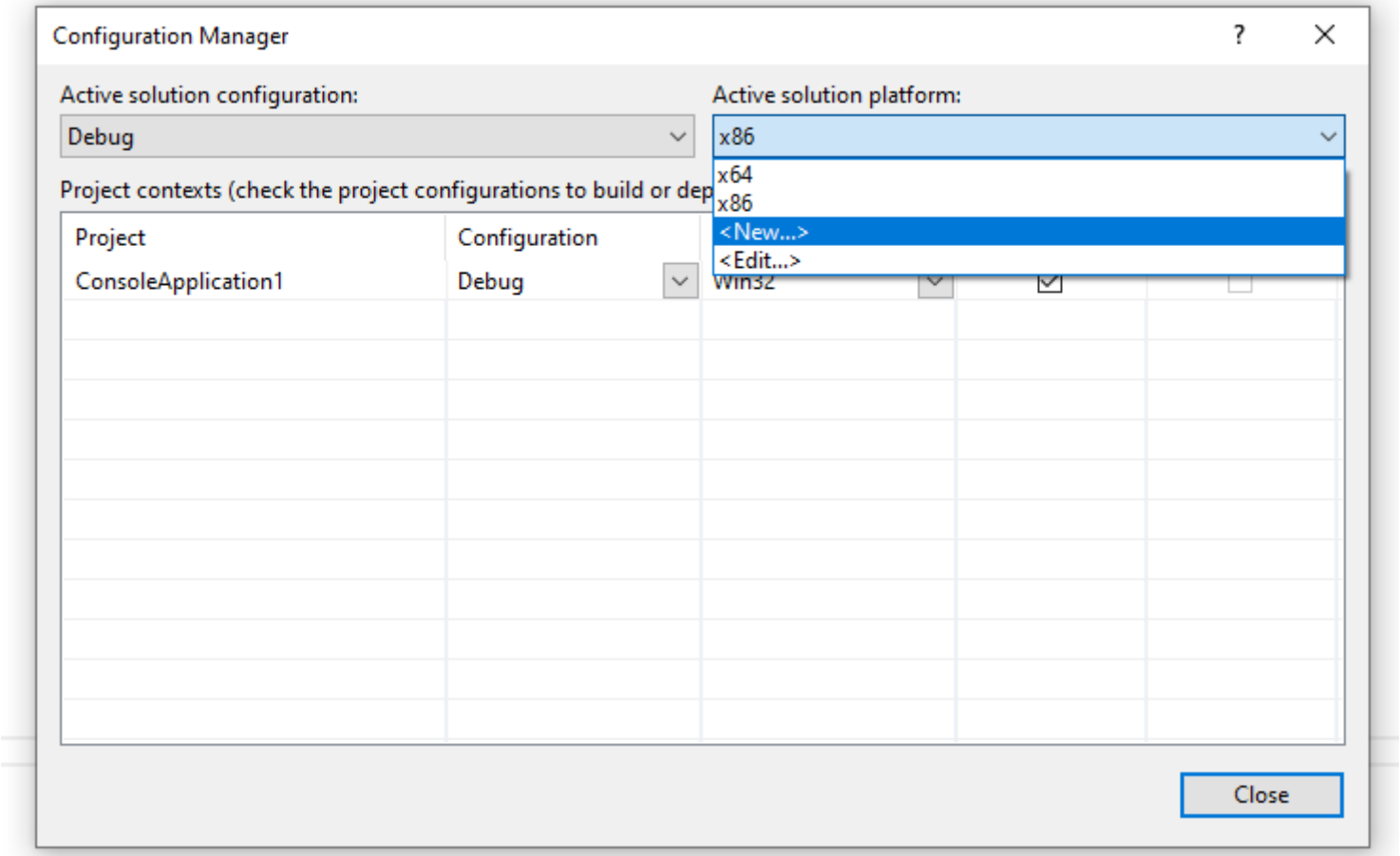


Pozostawiamy domyślne ustawienia i klikamy OK.

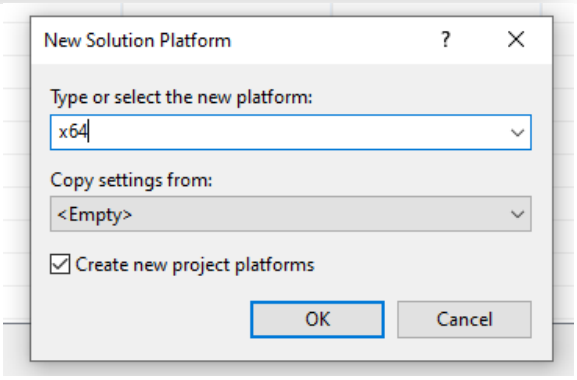
Został utworzony projekt w wersji 32-bitowej. Należy zamienić go na wersję 64-bitową. W tym celu trzeba rozwinąć pole z „x86” i wybrać „x64”.



➔ W przypadku gdy nie ma takiej opcji wyboru należy dodać ją w *Configuration Manager…*
➔ (**PL:** Menedżer konfiguracji / **UA:** / **RU:**)

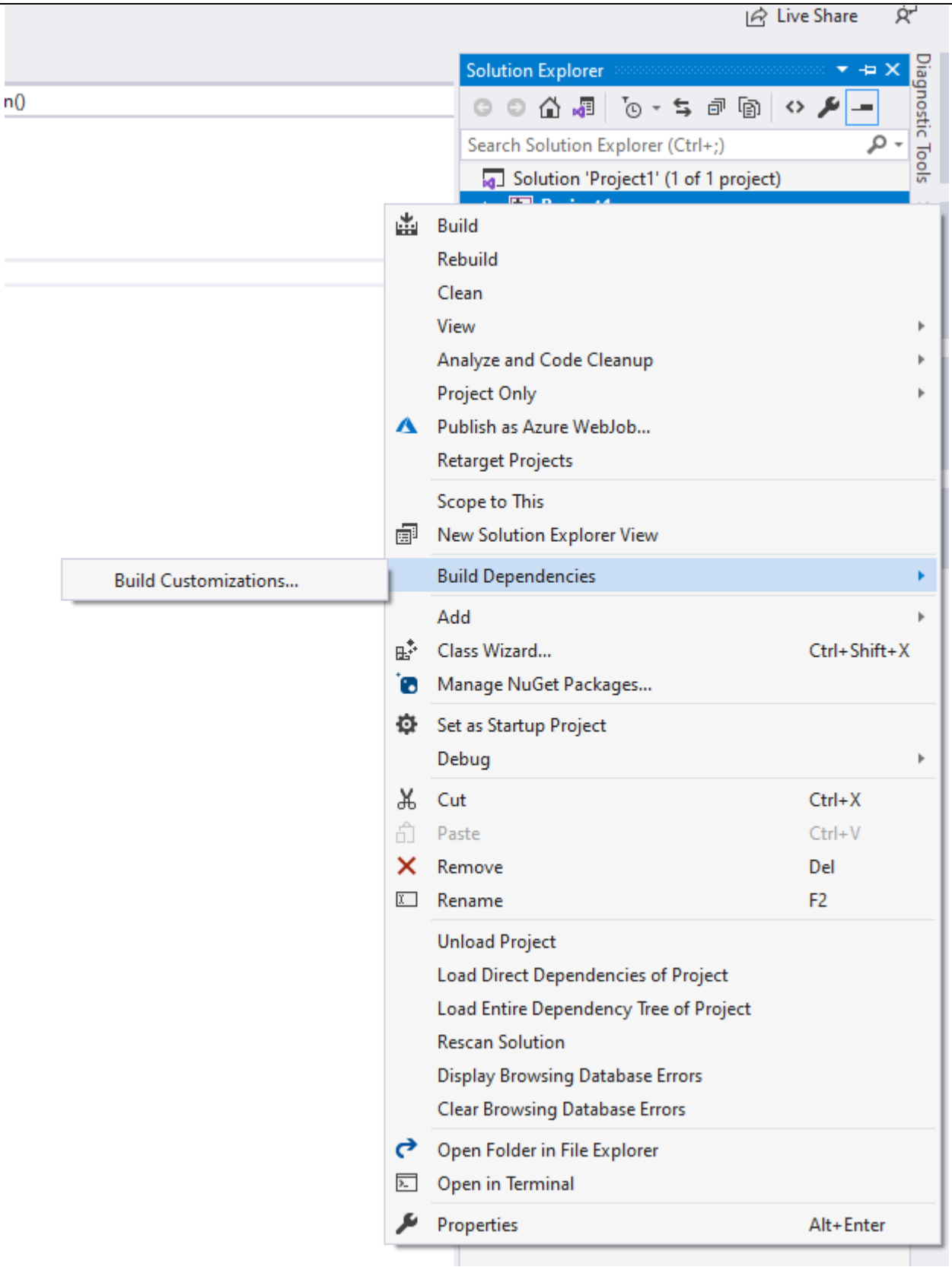


Klikamy na <New…> (**PL:** Nowy / **UA:** / **RU:**) i wybieramy z listy rozwijanej x64, dodatkowo wybieramy nowe ustawienia <Empty> (**PL:** Pusty / **UA:** / **RU:**) (brak kopiowania ustawień) oraz zaznaczamy „Create new project platrofms” (**PL:** Utwórz nowe platformy projektu / **UA:** / **RU:**).

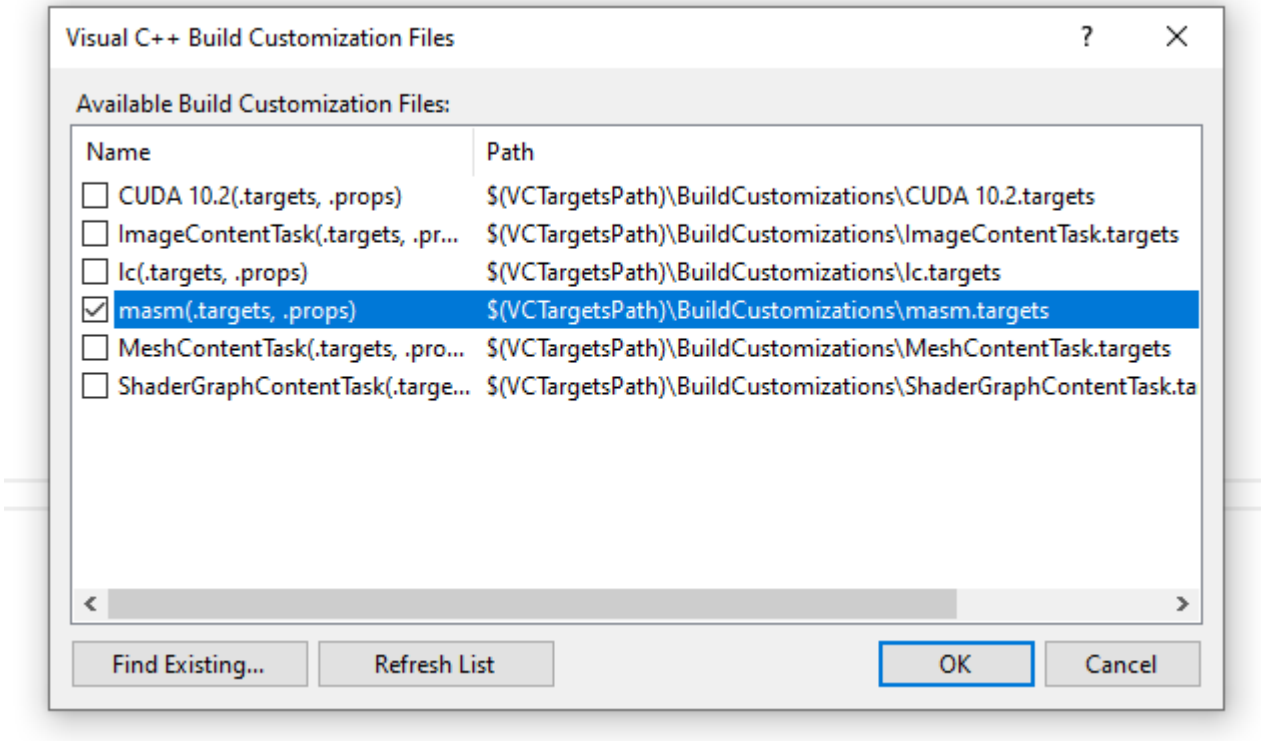


Wracamy do „Configuration Managera” i wybieramy „Active solution platform” (**PL:** Aktywne platformy rozwiązania / **UA:** / **RU:**) na dodaną „x64”. Zamykamy okno. Upewniamy się czy w głównym oknie kompilatora jest wybrana platforma „x64”.

Nasz projekt stał się 64-bitowy.
Kolejnym etapem jest konieczność włączenia możliwości użycia pakietu „masm” dla 64-bitowego kodu.
W tym celu należy kliknać prawym przyciskiem myszy na nazwie projektu w oknie „Solution Explorer” (**PL:** Eksplorator rozwiązań / **UA:** / **RU:**) i wybrać „Build Dependencies” oraz następnie „Build Customizations…” (**PL:** Zależności kompilacji (B) -> Dostosowywanie kompilacji / **UA:** / **RU:**).

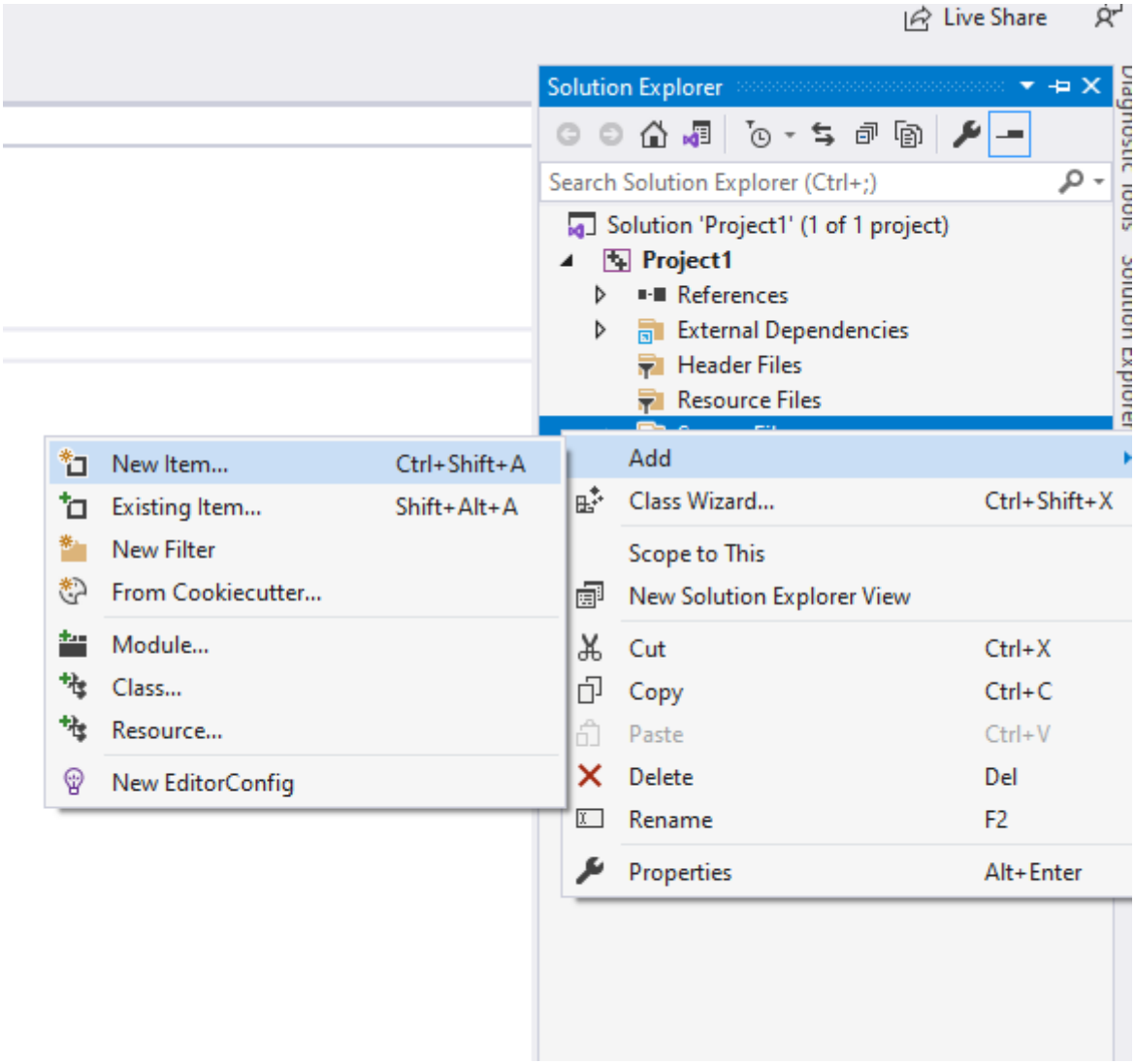

Live Share
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Project1' (1 of 1 project)
Diagnostic Tools
Build
Rebuild
Clean
View
Analyze and Code Cleanup
Project Only
Publish as Azure WebJob...
Retarget Projects
Scope to This
New Solution Explorer View
Build Customizations...
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Set as Startup Project
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename F2
Unload Project
Load Direct Dependencies of Project
Load Entire Dependency Tree of Project
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Open Folder in File Explorer
Open in Terminal
Properties Alt+Enter

W nowym okienku należy zaznaczyć używanie pakietu „masm” i potwierdzić przyciskiem OK.

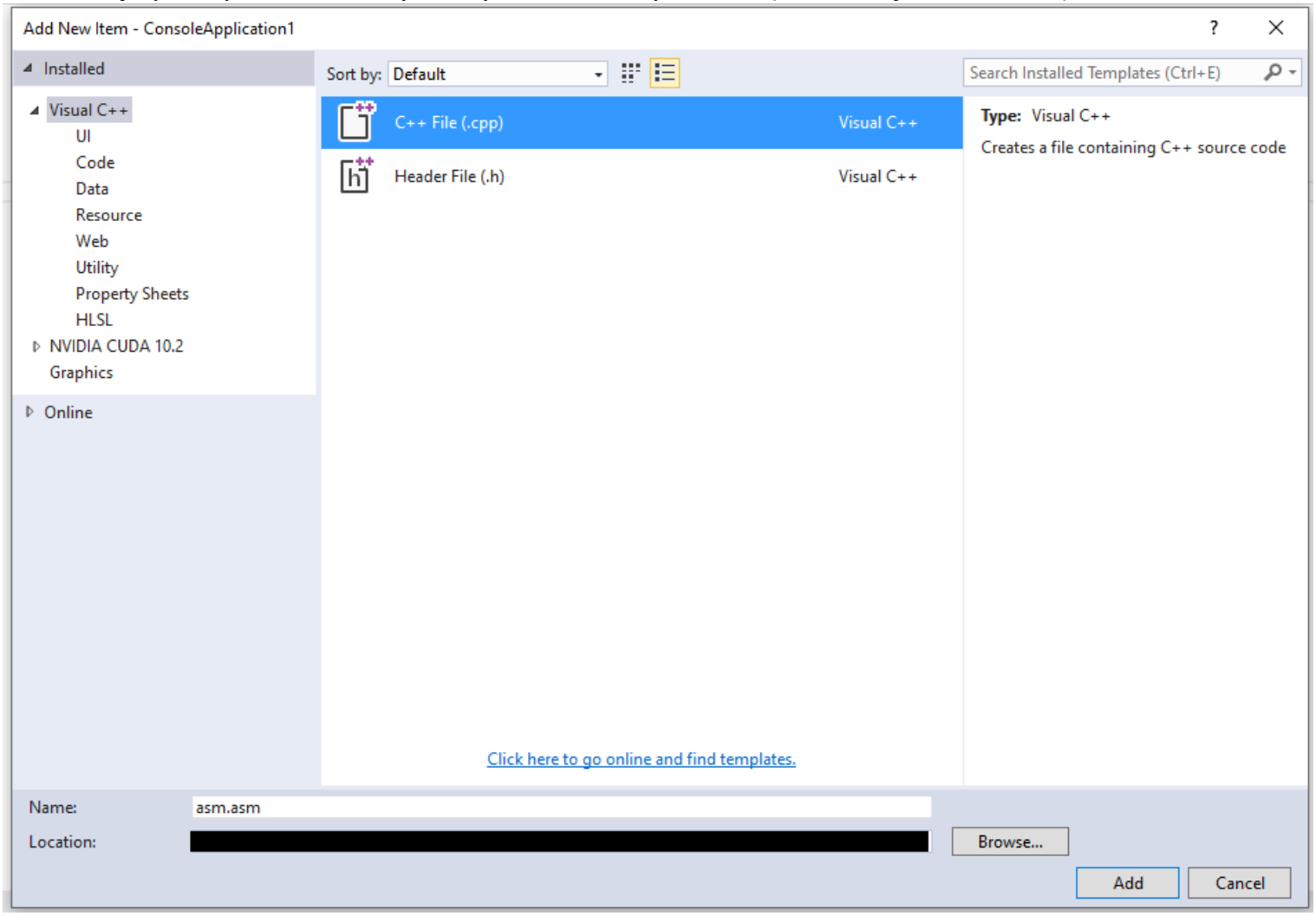


Na tym kończy się konfigurowanie Visual Studio w celu kompilacji z użyciem assemblera 64 bitowego. Jednakże należy wspomnieć, że kod assemblera 32 lub 64bitowy w trybie kompilacji 64bitowej nie może być kompilowany w metodzie „Inline” – jest to nie dozwolone z poziomu kompilatora. Dlatego należy przygotować odpowiednio kod do prawidłowej kompilacji.

W tym celu dodajemy nowy plik źródłowy o nazwie „asm.asm” (nazwa może być dowolna, rozszerzenie asm służy do oznaczania kodu assemblera). Można również dodać istniejący już plik z kodem w asemblerze i rozszerzeniem asm. (**PL:** Dodaj -> Nowy element / **UA:** / **RU:**)



Lokalizacje pliku pozostawiamy domyślna. Klikamy „Add”. (**PL:** Dodaj / **UA:** / **RU:**)



Na tym kończy się konieczność dodawania nowych plików. Teraz wystarczy uzupełnić odpowiednio kod w pliku *.cpp i asm.asm.

Przykładowy kod w pliku *.cpp może wyglądać tak:

Kod w C++:

```
#include <iostream>

using namespace std;

// deklaracja funkcji zewnętrznej
extern "C" __int64 suma(int a, int b);

int main(int argc, char * argv[])
{
	int a = 10;
	int b = 20;
	int c;
	c = suma(a, b);
	cout << "Suma=" << c << endl;
	system("PAUSE");
	return 0;
}
```

Widzimy tutaj deklaracje funkcji zewnętrznej `suma` (jest to funkcja napisana w assemblerze), oraz jej wywołanie w ciele funkcji głównej „main”.

Schemat tworzenia pliku ASM.

Poniższy szablon pozwala umieszczać w pliku *.asm dane i kod w odpowiednich segmentach:

```
.CODE
_DATA SEGMENT

_DATA ENDS

_TEXT SEGMENT

_TEXT ENDS
END
```

Dane do odczytu / zapisu definiuje się w segmencie _DATA.
Dane tylko do odczytu i kod programu w segmencie _TEXT.

Przykład użycia:

Kod w asemblerze:

```
.CODE
_DATA SEGMENT
_DATA ENDS
_TEXT SEGMENT
PUBLIC suma
suma PROC
    movsxd rax, ecx
    movsxd rdx,edx
    add rax,rdx
    ret
suma ENDP
_TEXT ENDS
END
```

Kod w C++:

```
#include <iostream>

using namespace std;

// deklaracja funkcji zewnętrznej
extern "C" __int64 suma(int a, int b);

int main(int argc, char * argv[])
{
    int a = 10;
    int b = 20;
    int c;
    c = suma(a, b);
    cout << "Suma=" << c << endl;
    system("PAUSE");
    return 0;
}
```

Słowo PUBLIC powoduje, że nazwa suma staje się dostępna dla zewnętzrnego kodu.
Deklaracja:
suma PROC
…
ret
suma ENDP
tworzy podprogram w asemblerze. Argumenty zostały przekazane poprzez rejestry RCX i RDX (patrz tabela na wykładzie.

Zadanie 1

Proszę napisać program obliczający objętość sześcianu o boku `a`.

Zadanie 2

Proszę obliczyć wartość y:

$y=ax^2 + bx + c$

dla zadanych parametrów `a`, `b`, `c` i `x`.

Zadanie 3

Proszę napisać w C++ i w asemblerze podprogramy konwertujące wartości 32-bitowe *little endian* na *big endian*.

Wywołanie funkcji systemu Windows - **okna komunikatu**.

```
_DATA SEGMENT
_DATA ENDS

_TEXT SEGMENT
hello_msg db "Hello world!", 0          ;dekl. danych – zestawu bajtów
info_msg db "Info", 0                   ;łańcuchy znaków zakończ zerem

EXTERN MessageBoxA: PROC                ;dekl. zewnętrznej funkcji
EXTERN GetForegroundWindow: PROC

PUBLIC hello_world_asm

hello_world_asm PROC                    ;deklaracja podprogramu
push rbp                                ;zabezpiecz rej. rbp wskaźnik ramy
mov rbp, rsp                            ; ustaw nowy wskaźnik ramy stosu
sub rsp, 8 * (4 + 2)                    ; alokacja miejsca dla parametrów
                                        ; wywołania (co najmniej 4)
                                        ; + 2 QWORD dla wyrównania stosu
and rsp, not 15                         ; wyrównanie stosu do 16 bajtów przed
                                        ; wywołaniem funkcji API
call GetForegroundWindow                ; Pobierz uchwyt okna
mov rcx, rax
                                        ; Sposób wywołania funkcji MessageBoxA:
                                        ; WINUSERAPI int WINAPI MessageBoxA(
                                        ; RCX => _In_opt_ HWND hWnd,
                                        ; RDX => _In_opt_ LPCSTR lpText,
                                        ; R8 => _In_opt_ LPCSTR lpCaption,
                                        ; R9 => _In_ UINT uType);
mov rdx, offset hello_msg
mov r8, offset info_msg
mov r9, 0                               ; MB_OK
call MessageBoxA

mov rsp, rbp                            ; przywrócenie wskaźnika stosu
pop rbp                                 ; przywrócenie wskaźnika ramy stosu
ret                                     ;powrót z podprogramu

hello_world_asm ENDP
_TEXT ENDS
END
```

Kod w C++:

```
extern "C" void hello_world_asm();

int main(int argc, char * argv[])
{
     hello_world_asm();
     return 0;
}
```

Istnieje również możliwość użyca uproszczonego pliku ASM

ConsoleApplication1.cpp | asm.asm

ConsoleApplication1

```cpp
// ConsoleApplication1.cpp : Defines the
//

#include "stdafx.h"
#include <iostream>
#include <windows.h>

using namespace std;

extern "C" INT64 ADD64(INT64, INT64);
extern "C" INT64 Max64(INT64, INT64);
extern "C" INT64 Min64(INT64, INT64);

int main()
{
    INT64 a = 30;
    INT64 b = 20;
    INT64 w;

    w = ADD64(a, b);
    cout << "Wynik: " << w << endl;

    w = Max64(a, b);
    cout << "Max: " << w << endl;

    w = Min64(a, b);
    cout << "Min: " << w << endl;

    system("PAUSE");
    return 0;
}

```

ConsoleApplication1.cpp | asm.asm*

```
.code

ADD64 proc
    add rcx, rdx;
    mov rax,rcx;
    ret;
ADD64 endp


Max64 proc
    cmp rcx, rdx;
    jnc Mexit;
    mov rax, rdx;
    jmp Exit;
Mexit:
    mov rax, rcx;
Exit:
    ret;

Max64 endp


Min64 proc

    cmp rcx, rdx;
    jnc Mexit;
    mov rax, rcx;
    jmp Exit;
Mexit:
    mov rax, rdx;
Exit:
    ret;
Min64 endp

end
```

Uproszczony plik ASM charakteryzuje się minimalną ilością kodu. Niezbędny jest tylko „.code” oraz „end”, wszystko co jest pomiędzy to już definicja funkcji zewnętrznych.

UWAGA!!! Podprogramy w asemblerze nie mają dostępu do zmiennych zadaeklarowanych w kodzie C++, dlatego muszą pobierać wszystkie dane w postaci parametrów aktualnych funkcji. Parametry funkcji domyślnie przekazywane są w rejestrach RCX, RDX, R8, R9, każdy kolejny parametr jest zapisany na stosie. Funkcja zwraca wartość w rejestrze RAX.

Zadania do samodzielnego wykonania:

1. Przepisać kod z ostatniego przykładu i przejść krokowo.
2. Zgłosić prowadzącemu, że kod działa prawidłowo lub że są problemy – spróbować je rozwiązać razem z prowadzącym zajęcia.