Table of Contents

1. Ćwiczenie 3: Integracja kodu niezarządzalnego, tworzonego w języku Assembler wraz z kodem zarządzalnym .NET C# 2
1.1. Środowisko sprzętowe i programistyczne 2
1.2. Wymagania 2
1.3. Wstęp 2
1.4. Powtórka z teorii: Przekazywanie danych do funkcji 2
1.4.1. Typy danych w .NET a typy danych w assemblerze 3
1.4.2. Automatyczne zarządzanie pamięcią w .NET 3
1.5. Dołączanie biblioteki DLL do kodu w C# 5
1.6. Konfiguracja środowiska 6
1.6.1. Ustawienie kolejności kompilacji (zależności) 7
1.6.2. Zmiana projektu DLL w C++ na projekt DLL w assemblerze 8
1.6.3. Ustawienie wspólnego folderu docelowego 13
1.6.4. Konfiguracja na potrzeby kodu unsafe oraz debuggowania 15
1.7. Przekazywanie wskaźników z poziomu C# do funkcji w assemblerze 16
1.7.1. Przykład 1 16
1.7.2. Przykład 2 17
1.8. Zadanie do wykonania 21
1.9. Metoda weryfikacji wyników 22
1.10. Pomocne zasoby 22

1. Ćwiczenie 3: Integracja kodu niezarządzalnego, tworzonego w języku Assembler wraz z kodem zarządzalnym .NET C#

1.1. Środowisko sprzętowe i programistyczne

Do wykonania ćwiczenia niezbędny jest komputer PC w postaci maszyny fizycznej lub wirtualnej z systemem Windows 7, 8, 8.1 lub 10, z zainstalowanym i skonfigurowanym środowiskiem programistycznym Microsoft Visual Studio 2019 Community Edition[1)].



Proszę nie pomylić z Visual Studio Code, to zupełnie inny produkt, który nie nadaje się wprost do wykorzystania w naszym laboratorium do realizacji zadań.



Szczegóły instalacji i konfiguracji środowiska przedstawiono w ramach Laboratorium 1 i 2.

1.2. Wymagania

Studenci muszą rozumieć sposób tworzenia i działania aplikacji typu “Command Line”, podstawy języka assembler oraz podstawy języka C#.

1.3. Wstęp

W niniejszym ćwiczeniu wykorzystamy wiedzę zdobytą w poprzednich ćwiczeniach. Tematyką ćwiczenia są struktury danych, dostęp do nich oraz przenoszenie informacji pomiędzy kodem zarządzalnym (C#) i kodem niezarządzalnym (assembler) w kontekście automatycznego zarządzania pamięcią w środowisku .NET i jego konsekwencji dla programowania w assemblerze.

1.4. Powtórka z teorii: Przekazywanie danych do funkcji

W kontekście przekazywania argumentów (danych) do funkcji, rozważamy dwa zasadnicze przypadki

- przekazywanie przez wartość,
- przekazywanie przez referencję (wskaźnik).

W większości języków programowania zmienne proste typu `int, word, float, byte, double` są przekazywane przez wartość. Oznacza to, że z punktu widzenia wywołującego, oryginalna zmienna jest kopiowana i przekazywana jest do funkcji jej wartość. Modyfikacja tej zmiennej po stronie wywoływanej funkcji nie zmienia wartości oryginału. Oczywiście tego typu wywołania bardzo mocno ograniczają możliwości przeakazywania parametrów do funkcji.

W przypadku złożonych typów danych, takich jak np: tablice zmiennych, struktury, klasy (instancje klas), zamiast przekazywania do funkcji wszystkich elementów tworzących strukturę danych, przekazuje się wskaźnik na tę strukturę. Dzięki temu, przekazując jedną wartość (adres=wskaźnik) przekazujemy niejako wszstykie dane, niemniej jednak nie są one kopiowane: funkcja wywoływana operuje na oryginałach i jeśli zmienia ich wartość to zmianie następuje wartość oryginalna, a nie jej kopia lokalna w funkcji wywoływanej,

Konsekwencją drugiego podejścia jest fakt, że struktura danych pozostaje niejako w przestrzeni pamięci wywołującego (nie jest kopiowana do wywoływanego), co w przypadku mieszania kodu zarządzalnego i niezarządzalnego ma istotne konsekwencje.

1.4.1. Typy danych w .NET a typy danych w assemblerze

Typy danych w .NET mogą być zaskakująco odmienne od tego, czego spodziewalibyśmy się patrząc w punktu widzenia assemblera. Poza takimi oczywistościami jak np. `byte`, należy pamiętać, że zmienna typu `char` w języku C# (oraz VB.NET) ma … 2 bajty długości. Wynika to z faktu stosowania unicode do przechowywania liter, co pozwala na znaczne zwiększenie po stronie .NET zakresu znaków, ale stanowi nie lada wyzwanie w przypadku kodu w assemblerze.
.NET stosuje dwa rodzaje zmiennych numerycznych: ze znakiem i bez znaku. Zasady prezentacji liczb ujemnych są takie same jak w assemblerze (i zgodnie ze stosowanym zapisem w uzupełnieniu do dwóch w procesorach Intel/AMD).
Ponadto wskaźniki w przypadku kodu 32-bitowego (x86) są 32-bitowe, a w przypadku kodu x64, 64-bitowe. W przypadku tego laboratorium używamy kodu 64-bitowego.
Odpowiednikiem typu word (16 bitów) po stronie assemblera jest typ `short` i `ushort` po stronie C# (odpowiednio ze znakiem i bez znaku).

1.4.2. Automatyczne zarządzanie pamięcią w .NET

W środowisku .NET, dowolna struktura danych (tabilca, obiekt, zmienna) może zostać przesunięta przez wewnœtrzny mechanizm z jednego miejsca w pamięci do innego. Mechanizm ten śledzi również na bieżąco kod źródłowy i rejestruje aktywne odwołania do zmiennej. Jeśli wykryje, że w kodzie nie ma więcej odwołań do danej zmiennej, jest ona usuwana z pamięci automatycznie, dzięki czemu w .NET nie trzeba samemu alokować i zwalniać pamięci. Za proces ten odpowiada tzw. `Garbage Collector`[2)].

Uwaga! W przypadku mieszania w jednym rozwiązaniu kodu zarządzalnego i niezarządzalnego należy mieć na uwadze, że `Garbage Collector` nie jest w stanie monitorować, do jakich zmiennych (fragmentów pamięci) odwołuje się kod w Assemblerze, ponieważ kod w Assemblerze jest “poza” środowiskiem .NET.



`Garbage Collector` nie jest w stanie monitorować, do jakich zmiennych odwołuje się kod w Assemblerze, ponieważ kod w Assemblerze jest “poza” środowiskiem .NET.

Konsekwencją powyższego faktu są przypadki, gdzie kod w assemblerze będzie “rzucał wyjątki” dostępu do pamięci i utraci (może utracić) dostęp do danych przekazanych z .NET, ponieważ (rys. 1, 2 i 3):

- W czasie przetwarzania danych przez funkcję w assemblerze, Garbage Collector wykonał cykl porządkowania, wykrył, że w kodzie .NET nie ma dalszych odwołań do zmiennej, skutkiem czego usunął ją i zwolnił pamięć po stronie kodu wywołującego w środowisku .NET; Kod funkcji w assemblerze utraci dostęp do danych i w konsekwencji pojawi się wyjątek odczytu/zapisu pamięci, ponieważ pod wskaźnikiem, który znajduje się po stronie kodu w assemblerze, nie ma już danych, na które ten wskaźnik wskazywał;
- W czasie przetwarzania danych przez funkcję w assemblerze, Garbage Collector wykonał cykl porządkowania i przesunął zmienną w inne miejsce w pamięci z powodów optymalizacyjnych; Kod w assemblerze nie otrzyma “aktualizacji” wskaźnika, ponieważ jest poza jurysdykcją środowiska .NET, skutkiem czego nastąpi błąd odczytu/zapisu pamięci, ponieważ wykonując dereferencję wskaźnika trafimy w miesce w pamięci, które nie przechowuje już zmiennej (jest wolne lub zajęte przez inną strukturę danych, po stronie .NET);
- Inne przypadki: np. asynchroniczny wątek aplikacji w środowisku .NET zmodyfikuje zmienną i np. skróci tablicę zmiennych, co może skutkować błędami po stronie kodu w assemblerze, w szczególności błędami logicznymi i błedami indeksacji i/lub wyjątkami związanymi z dostępem do pamięci;

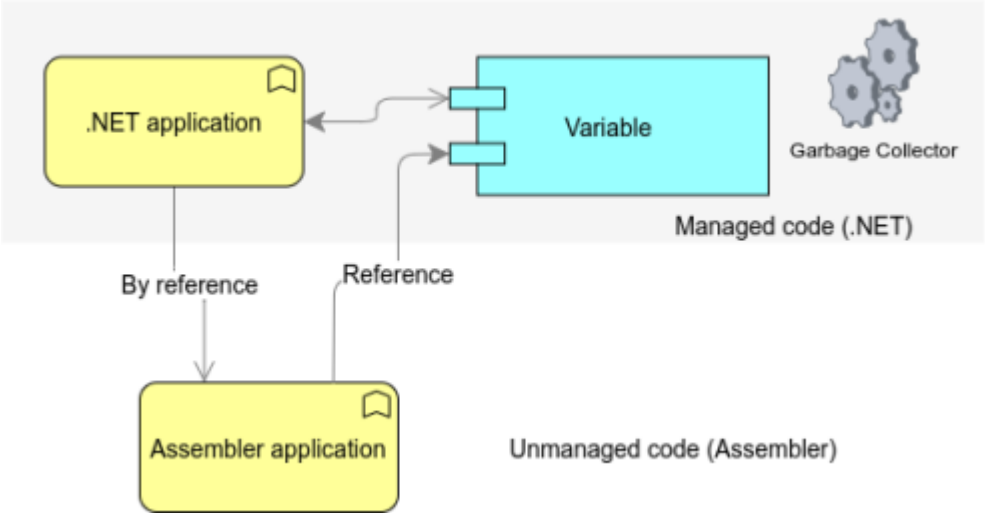


Fig. 1: Zmienna po stronie .NET jest w lokalizacji zgodnej z referencją

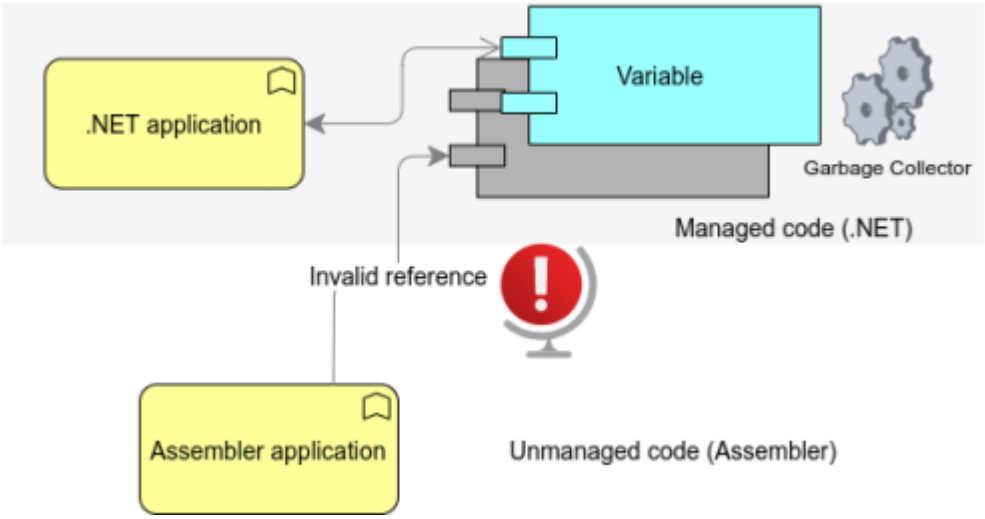


Fig. 2: Zmienna po stronie .NET została przesunięta przez GC w inne miejsce

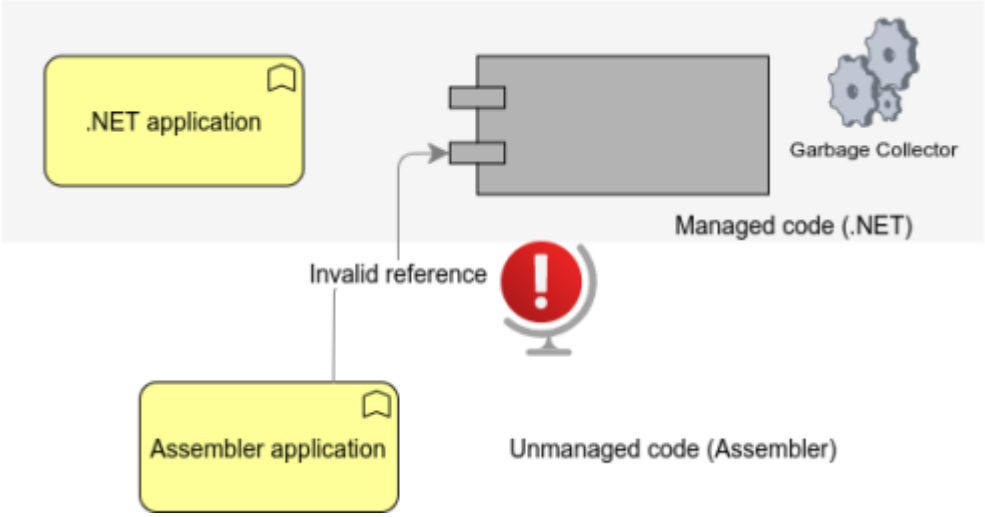


Fig. 3: Zmienna po stronie .NET została usunięta przez GC

W związku z powyższym, mieszając kod zarządzalny (.NET) oraz niezarządzalny (assembler) należy zapewnić aby:

- każdy fragment kodu w .NET (C#), który odwołuje się do kodu w assemblerze i jest związany z użyciem wskaźników (np. przekazywanie argumentów z wykorzystaniem wskaźnika) należy oznaczyć słowem kluczowym `unsafe` (references)[3];
- w celu poinformowania Garbage Collectora o zakazie relokacji zmiennej na czas wywołania funkcji w assemblerze należy użyć słowa kluczowego `fixed` po stronie kodu .NET (C#) [4] - fixed należy używać równolegle z `unsafe`;



Błędy związane z brakiem zastosowania powyższych reguł (relokacją pamięci przez Garbage Collector i następującym po nim odwołaniem z poziomu języka assembler) są niezwykle trudne do wykrycia, ponieważ aplikacja zachowuje się nieprzewidywalnie: raz działa, innym razem nie. Wynika to z faktu, że Garbage Collector może, ale nie musi przesunąć zmiennej, zatem kod w takim przypadku może działać "przez przypadek". Działanie `Garbage Collectora` jest zależne od wielu czynników, w tym np. od obciążenia pamięci operacyjnej przez inne procesy, co oznacza że dwukrotne uruchomienie aplikacji może powodować jej różne działanie.

1.5. Dołączanie biblioteki DLL do kodu w C#

W przypadku kodu w .NET, najwygodniej jest ładować bibliotekę w sposób dynamiczny. Niezaleznie od sposobu, musimy poinformować kompilator C# o tym, że funkcja w assemblerze znajduje się "poza" kodem źródłowym w C# (VB.NET), to znaczy że trzeba zadeklarować jej prototyp i wskazać go jako zewnętrzny (dyrektywa `extern`).
Poniżej znajduje się fragment aplikacji, która pokazuje, jak należy zadeklarować funkcję oraz jak ją wywołać:

```
using System;
using System.Runtime.InteropServices;

namespace ConsoleAsmTestTypes
{
```

```
    class Program
    {
        [DllImport("DLLAsm.dll")]
        private static extern int ProcAsm1(int a, int b);
        static void Main(string[] args)
        {
            int a = 10;
            int b = 20;
            unsafe
            {
                int c = ProcAsm1(a, b);
            }
        }
    }
}
```



Uwaga! Należy zapewnić, aby skompilowana biblioteka DLL (tutaj `DLLAsm.dll`) znalazła się w tym samym folderze, w którym znajdzie się skompilowana aplikacja .NET. W przeciwnym wypadku podczas uruchamiania zostanie rzucony wyjątek o braku możliwości zainstalowania biblioteki. W kolejnym rozdziale znajduje się infomacja, jak skonfigurować projekty, aby generowały wyniki do wspólnego folderu.



Można również pozostawić bibliotekę w pierwotnej lokalizacji i zmodyfikować dyrektywę `DLLImport` podając ścieżkę dostępu przed nazwą pliku DLL, względną lub absolutną. W przypadku użycia tej ostatniej, nie będzie można zmienić lokalizacji biblioteki, a w przypadku skopiowania biblioteki w inne miejsce na dysku, przestanie ona działać.

Uwaga na przekazywane i zwracane typy! Informacja o tym, gdzie znaleźć zmienne po stronie wywoływanej funkcji znajduje się w opisie do Laboratorium JA ćw. 1.
W powyższym przykładzie nie stosujemy dyrektywy `fixed` po stronie kodu w .NET (również użycie dyrketywy `unsafe` nie jest konieczne) ponieważ przekazujemy argumenty poprzez wartość, nie poprzez referencję. Dyrektywa `unsafe` została dołączona w celu jej wprowadzenia i prezentacji sposobu użycia.
Zewnętrzne funkcje mogą być deklarowane zarówno jako publiczne jak i prywatne.

1.6. Konfiguracja środowiska

Tak jak w każdym przypadku tworzenia projektu hybrydowego, tak i tutaj niezbędne jest stworzenie projektu i skonfigurowanie go pod kątem integracji z assemblerem. Zmiany są niezbędne w obu projektach tworzących rozwiązanie (solucję).
Struktura solucji zawiera dwa projekty, np jak na rysunku 4:

- Aplikacja typu Command Line w języku C# (główny projekt)
- Biblioteka DLL w C++, skonwertowana ręcznie do biblioteki w Assemblerze

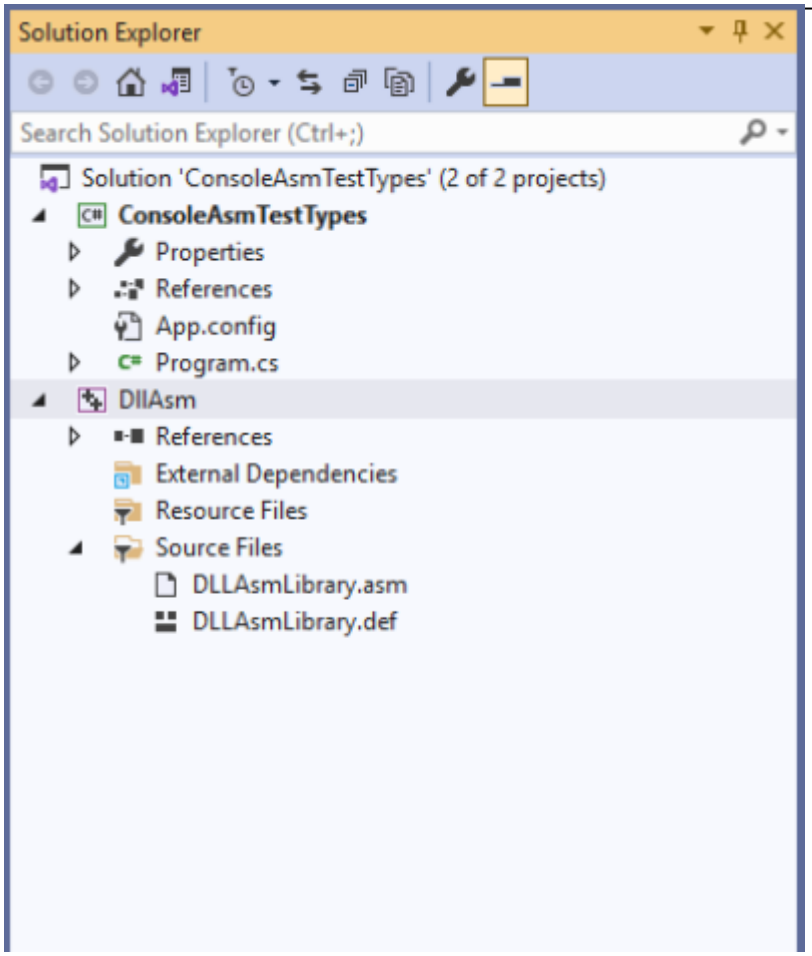


Fig. 4: Przykładowa konfiguracja solucji

1.6.1. Ustawienie kolejności kompilacji (zależności)

Prawym przyciskiem myszy należy kliknąć w projekt aplikacji konsolowej, a następnie wybrać “Build Dependencies”/“Project Dependencies...” i wskazać referencję (“Depends on:”) do projektu biblioteki DLL w assemblerze (jak na rysunku 5). Nazwy są zależne od nadanych nazw w trakcie tworzenia projektu:

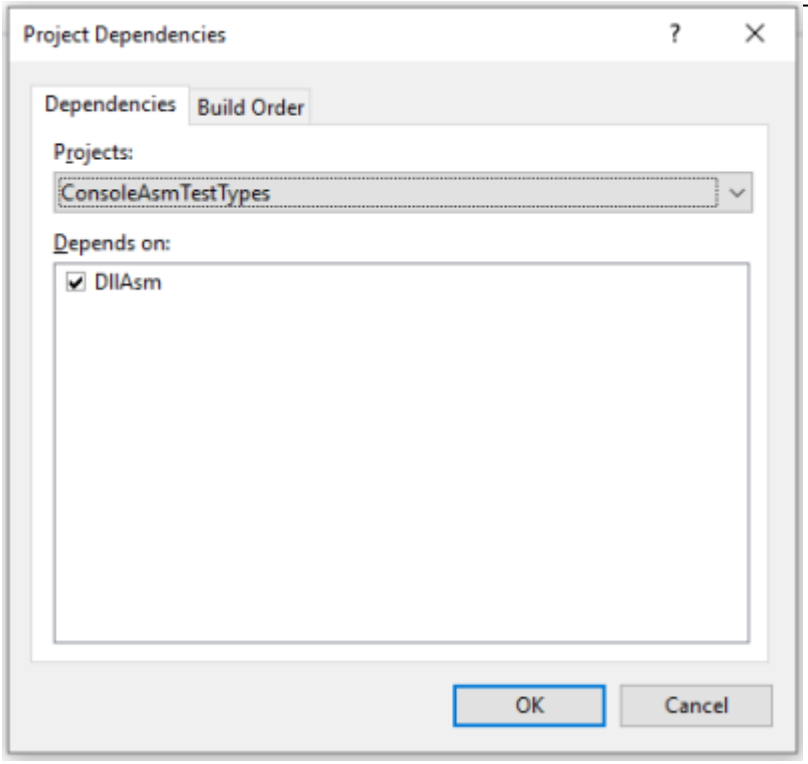


Fig. 5: Ustalenie zależności projektu konsolowego od projektu DLL (kompilacja)

Następnie w zakładce “Build Order” należy sprawdzić, czy kolejność kompilacji jest poprawna (najpierw DLL, potem zależna od niego aplikacja konsolowa, rys 6):

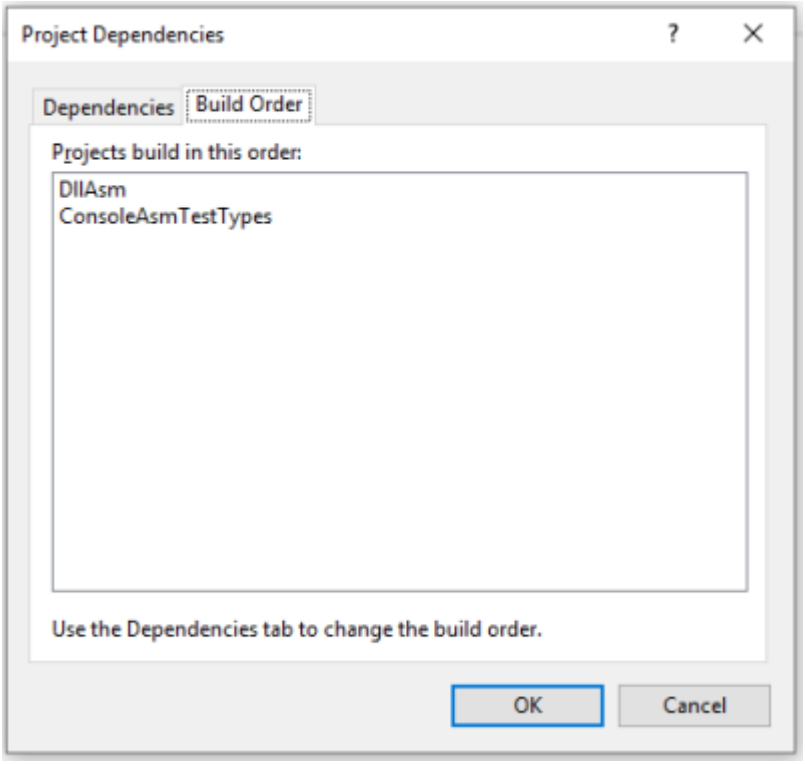


Fig. 6: Kontrola kolejności kompilacji

1.6.2. Zmiana projektu DLL w C++ na projekt DLL w assemblerze

W celu zmiany sposobu kompilacji (typu projektu) należy kliknąć prawym przyciskiem na nazwę projektu, a następnie wybrać “Build Dependencies”/“Build Customizations...” i wskazać “masm(.targets, .props)” (rysunek 7):

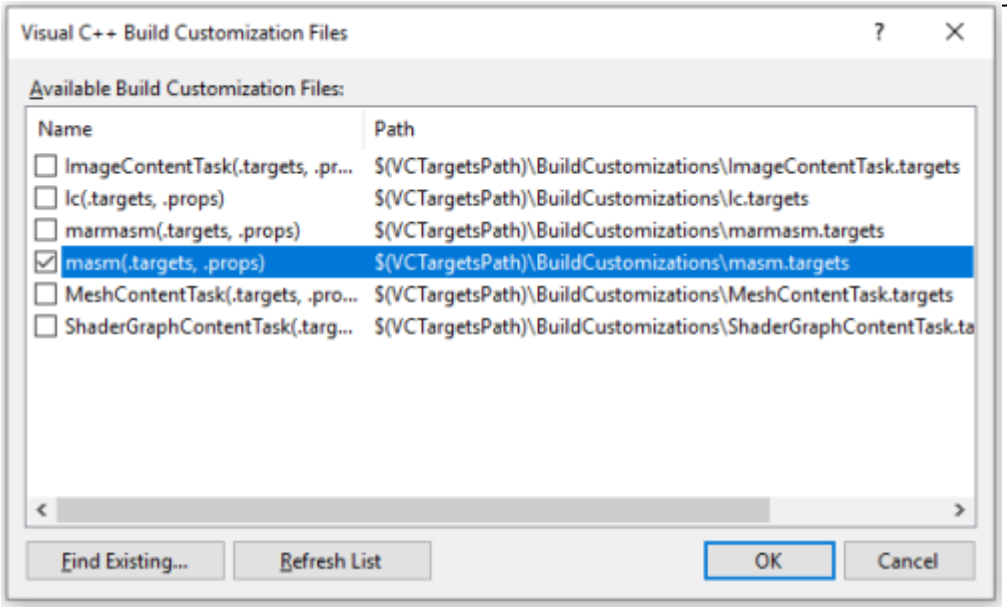


Fig. 7: Konfiguracja kompilatora dla projektu

Z solucji należy usunąć wszystkie pliki źródłowe, które zostały automatycznie wygenerowane podczas tworzenia projektu, a następnie dołożyć pliki źródłowe `.asm` i `.def`. Następnie plik `.asm` należy oznaczyć jako kod źródłowy assemblera (rysunek 8). W tym celu należy kliknąć prawym przyciskiem myszy na pliku `.asm`, wybierz “Properties” i w menu po lewej stronie wskaż “General”. Rodzaj pliku źródłowego należy wskazać w pozycji “General” i wybrać “Microsoft Macro Assembler” oraz ustawić wartość “Content” na “Yes”:

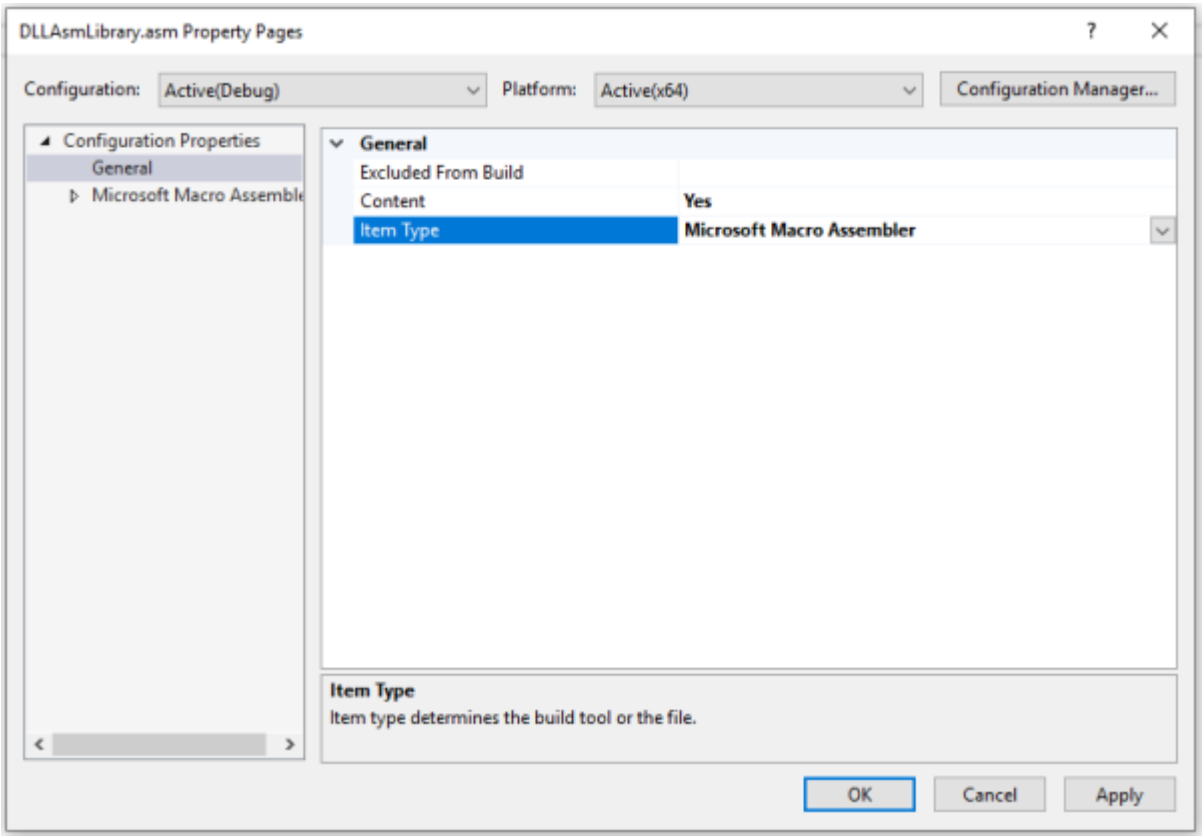


Fig. 8: Konfiguracja pliku źródłowego w assemblerze

We właściwościach projektu (prawym przyciskiem myszy należy klinąć nazwę projektu i wybrać “Properties”) wymagane jest ustawienie kilku parametrów konfiguracyjnych:

- “General”→“Configuration Type”: ustawić na “Dynamic Library (.dll)” (rys. 9);

- “Linker”/“Input”→“Module Definition File”: ustawić na nazwę pliku `.def`, który został dołączony do solucji (rys. 10);
- Sprawdzić (i ew ustawić) platformę docelową w “Linker”/“System”→“SubSystem”, powinna mieć wartość “Windows(/SUBSYSTEM:WINDOWS)” (rys. 11);
- “Linker”/“Advanced”→“No Entry Point”: ustawić na wartość “Yes” (rys. 12);
- Dodatkowo należy sprawdzić, czy w ww lokalizacji właściwość “Target Machine” jest ustawiona na “MachineX64 (/Machine:X64)” (rys 12);

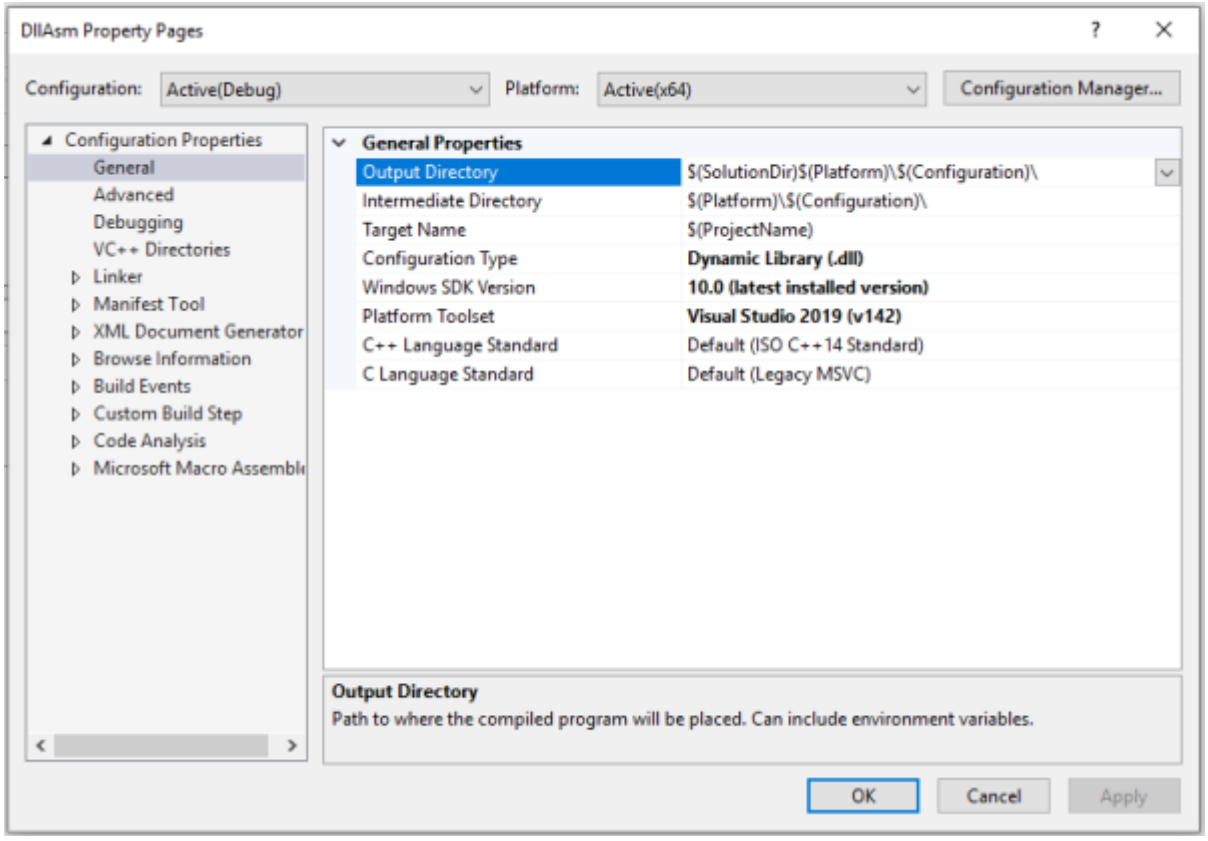


Fig. 9: Konfiguracja projektu w assemblerze cz. 1

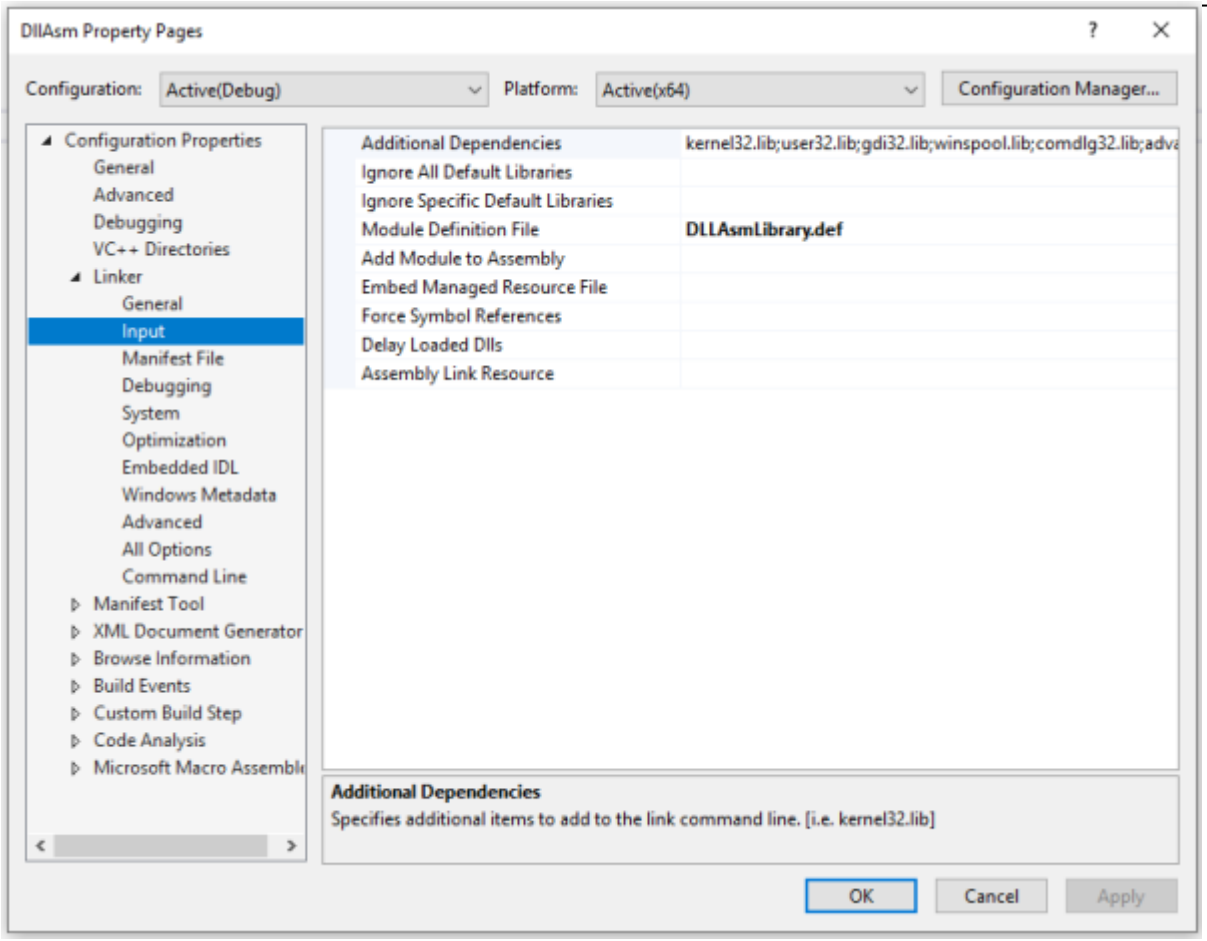


Fig. 10: Konfiguracja projektu w assemblerze cz. 2

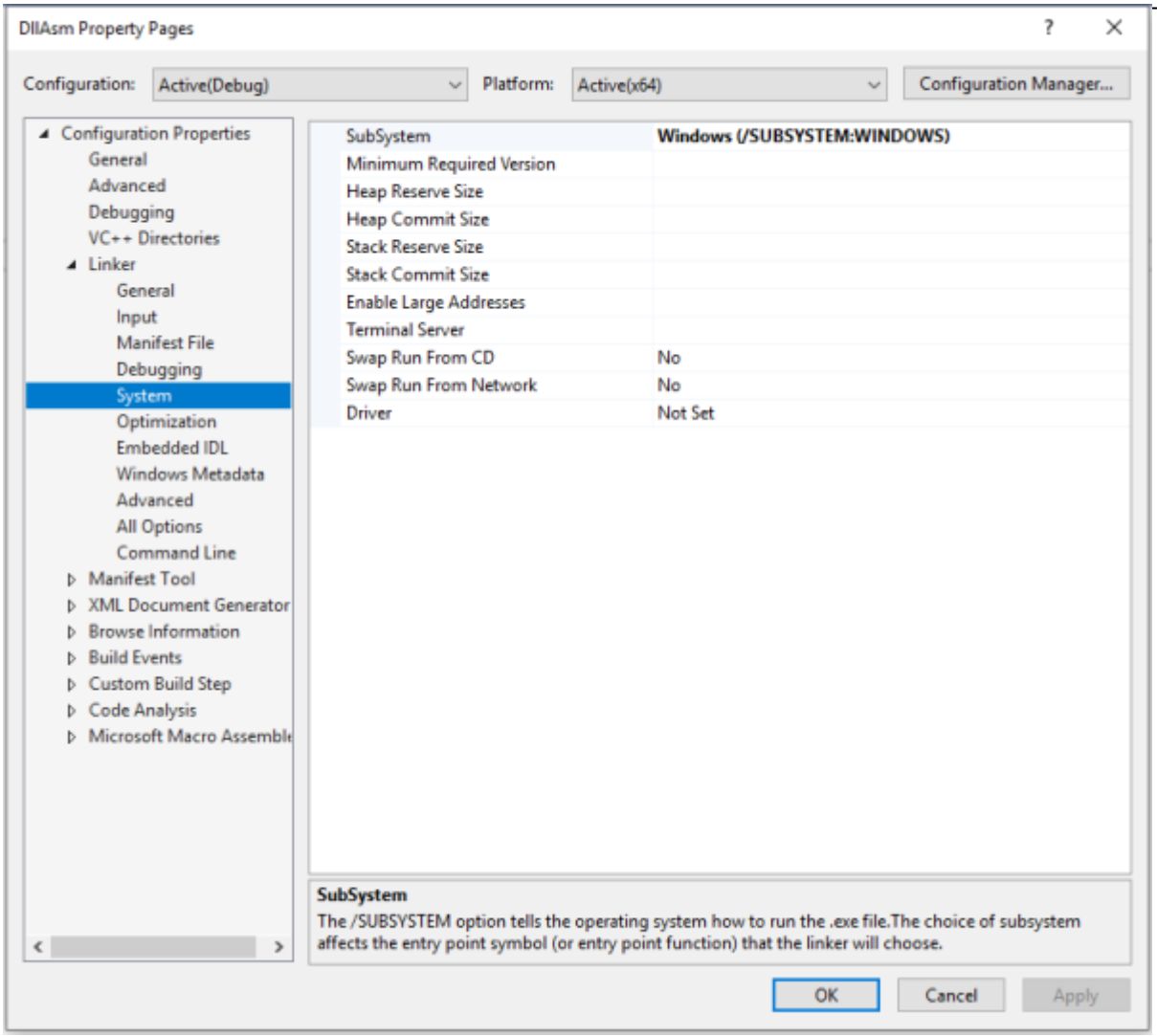


Fig. 11: Konfiguracja projektu w assemblerze cz. 3

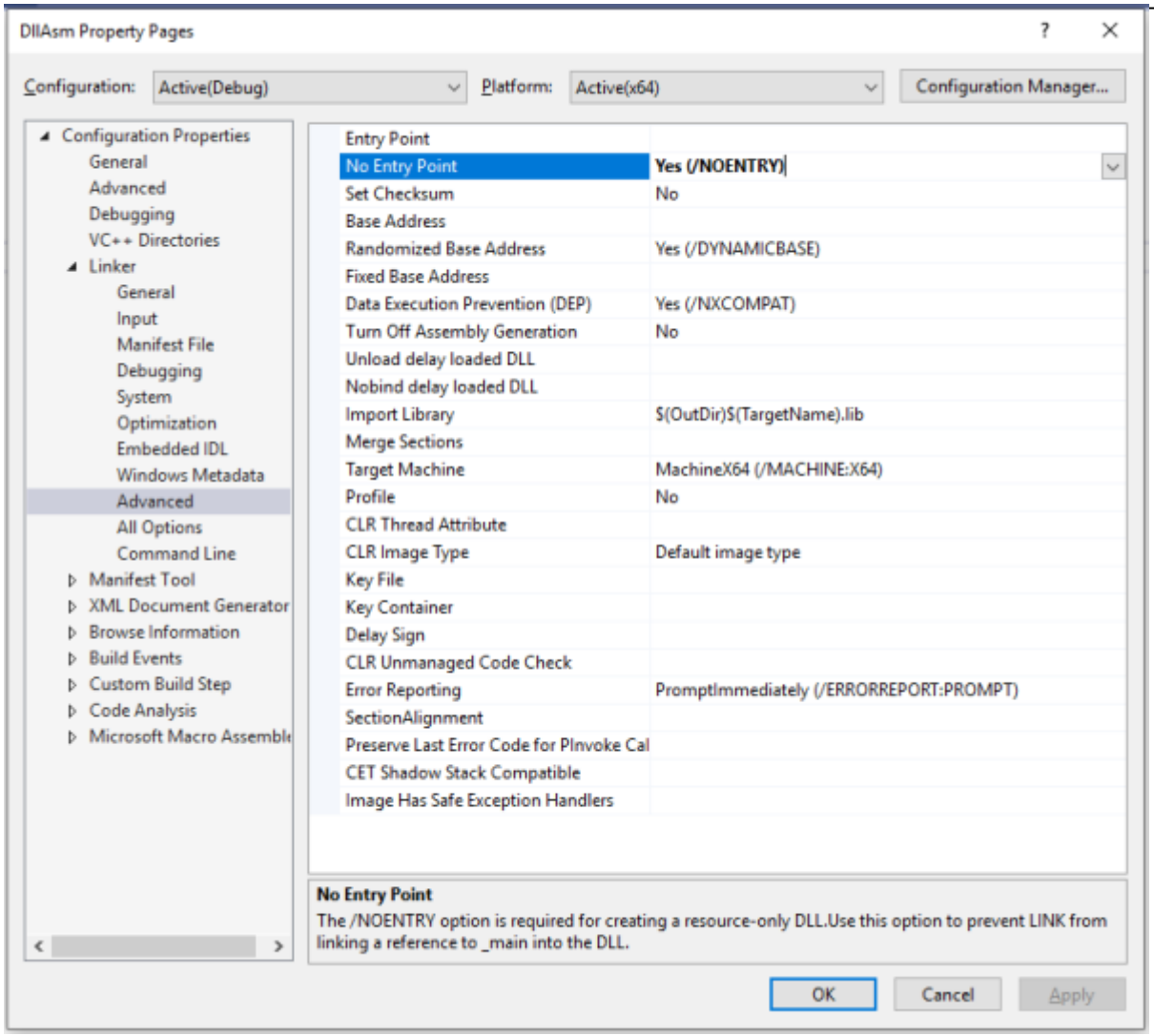


Fig. 12: Konfiguracja projektu w assemblerze cz. 4

1.6.3. Ustawienie wspólnego folderu docelowego

Aby można było ładować dynamicznie bibliotekę DLL w aplikacji C# bez konieczności podawania ścieżki, wynik kompilacji projektu DLL oraz projektu w C# musi znajdować się w tym samym folderze. W praktyce oznacza to, że w jednym z projektów należy zmienić konfigurację.
W projekcie DLL w języku assembler, domyślną lokalizacją wyniku kompilacji (m.in. wynikowego pliku `.dll` jest `$(SolutionDir)$(Platform)\$(Configuration)\`, co w praktyce oznacza, że na poziomie solucji zostanie utworzony folder `./x64/Debug/` lub `./x64/Release` (w zależności od trybu kompilacji), w którym znajdzie się skompilowana biblioteka DLL. Do tego folderu należy kompilować również główną aplikację konsolową (rysunek 13):

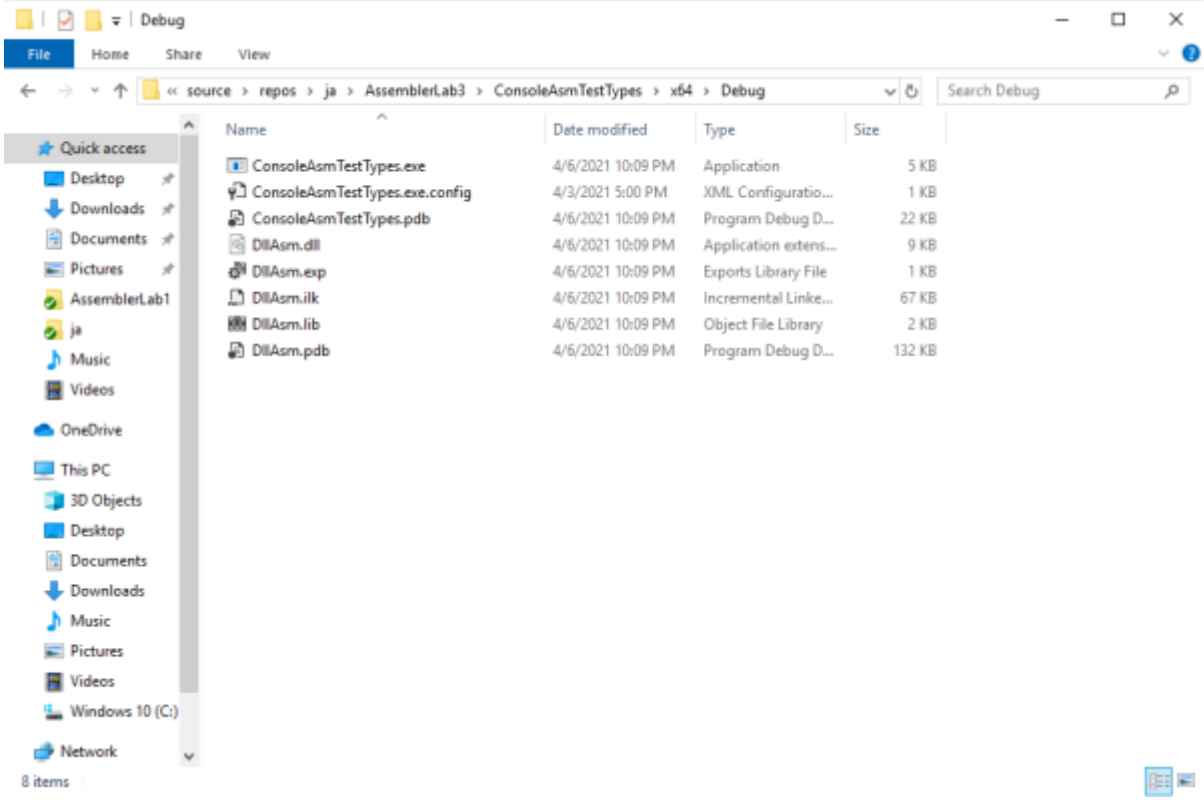
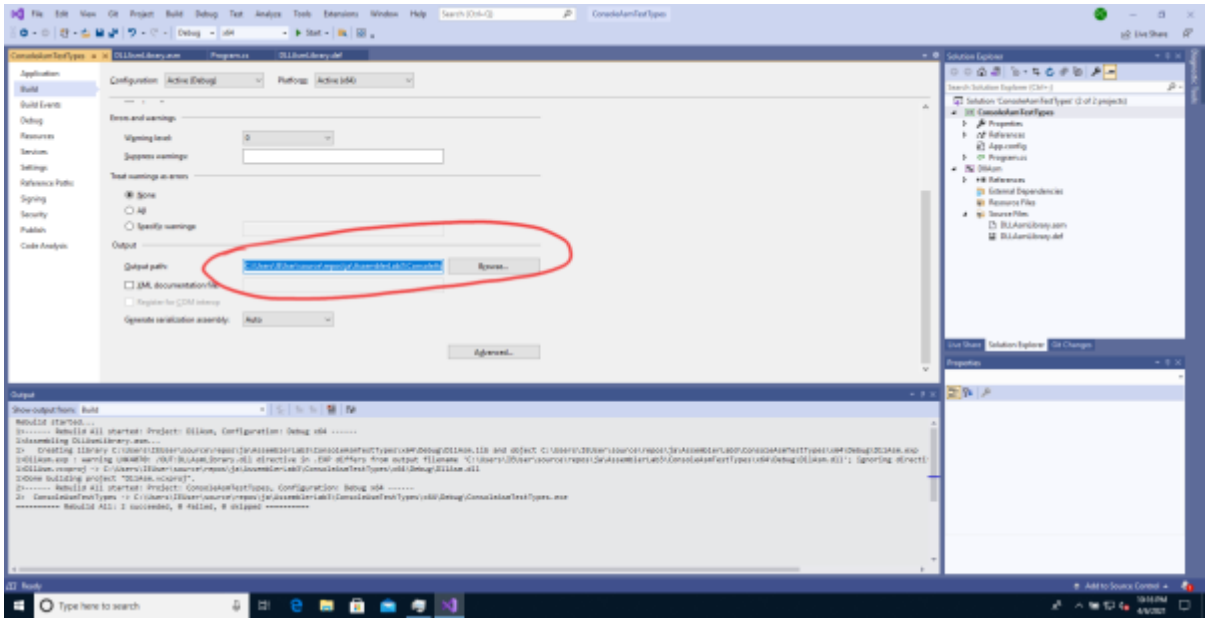


Fig. 13: Wspólny folder aplikacji konsolowej i bibloteki DLL

Aby uzyskać takie zachowanie, należy ustawić folder docelowy kompilacji w projekcie aplikacji w języku C#. We właściwościach projektu w języku C# należy wybrać opcję “Build” i w sekcji “Output” w pozycji “Output path” należy wskazać docelowy folder (rysunek 14):


Fig. 14: Konfiguracja folderu docelowego w aplikacji C# .NET


Przed wykonaniem pierwszej kompilacji folder nie zostanie założony, zatem nie będzie można go wybrać za pomoca przycisku [Browse...]. W związku z tym należy wykonać pustą kompilację wyłącznie projektu DLL w celu utworzenia folderu wynikowego.


Powyższa konfiguracja zapisuje na stałe ścieżkę w konfiguracji projektu. Oznacza to, że

zmiana z trybu Debug na Release wymaga zmiany ww. ścieżki!

W przypadku, gdyby pomimo ustawienia wspólnego folderu wynikowego nadal były zgłaszane błędy braku biblioteki DLL, istnieje możliwość, że włączony jest system tworzenia dodatkowych podfolderów, jak to opisano poniżej w notce z uwagami. Aby usunąć ten problem, konieczne jest ręczna edycja pliku konfiguracyjnego projektu.
Aby otworzyć plik konfiguracyjny projektu, należy dwukrotnie klinąć nazwe projektu w drzewie. Otworzy się plik typu XML, z rozszerzeniem `.csproj`.

Uwaga! W przypadku projektu WPF w folderze wynikowym może pojawić się podfolder z wersją frameworku .NET, co w praktyce spowoduje, że plik wykonywalny z frontside (GUI) zostanie uruchomiony w innym folderze (w podfolderze) w stosunku do wybranego w ustawieniach, skutkiem czego biblioteka DLL nie zostanie załadowana i zostanie zgłoszony wyjątek. Podfoldery te są dodawane automatycznie. Aby usunąć tę niedogodność, w pliku projektu WPF należy mieścić dodatkową sekcję[5]:

```
<PropertyGroup>
<AppendTargetFrameworkToOutputPath>false</AppendTargetFrameworkT
oOutputPath>
<AppendRuntimeIdentifierToOutputPath>false</AppendRuntimeIdentif
ierToOutputPath>
</PropertyGroup>
```

1.6.4. Konfiguracja na potrzeby kodu unsafe oraz debuggowania

Do ustawienia pozostają jeszcze dwie opcje:

- Włączenie użycia kodu “unsafe”, o ile dyrektywa “unsafe” jest wymagana. Dyrektywy tej nie trzeba używać m.in. w przypadku wykorzystywania struktury `IntPtr` do przekazywania wskaźników oraz w przypadku przekazywania argumentów przez wartość, niemniej jednak zalecane jest jej włączenie. W tym celu we właściwościach projektu w języku C# należy w sekcji “Build” w grupie“General” zaznaczyć opcję “Allow unsafe code” (rysunek 15);
- Włączenie debuggowania kodu natywnego, co pozwoli na płynne przełączanie pomiędzy kodem wywołującym w C# i wywoływanym w assemblerze w jednej sesji debuggera. W tym celu we właściwościach projektu w jezyku C# należy wybrać sekcję “Debug”, a następnie w grupie opcji “Debugger engines” zaznaczyć opcję “Enable natie code debugging” (rysunek 16);

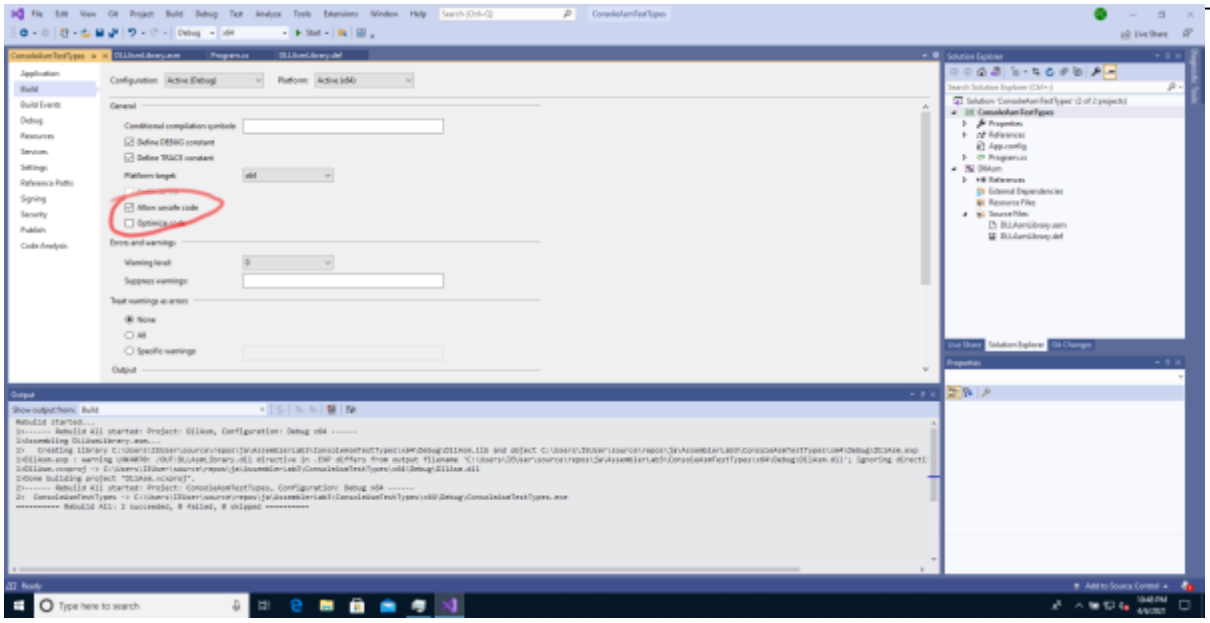


Fig. 15: Włączenie kodu unsafe

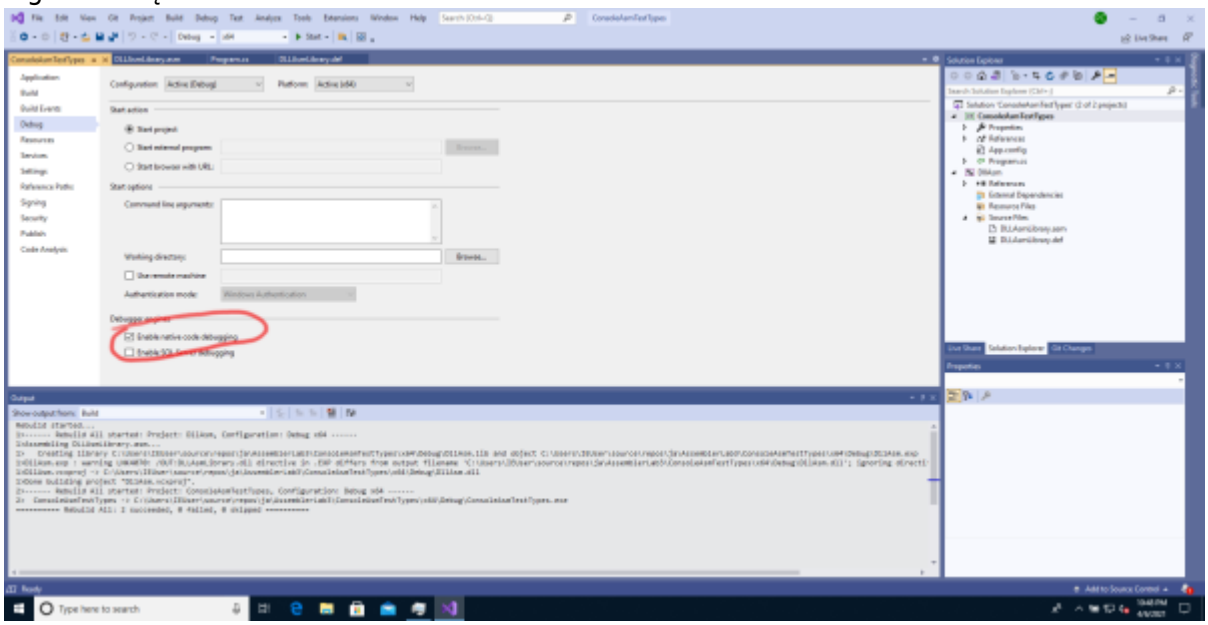


Fig. 16: Włączenie debuggowania kodu niezarządzalnego

1.7. Przekazywanie wskaźników z poziomu C# do funkcji w assemblerze

Przekazywanie typów złożonych (tablic, obiektów, struktur) jest możliwe wyłącznie za pomocą wskaźników. Wskaźniki te są przekazywane jako typy int, 64-bitowe, w rejestrach. Ich interpretacja po stronie funkcji wywoływanej należy do programisty. Nieniej jednak po stronie języka C# nie jest to tak oczywiste.

1.7.1. Przykład 1

Poniższy przykład prezentuje sposób uzyskania wskaźnika do tablicy. Deklaracja importowanej funkcji musi zostać uzupełniona o dyrektywę `unsafe`:

```
[DllImport("DLLAsm.dll")]
        private static unsafe extern int ProcAsm2(int* a, int pos);
```

Tablica wartości `int` (32-bitowe) została zadeklarowana po stronie kodu w .NET C# następująco:

```
int[] n1Array = { 1, 2, 3, 4, 5, 6 };
```

Wywołanie funkcji w bezpieczny sposób (z poinformowaniem Garbage Collectora o zablokowaniu relokacji / usuwania zmiennej z pamięci, stosując `fixed`):

```
unsafe
 {
   fixed(int* aAddress = &n1Array[0])
   {
      c = ProcAsm2(aAddress, 4);
   }
 }
```

W powyższym przykładzie pominięto przekazywanie do funkcji długości tablicy, gdyż jej rozmiar (6 elementów) jest z góry znany. W przypadku, gdy rozmiar tablicy jest zmienny, należy przekazać do funkcji w assemblerze jej długość jako osobny parametr.



Uwaga! Tablice po stronie kodu w C# są alokowane liniowo. Ponieważ jednak w C# nie da się w prosty sposób otrzymać wskaźnika do obiektu, który nie jest typem prostym (a tablica nim nie jest), używamy pewnego obejścia polegającego na tym, że pobieramy adres do pierwszego elementu tablicy, zamiast do całego obiektu tablicy. Ostatecznie, w przypadku tablicy typów prostych, adresy będą takie same. Elementy pojedynczej tablicy są alokowane bez wyrównania do 64 bitów. Dopiero po ostatnim elemencie może pojawić się wyrównanie (o ile będzie konieczne). Wyrównanie widać w kolejnym przykładzie, gdzie korzystamy ze struktury.

Przykładowy kod w assemblerze, który zwraca wskazany element tablcy, może wyglądać następująco:

```
ProcAsm2 proc
xor rax,rax
mov rax, rdx ;numer elementu tablicy do rax, potrzebujemy tylko ah (krótka
tablica)
mov dl,4 ; mnożnik, int to 4 bajty = 32 bity
mul dl ; rozmiar int to 4 bajty

mov eax, [rcx+rax]

ret
ProcAsm2 endp
```

1.7.2. Przykład 2

Przekazywanie złożonych struktur danych do assemblera wymaga zastosowania struktury przechowującej dane, aby uniknąć konieczności alokowania zmiennych na stosie. W szczególności wszystkie potrzebne dane można “spakować” po stronie C# i “rozpakować” po stronie assemblera, przekazując de-facto jedynie jeden wskaźnik na strukturę. Należy przy tym pamiętać, że jeśli

wewnątrz struktury są wskaźniki do tablic i innych złożonych typów danych, to i one, każdy element z osobna musi być obłożony klauzulą `fixed` tak, aby Garbage Collector nie zmodyfikował podrzędnych danych.

Na potrzeby poniższego przykładu przyjmujemy, że chcemy przekazać do assemblera trzy tablice typu `byte` o różnich długościach.



W przypadku, gdyby liczba parametrów, które trzeba przekazać do funkcji przekraczała 4, aby nie musieć umieszczać ich na stosie, można po stronie C# zadeklarować strukturę zawierającą argumenty funkcji, a następnie przekazać do funkcji w assemblerze wskaźnik na tę strukturę, zamiast każdy z argumentów osobno. Po stronie assemblera należy wtedy tę strukturę "rozkodować"

Deklaracja struktury po stronie języka C# może wyglądać następująco (ważna jest kolejność zmiennych, bo w takiej kolejności będą się one potem pojawiały w pamięci):

```
public unsafe struct StructToAssembler //Deklaracja typu struktury. Uwaga - to nie jest zmienna, tylko typ
    {
        public byte* ptrA1;
        public int lenA1;
        public byte* ptrA2;
        public int lenA2;
        public byte* ptrA3;
        public int lenA3;
    }
```

Znaczenie poszczególnych elementów:

- `ptrA1` to wskaźnik na początek pierwszej tablicy bajtów;
- `lenA1` to długość ww tablicy (w elementach);
- analogiczne dla **…**A2 i **…**A3;



Uwaga. Struktura MUSI być oznaczona jako `unsafe`, ponieważ zawiera typy wskaźnikowe `byte*`



Dane wewnątrz struktury (zmienne) są "wyrównanie" do 8 bajtów (64 bitów). A zatem każdy z ww elementów składowych struktury zajmuje w pamięci 8 bajtów nawet, jeśli długość zmiennej na to nie wskazuje. Przykładowo pojedynczy, `byte` (zmienna) zajmuje w pamięci również 8 bajtów, z czego znaczący jest tylko pierwszy, najmłodszy. Oczywiście w przypadku pojedynczej zmiennej typu tablicowego, dane są uszeregowane w ciągu, bez odstępu, zatem przykładowo `byte[8]` zajmuje 8 bajtów pod rząd, bez żadnego odstępu i wyrównywania.

Na potrzeby przykładu zostały tez utworzone 3 tablice bajtów oraz instancja ww. struktury typu `StructToAssembler`. Deklaracja ich jako statycznych wynika z faktu iż dane zostały zadeklarowane wewnątrz klasy statycznej `Program` odpowiadającej za przykładowy kod aplikacji typu CommandLine

w języku C# (w innym kontekście, tablica bajtów może być np. skonwertowaną listą wartości RGB z bitmapy, danymi numerycznymi ładowanymi z pliku lub dowolna, inna forma, w tym może być zadeklarowana dynamicznie (vide a3):

```
public static byte[] a1 = { 2, 4, 6, 8, 10, 12, 14, 16 };
public static byte[] a2 = { 3, 6, 9, 12, 15, 18 };
public static byte[] a3 = new byte[10]; //domyślnie zainicjalizowana zerami

public static StructToAssembler struktura = new StructToAssembler();
//instancja struktury
```

Prototyp funkcji importowanej z języka Assembler będzie zawierał wyłącznie jeden argument, który jest wskaźnikiem na instancję struktury przechowującej argumenty:

```
[DllImport("DLLAsm.dll")]
private static unsafe extern int ProcAsm3(StructToAssembler* structurePtr);
```

Bezpieczne przekazanie zmiennych (z zabezpieczeniem przed usunięciem / relokacją) wymaga użycia `fixed`. Uwaga - klauzule `fixed` mogą być zagnieżdżone, ponadto w ramach jednej klauzuli można "zablokować" więcej niż jedną zmienną, o ile są tego samego typu (występują tu drobne różnice w wersji frameworku .NET). Należy jednak pamiętać, że "zablokowanie" wskaźnika na strukturę nie oznacza automatycznie objęcia "blokadą" wszystkich zmiennych-członków tworzących strukturę, w szczególności w naszym przypadku, wskaźników na tablice `ptrA1`, `ptrA2` i `ptrA3`. W związku z tym w poniższym kodzie umieszczono dwie zagnieżdżone klauzule `fixed` (i oczywiście `unsafe`). Inicjalizacja elementów składowych struktury, tj. wyznaczenie wskaźników do tablic z danymi i ich zeskładowanie w strukturze oraz wyliczenie długości tablic jest dokonywane wewnątrz klauzul `fixed`, np. `byte* ptr = a1` oznacza wyznaczenie stałego, niemutowalnego za pomocą Garbage Collectora wskaźnika do tablicy `a1`. Wartość ta zostaje przypisana do struktury przed przekazaniem jej do funkcji: `struktura.ptrA1 = ptr1;`. Dodatkowo wyznaczona zostaje długość tablicy: `struktura.lenA1 = a1.Length;`. Całość przedstawiono na poniższym przykładzie. Oczywiście wskaźnik do struktury również musi być objęty klauzulą `fixed`, niezależnie od składowych struktury:

```
unsafe
{
   fixed (StructToAssembler* aAddress = &struktura)
   {
      fixed (byte* ptr1 = a1, ptr2 = a2, ptr3 = a3)
      {
         struktura.ptrA1 = ptr1;
         struktura.ptrA2 = ptr2;
         struktura.ptrA3 = ptr3;
         struktura.lenA1 = a1.Length;
         struktura.lenA2 = a2.Length;
         struktura.lenA3 = a3.Length;
         c = ProcAsm3(aAddress); //Wywołaj funkcję
       };
    }
}
```

Po stronie języka assembler można wykorzystać następujący kawałek kodu w celu "zdekodowania" struktury (kod jest nadmiarowy, niemniej jednak pozwala na debuggowanie krok po kroku i

podglądanie wartości w debuggerze):

```
.data
ptr1 dq  0
len1 dd  0
ptr2 dq  0
len2 dd  0
ptr3 dq  0
len3 dd  0

...

.code
ProcAsm3 proc
mov rax,rcx
mov rbx, [rcx] ;adres dla pierwszego wskaznika w strukturze (wskaźnik na a1)
mov ptr1, rbx
mov ebx, [rcx+8] ;dlugosc a1
mov len1, ebx
mov rbx, [rcx+16] ;adres dla drugiego wskaznika w strukturze (wskaźnik na
a2)
mov ptr2, rbx
mov ebx, [rcx+24] ;dlugosc a2
mov len2, ebx
mov rbx, [rcx+32] ;adres dla drugiego wskaznika w strukturze (wskaźnik na
a3)
mov ptr3, rbx
mov ebx, [rcx+48] ;dlugosc a3
mov len3, ebx
ret
ProcAsm3 endp
```

Na poniższym rysunku przedstawiono postać zmiennej (struktury) `struktura` typu `StructToAssembler` umieszczonej w pamięci po stronie języka C#, a widzianej po stronie kodu niezarządzalengo w assemblerze (rysunek 17).

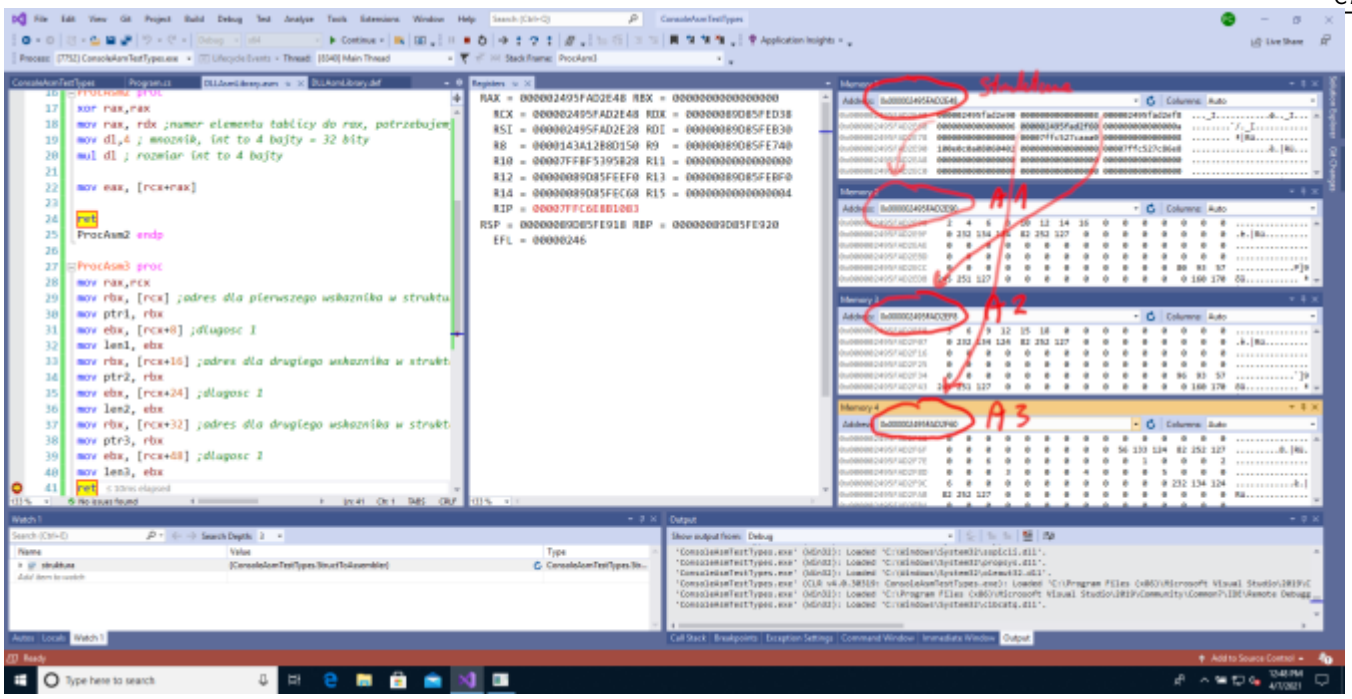


Fig. 17: Ułożenie zmiennej `struktura` i jej składowych w pamięci operacyjnej

Wskaźnik został przekazany przez rejestr `rcx`.
W okienku [Memory 1] dane prezentowane są w postaci liczb 64-bitowych (wskaźniki oraz `int` wyrównany do 64 bitów), odpowiedno w kolejności zgodnej z deklaracją typu `StructToAssembler`, a zatem jest to w kolejności:

- wskaźnik na tablicę `a1` (adres: 0x000002495fad2e90), czyli wartość przypisana za pomocą `struktura.ptrA1 = ptr1;`;
- rozmiar tablicy `a1` (8 elementów), czyli wartość pola `struktura.lenA1`;
- wskaźnik na tablicę `a2` (adres: 0x000002495fad2ef8), czyli wartość przypisana za pomocą `struktura.ptrA2 = ptr2;`;
- rozmiar tablicy `a2` (6 elementów), czyli wartość pola `struktura.lenA2`;
- wskaźnik na tablicę `a3` (adres: 0x000002495fad2f60), czyli wartość przypisana za pomocą `struktura.ptrA3 = ptr3;`;
- rozmiar tablicy `a3` (10 elementów, zapisane szesnastkowo jako '0x0a'), czyli wartość pola `struktura.lenA3`;

Uwaga - przy każdym uruchomieniu aplikacji te adresy będą (mogą być) inne, ponieważ pamięć jest alokowana dynamicznie. Natomiast klauzula `fixed` powoduje, że podczas wywołania procedury `ProcAsm3(aAddress);` wartości te nie ulegną zmianie.

1.7.2.1. Podgląd pamięci

Na rysunku 17 w okienku [Memory 2] widać zawartość pamięci odpowiadającej zmiennej `a1`, prezentowaną w postaci liczb dziesiętnych bez znaku, o długości bajtowej, zgodnie z tym, co zostało zadeklarowane po stronie języka C# (jest to w kolejności 2,4,6,8,10,12,14,16). Analogicznie [Memory 3] dla zmiennej `a2` i [Memory 4] dla zmiennej `a3`.

1.8. Zadanie do wykonania

W ramach laboratorium studenci zrealizują prostą aplikację konsolową, działającą w środowisku .NET w języku C#, która będzie wywoływała funkcje obliczeniowe, utworzone w kodzie niezarządzalnym, w języku Assembler, w postaci biblioteki dynamicznie dołączanej (DLL). Zmienne będą zadeklarowane po stronie .NET.

W szczególności, do wykonania są następujące zadania:

1. Zsumowanie wektora danych typu uint (zadeklarowanych po stronie ,NET w postaci tablicy jednowymiarowej o wymiarze 20 zmiennych). Funkcja ma zwracać pojedynczą wartość (przez rejestr). Wskaźnik na tablicę źródłową oraz jej długość należy podać do funkcji w assemblerze poprzez rejestry.
2. Obliczenie wartości średnich dla zestawu zmiennych przechowywanych w postaci dwóch wektorów o długości 16, typu float, pozycja po pozycji, wynikiem ma być wektor średnich (o długości 16). Funkcja nie zwraca żadnej wartości. Wyniki są umieszczane w tablicy zadeklarowanej po stronie ,NET i przekazanej do funkcji jako wskaźnik. Sugerowane jest użycie struktury do przekazania danych do funkcji. Docelową tablicę należy zadeklarować po stronie języka C#. Zalecane jest wykonanie operacji za pomocą pętli, tj. obliczanie osobno dla każdej pary. Można skorzystać albo z funkcji koprocesora arytmetycznego, albo z MMX/SSE.

Powyższe operacje są od siebie niezależne, a zatem powinny być zrealizowane w ramach dwóch, niezależnych funkcji assemblerowych.

1.9. Metoda weryfikacji wyników

Kalkulacje obliczone ręcznie powinny pokrywać się z obliczeniami zaimplementowanymi w algorytmie.

1.10. Pomocne zasoby

Visual Studio 2019: https://visualstudio.microsoft.com/vs/

1)
https://visualstudio.microsoft.com/vs/
2)
https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/
3)
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/unsafe-code-pointers/
4)
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/fixed-statement
5)
https://docs.microsoft.com/en-us/visualstudio/ide/how-to-change-the-build-output-directory?view=vs-2019

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

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/pl:edu:zmitac:n1z:ja:lab3?rev=1619276901

Last update: **2021/04/24 17:08**