

Laboratorium Architektury Komputerów i Systemów Operacyjnych

Ćwiczenie 2

Programowanie mieszane

Wprowadzenie

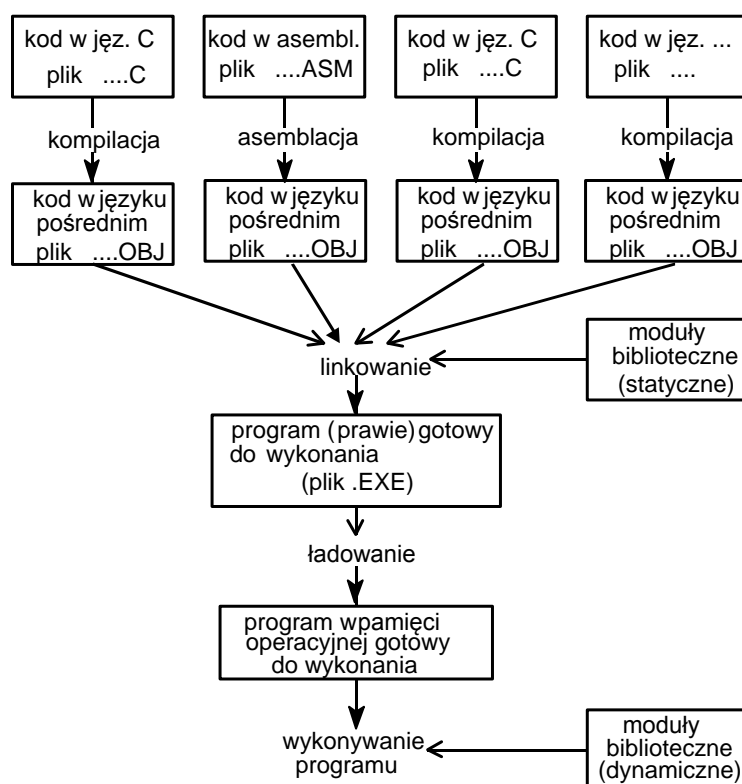
W przypadku tworzenia oprogramowania współpracującego z różnymi urządzeniami dołączonymi do komputera, zadaniem jednego z modułów funkcjonalnych oprogramowania jest organizowanie współpracy z tymi urządzeniami. W wielu przypadkach, ze względu na specyficzne wymagania urządzenia, taki moduł musi być kodowany w asemblerze, niekiedy przez konstruktora urządzenia. W rozpatrywanej sytuacji kod asemblerowy musi być przystosowany do współdziałania z pozostałym oprogramowaniem, kodowanym zazwyczaj w języku wysokiego poziomu (np. C/C++).

Niniejsze opracowanie przybliża zagadnienia związane z współdziałaniem kodu napisanego w asemblerze z kodem w języku C (i po pewnych rozszerzeniach C++), w środowisku 32-bitowym systemu Windows. Bardzo podobne, lub identyczne mechanizmy stosowane są w innych systemach operacyjnych.

Kompilacja, linkowanie i ładowanie

W wielu środowiskach programowania wytworzenie programu wynikowego wykonywane jest w dwóch etapach. Najpierw kod źródłowy każdego modułu programu zostaje poddany *kompilacji* (jeśli moduł napisany jest w języku wysokiego poziomu) lub *asemblacji* (jeśli moduł napisany jest w asemblerze). W obu tych przypadkach uzyskuje się plik w języku pośrednim (rozszerzenie `.OBJ`). Następnie uzyskane pliki `.OBJ` poddaje się konsolidacji czyli *linkowaniu*. W trakcie linkowania dołączane są także wszystkie niezbędne programy biblioteczne. W rezultacie zostaje wygenerowany plik zawierający program wynikowy z rozszerzeniem `.EXE`. Plik ten zawiera kod programu w języku maszynowym (czyli zrozumiałym przez procesor), aczkolwiek niektóre jego elementy wymagają korekcji uzależnionej od środowiska, w którym program będzie wykonany. Korekcja ta następuje w trakcie *ładowania* programu.

Niektóre programy biblioteczne mają charakter uniwersalny i są wykorzystywane przez wiele programów użytkowych. Wygodniej byłoby więc dołączać te programy dopiero w trakcie wykonywania programu, co pozwoliłoby na zmniejszenie rozmiaru pliku `.EXE`. W takim przypadku mówimy, że program korzysta z biblioteki dynamicznej (zapisanej w pliku z rozszerzeniem `DLL`). Omawiane fazy translacji pokazane są na poniższym rysunku.



Pliki .OBJ generowane przez różne kompilatory (w danym środowisku) zawierają kod w tym samym języku, który możemy uważać za język pośredni, stanowiący jak gdyby "wspólny mianownik" dla różnych języków programowania.

Podprogramy w technice programowania mieszanego

Problem tworzenia programu, którego fragmenty napisane są w różnych językach programowania wymaga m.in. ustalenia sposobu komunikowania się poszczególnych fragmentów ze sobą. Komunikacja taka staje się stosunkowo łatwa do zrealizowania, jeśli poszczególne fragmenty programu mają postać podprogramów (procedur). Podprogramy stanowią, ze swej natury, w pewien sposób wyizolowaną część programu, a komunikacja z nimi odbywa się wg ściśle ustalonego protokołu, określającego formaty danych i wzajemne obowiązki programu wywołującego i wywoływanego podprogramu. W ten sposób, w trakcie wykonywania programu, wywoływanie fragmentów napisanych w różnych językach programowania, sprowadza się do wywoływania odpowiednich podprogramów. W przypadku języka C wywołanie podprogramu oznacza po prostu wywołanie funkcji języka C, której kod został zdefiniowany w innym pliku, niekoniecznie napisanym w języku C.

Powyższe rozważania wskazują, że interfejs do podprogramów musi być jasno i przejrzysto zdefiniowany, a zarazem musi być na tyle uniwersalny, by mógł być implementowany przez kompilatory różnych języków programowania. Z tego powodu producenci oprogramowania (m.in. firma Microsoft) ustalają pewne niskopoziomowe protokoły wywoływania podprogramów, przeznaczone dla wytwarzanych przez nich kompilatorów języków programowania. Protokoły te stanowią fragment interfejsu oznaczonego symbolem ABI (ang. Application Binary Interface).

W szczególności ABI definiuje *standard wywoływania* (ang. calling convention), który określa sposób wywoływania funkcji, przekazywania jej argumentów, przejmowania

obliczonej wartości, podaje wykaz rejestrów procesora, których zawartości powinny być zachowane, itp. M.in. standard ABI określa czy parametry przekazywane są przez rejestry czy przez stos, a jeśli przez stos to w jakiej kolejności są ładowane, czy dopuszcza się zmienną liczbę argumentów, itd.

Obecnie, w oprogramowaniu komputerów osobistych rodziny PC, wyłoniły się trzy typy interfejsu procedur. Jeden z nich, używany jest przez translator języka C, drugi przez translator Pascala, a trzeci *StdCall* stanowi połączenie dwóch poprzednich. W dalszej części podamy więcej szczegółów na ten temat.

Organizacja stosu

Stos jest strukturą danych, która stanowi odpowiednik, np. stosu książek. Kolejne wartości zapisywane na stos ładowane są zawsze na jego wierzchołek. Również wartości odczytywane są zawsze z wierzchołka stosu, przy czym odczytanie wartości należy rozumieć jako usunięcie jej ze stosu. W literaturze technicznej tak zorganizowana struktura danych nazywana jest kolejką LIFO, co stanowi skrót od ang. "Last In, First Out". Oznacza to, że obiekt który wszedł jako ostatni, jako pierwszy zostanie usunięty.

W typowych komputerach stos umieszczony jest w pamięci operacyjnej komputera. W operacjach wykonywanych na stosie istotną rolę odgrywa rejestr nazywany *wskaźnikiem stosu*. Zawartość wskaźnika stosu określa położenie ostatnio zapisanej danej czyli położenie wierzchołka stosu. W komputerach z procesorem zgodnym z architekturą x86 wskaźnik stosu jest rejestrem 32-bitowym i jest oznaczony symbolem ESP. Zdefiniowano dwa podstawowe rozkazy wykonujące operacje na stosie:

PUSH — zapisanie danej na stosie

POP — odczytanie danej ze stosu.

W trybie 32-bitowym na stosie zapisywane są zawsze wartości 32-bitowe, czyli 4-bajtowe. Wskaźnik stosu ESP wskazuje zawsze położenie najmłodszego bajtu spośród czterech tworzących zapisaną wartość.

Rozkaz **PUSH** przed zapisaniem danej na stosie powoduje zmniejszenie rejestru ESP o 4, natomiast rozkaz **POP** po odczytaniu danej zwiększa rejestr ESP o 4. Oznacza to, że stos rośnie w kierunku malejących adresów, czyli każda kolejna wartość zapisywana na stosie umieszczana w komórkach pamięci o coraz niższych adresach.

Stos używany jest często do przechowywania zawartości rejestrów, np. rozkazy

```
push esi
push edi
```

powodują zapisanie na stos kolejno zawartości rejestrów ESI i EDI. W dalszej części programu można odtworzyć oryginalne zawartości rejestrów poprzez odczytanie ich ze stosu

```
pop edi
pop esi
```

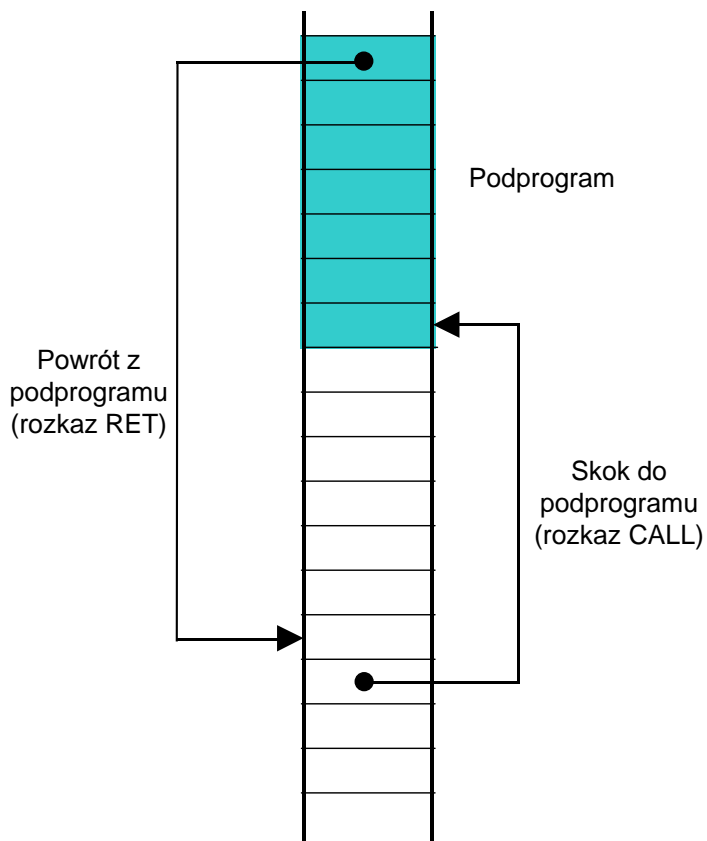
Wprawdzie rozkazy **PUSH** i **POP** stanowią dwa podstawowe rozkazy wykonujące działania na stosie, to jednak w istocie stos jest fragmentem pamięci głównej (operacyjnej) i wobec tego wartości zapisane na stosie mogą być także odczytywane za pomocą zwykłego rozkazu przesłania **MOV** — trzeba jednak znać adres potrzebnej danej. Obliczenie adresu nie jest trudne, ponieważ w każdej chwili rejestr ESP zawiera adres danej znajdującej się na wierzchołku stosu. Znając odległość potrzebnej danej od wierzchołka stosu można łatwo obliczyć jej adres i zastosować rozkaz **MOV**. W ten sposób można odczytywać dane znajdujące się wewnątrz stosu, a nie tylko na jego wierzchołku. Technika ta jest powszechnie

wykorzystywana przez kompilatory języków programowania wysokiego poziomu (przykład podany jest w dalszej części).

Stos pełni także ważną rolę w trakcie wywoływania podprogramów. Na poziomie rozkazów procesora wywołanie podprogramu (procedury, funkcji) polega po prostu na wykonaniu skoku bezwarunkowego do pierwszego rozkazu podprogramu (zob. rysunek). Dodatkowo, do podprogramu przekazywany jest także adres komórki pamięci, w której zawarty jest kolejny rozkaz programu głównego (znajdujący się w pamięci bezpośrednio za rozkazem, który wywołał podprogram). Omawiany adres, nazywany *śladem*, określa więc położenie rozkazu w pamięci, który będzie wykonany bezpośrednio po zakończeniu wykonywania podprogramu.

Do wywoływania podprogramów używany jest rozkaz CALL, który wykonuje skok do podprogramu i jednocześnie zapisuje na stosie adres powrotu, czyli *ślad*. W podanych dalej przykładach rozkaz CALL nie występuje w postaci jawnej — jest on automatycznie generowany przez kompilator języka C w trakcie tłumaczenia wiersza, w którym znajduje się instrukcja wywołania funkcji (podprogramu).

Z kolei na końcu podprogramu znajduje się rozkaz RET (ang. return, powrót), który zamyka działanie podprogramu i przekazuje sterowanie do programu głównego. W istocie rozkaz RET odczytuje z wierzchołka stosu liczbę stanowiącą ślad i wpisuje ją do wskaźnika instrukcji EIP.



Tryby adresowania

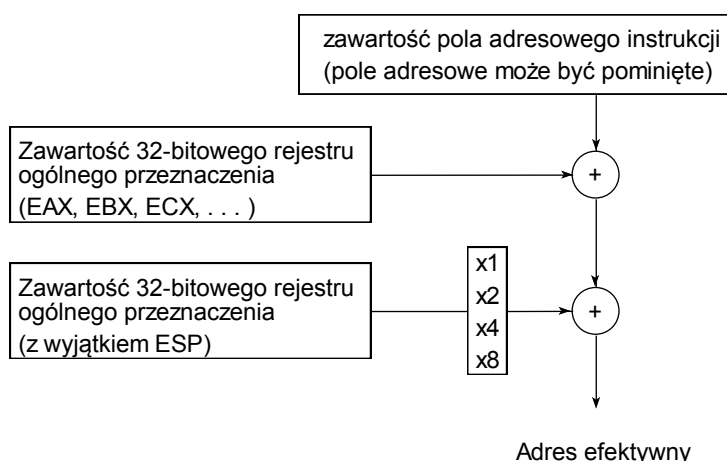
W wielu problemach informatycznych mamy do czynienia ze zbiorami danych w formie różnego rodzaju tablic, które można przeglądać, odczytywać, zapisywać, sortować itd. Na poziomie rozkazów procesora występują powtarzające się operacje, w których za każdym razem zmienia się tylko indeks odczytywanego lub zapisywanego elementu tablicy. Takie powtarzające się operacje koduje się w postaci pętli. W przypadku operacji na elementach tablicy muszą być dostępne mechanizmy pozwalające na dostęp do kolejnych elementów tablicy w trakcie kolejnych obiegów pętli. Ten właśnie problem rozwiązywany jest poprzez stosowanie odpowiedniego *trybu adresowania*.

Współczesne procesory udostępniają wiele trybów adresowania, dostosowanych do różnych problemów programistycznych. Między innymi pewne tryby adresowania zostały

opracowane specjalnie dla odczytywania wielobajtowych liczb, inne wspomagają przekazywanie parametrów przy wywoływaniu procedur i funkcji.

Znaczna część rozkazów procesora wykonujących operacje arytmetyczne i logiczne jest dwuargumentowa, co oznacza, że w kodzie w rozkazie podane są informacje o położeniu dwóch argumentów, np. odjemnej i odjemnika w przypadku odejmowania. Wynik operacji przesyłany jest zazwyczaj w miejsce pierwszego argumentu. Prawie zawsze jeden z argumentów znajduje się w jednym z rejestrów procesora, a drugi argument znajduje się w komórce pamięci, albo także w rejestrze procesora.

Omawiane tu tryby adresowania dotyczą przypadku, gdy jeden z argumentów operacji znajduje się w komórce pamięci. W szczególności tryb adresowania *bazowo-indeksowego* powoduje, że adres danej, na której ma być wykonana operacja, obliczany jest jako suma zawartości pola adresowego rozkazu (instrukcji) i zawartości jednego lub dwóch rejestrów 32-bitowych. Algorytm wyznaczania adresu ilustruje poniższy rysunek.



Przykładowo, jeśli chcemy obliczyć sumę elementów tablicy składającej się z liczb 16-bitowych (czyli dwubajtowych), to zwiększając w każdym obiegu pętli zawartość rejestru indeksowego o 2 powodujemy, że kolejne wykonania tego samego rozkazu dodawania **ADD** spowodują za każdym razem dodanie kolejnego elementu tablicy. Przykład programu, w którym występuje sumowanie elementów tablicy podany jest w dalszej części niniejszego opracowania.

Dodatkowo może być stosowany tzw. współczynnik skali (x1, x2, x4, x8), co ułatwia uzyskiwanie adresu wynikowego (jest to adresowanie bazowe-indeksowe ze skalowaniem).

Wyznaczanie adresu stanowi jeden z etapów wykonywania rozkazu przez procesor — jego wynikiem jest *adres efektywny* (zob. rysunek) rozkazu, czyli adres komórki pamięci zawierającej daną, na której zostanie wykonana operacja, np. mnożenie. W omawianych dalej przykładach pole adresowe instrukcji (rozkażu) jest zazwyczaj pominięte, co oznacza, że adres efektywny określony jest wyłącznie przez zawartość podanego rejestru bazowego i/lub indeksowego. Przykładowo, jeśli rejestr bazowy (np. **EBX**) zawierać będzie liczbę 724, to procesor odczyta wartość danej z komórki o adresie 724. Podkreślamy, że procesor wykonana wymaganą operację nie na liczbie 724, ale na liczbie która zostanie odczytana z komórki pamięci o adresie 724.

W zapisie asemblerowym, symbole rejestru, w którym zawarty jest adres komórki pamięci umieszcza się w nawiasach kwadratowych. Przykładowo, rozkaz

```
mov    edx, [ebx]
```

powoduje przesłanie do rejestru EDX wartości (np. liczby całkowitej) pobranej z komórki pamięci operacyjnej o adresie znajdującym się w rejestrze EBX. Zauważmy, że w omawianym przykładzie pole adresowe rozkazu nie występuje, a adres efektywny równy jest po prostu liczbie zawartej w rejestrze EBX.

Konwencje wywoływania procedur stosowane przez kompilatory języka C w trybie 32- i 64-bitowym

1. W trybie 32-bitowym parametry podprogramu przekazywane są przez stos. Parametry ładowane są na stos w kolejności odwrotnej w stosunku do tej w jakiej podane są w kodzie źródłowym, np. wywołanie funkcji `calc(a,b)` powoduje załadowanie na stos wartości `b`, a następnie `a`.
2. W trybie 64-bitowym pierwsze cztery parametry podprogramu przekazywane są przez rejestry: RCX, RDX, R8 i R9. Dopiero piąty parametr i następne, jeśli występują, przekazywane są przez stos, przy czym pierwszy z parametrów przekazywanych przez stos musi zajmować lokację pamięci o najniższym adresie, który musi być podzielny przez 8.
3. W trybie 64-bitowym do przekazywania liczb zmiennoprzecinkowych używa się odrębnych rejestrów związanych z operacjami multimedialnymi SSE: XMM0, XMM1, XMM2, XMM3 (zamiast rejestrów RCX, RDX, R8 i R9).
4. W trybie 32-bitowym jeśli parametr ma postać pojedynczego bajtu, to na stos ładowane jest podwójne słowo (32 bity), którego najmłodszą część stanowi podany bajt.
5. Jeśli parametrem jest liczba składająca się z 8 bajtów, to najpierw na stos ładowana jest 4-bajtowa starsza część liczby, a potem młodsza część (również 4-bajtowa). Taki schemat ładowania stosowany jest w komputerach, w których liczby przechowywane są w standardzie *mniejsze niż* (ang. *little endian*) i wynika z faktu, że stos rośnie w kierunku malejących adresów.
6. Obowiązek zdjęcia parametrów ze stosu po wykonaniu podprogramu należy do programu wywołującego. Funkcje systemowe Windows stosują standard `Stdcall`, w którym parametry zapisane na stosie zdejmowane są wewnątrz wywołanej funkcji.
7. Kompilatory języka C stosują dwa typowe sposoby przekazywania parametrów: przez wartość i przez adres. Jeśli parametrem funkcji jest nazwa tablicy, to przekazywany jest adres tej tablicy; wszystkie inne obiekty, które nie zostały jawnie zadeklarowane jako tablice, przekazywane są "przez wartość".
8. Wyniki podprogramu przekazywane są przez rejestr EAX (w trybie 32-bitowym) albo przez rejestr RAX (w trybie 64-bitowym). Wyniki 8-bitowe przekazywane są przez rejestr AL, a 16-bitowe przez rejestr AX. Jeśli wynikiem podprogramu jest adres (wskaźnik), to przekazywany jest także przez rejestr EAX (lub RAX).
9. Jeśli podprogram zmienia zawartość rejestrów EBX, EBP, ESI, EDI, to powinien w początkowej części zapamiętać je na stosie i odtworzyć bezpośrednio przed zakończeniem. Pozostałe rejestry robocze mogą być używane bez konieczności zapamiętywania i odtwarzania ich zawartości. Ograniczenia dotyczące trybu 64-bitowego podano w poniższej tabeli.

	Aplikacje 32-bitowe Windows, Linux	Aplikacje 64-bitowe Windows	Aplikacje 64-bitowe Linux
Rejestry używane bez ograniczeń	EAX, ECX, EDX, ST(0) ÷ ST(7) XMM0 ÷ XMM7	RAX, RCX, RDX, R8 ÷ R11, ST(0) ÷ ST(7) XMM0 ÷ XMM5	RAX, RCX, RDX, RSI, RDI, R8 ÷ R11, ST(0) ÷ ST(7) XMM0 ÷ XMM15
Rejestry, które muszą być zapamiętywane i odtwarzane	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12 ÷ R15, XMM6 ÷ XMM15	RBX, RBP, R12 ÷ R15
Rejestry, które nie mogą być zmieniane	DS, ES, FS, GS, SS		
Rejestry używane do przekazywania parametrów	(ECX)	RCX, RDX, R8, R9, XMM0 ÷ XMM3	RDI, RSI, RDX, RCX, R8, R9, XMM0 ÷ XMM7
Rejestry używane do zwracania wartości	EAX, EDX, ST(0)	RAX, XMM0	RAX, RDX, XMM0, XMM1, ST(0), ST(1)

Podprogram w assemblerze przystosowany do wywoływania z poziomu języka C musi być skonstruowany dokładnie wg tych samych zasad co funkcje w języku C. Wynika to z faktu, że program w języku C będzie wywoływał podprogram w taki sam sposób w jaki wywołuje inne funkcje w języku C.

Wszystkie nazwy globalne zdefiniowane w treści podprogramu w assemblerze muszą być wymienione na liście dyrektywy **PUBLIC**. Jednocześnie nazwy innych używanych zmiennych globalnych i funkcji muszą być zadeklarowane na liście dyrektywy **EXTRN**.

Ze względu na konwencję nazw stosowaną przez kompilatory języka C, każdą nazwę o zasięgu globalnym wewnątrz podprogramu assemblerowego należy poprzedzić znakiem podkreślenia _ (nie dotyczy to trybu 64-bitowego a także 32-bitowej konwencji *StdCall*).

Program przykładowy (wersja 32-bitowa)

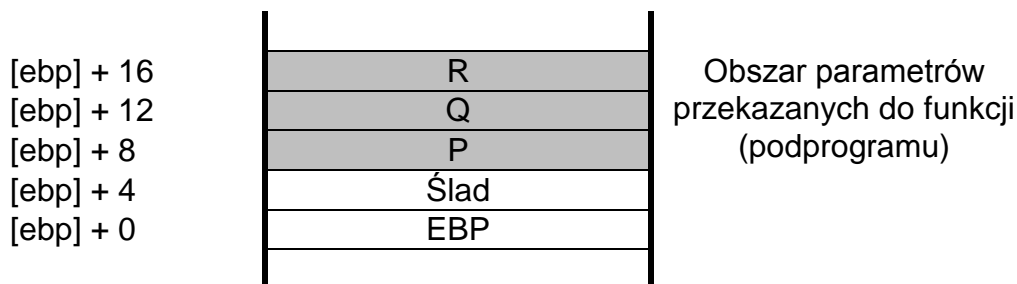
Podany niżej program (wersja 32-bitowa) składa się z dwóch modułów zawierających kod w języku C i kod w assemblerze. Program w języku C oblicza i wyświetla na ekranie sumę trzech liczb całkowitych (typu `int`), które są argumentami funkcji:

```
suma_liczb (int p, int q, int r)
```

Wartości typu `int` są 32-bitowymi liczbami ze znakiem w kodzie U2. Tak więc wartość typu `int` zajmuje jeden element na stosie, czyli 4 bajty.

W trakcie wykonywania programu w trybie 32-bitowym wartości liczbowe argumentów tej funkcji zostaną umieszczone na stosie, przy czym zgodnie z konwencją stosowaną przez kompilatory języka C parametry ładowane są na stos od prawej do lewej, czyli kolejno zostaną wpisane wartości parametrów `r`, `q`, `p`. Następnie zostanie wywołana funkcja (podprogram) `suma_liczb`. Kod tego podprogramu został napisany w assemblerze i podany jest na str. 8-9 (wersja 64-bitowa podana jest na str. 10).

W wersji dla trybu 32-bitowego, w początkowej części podprogramu na stosie zostaje przechowana zawartość rejestru **EBP** (rozkaz `push ebp`), a następnie do rejestru **EBP** zostanie załadowany adres wierzchołka stosu (skopiowany z rejestru **ESP**). Zatem położenie wierzchołka stosu wskazywane jest teraz także przez rejestr **EBP** — ten właśnie rejestr używany będzie do pobierania wartości parametrów. Sytuacja na stosie w tym momencie realizacji programu pokazana jest rysunku.



Jeśli więc rejestr **EBP** wskazuje położenie wierzchołka stosu, to zwiększając odpowiednio adres zawarty w rejestrze **EBP** o 8, 12 i 16 uzyskujemy adresy komórek pamięci wewnątrz stosu, w których zostały umieszczone wartości liczbowe przekazanych argumentów. Dysponując adresem komórki pamięci można, korzystając z mechanizmu adresowania indeksowego, odczytać zawartość tej komórki. W zapisie asemblerowym, symbole rejestru, w którym zawarty jest adres komórki pamięci umieszcza się w nawiasach kwadratowych, np. `mov eax, [ebp+8]`. Uwaga: zapis `mov eax, [ebp]+8` jest całkowicie równoważny.

Tak więc w początkowej części omawianego przykładu rozkaz `mov eax, [ebp+8]` wpisuje do rejestru **EAX** wartość pierwszego argumentu, a kolejne dwa rozkazy (`add eax, [ebp+12]` oraz `add eax, [ebp+16]`) dodają do rejestru **EAX** wartości drugiego i trzeciego argumentu.

Po wykonaniu tych operacji w rejestrze **EAX** znajdować się będzie suma trzech argumentów funkcji `suma_liczb`. Ponieważ, zgodnie z przyjętą konwencją obliczone wartości funkcji przekazywane są przez rejestr **EAX**, więc można zakończyć wykonywanie podprogramu i przekazać sterowanie do programu głównego (rozkaz `ret`). Przedtem jeszcze odtwarzana jest zawartość rejestru **EBP**.

Kod w języku C dla trybu 32-bitowego (w dalszej części ćwiczenia kod ten należy wpisać do pliku `cw2c32.c`)

```
#include <stdio.h>
int suma_liczb (int p, int q, int r);

int main()
{
    int wynik;
    wynik = suma_liczb(3, 5, 7);
    printf("\nSuma = %d\n", wynik);
    return 0;
}
```

Kod w asemblerze dla trybu 32-bitowego (w dalszej części ćwiczenia kod ten należy wpisać do pliku `cw2a32.asm`)

```
.686
.model flat

public _suma_liczb
```



```

; prototyp na poziomie języka C
; int suma_liczb (int p, int q, int r);

.code
_suma_liczb    PROC
                push    ebp
                mov     ebp, esp

; wpisanie wartości parametru p do rejestru EAX
                mov     eax, [ebp+8]

; dodanie do rejestru EAX wartości parametru q
                add     eax, [ebp+12]

; dodanie do rejestru EAX wartości parametru r
                add     eax, [ebp+16]

; odtworzenie pierwotnej zawartości rejestru EBP
                pop     ebp

                ret     ; powrót do programu głównego
_suma_liczb    ENDP

END

```

Program przykładowy (wersja 64-bitowa)

Podobnie jak poprzednio, program w wersji 64-bitowej składa się z dwóch modułów zawierających kod w języku C i kod w assemblerze. Program w języku C oblicza i wyświetla na ekranie sumę trzech liczb całkowitych (typu `__int64`), które są argumentami funkcji:

```
suma_liczb64 (__int64 p, __int64 q, __int64 r);
```

Wartości typu `__int64` są 64-bitowymi liczbami ze znakiem w kodzie U2.

W trakcie wykonywania programu w trybie 64-bitowym wartości liczbowe argumentów funkcji przekazywane są przez rejestry `RCX`, `RDX` i `R8`. Jeśli występowałby czwarty parametr funkcji, to przekazywany jest przez rejestr `R9`, natomiast ewentualne następne parametry: piąty, szósty, itd. przekazywane są przez stos tak jak w trybie 32-bitowym. Obliczenie sumy wykonuje podprogram, którego kod został napisany w assemblerze (plik `cw2a64.asm`).

Tak więc w początkowej części omawianego podprogramu rozkaz `mov rax, rcx` wpisuje do rejestru `RAX` wartość pierwszego argumentu, a kolejne dwa rozkazy (`add rax, rdx` oraz `add rax, r8`) dodają do rejestru `RAX` wartości drugiego i trzeciego argumentu.

Po wykonaniu tych operacji w rejestrze `RAX` znajdować się będzie suma trzech argumentów funkcji `suma_liczb64`. Ponieważ, zgodnie z przyjętą konwencją obliczone wartości funkcji przekazywane są przez rejestr `RAX`, więc można zakończyć wykonywanie podprogramu i przekazać sterowanie do programu głównego (rozkaz `ret`).

Kod w języku C dla trybu 64-bitowego (w dalszej części ćwiczenia kod ten należy wpisać do pliku *cw2c64.c*)

```
#include <stdio.h>
__int64 suma_liczb64 (__int64 p, __int64 q, __int64 r);

int main()
{
    __int64 wynik;
    wynik = suma_liczb64 (3, -1, 7000000000000000000);
    // trzecim argumentem jest liczba 7 trylionów (18 zer)
    printf("\nSuma = %I64d\n", wynik);
    return 0;
}
```

Kod w asemblerze dla trybu 64-bitowego (w dalszej części ćwiczenia kod ten należy wpisać do pliku *cw2a64.asm*)

```
public suma_liczb64
.code

suma_liczb64    PROC

; załadowanie do rejestru RAX wartości pierwszego argumentu
; funkcji
    mov rax, rcx

; dodanie do rejestru RAX wartości drugiego argumentu funkcji
    add rax, rdx

; dodanie do rejestru RAX wartości trzeciego argumentu funkcji
    add rax, r8

; wynik sumowania znajduje się w rejestrze RAX

    ret ; powrót do programu głównego
suma_liczb64    ENDP
END
```

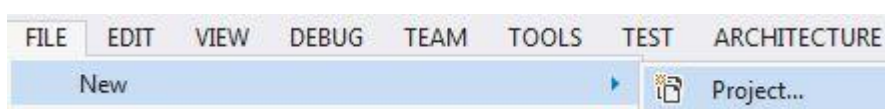
Edycja i uruchamianie programu przykładowego w środowisku Microsoft Visual Studio

Uwaga: nazwy plików zawierających część programu napisaną w języku C i w asemblerze **nie mogą być identyczne!**

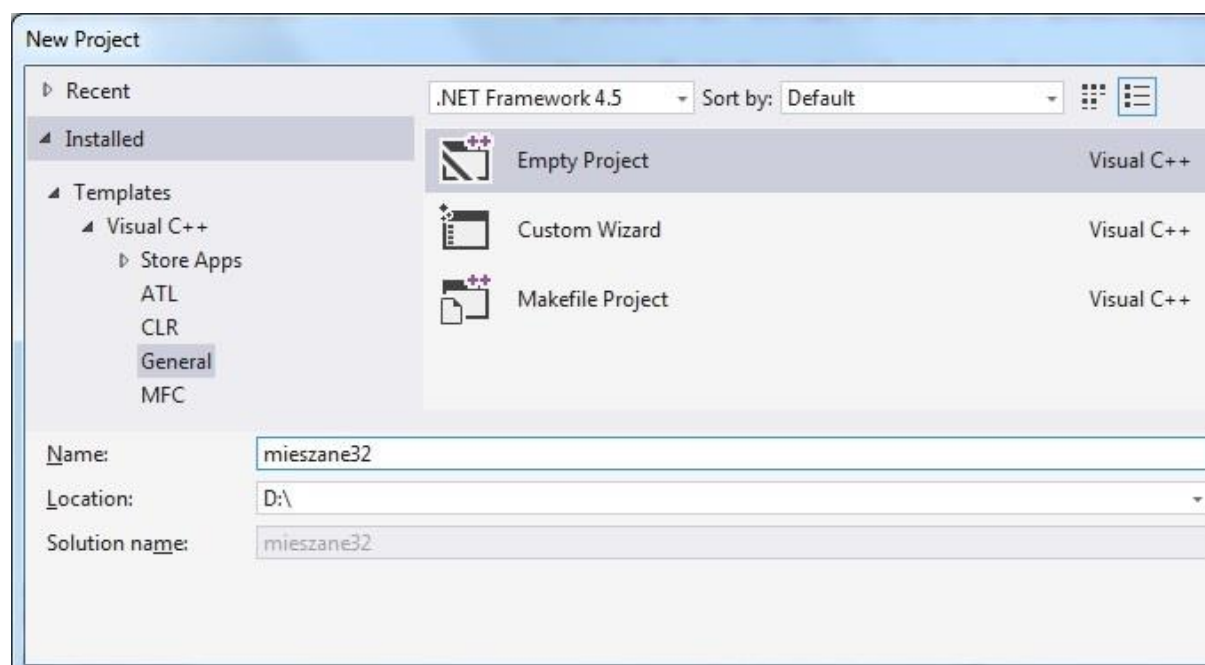
Zakładamy, że program źródłowy zostanie umieszczony w dwóch podanych wyżej plikach. W celu przeprowadzenia kompilacji programu w języku C, następnie asemblacji programu w języku asemblera i w końcu konsolidacji obu plików w języku pośrednim (z rozszerzeniem .OBJ) należy wykonać niżej opisane działania. Opis obejmuje wersje dla

trybu 32- i 64-bitowego. Dla trybu 32-bitowego i trybu 64-bitowego należy zbudować oddzielne projekty.

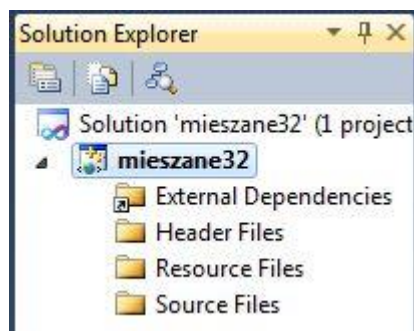
1. Po uruchomieniu MS Visual Studio 2013 należy wybrać opcje: File / New / Project



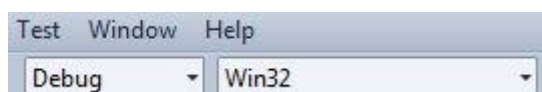
2. W oknie nowego projektu (zob. rys.) określamy najpierw typ projektu poprzez rozwinięcie opcji Visual C++. Następnie wybieramy opcje General / Empty Project. Do pola Name wpisujemy nazwę programu (tu: mieszane32 albo mieszane64) i naciskamy OK. W polu Location powinna znajdować się ścieżka D:\. Znacznik Create directory for solution należy ustawić w stanie nieaktywnym.



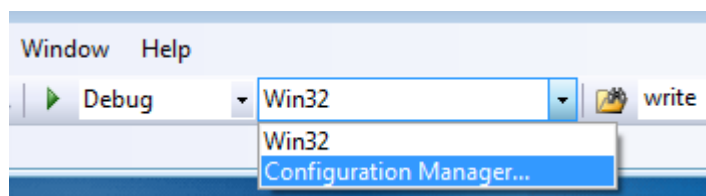
3. W rezultacie wykonania opisanych wyżej operacji pojawi się okno Solution Explorer, którego fragment pokazany jest na poniższym rysunku.



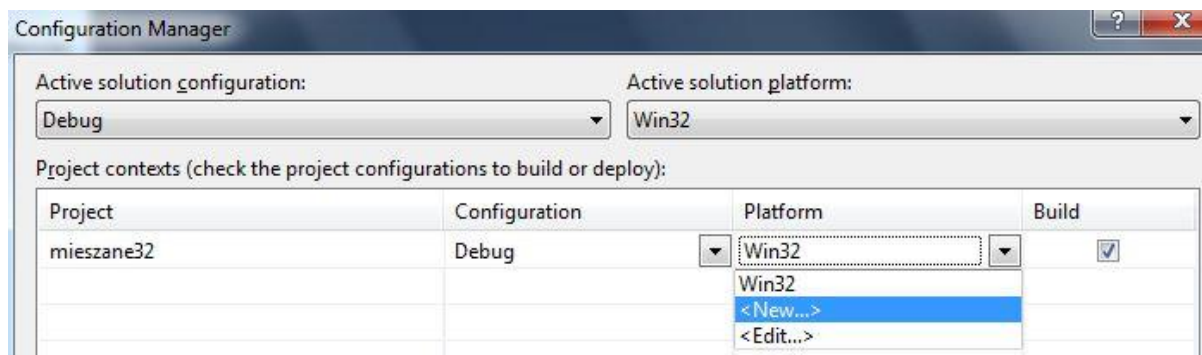
4. W kolejnym kroku należy określić tryb kompilacji i konsolidacji. W tym celu należy wybrać odpowiednią opcję na pasku w górnej części ekranu — ilustruje to poniższy rysunek. Dostępny jest tryb 32-bitowy oznaczony jako Win32 i tryb 64-bitowy oznaczony jako x64.



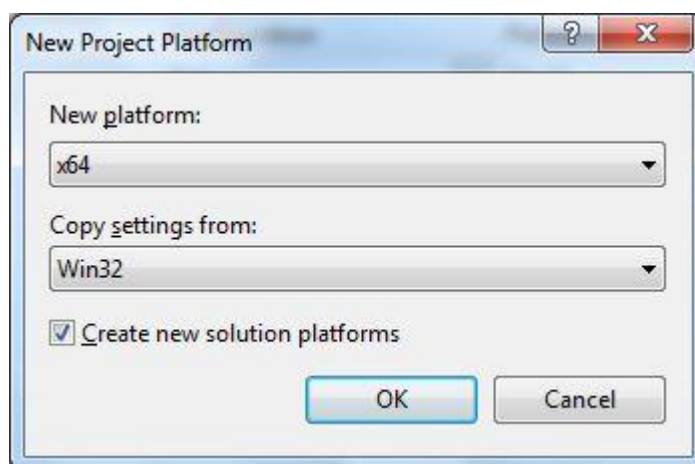
5. Zazwyczaj opcja x64 jest inicjalnie niedostępna i wymaga uaktywnienia (dla programu przykładowego w wersji 64-bitowej). W tym celu w górnej części ekranu trzeba wybrać opcję Configuration Manager tak jak pokazano na poniższym rysunku.



W rezultacie zostanie otwarte pokazane niżej okno. Następnie w tym oknie w kolumnie Platform należy wybrać opcję New.



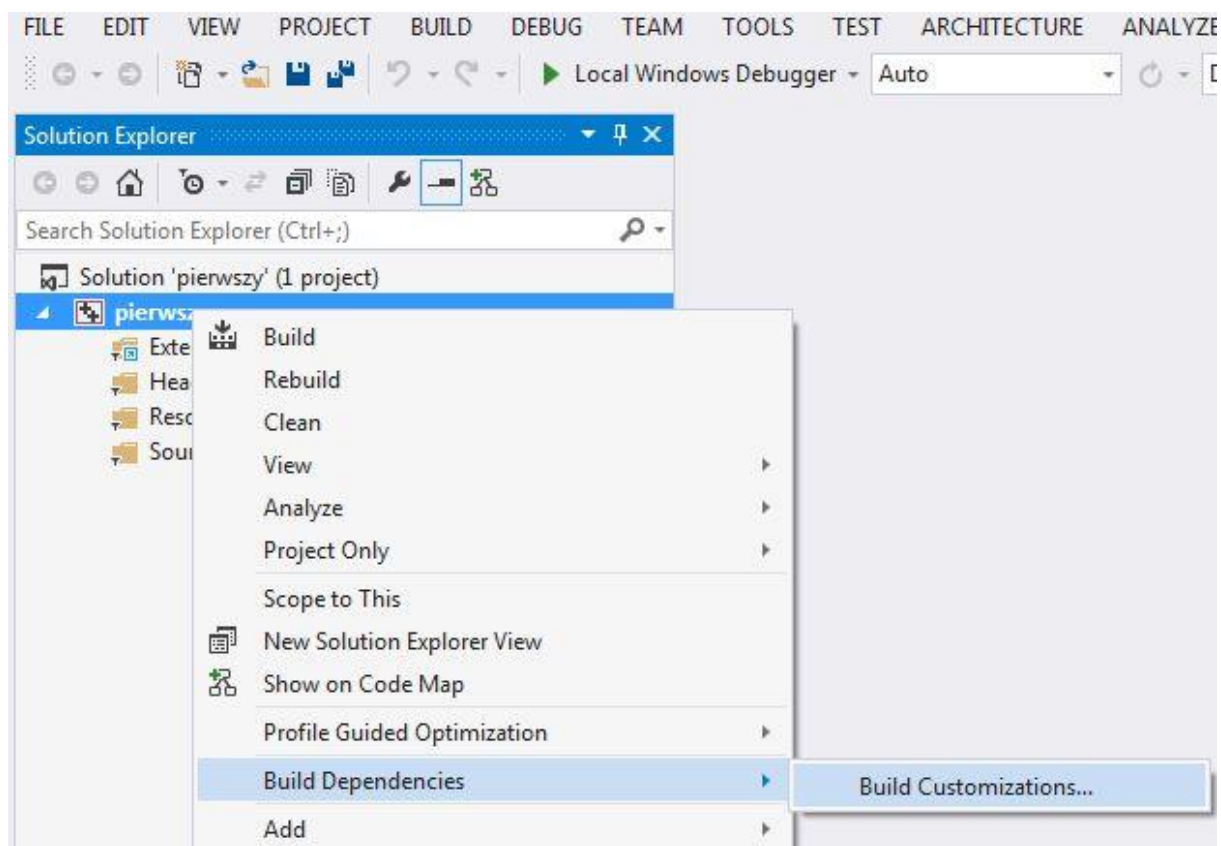
Z kolei pojawi się kolejne okno dialogowe, w którym należy wybrać wiersz x64, tak jak pokazano na poniższym rysunku.



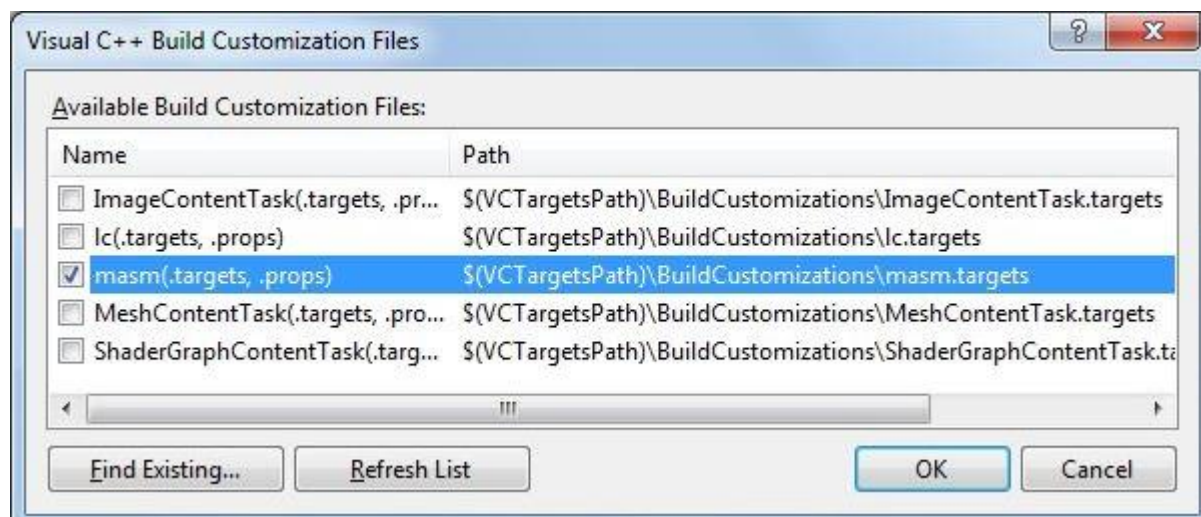
Następnie należy nacisnąć przycisk OK, potem Close, co spowoduje pojawienie się napisu x64 w górnej części ekranu (zob. rysunek).



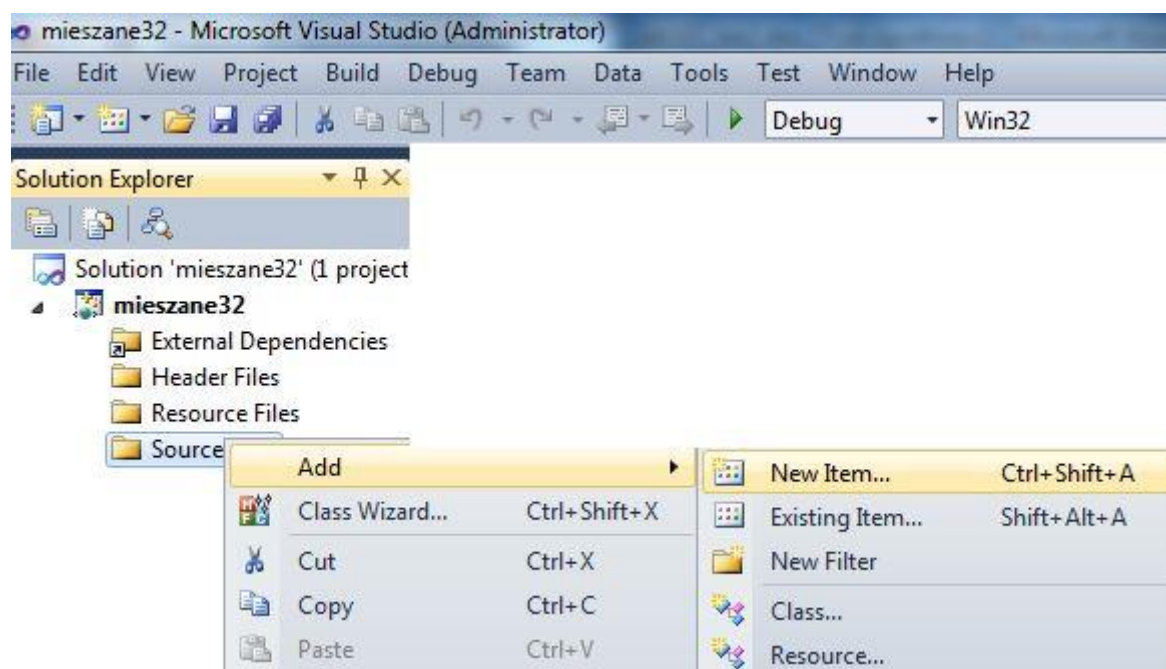
6. Teraz trzeba wybrać odpowiedni asembler. W tym celu należy kliknąć prawym klawiszem myszki na nazwę projektu pierwszy i z rozwijanego menu wybrać opcję Build Dependencies / Build Customizations.



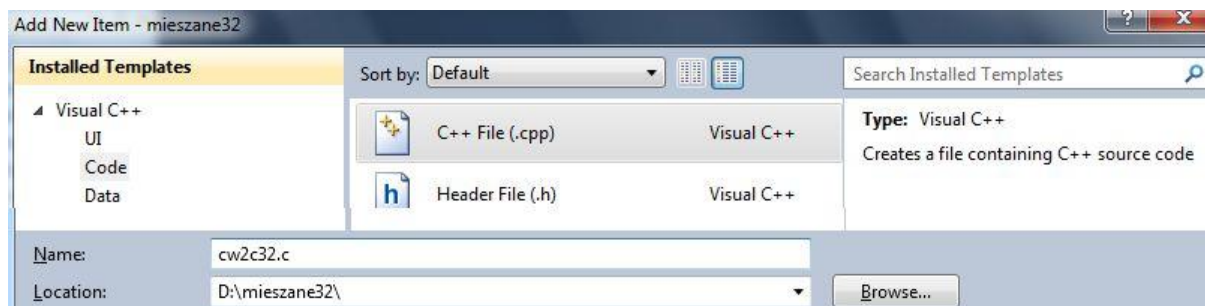
7. W rezultacie na ekranie pojawi się okno (pokazane na poniższym rysunku), w którym należy zaznaczyć pozycję **masm** i nacisnąć OK.



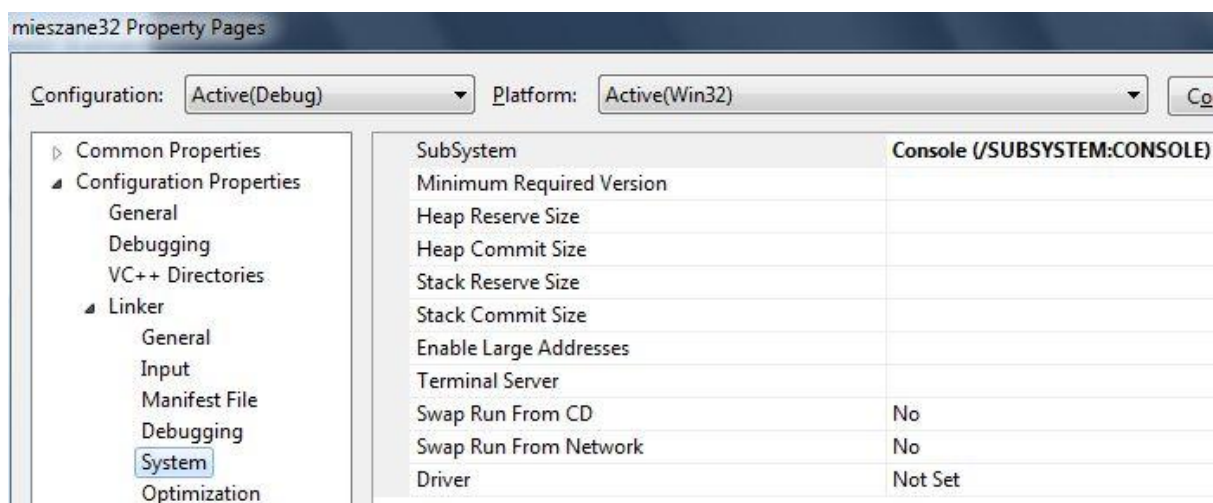
8. Następnie prawym klawiszem myszki w oknie **Solution Explorer** należy kliknąć na **Source File** i wybrać opcję **Add / New Item**.



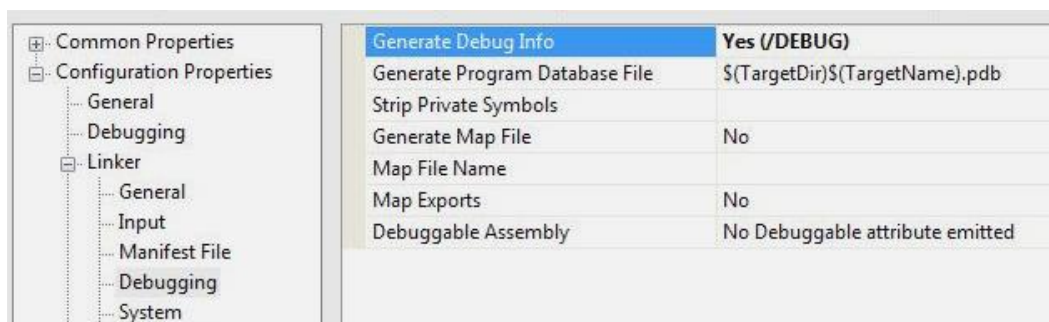
9. W ślad za tym pojawi się kolejne okno, w którym w polu **Name** wpisujemy nazwę pliku zawierającego kod w języku C (tu: **cw2c32.c**). Naciskamy przycisk **Add**, i zaraz po tym w identyczny sposób wprowadzamy nazwę pliku w asemblerze (tu: **cw2a32.asm**). Dla trybu 64-bitowego nazwy wprowadzanych plików będą miały postać: **cw2c64.c** i **cw2a64.asm**.



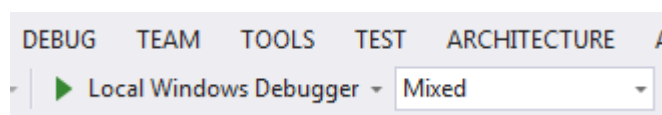
10. Po tych przygotowaniach do odpowiednich okien edycyjnych należy wprowadzić kod źródłowy programu w języku C i w asemblerze (podany na stronach 8-9). Dla trybu 32-bitowego kod źródłowy należy wprowadzić do okien skojarzonych z plikami `cw2c32.c` i `cw2a32.asm`, a trybu 64-bitowego do okien: `cw2c64.c` i `cw2a64.asm`.
11. W kolejnym kroku należy uzupełnić ustawienia konsolidatora (linkera). W tym celu należy kliknąć prawym klawiszem myszki na nazwę projektu `mieszane32` i z rozwijanego menu wybrać opcję **Properties**. W ramach pozycji **Linker** należy ustawić typ aplikacji. W tym celu zaznaczamy grupę **System** (zob. rysunek) i w polu **SubSystem** wybieramy opcję **Console (/SUBSYSTEM:CONSOLE)**.



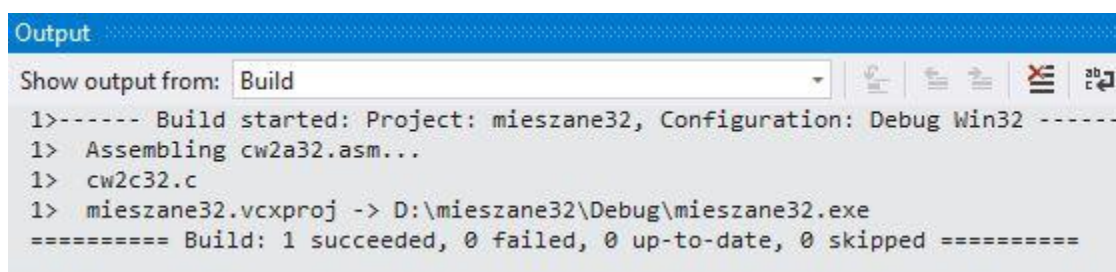
12. Przy okazji warto ustawić opcje aktywizujące debugger. W tym celu wybieramy opcję **Debugging** (stanowiącej rozwinięcie opcji **Linker**), a następnie pole **Generate Debug Info** ustawiamy na **YES**. Następnie naciskamy **OK**.



Ponadto w górnej części okna Visual Studio należy ustawić opcję debuggowania „Mixed” tak jak pokazano na poniższym rysunku.



13. W celu wykonania kompilacji (programu w języku C), asemblacji (programu w języku asemblera) i konsolidacji programu należy wybrać opcję **Build / Build Solution** (lub nacisnąć klawisz F7). Opis przebiegu tych operacji pojawi się w oknie umieszczonym w dolnej części ekranu. Przykładowa postać takiego opisu pokazana jest poniżej.



14. Jeśli w trakcie kompilacji, asemblacji lub konsolidacji wystąpią błędy, to program wynikowy nie jest tworzony.
15. W przypadku błędów sygnalizowanych podczas kompilacji lub asemblacji, identyfikacja błędu nie przedstawia większych trudności — wystarczy tylko, w oknie zawierającym opis kompilacji, dwukrotnie kliknąć myszką na wiersz zawierający słowo **error**, co spowoduje pojawienie się strzałki obok błędnego wiersza kodu źródłowego. Poniżej podano fragment opisu asemblacji programu, w którym wykryto błąd:

```
1>_MASM:
1> Assembling [Inputs]...
1>cw2a32.asm(14): error A2006: undefined symbol : esx
```

```

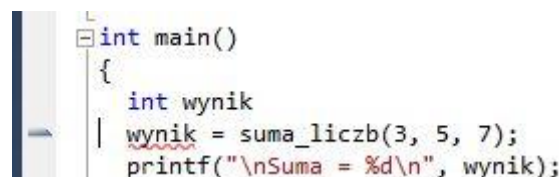
; wpisanie wartości parametru p do rejestru EAX
mov    esx, [ebp+8]

; dodanie do rejestru EAX wartości parametru q
add    eax, [ebp+12]
```


Zauważmy, że błędny wiersz został poprzedzony strzałką — w tym przypadku błąd polega na użyciu symbolu nieistniejącego rejestru `esx`.

16. Tak samo sygnalizowane są błędy wykryte podczas kompilacji programu w języku C:

```
1>ClCompile:
1> cw2c32.c
1>d:\cw2\cw2c32.c(7): error C2146: syntax error : missing ';' before
identifier 'wynik'
```

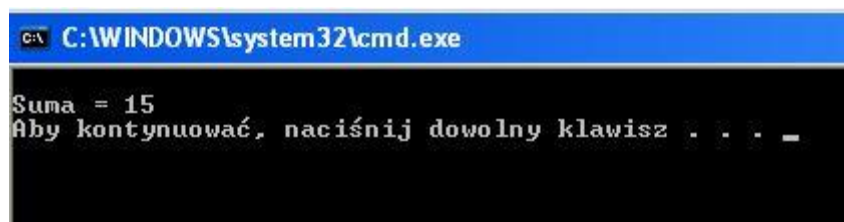


```
int main()
{
    int wynik
    |
    wynik = suma_liczb(3, 5, 7);
    printf("\nSuma = %d\n", wynik);
}
```

W podanym przykładzie błąd polega na pominięciu znaku średnika po deklaracji zmiennej `wynik`. Zauważmy, że strzałka nie wskazuje błędnego wiersza, lecz następny.

17. Podczas konsolidacji (linkowania) często występuje błąd nierozwiązanego odwołania zewnętrznego (ang. `unresolved external symbol`). Zazwyczaj błąd ten wynika z podania błędnej nazwy funkcji bibliotecznej, a przypadku programowania mieszanego przyczyną błędu jest często pominięcie asemblacji mimo istnienia pliku z rozszerzeniem `.asm`. W omawianym przypadku trzeba sprawdzić ustawienie opcji **Build Customizations** (pole `masm` powinno być zaznaczone). W dalszej kolejności należy usunąć z projektu plik w asemblerze poprzez (w oknie **Solution Explorer**) kliknięcie prawym klawiszem myszki na nazwę pliku w asemblerze i wybranie opcji **Remove / Remove**. Zaraz potem należy ponownie wprowadzić plik w asemblerze (**Source Files / Add / Existing Item / plik w asemblerze**).

18. Jeśli nie zidentyfikowano błędów, to można uruchomić program naciskając kombinację klawiszy **Ctrl F5**. Na ekranie pojawi się okno programu, którego przykładowy fragment pokazany jest poniżej.



```
C:\WINDOWS\system32\cmd.exe
Suma = 15
Aby kontynuować, naciśnij dowolny klawisz . . . _
```

Operacje na elementach tablic

Korzystając z wcześniej omawianej techniki adresowania rozpatrzmy teraz przykład wyznaczania sumy elementów tablicy liczb całkowitych. Kod programu głównego napisany jest w języku C, a funkcja (podprogram) wykonująca sumowanie napisana jest w asemblerze. Program napisany jest w wersji 32-bitowej. Tablica składa się z liczb całkowitych typu `int`, które kodowane są jako wartości 32-bitowe (4-bajtowe). Kod w języku C ma postać:

```
#include <stdio.h>
int suma_elementow (int tabl[], int n);

int main()
{
    int wynik, liczby[7] = {24, -20000, 0, 1, 20001, 19, 2};
    wynik = suma_elementow(liczby, 7);
    printf("\nSuma elementow tablicy = %d\n", wynik);
    return 0;
}
```

Do obliczenia sumy elementów tablicy używana jest funkcja `suma_elementow`, której kod został napisany w assemblerze. Funkcja ta ma dwa argumenty: adres tablicy i liczba elementów tablicy — wywołanie tej funkcji w programie przykładowym ma postać:

```
wynik = suma_elementow(liczby, 7);
```

Jeśli argumentem funkcji w języku C jest nazwa tablicy, to na stos ładowany jest jedynie adres tej tablicy, a nie wszystkie elementy. Zatem kod w assemblerze powinien odczytać ten adres i na jego podstawie określić wartości kolejnych elementów. Czynności te realizuje niżej podany kod w assemblerze.

```
.686
.model flat
public _suma_elementow
; prototyp funkcji na poziomie języka C ma postać:
; int suma_elementow (int tabl[], int n);

.code
_suma_elementow PROC
    push    ebp
    mov     ebp, esp
    push    ebx                ; przechowanie rejestru EBX
    mov     ebx, [ebp+8]       ; ładowanie adresu tablicy
    mov     ecx, [ebp+12]      ; liczba obiegów pętli
    mov     eax, 0              ; początkowa wartość sumy

; dodanie do EAX kolejnego elementu tablicy
ptl: add     eax, [ebx]

; obliczenie adresu kolejnego elementu
    add     ebx, 4

; zmniejszenie o 1 licznika obiegów pętli
    sub     ecx, 1

    jnz     ptl                ; skok, gdy licznik obiegów różny od 0

    pop     ebx                ; odtworzenie zawartości EBX
    pop     ebp                ; odtworzenie zawartości EBP
    ret                                     ; powrót do programu głównego
_suma_elementow ENDP

END
```

Technika porównywania liczb

W prawie wszystkich programach komputerowych sposób działania programu zależy od wartości wyników pośrednich. Konieczne jest więc odpowiednie sterowanie przebiegiem wykonywania programu w zależności od wartości tych wyników. W językach wysokiego poziomu sterowanie wykonywane jest za pomocą różnych wersji instrukcji warunkowej `if`, natomiast na poziomie kodu assemblerowego stosuje się rozkaz porównania `cmp` (skrót od ang. *compare* – porównywać) wraz z odpowiednio dobranymi rozkazami skoku warunkowego.

W celu porównania zawartości dwóch rejestrów należy wykonać ich odejmowanie, jednak bez wpisywania wyniku końcowego — operację tę wykonuje rozkaz `cmp`. Następnie wykonywany jest rozkaz skoku warunkowego: jeśli warunek jest spełniony, to następuje skok do miejsca w programie poprzedzonego etykietą, jeśli nie, to program wykonywany jest w naturalnej kolejności. Do porównywania stosuje się rozkazy podane w poniższej tabeli.

Warunek skoku	dla liczb bez znaku	dla liczb ze znakiem
Skocz, gdy większy	<code>ja</code>	<code>jg</code>
Skocz, gdy większy lub równy	<code>jae</code>	<code>jge</code>
Skocz, gdy mniejszy	<code>jb</code>	<code>jl</code>
Skocz, gdy mniejszy lub równy	<code>jbe</code>	<code>jle</code>
Skocz, gdy równy	<code>je</code>	
Skocz, gdy nierówny	<code>jne</code>	
Skok bezwarunkowy	<code>jmp</code>	

Poniżej podano przykładowy fragment, w którym porównywane są dwie liczby ze znakiem zawarte w rejestrach `ESI` i `EDI`.

```

cmp esi, edi
jge oblicz2

; fragment programu wykonywany, gdy ESI < EDI
- - - - -
- - - - -
    jmp dalej

oblicz2:
; fragment programu wykonywany, gdy ESI ≥ EDI
- - - - -
- - - - -

dalej:
- - - - -
```

Specyfika kompilatorów języka C++

Opisane tu zasady w pewnym stopniu dotyczą także kompilatorów języka C++. Główna trudność polega na konieczności uwzględnienia zmian nazw funkcji wykonywanych przez kompilator C++. Zmiany te opisane są zazwyczaj w dokumentacji kompilatora, ale ich uwzględnienie jest dość kłopotliwe. Z tego powodu zazwyczaj funkcje zakodowane w assemblerze wywołujemy w programie w języku C++ przy zastosowaniu interfejsu języka C. W takim przypadku obowiązują podane wyżej zasady, a prototyp funkcji musi być poprzedzony kwalifikatorem `extern "C"`, np.:

```
extern "C" int szukaj_max (int * tablica, int n);
```

Specyfika programowania mieszanego dla funkcji kodowanych wg standardu `stdcall` w kodowaniu 32-bitowym

W funkcjach zdefiniowanych w interfejsie Win32 API stosowany jest zazwyczaj standard `_stdcall`. W standardzie `_stdcall` stosowanym przez kompilatory firmy Microsoft nazwa funkcji po kompilacji (zawarta w pliku `.obj`) zawiera także liczbę bajtów zajmowanych przez parametry przekazywane do funkcji, przy czym nazwa poprzedzona jest znakiem podkreślenia `_`. Przykładowo, nazwa funkcji

```
iloczyn_liczb (int a, int b, int c);
```

zawarta w programie w języku C po kompilacji przyjmie postać `_iloczyn_liczb@12`. Do podanej funkcji przekazywane są bowiem trzy parametry, z których każdy zajmuje 32 bity (4 bajty).

Jeśli funkcję w standardzie `_stdcall` zamierzamy zakodować w assemblerze, to konieczne jest przekazanie assemblerowi informacji o liczbie i rozmiarach parametrów przekazywanych do funkcji. Informacje takie podaje się w wierszu dyrektywy `PROC`, np.

```
suma_liczb PROC stdcall, arg1:dword, arg2:dword, arg3:dword
```

Taka konstrukcja powoduje jednak pewne dodatkowe działania assemblera:

1. Nazwa funkcji (podprogramu) zostaje poprzedzona znakiem podkreślenia `_`.
2. Assembler automatycznie generuje rozkazy `push ebp` oraz `mov ebp, esp`, więc należy je pominąć w kodzie funkcji w assemblerze.
3. Assembler automatycznie generuje rozkaz `pop ebp` przed rozkazem `ret` (ściśle: generowany jest rozkaz `leave`, który w tym przypadku działa tak jak `pop ebp`).
4. Do rozkazu `ret` dopisywany jest dodatkowy parametr, np. `ret 12`, tak by rozkaz ten usunął parametry ze stosu (standard `stdcall` wymaga, by parametry ze stosu zostały usunięte przez wywołaną funkcję — czynność tę wykonuje właśnie rozkaz `ret` z parametrem).

W omawianym przypadku można (ale nie jest to obowiązkowe) używać podanych argumentów `arg1, arg2, ...` zamiast wyrażeń adresowych `[EBP+8]`, `[EBP+12]`, itd.

Jeśli funkcja kodowana w assemblerze ma wejść w skład biblioteki dynamicznej DLL, to po słowie `PROC` trzeba umieścić parametr `EXPORT`, np.

```
suma_liczb PROC stdcall EXPORT, arg1:dword, ....
```

Uzupełnienia dotyczące wywoływania funkcji w trybie 64-bitowym w systemie Windows¹

1. W trybie 64-bitowym pierwsze cztery parametry podprogramu przekazywane są przez rejestry: RCX, RDX, R8 i R9. Dopiero piąty parametr i następne, jeśli występują, przekazywane są przez stos, przy czym pierwszy z parametrów przekazywanych przez stos musi zajmować lokację pamięci o najniższym adresie, który musi być podzielny przez 8. Tak więc jeśli liczba parametrów przekracza 4, to parametry ładowane są na stos w kolejności od prawej do lewej, z wyłączeniem czterech pierwszych parametrów z lewej strony (które przekazywane są przez rejestry).
2. W trybie 64-bitowym do przekazywania liczb zmiennoprzecinkowych używa się odrębnych rejestrów związanych z operacjami multimedialnymi SSE: XMM0, XMM1, XMM2, XMM3 (zamiast rejestrów RCX, RDX, R8 i R9).
3. Bezpośrednio przed wywołaniem funkcji trzeba zarezerwować na stosie obszar 32-bajtowy (w przypadku programowania mieszanego rezerwację tę wykonuje kod generowany przez kompilator języka C). Obszar ten może wykorzystany w wywołanej funkcji (podprogramie) do przechowywania zawartości czterech rejestrów ogólnego przeznaczenia. Rezerwacja omawianego obszaru, który określany czasami angielskim terminem *shadow space*, jest wymagana także w przypadku, gdy liczba przekazywanych parametrów jest mniejsza niż 4. Rezerwację wykonuje się poprzez zmniejszenie wskaźnika stosu RSP o 32.

Parametry przekazywane przez stos
Obszar 32-bajtowy używany przez wywołaną funkcję
Ślad rozkazu CALL (adres powrotu)
Zmienne lokalne

4. Ponadto istnieje dodatkowe wymaganie: przed wykonaniem rozkazu skoku do podprogramu (rozkaz CALL) wskaźnik stosu RSP musi wskazywać adres podzielny przez 16. Pominięcie tego wymagania powoduje zazwyczaj zakończenie wykonywania programu wraz z komunikatem, że program wykonał niedozwoloną operację. W praktyce programowania mieszanego omawiany rozkaz CALL zawarty jest w kodzie generowanym przez kompilator języka C.
5. Zauważmy, że warunek podany w pkt. 4 nie jest spełniony bezpośrednio po rozpoczęciu wykonywania kodu wywołanej funkcji — rozkaz CALL zapisał bowiem 8-bajtowy ślad na stosie, wskutek czego rejestr RSP nie będzie podzielny przez 16. Oznacza to, że jeśli wewnątrz wywołanej funkcji zamierzamy wywołać inną funkcję (z co najwyżej czterema parametrami), to musimy zarezerwować (32 + 8) bajtów — rezerwacja dodatkowych 8 bajtów wynika z konieczności spełnienia warunku by rejestr RSP był podzielny przez 16.
6. Dodatkowo, liczba bajtów obszaru zajmowanego przez parametry (zob. pkt. 1) musi stanowić wielokrotność 16. Przykładowo, jeśli wywoływana funkcja ma 7 parametrów, to przed wywołaniem tej funkcji trzeba zarezerwować 72 bajty: 3 parametry przekazywane przez stos (24 bajty), obszar przewidziany do wykorzystania przez wywołaną funkcję (32 bajty), dopełnienie do wielokrotności 16 bajtów (8 bajtów), spełnienie warunku aby RSP był podzielny przez 16 (8 bajtów).

¹ Materiał zawarty w tym podrozdziale nie jest obowiązkowy

7. Zwolnienie stosu wykonuje program, który umieścił dane na stosie lub zarezerwował obszar.
8. Wyniki podprogramu przekazywane są przez rejestr RAX lub XMM0 (w przypadku wartości zmiennoprzecinkowych).
9. Wymienione rejestry muszą być zapamiętywane i odtwarzane (o ile są używane w programie): RBX, RSI, RDI, RBP, R12 ÷ R15, XMM6 ÷ XMM15

Zadania do wykonania

1. Uruchomić podany wcześniej program przykładowy w wersji 32-bitowej (pliki cw2c32.c i cw2a32.asm) i 64-bitowej (pliki cw2c64.c i cw2a64.asm).
2. Zmodyfikować i uruchomić obie wersje programu przykładowego (32- i 64-bitową) w taki sposób, by funkcja w assemblerze sumowała wartości 4 argumentów. *Wskazówka:* w trybie 64-bitowym cztery pierwsze parametry przekazywane są przez rejestry.
3. (zadanie nadobowiązkowe) Zmodyfikować i uruchomić wersję 64-bitową programu przykładowego w taki sposób, by funkcja w assemblerze sumowała wartości 5 argumentów. *Wskazówka:* w trybie 64-bitowym cztery pierwsze parametry przekazywane są przez rejestry, a następne przez stos, przy czym kompilator języka C tworzy na stosie dodatkowy obszar pomocniczy o rozmiarze 32 bajtów. W rezultacie piąty parametr dostępny będzie pod adresem RBP+16+32. Dalsze wyjaśnienia dotyczące tego zagadnienia podane są na poprzedniej stronie.
4. Napisać w assemblerze kod funkcji

```
void podaj_znak (int tabl[], int n);
```

przystosowanej do wywoływania z poziomu języka C w trybie 32-bitowym. Funkcja podaj_znak zastępuje wszystkie wartości dodatnie w n-elementowej tablicy tabl przez liczbę +1, wszystkie wartości ujemne przez liczbę -1, a liczby 0 pozostawia bez zmiany. Napisać także krótki program przykładowy w języku C, w którym nastąpi wywołanie ww. funkcji dla tablicy złożonej 7 elementów (wartości elementów wpisać bezpośrednio do kodu programu lub wprowadzać z klawiatury). W programie w języku C wyświetlać zawartość tablicy przed i po wywołaniu ww. funkcji. Utworzyć nowy projekt w środowisku MS Visual Studio 2013, wpisać kod w języku C i w assemblerze i uruchomić program.