# Ćwiczenie nr 6

# Programowanie mieszane

## 6.1 Wstęp

Współczesne języki programowania posiadają bardzo rozbudowane elementy językowe, co pozwala w większości przypadków na zdefiniowanie całego kodu programu w jednym języku. Czasami jednak napisanie fragmentu kodu za pomocą innego języka programowania pozwala na bardziej czytelne wyrażenie algorytmu, czy też prowadzi do bardziej efektywnego kodu wynikowego. Występują też sytuacje, gdy np. tworzymy program w języku C, a procedura numeryczna jest dostępna w postaci źródłowej w Fortranie czy Pascalu.

W odniesieniu do asemblera, wiele współczesnych kompilatorów języków programowania pozwala na bezpośrednie wprowadzenie kodu asemblerowego do kodu programu w języku wysokiego poziomu. W bardziej skomplikowanych przypadkach wskazane jest jednak utworzenie odrębnego modułu w asemblerze, który jest oddzielnie kompilowany (asemblowany) i następnie integrowany (linkowany) z innymi, wcześniej skompilowanymi modułami. Podstawą tej metody jest zasada, że 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.

Oczywiście, taki moduł asemblerowy musi spełniać standardy stosowane przez kompilator języka wysokiego poziomu. Zazwyczaj moduł asemblerowy składa się z jednego lub kilku podprogramów, które przystosowane są do wywoływania z programu w języku wysokiego poziomu

## 6.2 Wstawki asembler owe w języku C

Niektóre kompilatory języka C, np. Borland C/C++ posiadają wbudowany asembler, co pozwala na bezpośrednie dołączanie kodu asemblerowego do treści funkcji w języku C. Technika ta, określana terminem wstawki asemblerowe, może być zalecana wówczas, gdy długość dołączanego kodu asemblerowego nie przekracza kilkudziesięciu linii. Dostęp do zmiennych lokalnych i globalnych nie wymaga żadnych dodatkowych przygotowań; nie potrzeba też wykonywać czynności organizacyjnych. Wstawka asemblerowa ma postać

```
asm < instrukcja języka asembler >;
```

przykładowo

```
asm   mov ax, dx;
```

We wstawkach asemblerowych można też stosować etykiety – wyjaśnia to poniższy przykład.

```
int jakas_funkcja (int n)
{
      …
      etykleta1:
      asm   mov   cx, n;
      asm   mov   [si], 0;
      asm   inc   si;
      asm   loop  etykleta1;
      …
      return 0;
};
```

Przy większej liczbie instrukcji asemblerowych możliwe jest ich grupowanie przy pomocy instrukcji:

```
asm {
      < instrukcja języka asembler >;
      …
      < instrukcja języka asembler >;
      };
```

## 6.3 Struktury programów (modele pamięci)

Tryb 32-bitowy został wprowadzony w procesorach 386, i rozwinięty w 486 i w Pentium. Jednak w praktyce programowania dopiero rozpowszechnienie się systemów Windows 95/98 pozwoliło na pełne wykorzystanie jego zalet. Wcześniej stosowany był tryb 16-bitowy, co wiązało ze znacznymi utrudnieniami w zakresie programowania. Szczególnie kłopotliwe było ograniczenie rozmiarów segmentu do 64 KB, co wywarło istotny wpływ na struktury tworzonych programów. Jeśli obszar zajmowany przez dane lub instrukcje przekraczał 64 KB, to pojawiała się konieczność tworzenia w programie kilku segmentów danych czy instrukcji. W ślad za tym pojawiały się problemy związane z adresowaniem zmiennych zdefiniowanych w segmentach danych, a także problemy adresowania instrukcji sterujących, szczególnie w przypadku przekazywania sterowania do innego segmentu.

W celu uporządkowania możliwych sytuacji w zakresie adresowania zmiennych i przekazywania sterowania, zdefiniowano typowe struktury programów, określane jako modele pamięci. Na ogół struktury te opisywały programy złożone z wielu segmentów. Obecnie, wskutek coraz szerszego stosowania trybu 32-bitowego, w którym rozmiary segmentów mogą dochodzić do 4 GB, typowe programy zawierają zwykle pojedynczy segment instrukcji, pojedynczy segment danych i segment stosu. Taka struktura programu, określana jako model `flat`, staje się obecnie standardowa dla programowania w trybie 32-bitowym.

W przypadku programowania w trybie 16-bitowym zdefiniowano niżej wymienione struktury programów wielosegmentowych (modele pamięci):

- `small` – program zawiera pojedynczy segment kodu i pojedynczy segment danych;
- `medium` – program zawiera pojedynczy segment danych i wiele segmentów kodu;
- `compact` – program zawiera wiele segmentów danych i pojedynczy segment kodu;
- `large` – program zawiera wiele segmentów danych i wiele segmentów kodu.

Używane były także odmiany podanych modeli: `tiny` i `huge`. Poza wymienionymi tu segmentami kodu i danych statycznych, każdy z rozpatrywanych programów zawiera też segment danych dynamicznych (segment stosu).

Wybór modelu pamięci realizowany jest zazwyczaj poprzez podanie opcji kompilatora. Następnie w trakcie linkowania należy wskazać odpowiednie biblioteki funkcji standardowych, właściwe dla przyjętego modelu.

Wobec wspomnianego na początku rozwoju programów pracujących w trybie 32-bitowym, dominujące znaczenie posiada model `flat`, który stanowi odpowiednik modelu smali dla trybu 16-bitowego. Ponieważ jest to praktycznie jedyny używany model w trybie 32-bitowym, więc termin model pamięci nie musi być jawnie stosowany.

Kompilatory języka C stosują ustalone nazwy segmentów. Jeżeli jeden z modułów programu napisany jest w asemblerze, to nazwy i parametry segmentów muszą być zgodne z konwencją stosowaną przez kompilatory C.

W aktualnie używanych asemblerach dostępne są dyrektywy, które automatycznie generują potrzebne dyrektywy SEGMENT, GROUP i ASSUME, wraz z wymaganymi parametrami, zależnie od przyjętego modelu pamięci. Zestawienie tych dyrektyw zawiera poniższa tablica.

| Dyrektywa | Interpretacja |
|---|---|
| .STACK [rozmiar] | segment stosu |
| .CODE | segment kodu |
| .DATA | segment zawierające dane zainicjalizowane |
| .DATA? | segment zawierający dane niezainicjalizowane |
| .CONST | dane stałe |

Każda z podanych dyrektyw powoduje zamknięcie dotychczas kompilowanego segmentu (a więc generuje dyrektywę ENDS), i następnie generuje dyrektywę SEGMENT z odpowiednimi parametrami.

Wybór modelu pamięci realizowany jest za pomocą dyrektywy .MODEL, którą umieszcza się na początku pliku źródłowego (np. `.model flat`).

Zauważmy, że omawiane dyrektywy wyróżniają segmenty zawierające dane zainicjalizowane i niezainicjalizowane, co pozwala na bardziej racjonalne ich rozmieszczenie. Segmenty definiowane za pomocą dyrektywy .CONST zawierają dane stałe, tj. dane, które nie ulegają zmianie w trakcie wykonywania programu.

## 6.4 Łączenie kodu napisanego w języku C z kodem w asemblerze

Technika tworzenia programów, których kod składa się z fragmentów napisanych w różnych językach programowania, wymaga szczegółowej znajomości zasad wywoływania podprogramów stosowanych przez kompilatory tych języków. Informacje te dostępne są na ogół w dokumentacji oprogramowania. Wśród stosowanych standardów można zauważyć dwie charakterystyczne techniki – jedna z nich stosowana jest przez kompilatory języka C, druga zaś przez kompilatory języka Pascal. W dalszej części niniejszej ograniczymy się do rozpatrzenia techniki łączenia kodu asemblerowego z kodem napisanym w języku C.

### *6.4.1 Konwencje wywoływania procedur stosowane przez kompilatory języka C*

1. Parametry procedury 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. Jeśli parametr ma postać pojedynczego bajtu, to na stos ładowane jest podwójne słowo (32 bity), którego młodszą część stanowi podany bajt.
3. Jeśli parametrem jest liczba składająca się z kilku bajtów, to najpierw na stos ładowana jest najstarsza część liczby i kolejno coraz młodsze. Taki schemat ładowania stosowany jest w komputerach, w których liczby przechowywane są w standardzie mniejsze niżej, i wynika z faktu, że stos rośnie w kierunku malejących adresów.
4. Obowiązek zdjęcia parametrów ze stosu po wykonaniu procedury należy do programu wywołującego.
5. Kompilatory języka C stosują dwa typowe sposoby przekazywania parametrów: przez wartość i przez adres. Jeśli parametrem funkcji jest nazwa tablicy, to na stos ładowany jest adres tej tablicy; wszystkie inne obiekty, które nie zostały jawnie zadeklarowane jako tablice, przekazywane są „przez wartość”.
6. Wyniki podprogramu przekazywane są przez rejestry:

- wynik 1-bajtowy przez AL,
- wynik 2-bajtowy przez AX,
- wynik 4-bajtowy przez EAX.

Jeśli wynikiem podprogramu jest adres (wskaźnik), to przekazywany jest także przez rejestry.

7. Jeśli podprogram zmienia zawartość rejestrów EBX, EBP, ESI, EDI, SS, DS, 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.

Obok opisanej powyżej konwencji przekazywania parametrów, większość funkcji używanych w 32-bitowym systemie Windows (Win32 API) stosuje konwencję znaną jako `StdCall`, co stanowi skrót od „*Standard call*”. Konwencja ta jest bardzo zbliżona opisanej wyżej, z tą różnicą, że obowiązek zdjęcia parametrów ze stosu należy do wywołanego podprogramu. Konwencja wywoływania procedur stanowi na ogół parametr kompilacji, np. w środowisku Borland C++ 5.1 należy wybrać `Options|Project|Calling convention: C, Pascal, Register, Standard call`.

### *6.4.2 Elementy podprogramów asemblerowych wywoływanych z poziomu języka C*

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

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

Ze względu na konwencję nazw stosowaną przez kompilatory języka C, każdą nazwę o zasięgu globalnym wewnątrz podprogramu asemblerowego należy poprzedzić znakiem podkreślenia _ (nie dotyczy to konwencji StdCall). Niektóre kompilatory C rozróżniają tylko 8 pierwszych znaków nazwy, co należy brać pod uwagę przy tworzeniu nazw globalnych.

Należy też pamiętać, że w języku C małe i duże litery nie są utożsamiane, podczas gdy asembler zamienia wszystkie małe litery na duże (opcja `/ml` lub `/mx` powoduje, że asembler traktuje małe i duże litery niezależnie).

### *6.4.3 Przykład wywoływania podprogramu asemblerowego z poziomu języka C*

W instrukcji do ćwiczenia 5 omówiono podprogram asemblerowy koszt, który obliczał wartość wyrażenia $(\text{długość} + 5) \cdot \text{szerokość} + \text{wysokość}$. Spróbujmy dostosować ten podprogram do standardów wywoływania funkcji w języku C, w trybie 32-bitowym, przyjmując model `flat`. Przykładowy program w języku C, który wywołuje funkcję koszt może mieć postać:

```
#include <stdio.h>
extern int koszt (unsigned int dlugosc,
                  unsigned int szerokosc,
                  unsigned int wysokosc);
int main ()
{
      printf("\nObliczenie kosztow (model flat)");
      printf("\nKoszty (3, 4, 5) = %u", koszt(3, 4, 5));
 return 0;
}
```

Funkcja koszt, stanowiąca zmodyfikowaną wersję funkcji wcześniej omawianego podprogramu koszt, przystosowany do modelu `flat`, została umieszczona w odrębnym module asemblerowym.

```
.386 PUBLIC koszt

$param      STRUC
                  dd    ?     ;EBP
                  dd    ?     ;ślad instrukcji CALL
      dlugosc     dd    ?
      szerokosc   dd    ?
      wysokosc    dd    ?
$param ENDS

_TEXT       SEGMENT word public 'CODE' use32
            ASSUME cs:_TEXT
_koszt      PROC
            push  ebp                 ;przechowanie EBP na stosie
            mov   ebp, esp            ;po wykonaniu tej instrukcji
                                      ;rejestr EBP będzie wskazywał
                                      ;wierzchołek stosu
            mov   eax, [ebp].dlugosc  ;ładowanie parametru długość
            add   eax, 5              ;AX←AX + 5
            mul   [ebp].szerokosc     ;mnożenie przez szerokość
            add   eax, [ebp].wysokosc ;dodanie wartości parametru
                                      ;wysokość
            pop   ebp                 ;odtworzenie rejestru ebp
            ret                       ;powrót do programu wywołującego
_koszt      ENDP
_TEXT       ENDS
            END


mnożenie przez szerokość
; ; wysokość ; odtworzenie rejestru EBP
ret   ; powrót do programu wywołującego
koszt ENDP
TEXT  ENDS
END
```