Wykład 10

10. Tryby adresowania i modele pamięci

10.1. Tryb adresowania rzeczywistego

10.2. Tryb adresowania wirtualnego

10.3. Interpretacja pamięci

10.4. Stronicowanie pamięci

10.5. Modele pamięci

10.6. Struktura programu wykonywalnego

10. Tryby adresowania i modele pamięci

W kompilatorach 16-bitowych adresy logiczne przechowywane we wskaźnikach mają następującą postać:

adres_logiczny = segment : offset.

Segment jest numerem bloku z zakresu od 0 do 0xFFFF o rozmiarze 64KB względem początku, którego jest liczone przesunięcie offset, w ramach bloku.

Sposób wyznaczania adresu fizycznego na podstawie adresu logicznego zależy od trybu adresowania. W kompilatorach 32-bitowych (VC++, BuilderC++) wskaźniki zawierają 32-bitowe adresy logiczne, które są przeliczane przez system operacyjny na adresy fizyczne.

10.1. Tryb adresowania rzeczywistego

W trybie adresowania rzeczywistego (tryb rzeczywisty procesora, np. tryb DOS-u; kompilator BC++) adres fizyczny określa zależność:

adres_fizyczny = adres_bazowy + offset =
= 16∗segment + offset.

Segment (nr segmentu) i offset (przesunięcie w ramach bloku 64 KB o numerze segment) są liczbami 16 bitowymi bez znaku (typu unsigned). Adresy fizyczne są liczbami 20 bitowymi o wartościach od 0 do 0xFFFFF (adresowanie do 1 MB). Wartość 16∗segment jest adresem bazowym względem, którego liczone jest przesunięcie określone przez offset. Na przykład dla wskaźnika 0xB800 : 0x0002 adres bazowy wynosi 0xB8000 (mnożenie przez 16 polega na dopisaniu zera do wartości segmentu w postaci szesnastkowej), natomiast adres fizyczny wynosi 0xB8002.

Funkcja wyznaczająca adres fizyczny wskaźnika typu void far * w trybie rzeczywistym może być zdefiniowana następująco (BC++ 3.1):

```
 unsigned long adres(void far *x)
{
   return 16UL* FP_SEG(x) + FP_OFF(x);   // 16UL – zamiana na typ ul
 }
```

10.2. Tryb adresowania wirtualnego

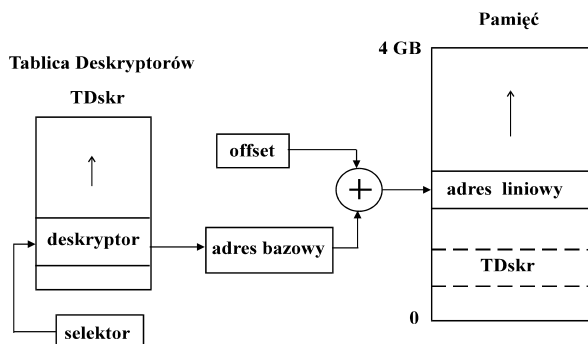
W trybie adresowania wirtualnego (tryb wirtualny, chroniony – wymaga przełączenia procesora, np. Windows NT; kompilator BCW) adresy logiczne przechowywane we wskaźnikach mają następującą postać:

adres_logiczny = selektor : offset.

Selektor jest liczbą dwubajtową (typu unsigned), która zawiera numer (indeks) pola w tablicy deskryptorów (TDskr) opisującego właściwości wskazywanego obszaru pamięci. Każdy deskryptor składa się z 8 bajtów i zawiera informacje o adresie początkowym (**adresie bazowym**) bloku, rozmiarze bloku oraz prawach dostępu do bloku (czytanie, zapis, usuwanie). Offset jest liczbą dwubajtową (typu unsigned), która określa przesunięcie od początku adresu bazowego (przesunięcie nie może przekroczyć rozmiaru bloku).

Na podstawie adresu bazowego można wyznaczyć **adres liniowy** (adres wirtualny) bloku (segmentu), który w przypadku braku stronicowania pamięci (automatycznej wymiany ramek stron pomiędzy pamięcią operacyjną i dyskową) jest równy adresowi fizycznemu bloku W systemach 32-bitowych - VC++, BuilderC++, przy wydruku zawartości wskaźnika jest wyprowadzany skojarzony z nim 32-bitowy adres logiczny zdefiniowany w 4-gigabajtowej przestrzeni przydzielonej dla aplikacji.

adres_liniowy = adres_bazowy + offset.



Począwszy od procesorów 80386 adres bazowy może być 32-bitowy (czterobajtowy). Pozwala to adresować liniowy blok pamięci o rozmiarze do 4GB. Adresy wykraczające poza obszar pamięci operacyjnej komputera znajdują się w pamięci dyskowej. Dane lub fragmenty programu znajdujące się pod tymi adresami są ładowane z dysku do pamięci w momencie, gdy są potrzebne.

W trybie wirtualnym istnieje możliwość ograniczenia swobody dostępu programów (aplikacji) do bloków pamięci wykorzystywanych przez inne programy. Wystarczy w tym celu ustawić atrybuty segmentów i zapamiętać je w odpowiednich polach statusowych związanych z nimi deskryptorów (np. segmenty kodu mają domyślnie ustawiony atrybut tylko do czytania). Dzięki wspomnianej własności tryb adresowania wirtualnego często jest nazywany **trybem chronionym.** Praca w trybie chronionym pozwala na implementację wielozadaniowości.

Struktura deskryptora segmentu

Każdy deskryptor składa się z 8 bajtów, opisujących fizyczny blok (segment) w pamięci wirtualnej. Znaczenie elementów deskryptora segmentu jest następujące:

- rozmiar segmentu (16 bitów); bajty 1,2;
- adres początkowy (bazowy) segmentu (bity 0...15); bajty 3,4;
- adres początkowy (bazowy) segmentu (bity 16...23); bajt 5;
- prawa dostępu do segmentu (dla 80286); bajt 6;
- rozszerzenie dla procesora 80386; bajty 7,8.

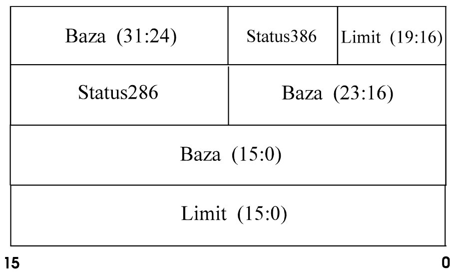
W przypadku procesora 80286, bajty 7, 8 deskryptora segmentu są równe zero. W przypadku procesora 80386 ich znaczenie jest następujące:
- bity 0...3 stanowią rozszerzenie rozmiaru segmentu
  (20-bitowy rozmiar segmentu dla 80386),
- bity 4...7 określają rozszerzenie praw dostępu,
- bity 8...15 stanowią rozszerzenie adresu początkowego segmentu
  (32-bitowy adres bazowy dla 80386).

Z przedstawionych rozważań wynika, że wartości kolejnych selektorów w adresach logicznych różnią się zawsze o stałą 8.

Tablica deskryptorów segmentów ma rozmiar 64KB, a więc można w niej zapamiętać 8K (8192) deskryptorów.

W praktyce dostępne są dwie tablice deskryptorów, globalna (GDT) i lokalna (LDT). Adresy tablic deskryptorów pamiętane są w 40-bitowych rejestrach GDTR i LDTR.

Deskryptor segmentu



Limit - rozmiar segmentu;

Baza - adres początkowy (bazowy) segmentu; baza adresu liniowego;

Status286 - prawa dostępu do segmentu
w przypadku procesora 80286;

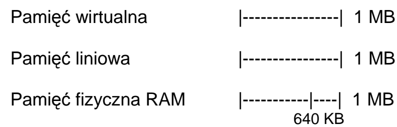
Status386 - prawa dostępu do segmentu
w przypadku procesora 80386.

W przypadku procesora 80286 maksymalna długość segmentu wynosi 64KB, natomiast w przypadku procesora 80386 maksymalna długość segmentu wynosi 1MB, a przy włączonym bicie granulacji (bit nr 3 w polu Status386 deskryptora) wynosi 4K $*$ 1MB = 4GB.

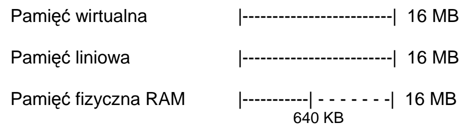
Dostępne są dwie tablice deskryptorów, globalna (GDT) i lokalna (LDT), co daje łączną, wirtualną przestrzeń adresową 16K $*$ 4GB = 64TB (tera bajtów).

10.3. Interpretacja pamięci

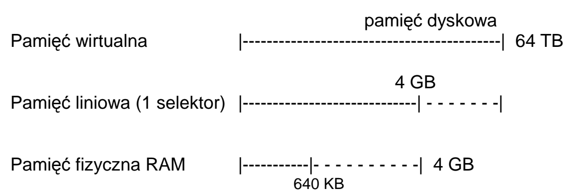
Procesor 808x



Procesor 80286



Procesor 80386



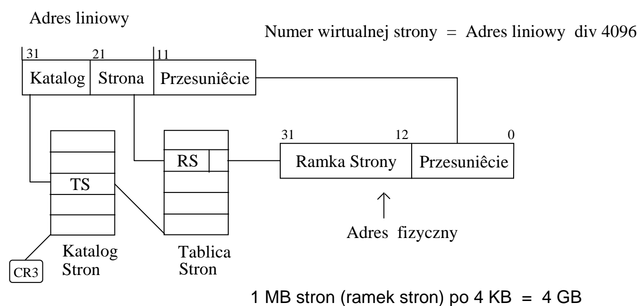
W systemach 32-bitowych VC++ i BBuilderC++ każda aplikacja korzysta z 4-gigabajtowej, logicznej przestrzeni adresowej. Liniowe adresy logiczne są przeliczane za pomocą odpowiedniej tabeli stron na adresy fizyczne przez system zarządzania pamięcią wirtualną.

10.4. Stronicowanie pamięci

Procesor 80386 implementuje wirtualne stronicowanie pamięci sprzętowo. Wszystkie informacje niezbędne do jego realizacji są przechowywane na dysku w postaci *katalogu stron* i kilku *tablic stron*.

Mechanizm stronicowania wykorzystuje pojęcie *adresu liniowego* (adresu wirtualnego). W procesorach 808x i 80286, które nie posiadają możliwości stronicowania pamięci oraz w przypadku pracy w trybie rzeczywistym lub przy wyłączonym stronicowaniu pamięci, adresy liniowe równają się adresom fizycznym. Jeżeli natomiast stronicowanie jest dostępne, to adresy wirtualne muszą być tłumaczone na adresy fizyczne - co jest realizowane sprzętowo.

Stronicowanie pamięci - wyznaczanie adresu fizycznego



CR3 – rejestr procesora.
Katalog określa numer tablicy stron, w której znajduje się ramka strony.
Strona określa pozycję adresu strony (ramki strony).

Sposób implementacji trybu chronionego zależy od wykorzystywanego kompilatora (np. w systemie Borland C++ 3.1 tryb chroniony można wykorzystywać w środowisku Windows w oparciu o kompilator BCW). Do zarządzania pamięcią w trybie chronionym można wykorzystywać funkcje klasy **Global** (np. GlobalAlloc, GlobalLock, GlobalFree). Za pomocą funkcji GlobalAlloc można przydzielić dynamicznie w pamięci rozszerzonej (powyżej 1 MB) blok o rozmiarze do 16 MB.

Przykład. 10.1. Wyznaczanie adresów liniowych elementów tablicy.

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <dos.h>
#include <windows.h>
#include <string.h>

// Wyznaczanie adresu liniowego i dostęp do danych
// ustawic sciezki do c:\bc\owl\ lib oraz include

unsigned long AdresLiniowy(const void far * p)
{
   unsigned sel = FP_SEG(p);
   unsigned long a;

   a = GetSelectorBase(sel);
   a = a + FP_OFF(p);

   return a;
}

void main()
{
   char far * p;
    char far *tab = new char[6];   // utworzenie tablicy

 // char naz[80] = "c:\\rc\\p10_1.txt";    // nazwa pliku z danymi

 // przekierowanie standardowego strumienia wyjscia stdout
// (normalnie skojarzonego z ekranem) do pliku o nazwie naz
// dane z stdout będą odbierane w pliku naz

   // if (freopen(naz, "w", stdout)== NULL)
  //  fprintf(stderr, "Blad przekierowania stdout\n");

   _fstrcpy(tab, "Wynik");   // kopiowanie "Wynik" do tab
```

```
    cout << "Elementy tablicy: tab[0] - tab[5]:" << endl;

    for (int j=0; j<6; j++)
    printf("%c - %3d %Fp %lu\n", tab[j], (int) tab[j], (void far *) &tab[j],
             AdresLiniowy(&tab[j]));

    printf("Dostep do tablicy za pomoca wskaznika p = &tab[0]\n");
    p=tab;
    for (j=0; j<6; j++)
    printf("%c - %3d %Fp  %lu\n", p[j], (int) p[j],(void far *) &p[j],
             AdresLiniowy(&p[j]));

    // zamkniecie standardowego strumienia stdout
    // i pliku z nim skojarzonego
  //   fclose(stdout);

    delete tab;
getch();
}
```

Przykładowe wyniki programu 10.1.

```
Elementy tablicy: tab[0] - tab[5]:
W -  87 1AD7:21E4 209796
y - 121 1AD7:21E5 209797
n - 110 1AD7:21E6 209798
i - 105 1AD7:21E7 209799
k - 107 1AD7:21E8 209800
□ -   0 1AD7:21E9 209801
Dostep do tablicy za pomoca wskaznika p = &tab[0]
W -  87 1AD7:21E4  209796
y - 121 1AD7:21E5  209797
n - 110 1AD7:21E6  209798
i - 105 1AD7:21E7  209799
k - 107 1AD7:21E8  209800
□ -   0 1AD7:21E9  209801
```

Przykład. 10.2. Dostęp do bloku pamięci o rozmiarze 200000 bajtów przydzielonej za pomocą funkcji GlobalAlloc z wykorzystaniem wskaźników typu huge.

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <dos.h>
#include <windows.h>

// Dynamiczna alokacja bloku 200 000 bajtów; losowa inicjacja;
// wyprowadzanie danych

// ustawic sciezki do c:\bc\owl\ lib oraz include

void huge* pbuf1;        // wskaznik pocz. bufora

HGLOBAL handle1;    // uchwyt bloku


void main()
{
   unsigned char huge * p;

   randomize();            // inicjacja generatora

// char naz[80] = "c:\\rc\\p10_2.txt";    // nazwa pliku z danymi

// przekierowanie standardowego strumienia wyjscia stdout
// (normalnie skojarzonego z ekranem) do pliku o nazwie naz
// dane z stdout będą odbierane w pliku naz

//  if (freopen(naz, "w", stdout)== NULL)
//  fprintf(stderr, "Blad przekierowania stdout\n");

//  alokacja bloku przesuwalnego
     handle1 = GlobalAlloc(GMEM_MOVEABLE, 200000);

// zablokowanie bloku i utworzenie wskaźnika
     pbuf1 = GlobalLock(handle1);
```

```
    printf("Wskaźnik bloku: %Fp   Uchwyt: %#X\n", pbuf1, handle1);
    if (pbuf1)
    {  p = (unsigned char huge *) pbuf1;  // konwersja typu

        // pbuf1 adresuje 200 000 bajtów = 3 * 64 KB + reszta
        // zajmuje selektory:
        // seg(pbuf1)   -  64KB = 65536 bajtów,
        // seg(pbuf1)+8 -  64KB = 65536 bajtów,
        // seg(pbuf1)+16 - 64KB = 65536 bajtów,
       // seg(pbuf1)+24 - 3392 bajty

        for (long i=0; i<200000; i++)   // inicjacja bloku
        {
            p[i] = random(255);    // we wskaźniku typu huge
                              // automatyczna korekcja segmentu p o 8 co 64KB
              if( !(i%20000) || i>199995  || (65531<i && i<65540) )
              { printf("i = %9ld %4d %Fp\n", i, p[i], &p[i]);
                getch(); }
        }
      printf("Odczyt komórek: 65532-65539\n");
      for (i=0; i<200000; i++)
      if (65531<i && i<65540) printf ("%4d", p[i]);

      printf("\n");
      printf ("Odczyt komórek: 199996-199999\n");
      for (i=0; i<200000; i++)   // dla i < 200 001 blad ochrony
      if (i>199995) printf ("%4d", p[i]);

        GlobalUnlock(handle1);  // odblokowanie dostępu do bloku

        GlobalFree(handle1);    // zwolnienie pamięci
    }
    else { cout << "Blad alokacji pamieci !" << endl; }

   // zamkniecie standardowego strumienia stdout
   // i pliku z nim skojarzonego
   // fclose(stdout);

getch();
}
```

Przykładowe wyniki programu 10.2.

```
Wskaźnik bloku: 1967:0000   Uchwyt: 0X1966
i =         0  218 1967:0000
i =     20000  246 1967:4E20
i =     40000  100 1967:9C40
i =     60000   13 1967:EA60
i =     65532  102 1967:FFFC
i =     65533   77 1967:FFFD
i =     65534   80 1967:FFFE
i =     65535    0 1967:FFFF
i =     65536  116 196F:0000
i =     65537  170 196F:0001
i =     65538  154 196F:0002
i =     65539  199 196F:0003
i =     80000    6 196F:3880
i =    100000  135 196F:86A0
i =    120000  245 196F:D4C0
i =    140000  220 1977:22E0
i =    160000  237 1977:7100
i =    180000    5 1977:BF20
i =    199996   84 197F:0D3C
i =    199997   94 197F:0D3D
i =    199998  158 197F:0D3E
i =    199999   55 197F:0D3F
Odczyt komórek: 65532-65539
 102  77  80   0 116 170 154 199
Odczyt komórek: 199996-199999
  84  94 158  55
```

W przypadku wskaźników typu huge następuje automatyczna modyfikacja numeru selektora przy przejściu w bloku o rozmiarze 200000 przez granice segmentów o rozmiarze 64KB (65536 B). W przypadku wykorzystywania wskaźników typu far należy modyfikować selektory samodzielnie.

Przykład. 10.3. Dostęp do bloku pamięci o rozmiarze 200000 bajtów przydzielonej za pomocą funkcji GlobalAlloc z wykorzystaniem wskaźników typu far.

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <dos.h>
#include <windows.h>

 // Alokacja bloku 200 000 bajtów z wukorzystaniem wskaznika typu far;
 // inicjacja 10 ostatnich bajtów bloku - komórki 199990 do 199999

 // ustawic sciezki do c:\bc\owl\ lib oraz include

void far* pbuf1;   // wskaznik pocz. bufora

HGLOBAL handle1;        // uchwyt bloku


void main()
{
     unsigned char far * pom;
     unsigned char far * pom1;

     randomize();           // inicjacja generatora

// char naz[80] = "c:\\rc\\p10_3.txt";    // nazwa pliku z danymi

// przekierowanie standardowego strumienia wyjscia stdout
// (normalnie skojarzonego z ekranem) do pliku o nazwie naz
// dane z stdout będą odbierane w pliku naz

//   if (freopen(naz, "w", stdout)== NULL)
//   fprintf(stderr, "Blad przekierowania stdout\n");

     handle1 = GlobalAlloc(GMEM_MOVEABLE, 200000);
     pbuf1 = GlobalLock(handle1);

     printf("Wskaźnik bloku: %Fp     Uchwyt: %#X\n", pbuf1, handle1);
     if (pbuf1)
     {
         // pbuf1 adresuje 200 000 bajtów = 3 * 64 KB + reszta
         // zajmuje selektory:
// seg(pbuf1)   -  64KB = 65536 B,    // seg(pbuf1)+8 - 64KB = 65536 B,
// seg(pbuf1)+16 - 64KB = 65536 B,  // seg(pbuf1)+24 - 3392 B

// przy przechodzeniu przez granice segmentu 64KB
// nalezy zwiekszyc segment wskaznika typu far o 8

// ustawienie wskaznika pom na adres
// (pbuf1 + 3*64KB + 3382 = 199990)
// próba dostepu do komorki pamieci o adresie
// (seg(pbuf1): ofs(pbuf1)+10) - zakonczy sie bledem ochrony, gdyż
// pbuf1 nie ma dostepu do komórki o adresie 199 990 + 10 = 200000

pom =(unsigned char far*) MK_FP(FP_SEG(pbuf1)+24,
FP_OFF(pbuf1)+3382);

printf("Wskaźnik do komórki 199990 (%x+0x18, 3382=0xD36): %Fp \n",
FP_SEG(pbuf1),pom);
         pom1 = pom;   // zapamietanie pom

         printf("Inicjacja komórek 199990 - 199999\n");
         for (long i=0; i<10; i++)  // dla i < 11 blad ochrony
         {
            pom[i] = random(255);  // inicjacja
            printf("i = %9ld %4d %Fp\n", i, pom[i], &pom[i]);
             getch();
         }
          printf("Odczyt komórek 199990 - 199999\n");
          for (i=0; i<10; i++) printf ("%4d", pom1[i]);

         GlobalUnlock(handle1);  // odblokowanie bloku
         GlobalFree(handle1);   // zwolnienie pamieci
     } else { cout << "Blad alokacji pamieci !" << endl; }

// zamkniecie standardowego strumienia stdout
// i pliku z nim skojarzonego
//     fclose(stdout);
   getch();
}
```

Przykładowe wyniki programu 10.3.

```
Wskaźnik bloku: 1967:0000      Uchwyt: 0X1966
Wskaźnik do komórki 199990 (1967+0x18, 3382=0xD36):
197F:0D36
Inicjacja komórek 199990 - 199999
i =         0  184 197F:0D36
i =         1   75 197F:0D37
i =         2  212 197F:0D38
i =         3  207 197F:0D39
i =         4  149 197F:0D3A
i =         5   55 197F:0D3B
i =         6  138 197F:0D3C
i =         7  109 197F:0D3D
i =         8  114 197F:0D3E
i =         9   21 197F:0D3F
Odczyt komórek 199990 - 199999
 184 75 212 207 149  55 138 109 114  21
```

10.5. Modele pamięci

Adresowanie oparte na segmentach (selektorach) jest ściśle powiązane ze strukturą programu wykonywalnego (exe), który zawiera instrukcje programu oraz wykorzystywane stałe i zmienne globalne. W momencie uruchomienia programu przydzielany jest w pamięci komputera obszar przeznaczony dla kodu programu (segment kodu), obszar przeznaczony dla danych (segment danych) i obszar przeznaczony dla zmiennych lokalnych, parametrów i adresów powrotu z wywołań funkcji (segment stosu) oraz rezerwowany jest obszar przeznaczony dla zmiennych dynamicznych - nazywany stertą (stosem zmiennych dynamicznych).

W języku C/C++ program może składać się z wielu modułów. Po kompilacji programu (modułu) utworzony zostanie kod wykonywalny (wynikowy), który może składać się z następujących segmentów:

- kodu (instrukcje do wykonania),
- danych zainicjowanych (np. stałe),
- danych niezainicjowanych,
- stosu (zmienne automatyczne i parametry funkcji),
- sterty (stosu zmiennych dynamicznych).

Każdy z tych segmentów posiada swoją nazwę i klasę, które są łańcuchami znaków (np. segment kodu jest klasy CODE i ma domyślną nazwę _TEXT). Klasa stanowi informację dla programu łączącego o sposobie łączenia i kolejności rozmieszczania w pamięci segmentów należących do poszczególnych modułów.

W systemie BC++ 3.1 rozmiar segmentów kodu i danych *nie może przekroczyć* 64KB dla *pojedynczego* modułu programu. Stos programu nie może przekroczyć 64KB, natomiast sterta może zajmować całą dostępną pamięć (ograniczenia wynikają z możliwości kompilatora).

W przypadku programów wielomodułowych całkowity rozmiar kodu i danych programu oraz rozmieszczenie modułów w pamięci zależy od przyjmowanego w kompilatorze **modelu pamięci**. Na przykład system BC++ 3.1 oferuje sześć modeli: *TINY*, *SMALL*, *MEDIUM*, *COMPACT*, *LARGE* i *HUGE* (BCW++3.1 – modele: *SMALL*, *MEDIUM*, *COMPACT*, *LARGE*). Każdy z tych modeli nakłada pewne ograniczenia na maksymalny rozmiar obszaru danych, rozmiar kodu programu oraz domyślny typ wskaźników kodu i danych (wskaźniki dalekie lub bliskie).

We wszystkich modelach, oprócz modelu HUGE, obszar danych **statycznych** jest wspólny dla wszystkich modułów programu, a jego rozmiar nie może przekroczyć 64KB.

W modelu TINY maksymalny rozmiar, łącznie, kodu i danych (łącznie ze stertą, tj. obszarem zmiennych dynamicznych) nie może przekroczyć 64KB. Model SMALL ogranicza kod do 64KB i dane do 64KB. W modelu MEDIUM kod jest ograniczony do 1MB, natomiast dane do 64KB. Model COMPACT ogranicza kod do 64KB oraz dane do 1MB. W modelach LARGE i HUGE kod i dane, każde z osobna, nie mogą przekroczyć 1MB.

Model pamięci określa domyślny typ wskaźników kodu i danych (wskaźniki dalekie lub bliskie). W szczególności, wskaźnik zdefiniowany jako int *wsk jest wskaźnikiem bliskim (dwubajtowym) w modelach TINY, SMALL i MEDIUM, natomiast wskaźnikiem dalekim w pozostałych modelach.

Model	*Ograniczenia*		*Wskaźniki: Kod*	*Dane*
TINY	Kod + Dane < 64KB		Bliskie	Bliskie
SMALL	Kod < 64KB,	Dane < 64KB	Bliskie	Bliskie
MEDIUM	Kod < 1MB,	Dane < 64KB	Dalekie	Bliskie
COMPACT	Kod < 64KB,	Dane < 1MB	Bliskie	Dalekie
LARGE	Kod < 1MB,	Dane < 1MB	Dalekie	Dalekie
HUGE	Kod < 1MB,	Dane < 1MB	Dalekie	Dalekie

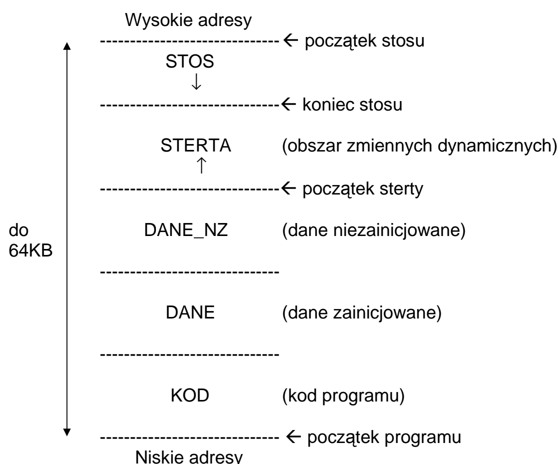
We wszystkich modelach pamięci jedynie zmienne automatyczne i parametry funkcji posiadają ściśle określone, niezależne od modelu pamięci miejsce przechowywania - stos procesora (segment stosu).

10.6. Struktura programu wykonywalnego

Program wykonywalny ma różną strukturę w zależności od wykorzystywanego modelu pamięci.

Model TINY

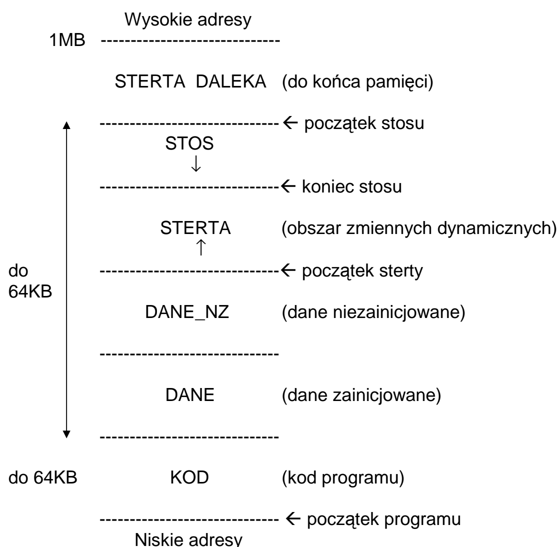
Model ten jest przeznaczony do tworzenia programów o niewielkich rozmiarach. W modelu tym rozmiar obszaru przeznaczonego na kod, dane i stos nie może przekroczyć 64KB. Wskaźniki kodu i danych są bliskie. Programy tej klasy mogą być przekształcone do wersji com.



Model SMALL

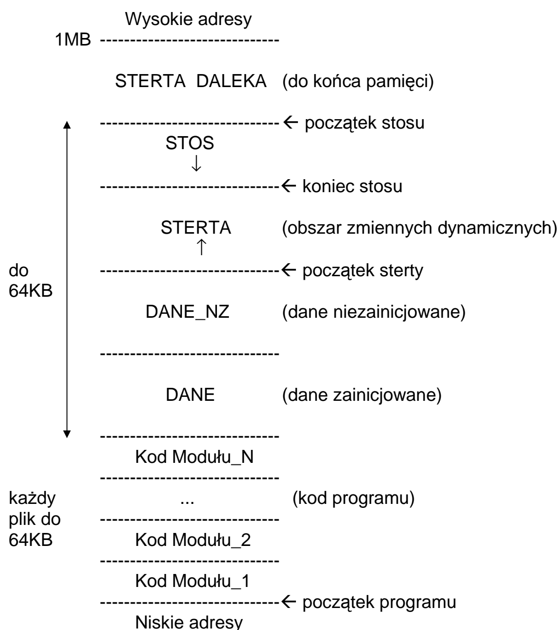
Model ten jest przeznaczony do tworzenia programów, w których kod i dane znajdują się w osobnych segmentach. Rozmiar kodu nie może przekroczyć 64 KB oraz rozmiar danych nie może przekroczyć 64KB. Wskaźniki kodu i danych są bliskie. Istnieją dwa rodzaje sterty: *bliska* – w obszarze segmentu danych oraz *daleka* – od początku stosu do końca dostępnej pamięci. Segmenty danych, sterty bliskiej i stosu zajmują ten sam obszar pamięci (segment danych).

Do przydzielania pamięci w obszarze *sterty bliskiej* służą funkcje *malloc* i *calloc* – ANSI C (zwolnienie pamięci – funkcja *free*), natomiast w obszarze *sterty dalekiej* funkcje *farmalloc* i *farcalloc* – BC++3.1 (zwolnienie pamięci – funkcja *farfree*). Operator new przydziela pamięć domyślnie w obszarze sterty bliskiej.



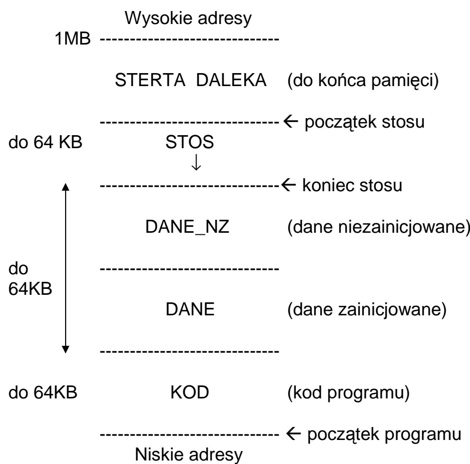
Model MEDIUM

Model ten jest przeznaczony do tworzenia programów o znacznych rozmiarach kodu i niewielkiej ilości danych. Segmenty kodu należące do różnych modułów są umieszczane kolejno w pamięci. Rozmiar każdego z nich może mieć maksymalnie do 64KB, ale łącznie nie mogą zajmować więcej niż 1 MB. Adres segmentu kodowego jest ustalany w momencie odwołania do funkcji zdefiniowanej w danym module. Dane statyczne, sterta bliska i stos są połączone w jeden segment danych, którego rozmiar nie może przekroczyć 64KB. Wskaźniki kodu są dalekie, natomiast danych bliskie.



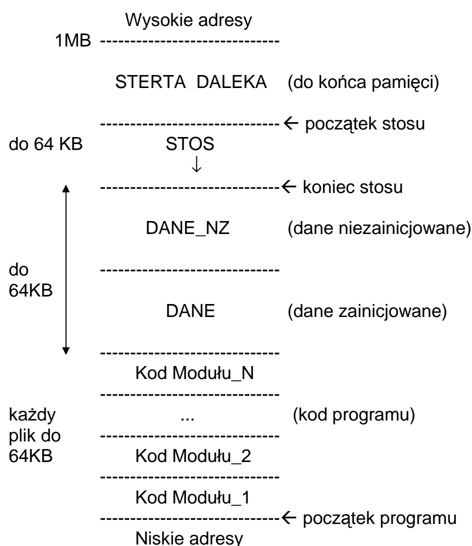
Model COMPACT

Model ten jest przeznaczony do tworzenia programów o niewielkich rozmiarach (do 64KB), ale zarządzających wielkimi strukturami danych. Rozmiar obszaru danych nie może przekroczyć 64KB. Składa się on z segmentu danych zainicjowanych i niezainicjowanych. Dane i sterta łącznie nie mogą zajmować więcej niż 1MB. Stos jest niezależnym segmentem o rozmiarze do 64KB. Istnieje tylko daleka sterta zmiennych dynamicznych. Wskaźniki kodu są bliskie, danych – dalekie.



Model LARGE

Model ten jest przeznaczony do tworzenia programów o znacznych rozmiarach kodu i danych. Segmenty kodu należące do różnych modułów są umieszczane kolejno w pamięci. Rozmiar każdego z nich może mieć maksymalnie do 64KB, ale łącznie nie mogą zajmować więcej niż 1MB. Adres segmentu kodowego jest ustalany w momencie odwołania do funkcji zdefiniowanej w danym module. Obszar danych nie może przekroczyć 64KB. Składa się on z segmentu danych zainicjowanych i niezainicjowanych. Dane i sterta łącznie nie mogą zajmować więcej niż 1MB. Stos jest niezależnym segmentem o rozmiarze do 64KB. Istnieje tylko daleka sterta zmiennych dynamicznych. Wskaźniki kodu i danych są dalekie.



Model HUGE

Model ten jest przeznaczony do tworzenia programów o znacznych rozmiarach kodu i danych. Segmenty kodu należące do różnych modułów są umieszczane kolejno w pamięci. Rozmiar każdego z nich może mieć maksymalnie do 64KB. Obszar danych statycznych może mieć rozmiar większy niż 64KB. Składa się on z segmentów danych należących do poszczególnych modułów każdy o rozmiarze nie przekraczającym 64KB. Segmenty danych należące do różnych modułów są umieszczane kolejno w pamięci. Dane i sterta łącznie nie mogą zajmować więcej niż 1MB. Adres segmentu kodowego jest ustalany w momencie odwołania do funkcji zdefiniowanej w danym module. Podobnie adres segmentu danych jest ustalany w momencie odwołania do danej zdefiniowanej w odpowiednim segmencie. Stos jest niezależnym segmentem o rozmiarze do 64KB. Istnieje tylko daleka sterta zmiennych dynamicznych. Wskaźniki kodu i danych są dalekie.

W modelu HUGE można tworzyć programy, w których łączny obszar danych statycznych przekracza 64KB. Na przykład

Moduł_1.cpp

```
char tab[40000];        // tablica statyczna o rozmiarze 40000 bajtów
```

Moduł_2.cpp

```
int w[30000];           // tablica statyczna o rozmiarze 60000 bajtów
```

Glowny.cpp

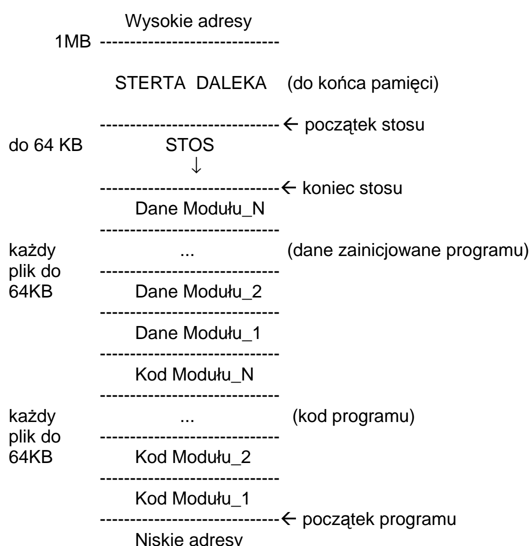
```
extern char tab[];      // deklaracja tablicy zewnętrznej typu char
extern int w[];         // deklaracja tablicy zewnętrznej typu int

void main()
{        tab[0] = 5;     w[0] = 7;     }
```

projekt: Glowny.prj

Moduł_1.cpp
Moduł_2.cpp
Glowny.cpp

Struktura pamięci w modelu HUGE



W przypadku kompilatora BCW dla Windows występują modele: SMALL, MEDIUM, COMPACT i LARGE. Korzystanie z pamięci w trybie chronionym umożliwiają funkcje pokrewne z GlobalAlloc,

Niezależnie od modelu pamięci w kompilatorze BC++3.1 wskaźniki definiowane jako:

- **near** (np. double near *y) są zawsze bliskie,
- **far** (np. double far *x, int far *d) są zawsze dalekie,
- **huge** (np. double huge *u) są zawsze dalekie znormalizowane, tzn. ich offsety należą zawsze do przedziału od 0 do 15 (przy wszelkich operacjach na wskaźnikach tego typu następuje automatyczna korekcja offsetu i segmentu).

W przypadku programów wielomodułowych istnieje możliwość łączenia programów skompilowanych w różnych modelach. Można na przykład skompilować jeden moduł (mod_1) w modelu COMPACT w celu zapewnienia bliskich (szybkich) odwołań między funkcjami modułu, natomiast drugi moduł (mod_2) w modelu LARGE w celu udostępnienia jego funkcji innym modułom. W przypadku odwołania z modułu mod_1 do funkcji mfun(), znajdującej się w module mod_2, należy zapewnić, aby było to wywołanie dalekie. Wystarczy w tym celu umieścić w module mod_1 prototyp funkcji informujący, że jest ona daleka:

```
mod_1.cpp               // skompilowany w modelu COMPACT

int far mfun();         // prototyp dalekiej funkcji z modułu mod_2;


mod_2.cpp               // skompilowany w modelu LARGE

int mfun() { return –1; }  // definicja funkcji mfun()
```

Biblioteki standardowe są kompilowane zwykle w modelu LARGE lub HUGE, aby zapewnić możliwość dalekich odwołań do zawartych w nich funkcji niezależnie od modelu przyjętego podczas kompilacji modułu.

Odczytanie wartości segmentu i offsetu wskaźnika umożliwiają funkcje:

```
unsigned FP_SEG(void far *wsk);     // segment wskaźnika
unsigned FP_OFF(void far *wsk);     // offset wskaźnika
```

Utworzenie wskaźnika o określonym segmencie i offsecie umożliwia funkcja: void far *MK_FP(unsigned segment, unsigned offset).

Adres logiczny zawarty we wskaźniku void *wsk można wyprowadzić na ekran w postaci szesnastkowej *ssss* : *oooo* (*segment* : *offset*) za pomocą funkcji printf, korzystając z formatu %p. Sposób wyprowadzania zależy od modelu. Na przykład w modelach TINY, SMALL, MEDIUM printf(”%p”,wsk) wyprowadzi wskaźnik w postaci *oooo*, natomiast w modelach COMPACT, LARGE i HUGE wszystkie wskaźniki niezależnie od typu i formatu, są zawsze wyprowadzane w postaci *ssss* : *oooo* (domyślnie tylko wskaźniki dalekie). Wskaźniki jawnie zdefiniowane jako bliskie, np. void near *p, należy wyprowadzać w formacie %Np, natomiast wskaźniki jawnie zdefiniowane jako dalekie, np. void far *q, należy wyprowadzać w formacie %Fp.