Wykład 9

9. Wskaźniki i zmienne dynamiczne

Wskaźnik jest zmienną, która służy do przechowywania programowego adresu:

- innej zmiennej (zmiennej prostej, tablicy, obiektu),
- komórki pamięci,
- funkcji.

W systemach 16-bitowych (np. kompilator BC++ 3.1) wskaźniki zawierały adresy logiczne zapisane w postaci *segment : offset*, gdzie część segmentowa określała numer bloku pamięci o rozmiarze 64 KB względem, którego liczono przesunięcie offset. Na podstawie adresów logicznych zmiennych i funkcji wyznaczane były ich adresy fizyczne. Wyprowadzenie wskaźnika na ekran powodowało wyświetlenie segmentu i po dwukropku jego offsetu (segment : offset). W kompilatorach 32-bitowych VC++, BuilderC++ przy wyprowadzaniu wskaźnika wyprowadzany jest płaski, 32-bitowy adres logiczny przeliczany przez system na adres fizyczny.

9.1. Klasyfikacja wskaźników

Wskaźniki dzielą się na:

- wskaźniki danych (zawierają adresy zmiennych),
- wskaźniki kodu (zawierają adresy funkcji).

W systemach 16-bitowych wskaźniki mogły być **dalekie** (czterobajtowe), definiowane z wykorzystaniem słowa kluczowego far i złożone z dwubajtowej części segmentowej i dwubajtowej części offsetowej – dane typu unsigned (np. void far *daleki), lub **bliskie** (dwubajtowe), definiowane za pomocą słowa kluczowego near i złożone tylko z dwubajtowego offsetu (np. void near *bliski). W przypadku braku słowa near lub far (ANSI C) rodzaj wskaźnika zależał od przyjmowanego w kompilatorze modelu pamięci (np. tiny, small, medium).

Na przykład w systemie BC++ można było definiować wskaźniki typu

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

W kompilatorach VC++ i BuilderC++ tego typu podział nie występuje.

9.2. Adresy logiczne i fizyczne

W systemach 16-bitowych wskaźniki zawierają adresy logiczne, które mają następującą postać:

adres_logiczny = segment : offset.

Interpretacja części segmentowej i offsetowej zależy od **trybu adresowania**.

Sposób wyznaczania adresu fizycznego na podstawie znajomości adresu logicznego (segmentu i offsetu – dane typu unsigned) zależy od trybu adresowania.

W trybie **adresowania rzeczywistego** (aplikacje dla systemu Dos; kompilator BC++3.1) segment określa numer bloku pamięci o rozmiarze 64KB względem, którego liczone jest przesunięcie określone przez offset. Każdy taki blok może zaczynać się od adresu fizycznego (adresu bazowego) podzielnego przez 16 (przykładowe adresy bazowe 0, 16, 32, 48, itd.). Adres bazowy jest równy 16∗segment, natomiast adres fizyczny oblicza się ze wzoru 16∗segment + offset. Na przykład wskaźnik 0xb800:0x0001 reprezentuje adres fizyczny równy 0xb8001.

W trybie **adresowania wirtualnego** (aplikacje dla systemu Windows; kompilator BCW) segment jest numerem selektora, który identyfikuje deskryptor definiujący adres początku (adres bazowy) bloku pamięci związanego ze wskaźnikiem. Selektor określa pozycję deskryptora bloku w tablicy deskryptorów, natomiast offset – przesunięcie w bloku. Na podstawie selektora obliczany jest adres bazowy, a następnie wirtualny adres liniowy wg wzoru: adres_bazowy + offset. Adres liniowy jest przeliczany przez system na adres fizyczny.

Wskaźniki bliskie umożliwiają dostęp do komórek pamięci w obrębie bloku 64KB (zmiana offsetu od 0 do 0xFFFF). W trybie adresowania rzeczywistego wskaźniki dalekie umożliwiają dostęp do komórek pamięci o adresach z zakresu od 0 do 0xFFFFF (1 MB), natomiast w trybie adresowania wirtualnego dostęp do komórek pamięci o adresach 32-bitowych, tj. od 0 do 0xFFFFFFFF (4GB).

W kompilatorach 32-bitowych VC++ oraz BuilderC++ występuje tryb adresowania wirtualnego a wskaźniki zawierają wirtualne, 32-bitowe **adresy logiczne** określone w przestrzeni 4 gigabajtowej.

9.3. Definiowanie wskaźników

Wskaźnik wsk służący do przechowywania adresu zmiennej typu typ definiuje się następująco:

```
typ *wsk;          // wskaźnik do zmiennej typu typ
```

Definicję wskaźnika rozpoznajemy po * i czytamy od prawej do lewej (wsk jest nazwą wskaźnika * do zmiennej typu typ).

W szczególności:

```
char  *wsk_c;          // wskaźnik do zmiennej typu char
unsigned char *wsk_uc; // wsk. do zmiennej typu unsigned char
int   *wsk_i;          // wskaźnik do zmiennej typu int
long  *wsk_l;          // wskaźnik do zmiennej typu long
float  *wsk_f;         // wskaźnik do zmiennej typu float
double  *wsk_d;        // wskaźnik do zmiennej typu double
```

Wskaźnik danego typu powinien być wykorzystywany do przechowywania adresów zmiennych tego samego typu.

W celu inicjacji wskaźnika należy przypisać mu adres zmiennej wykorzystując operator &, który podaje adres swojego argumentu. Jeżeli x jest zmienną, to &x jest **stałą** określającą adres (położenie w pamięci) zmiennej x. Stała &x może być wykorzystana do zainicjowania wskaźnika pokazującego na początek bloku pamięci, w którym przechowywana jest zmienna x. Na przykład:

```
int x = 5;          // zmienna x typu int zainicjowana stałą 5
int *px = &x;       // zmienna wskaźnikowa px zainicjowana stałą &x

float v = 1.2;      // zmienna v typu float zainicjowana stałą 1.2

// px = &v;         // błąd – nie można inicjować wskaźnika do typu int
                    // adresem (wskaźnikiem) do typu float (&v jest typu float *)
```

Jeżeli w miejscu deklaracji wskaźnika jego wartość początkowa nie zostanie określona (ustawiona na adres zmiennej odpowiedniego typu), to zostanie on zainicjowany automatycznie zgodnie z regułami obowiązującymi dla zmiennych.

W szczególności, wskaźniki statyczne i zewnętrzne są domyślnie inicjowane zerami (wsk = NULL), natomiast wartości początkowe wskaźników lokalnych nie są określone (są wartościami przypadkowymi). Wygodnie jest inicjować wskaźniki lokalne wartością NULL, gdyż można taki przypadek łatwo wykryć w programie. Na przykład: if (wsk!=NULL) { wykonaj instrukcje; }.

W kompilatorze BC++3.1 istnieją specjalne funkcje, które umożliwiają odczytanie wartości segmentu i offsetu wskaźnika:

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

Utworzenie w systemie BC++3.1 dalekiego wskaźnika o określonym segmencie i offsecie umożliwia funkcja:
void far *MK_FP(unsigned segment, unsigned offset).
Np. void far *d = MK_FP(0xB800, 0x0000);

```
unsigned  s = FP_SEG(d);   // s = 0xB800;
unsigned  o = FP_OFF(d);   //  o = 0x0000;
```

W innych kompilatorach możemy ustawić wskaźnik na określoną wartość: void *d = (void *) 0xB8000000;

9.4. Wskaźniki i zmienne wskazywane

Wartością wskaźnika jest adres innej zmiennej. W celu odczytania zawartości zmiennej (określonego typu) wskazywanej przez wskaźnik należy użyć operatora wyłuskania *. Jeżeli px jest wskaźnikiem na zmienną int x, to można wykonywać operacje z udziałem zmiennej x posługując się wyrażeniem *px, które jest l-wartością i może stać po lewej stronie operatora przypisania.

```
int x = 7;    int a = 2;    int *px = &x;

*px = 8;                    // x = 8
 a = *px;                   // a = x = 8
 a = 30;                    // a = 30
*px = a;                    // x = 30
```

```
printf("%d", *px);          // pisz zawartość zmiennej *px typu int
                            // wskazywanej przez px;
printf("%d %d", *px, x);    // dwa razy jest wyprowadzane x = 30
x = 5;                      // x = 5
cout << *px << ' ' << x << endl;    // dwa razy jest wyprowadzane x = 5
```

Uwaga!
Nie wolno podstawiać wartości do komórki pamięci wskazywanej przez wskaźnik, który nie jest zainicjowany !

```
int *py;            // wskaźnik bez inicjacji zawiera 0 (NULL) lub
                    // adres przypadkowy
*py = 7;            // podstawienie może prowadzić do błędu w programie !
```

Każdy wskaźnik jest zmienną, która posiada swój adres programowy. Na przykład, adres wskaźnika int *wsk można zapamiętać bezpośrednio w zmiennej wskazującej na wskaźnik do int , to jest w zmiennej int * *pw.

```
int *wsk;           // wskaźnik do int
int * *pw = &wsk;   // pw zawiera adres zmiennej wskazującej na int
```

9.5. Wskaźniki do stałych

Wskaźnik może służyć do przechowywania adresu stałej. Zmienna typu const nie może być modyfikowana dlatego adres stałej można zapamiętać jedynie we wskaźniku do stałej.

Wskaźniki do stałych mogą zawierać adresy dowolnych zmiennych i mogą być modyfikowane w programie, ale nie można za ich pomocą modyfikować zmiennych wskazywanych.

```
const int stala = 10;       // definicja stałej typu int
const int *wsk_st;          // wskaźnik do stałej typu int

void main() {
int i = 5;
const int *w = &i;      // wskaźnik zainicjowany
wsk_st = &i;            // inicjacja adresem zmiennej automatycznej i

cout << *wsk_st << endl;        // 5
cout << *w << endl;             // 5

// *w = *w + 5;                 // błąd kompilatora !
```

```
// *wsk_st+= 7;                 // nie można modyfikować stałej

wsk_st = &stala;                // wskaźnik inicjowany adresem stałej
w = &stala;                     // wskaźnik inicjowany adresem stałej
cout << *wsk_st << endl;        // 10
cout << *w << endl;             // 10
}
```

Wskaźniki do stałych nie mogą być wykorzystywane do modyfikacji zmiennych wskazywanych, nawet jeśli zawierają adresy zmiennych automatycznych.

9.6. Stałe wskaźniki

Stały wskaźnik to wskaźnik, który zawsze pokazuje na to samo. Wskaźnik tego typu musi być zainicjowany w miejscu definicji - tak jak każda stała. W programie nie można już modyfikować jego wartości.

Za pomocą wskaźnika stałego można jednak modyfikować zawartość zmiennej wskazywanej, ale tylko tej, której adresem wskaźnik został zainicjowany.

```
const  int st = 4;          // stała
int zm = 10;                // definicja zmiennej

int * const w = &zm;        // stały wskaźnik do zmiennej zm
// int * const x = &st;     // błąd – wskaźnik musi wskazywać na stałą
const * const x = &st;      // dobrze - stały wskaźnik do stałej

int i = 7;
int * const wi = &i;        // stały wskaźnik do zmiennej lokalnej i
int * const pz = &zm;       // stały wskaźnik do zmiennej zm

cout << *wi << endl;        // i=7 = i
cout << *pz << *w << endl;  // 10 i 10; zm=10

*wi +=8;                    // i = 7 + 8 = 15
cout << i << endl;          // i =15
// wi = &zm;                // błąd – nie wolno zmieniać stałej wi
*pz += 2;                   // zm = 10 + 2
cout << zm << endl;         // zm = 12
```

Można również zdefiniować stały wskaźnik do obiektu przyjmowanego za stałą (stały wskaźnik do stałej). Wskaźnik tego typu może być jedynie wykorzystywany do odczytu zawartości zmiennej lub stałej, której adres zawiera, i nie może być zmodyfikowany.

```
const int k = 3;    // stała
int a = 11;         // zmienna

// wk, wa – stałe wskaźniki do obiektów stałych
const int * const wk = &k;      // inicjacja adresem stałej k
const int * const wa = &a;      // inicjacja adresem zmiennej a

int b = *wk;                    // b = k = 3
    b = *wa;                    // b = a = 11
```

Wskaźniki stałe i wskaźniki do stałych znajdują zastosowanie podczas przekazywania do funkcji obiektów (zmiennych, tablic, struktur), które są przekazywane za pomocą wskaźników, a nie powinny być modyfikowane wewnątrz funkcji. Na przykład, jeżeli chcemy tylko wyświetlić zawartość zmiennej, to możemy wykorzystać funkcję:

```
void Pisz(const int *z)
{
      // *z +=2;              // błąd kompilatora – modyfikacja stałej !
      cout << *z << endl;
}

void Pisz1(int * const z)
{
       //  z++;              // nie można modyfikować stałego wskaźnika
          *z +=2;           // modyfikacja zmiennej
          cout << *z << endl;
       }
```

9.7. Wskaźniki typu void

Wskaźnik do zmiennej określonego typu zawiera informację o adresie zmiennej oraz jej typie (co ma znaczenie podczas wykonywania operacji arytmetycznych z udziałem wskaźnika oraz interpretacji wskazywanego obszaru pamięci). Z definicji

```
int *wi;            // wskaźnik do obiektu typu int
```

wynika, że wi wskazuje na obszar pamięci o rozmiarze sizeof(int), w którym można zapamiętać zmienną typu int. Można bezpośrednio odwołać się do pamięci o adresie wi za pomocą *wi.

Istnieje możliwość definiowania wskaźników, które wskazują na typ void.

```
void *x;            // wskaźnik na dowolny typ (typ void)
```

Wskaźniki tego typu mogą zawierać adres zmiennej dowolnego typu.

W przypadku definicji

```
int i = 5;    float z = 1.3;    int *wi = &i;    float *wz = &z;
```

możliwe są przypisania x = wi oraz x = wz.

Wskaźnikowi typu void można przypisać wskaźnik dowolnego typu z wyjątkiem wskaźnika do zmiennej typu const.

```
const long s = 4;          // stała s typu long

// void *x = &s;           // błąd – s jest stałą
```

Można jednak wpisać adres obiektu stałego do wskaźnika typu void do stałej: const void *x = &s.

9.8. Rzutowanie wskaźników

W celu przypisania wskaźnika typu void innemu wskaźnikowi należy dokonać konwersji typu wskaźnika (rzutowania wskaźnika) do typu wymaganego. Operacja rzutowania wskaźników jest również niezbędna w przypadku, gdy przypisujemy wskaźnik jednego typu do wskaźnika innego typu.

```
double zm = 11.2;    void *x = &zm;
double *d;   int *wi;

// d = x;                  // błąd – brak zgodności typów
d = (double *) x;          // rzutowanie wskaźnika void * do double *
cout << *d << endl;
wi = (int *) d;            // rzutowanie double * do  int *
```

Wskaźniki typu void umożliwiają przekazywanie do funkcji wskaźników dowolnego (niestałego) typu. Wewnątrz funkcji można dokonać konwersji typu wskaźnika (rzutowania wskaźnika) do innego typu w celu interpretacji wskazywanego obszaru pamięci.

```
void CzytajPamiec(void *x)
 {
      char *c = (char *) x;       // konwersja typu (rzutowanie) wskaźnika
      int *i    = (int *) x;
      long *z = (long *) x;
            / interpretuj pamięć
      cout << *c << endl;    // pobierz sizeof(char) bajtów od adresu c
      cout <<  *i << endl;   // pobierz sizeof(int) bajtów od adresu i
      cout << *z << endl;    // pobierz sizeof(long) bajtów od adresu z
}
```

9.9. Zmienne dynamiczne

Wskaźniki na ogół zawierają adresy zmiennych (obiektów), dla których pamięć jest przydzielana automatycznie przez kompilator na podstawie ich definicji. Dostęp do takich zmiennych odbywa się za pomocą ich identyfikatorów. Zmienne te posiadają pewne własności wynikające z ich zasięgu, łączności i czasu trwania.

Stałe, zmienne statyczne i zewnętrzne mają zarezerwowane miejsce w kodzie wykonywalnym programu (są umieszczane w obszarze danych programu).

Zmienne lokalne (automatyczne) funkcji są umieszczane na stosie w momencie, gdy sterowanie wejdzie do bloku, w którym zostały zdefiniowane. Zmienne tej klasy znikają po wyjściu sterowania z bloku.

Istnieje możliwość dynamicznego tworzenia i kasowania zmiennych w trakcie działania programu. Służą do tego celu operatory **new** i **delete** (język C++) oraz funkcje **malloc**, **calloc** i **free** (język C/C++).

9.10. Operatory new i delete

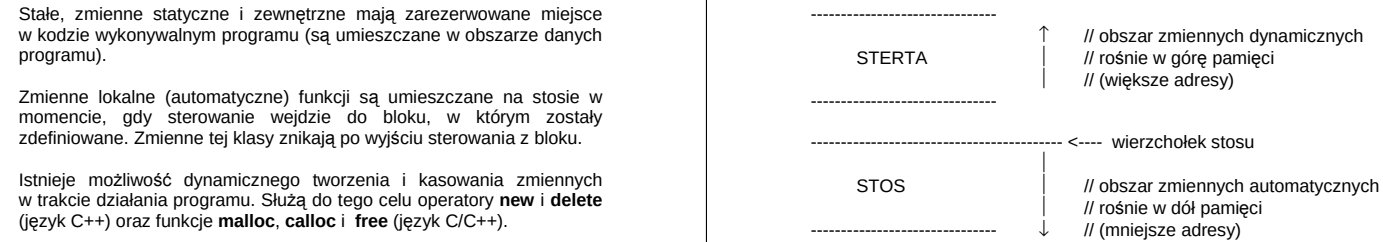
Operator **new** pozwala utworzyć obiekt wskazywany przez wskaźnik określonego typu dynamicznie w trakcie działania programu, czyli utworzyć tzw. zmienną dynamiczną (new zwraca wskaźnik do tworzonego obiektu). Zmienna tego typu jest tworzona w obszarze pamięci nazywanym **stertą** (ang. heap).

W kompilatorze 16-bitowym BC++3.1 w trybie adresowania rzeczywistego (tryb DOS-u) sterta może zajmować pamięć o adresach od 0 do 1 MB. Natomiast w trybie adresowania wirtualnego może być wykorzystywana cała dostępna pamięć w tym obszar powyżej 1 MB. W kompilatorach 32-bitowych wykorzystywane są 32-bitowe adresy logiczne, które są przeliczane na adresy fizyczne uwzględniające fizyczną pamięć RAM oraz wirtualną pamięć dyskową. Fizyczne bloki pamięci mogą być wymieniane w sposób przeźroczysty dla użytkownika pomiędzy RAM a dyskami.

Rozmiar przydzielanej pamięci jest proporcjonalny do rozmiaru tworzonego obiektu (zawsze nie mniejszy niż rozmiar tworzonej zmiennej). Utworzona zmienna nie posiada nazwy, ale jej adres jest podstawiany do wskaźnika odpowiedniego typu. Można więc uzyskać do niej dostęp za pomocą operatora wyłuskania *.

```
int *x = NULL;    // wskaźnik do obiektu typu int – zainicjowany na 0

x = new int;      // utworzenie zmiennej typu int na stercie;
                  // adres początku przydzielonego obszaru pamięci
                  // został wpisany do wskaźnika x;
                  // jeśli alokacja powiodła się, to adres jest
                  // różny od NULL
```



Jeżeli alokacja pamięci powiodła się (na stercie była wystarczająca ilość wolnego miejsca), to wskaźnik x jest różny od NULL. Wówczas, można wpisać do przydzielonego obszaru liczbę typu int i wykorzystywać wyrażenie *x jako zmienną typu całkowitego. Jeżeli alokacja nie powiodła się, to wskaźnik przyjmuje wartość NULL.

```
int a = 7;

if (x != NULL)
{
*x = 5;                    // wpisz 5 do utworzonej zmiennej
cout << *x << endl;        // 5

*x = a;                    // wpisz do obszaru liczbę a
cout << *x << endl;        // 7
}
```

Istnieje możliwość utworzenia obiektu określonego typu za pomocą operatora new i nadania mu wartości początkowej.

```
char *zn =  new char (65);    // utworzenie obszaru typu char
          // i wstawienie do tego obszaru liczby 65 - równoważne *zn = 65;
```

Do likwidacji zmiennej utworzonej na stercie za pomocą **new** służy operator **delete**.

```
delete x;          // zwolnienie obszaru pamięci związanego
                   // ze wskaźnikiem x (obszar typu int)
x = NULL;          // zerowanie po zwolnieniu - zalecane
```

Własności zmiennych utworzonych za pomocą operatora new:

- istnieją od momentu utworzenia operatorem new do momentu likwidacji za pomocą operatora delete;
- nie posiadają nazwy; dostęp do nich jest możliwy wyłącznie za pomocą wskaźników;
- nie podlegają regułom obowiązującym dla zwykłych zmiennych, a dotyczącym zasięgu, łączności i czasu trwania;
- nie są automatycznie inicjowane wartościami początkowymi w momencie utworzenia; przydzielony blok pamięci zawiera wartość przypadkową;
- po zwolnieniu pamięci wskaźnik zmiennej nadal zawiera ten sam adres, ale zwolniona pamięć może być udostępniona dla innych zmiennych (należy uważać, aby nie wstawiać danych do pamięci wskazywanej przez wskaźnik, który wskazuje na zwolniony obszar pamięci).

Uwagi!

- Nie należy stosować operatora delete dla obiektów, które nie zostały utworzone za pomocą new.

- Nie wolno również kasować dwa razy tego samego obiektu. Aby się ustrzec przed taką możliwością wystarczy po skasowaniu obiektu ustawić jego wskaźnik na NULL. Zwalnianie wskaźnika pustego nie powoduje błędu.

- Należy również uważać, aby nie utracić zawartości wskaźnika, który zawiera adres zmiennej utworzonej za pomocą operatora new. W takim przypadku nie będzie można zwolnić przydzielonej pamięci co może doprowadzić do wyczerpania sterty.

Zmienna dynamiczna może być utworzona wewnątrz funkcji. Przydzielony obszar pamięci nie znika po zakończeniu funkcji. Należy jednak zadbać o to, aby funkcja przekazała wskaźnik do utworzonego obszaru na zewnątrz.

Przykład. 9.1. Tworzenie zmiennych dynamicznych i zapamiętywanie ich adresów za pomocą wskaźników.

```
int * Tworz1()
{  return new int; }      // funkcja zwraca wskaźnik do int

void Tworz2(int* *x)      // przekazywanie przez wskaźnik
{  *x = new int; }        // funkcja ustawia wskaźnik do int

void Tworz3(int* &x)      // przekazywanie przez referencję
{  x = new int;   }       // funkcja ustawia wskaźnik do int

void main()
{      int *pw = NULL;          // wskaźnik do int

       pw = Tworz1();   // funkcja zwraca wskaźnik do int
       if (pw)  { *pw = 1;   cout << *pw << endl;        // 1
       delete pw;   pw = NULL; }

       Tworz2(&pw);     // przekazywanie pw przez wskaźnik
       if (pw)  { *pw = 2;   cout << *pw << endl;        // 2
       delete pw;   pw = NULL; }

       Tworz3(pw);      // przekazywanie pw przez referencję
       if (pw)  { *pw = 3;   cout << *pw << endl;        // 3
       delete pw;   pw = NULL; }
}
```

9.11. Funkcje malloc, calloc, free

W języku C/C++ istnieją standardowe funkcje umożliwiające dynamiczną alokację pamięci (malloc i calloc) oraz funkcja zwalniająca przydzieloną pamięć (free). Prototypy funkcji (również ANSI C):

```
void *malloc(size_t  K);       //  alokacja K bajtów

void *calloc(size_t  N, size_t K);    // alokacja N razy po K bajtów

void free(void *x);            // zwolnienie obszaru
```

Typ **size_t** jest zdefiniowany jako **unsigned**.

Funkcje malloc i calloc przydzielają spójne obszary pamięci, których rozmiary nie przekraczają zakresu typu size_t (unsigned) i zwracają wskazanie do przydzielonego obszaru. Jeżeli alokacja nie jest możliwa, to zwracany jest wskaźnik NULL. Funkcja calloc dodatkowo zeruje przydzieloną pamięć. Funkcja free zwraca do systemu przydzieloną pamięć.

Funkcje malloc i calloc zwracają wskaźniki do typu void dlatego niezbędne są konwersje typu przy podstawieniach do wskaźników innych typów. Należy uważać, aby za pomocą funkcji free nie zwalniać pamięci, która nie została przydzielona.

Przykład. 9.2. Dynamiczna alokacja pamięci dla zmiennych z wykorzystaniem malloc i calloc.

```
void main()
{
       int *pw = NULL;          // wskaźnik do int

       pw = (int *) malloc( sizeof(int) );     // alokacja sizeof(int) bajtów
       if (pw)   {  *pw = 1;
                    cout << *pw << endl;                  // 1
                    free(pw);   pw = NULL;  }

       pw = (int *) calloc(1, sizeof(int) );   // 1 blok sizeof(int) bajtowy

       if (pw)  {
        cout << *pw << endl;   // wartość 0 – zawarta
                               // w komórce o adresie pw
       *pw = 2;
       cout << *pw << endl;                               // 2

       free(pw);   pw = NULL;  }
}
```

W kompilatorach 16-bitowych do przydzielania, w obszarze od 0 do 0xFFFFF (1 MB), bloków pamięci o rozmiarze większym niż 64 KB oraz alokowania pamięci w obszarze tzw. *sterty dalekiej*, adresowanej przez wskaźniki typu far (np. void far *p), służą np. w środowisku BC++3.1 odpowiedniki przedstawionych funkcji, tj. *void far *farmalloc(long K)* oraz *void far *farcalloc(long N, long K)*. Do zwolnienia - funkcja *farfree*.

Przykład. 9.3. Dynamiczna alokacja pamięci dla zmiennych z wykorzystaniem malloc i farmalloc.

```
// kompilować w modelu large, huge w BC++ 3.1
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <alloc.h>

void main()
{
 void     *p = NULL;
 void far *q = NULL;

 long a, b;
 a = 65000;
 b = 400000;

 q = farmalloc(b);              // alokacja bloku 400000 bajtow
 cout<< q << endl;              // np. 0x0cef: 0x0004

 p = malloc(a);                 // alokacja bloku 65000 bajtow
 cout << p << endl;             // np. 0x6e98 : 0x0004

 free(p);       p = NULL;     // zwolnienie pamięci w odwrotnej kolejności
 farfree(q);    q = NULL;

 getch();
 }
```

W kompilatorach 16-bitowych dostęp do bloku pamięci o rozmiarze większym niż 64 KB wymaga zmiany numeru segmentu wskaźnika po każdym przejściu przez granicę bloku o rozmiarze 64 KB lub wykorzystania wskaźników typu huge, dla których zmiany takie są realizowane automatycznie.

W kompilatorach 32-bitowych VC++ oraz BuilderC++ przesunięcie wskaźnika o dowolny offset automatycznie prowadzi do zmiany płaskiego adresu logicznego, który może być traktowany w cztero-gigabajtowej przestrzeni adresowej programu tak jak adres fizyczny.