Wykład 11

11. Wskaźniki a tablice jednowymiarowe

Dostęp do elementów tablic można zrealizować za pomocą wskaźnika zawierającego adres pierwszego elementu tablicy i korzystając z zasad arytmetyki na wskaźnikach.

11.1. Arytmetyka wskaźników

Wskaźnik zawiera informacje o adresie obiektu, na który wskazuje oraz o typie obiektu. Do wskaźnika typu różnego niż void można dodać (odjąć) liczbę całkowitą.

Dodanie do wskaźnika **px** typu **tdana** liczby całkowitej 1 spowoduje zwiększenie offsetu wskaźnika o rozmiar obiektu wskazywanego, tzn. o wartość **sizeof(tdana)**. Na przykład,

```
long a = 123;
long *wl = &a;          // wskaźnik na obiekt typu long

printf("%p\n", wl);     //  BC++ 3.1   0x8F00 : 0x0000 – segment : offset
printf("%p\n", wl);     //  VC++ i BBC++  - adres logiczny 0x0086580C
wl = wl + 1;        //  BC++3.1 - offset wskaźnika zwiększa się o
    //  sizeof(long)=4; tj. 0x8F00: 0x0004;
    //  VC++ i BBC++  - adres logiczny zwiększa się o 4, tj. 0x00865810
```

Operacja dodania stałej całkowitej n do wskaźnika px typu tdana

px = px + n;

prowadzi do następującej zmiany wartości wskaźnika

BC++3.1 : **offset = offset + n * sizeof(tdana)**

VC++, BuilderC++: **adr_log = adr_log + n * sizeof(tdana)**

Na wskaźnikach typu różnego niż void można wykonywać następujące operacje arytmetyczne:

- dodać lub odjąć liczbę całkowitą;
- odjąć od siebie dwa wskaźniki tego samego typu;
- porównać wskaźniki tego samego typu (<, >, <=, >=);
- przyrównać wskaźniki tego samego typu (==, !=);
- przyrównać wskaźnik do wskaźnika pustego NULL.
  (jeśli operacje będą na wskaźnikach różnych typów, to kompilator może sam dokonać konwersji typów i wtedy nie zgłosi błędu)

W kompilatorze BC++3.1 w przypadku wskaźników bliskich i dalekich operacje arytmetyczne +, -, +=, -=, ++, -- oraz porównania <, <=, >, >= są realizowane tylko na offsecie. Porównanie wskaźników sprowadza się do porównania ich offsetów traktowanych jako liczby typu unsigned.

W systemie BC++3.1 operacje przyrównania == oraz != są zawsze wykonywane na całym wskaźniku. Dwa wskaźniki są równe tylko wtedy, gdy mają identyczne segmenty i offsety.

W kompilatorach VC++ oraz BuilderC++ operacje arytmetyczne na wskaźnikach oraz przyrównania wskaźników są realizowane na adresach logicznych zapisanych w tych wskaźnikach.

```
char * w_1 = (char *) 0xB8000010;

char * w_2 = (char *) 0xB8010000;
```

Wskaźniki w_1 i w_2 wskazują na różne adresy logiczne i dlatego z punktu widzenia operacji przyrównania są różne.

```
int w = (w_1 == w_2);          // w = 0

if (w) cout << "w_1 == w_2" << endl;    // wskaźniki są różne
else  cout << "w_1 !=  w_2" << endl;
```

Porównanie wskaźników:

```
w = (w_1 < w_2);               // w = 1;  porównywane są adresy

if (w) cout << "w_1 < w_2" << endl;  // nierówność jest spełniona
```

Dodanie liczby całkowitej do wskaźnika prowadzi w kompilatorze BC++3.1 tylko do zmiany offsetu wskaźnika:

```
w_2 = w_2 + 0xFFFF;        // w_2 = 0xB801:0x0000
printf("%p", w_2);         //  0xB801 : 0xFFFF

w_1 = w_1 + 0xFFFF;        // w_1 = 0xB800:0x0010
printf("%p", w_1);         //  0xB800 : 0x000F (0x0010 + 0xFFFF)
```

Przekroczenie zakresu offsetu (typ unsigned) nie prowadzi w przypadku wskaźników typu near i far do automatycznej modyfikacji segmentu (BC++ 3.1).

W kompilatorach 32-bitowych dodanie do wskaźnika liczby całkowitej powoduje zmianę adresu logicznego o tę wartość przemnożoną przez rozmiar obiektu wskazywanego.

```
char * w_1 = (char *) 0xB8000010;

w_1 = w_1 + 0xFFFF;

printf("%p", w_1);             //  w_1 = 0xB801000F – adres logiczny
```

Odejmowanie wskaźników tego samego typu wyznacza liczbę całkowitą obiektów między tymi wskazaniami.

```
long *a1 = (long *) 0xB8000000;
long *a2 = (long *) 0xB8000010;

a1 = a2;                   // 0xB8000010

a1 += 3;                   // 0xB800001C ;   0x0010 + 3*sizeof(long)

w = a1 – a2;               // w = 3
```

Operacje odejmowania, porównywania i przyrównywania dwóch wskaźników są najczęściej wykonywane na wskazaniach dotyczących elementów tej samej tablicy.

```
long tab[10];              // tablica

a1 = &tab[0];
a2 = &tab[9];

int w = a2 – a1;           // w = 9

w = a2 > a1;               // w = (a2 > a1) – priorytet operatora > jest
                           // większy niż operatora =;   w = 1
```

W kompilatorze BC++3.1 w przypadku wskaźników typu huge **wszystkie** operacje są wykonywane na całych wskaźnikach. Po wykonaniu operacji prowadzących do zmiany wskaźników są one normalizowane w taki sposób, aby ich offsety należały do przedziału od 0 do 15. W celu zachowania adresów fizycznych wskazywanych przez wskaźniki są również modyfikowane ich segmenty. W przypadku zwiększania wskaźnika typu huge po przekroczeniu zakresu offsetu segmenty są zmieniane automatycznie.

```
long huge * x1 = (long huge *) MK_FP(0xB800, 0x0010);
long huge * x2 = (long huge *) MK_FP(0xB801, 0x0000);

printf("%Fp", x1);              // 0xB800 : 0x0010

x1++;                           // 0xB800 : 0x0014  = 0xB801 : 0x0004

printf("%Fp", x1);              // 0xB801 : 0x0004 – po normalizacji
```

W przypadku wskaźników typu huge po przekroczeniu zakresu offsetu (wartości 15) automatycznie zwiększany jest segment.

```
x1 = x1 + 6; // x1 = 0xB801: 0x0004 + 6*sizeof(long) = 0xB802:0x000C;
```

Operacje porównywania i przyrównywania dotyczą w przypadku wskaźników typu huge wskazywanych adresów fizycznych.

```
long huge * x1 = (long huge *) MK_FP(0xB800, 0x0010);  // 0xB8010
long huge * x2 = (long huge *) MK_FP(0xB801, 0x0000);  // 0xB8010
```

Dla trybu rzeczywistego adresy fizyczne wskazywane przez x1, x2 są takie same i równe 0xB8000 + 0x0010 = 0xB8010 + 0x0000 = 0xB8010.

```
int w = (x1 == x2);             // w = 1 - równość dla wskaźników huge

x2++;       w = (x2 > x1);      // x2 = 0xB801: 0x0004;    w = 1

x1++;       w = (x2 > x1);      // x1 = 0xB801: 0x0004;    w = 0
```

W przypadku kompilatorów 32-bitowych VC++ oraz BuilderC++ istnieje jeden rodzaj wskaźników a wszystkie operacje dotyczą liniowych adresów logicznych.

```
#include <stdio.h>      #include <stdlib.h>
#include <memory.h>   #include <conio.h>

void main()
{   // VC++6.0 i BuilderC++6.0 - wskaźniki zawierają liniowe adresy
// logiczne na podstawie, których wyznaczane są adresy fizyczne;
// operacje na wskaźnikach dotyczą 32-bitowych, płaskich
// adresów logicznych; jeden wskaźnik definiuje adres liniowy w
// przestrzeni adresowej aplikacji z zakresu od 0 do 4 GB

char *s = (char *) malloc(1);
char *w = (char *) malloc(1000000);

int *r = (int *)s;
char *x = s;

printf("%p %lu\n", s, (unsigned long) s);
printf("%p %lu\n", w, (unsigned long) w);

s = s + 130000;  // + 0x1FBD0 = 130 000 do adresu logicznego
r = r + 100;     // + 4*100 = 400 do adresu logicznego

printf("\nPocz =      %p %lu", x, (unsigned long)x);
printf("\n + 130000 %p %lu", s, (unsigned long)s);
printf("\n + 400       %p %lu", r, (unsigned long)r);

int k = s - x;              printf("\n%d", k);     // 130 000
k = (char *) r - x;         printf("\n%d", k);     // char - 400
k = r - (int *) x;          printf("\n%d", k);     // int - 100

/* wyniki
008657EC 8804332  - adres logiczny s
008657FC 8804348  - adres logiczny w

Pocz =      008657EC 8804332 - adres pocz
 + 130000 008853BC 8934332 - zwiększone o 130 000  = s
 + 400       0086597C 8804732 - zwiększone o        400  = r
130000  - odległość (s - pocz) w char
400        - odległość (r - pocz) w char
100        - odległość (r - pocz) w int       */
getch();
}
```

11.2. Dostęp do elementów tablic

W języku C/C++ nazwa tablicy jest **stałą wskaźnikową**, która zawiera adres pierwszego elementu tablicy. Stała wskaźnikowa jest wskazaniem na typ elementu tablicy. Na przykład, dla tablicy

```
int tab[5] = {0, 1, 2, 3, 4};
```

stała tab jest typu int *, gdyż elementy tablicy są typu int.

Zgodnie z zasadami arytmetyki wskaźników wyrażenie (tab + i) wskazuje i-ty element typu int za elementem wskazywanym przez tab. Spełnione są następujące zależności:

```
tab == &tab[0];          //  tab jest stałą == adresowi elementu tab[0]

(tab + i ) == &tab[i];   // wyrażenie (tab + i) == adresowi &tab[i]

*(tab + i) == tab[i];    // wyrażenie *(tab + i) == tab[i]

int k = sizeof(tab)/sizeof(int);          // rozmiar tablicy k = 5

i = 0;
int *wsk = tab;          // lub  wsk = &tab[0];

while ( i < k )          {
  printf("*wsk = %2d i = %d\n",*wsk++, i++);
}
```

W przypadku tablic znakowych można zainicjować wskaźnik adresem tekstu bezpośrednio w programie.

```
char *wt = "To jest C";    // automatyczne przydzielenie pamięci dla
                           // tekstu przez system
```

Analogicznie, w momencie definicji tablicy można napisać:

```
char tekst[] = "To jest C";
```

Nie jest możliwe ponowne przypisanie w postaci:

```
tekst = "To jest tekst";
```

Zmienna wt zawiera adres pierwszego elementu statycznego łańcucha "To jest tekst", a więc wt == &tekst[0] i *wt == 'T'. Wynika stąd, że

```
wt + 1 == tekst + 1;    *(wt+1)== 'o';
wt + 3 == tekst + 3;    *(wt+3)== 'j';
```

Wyrażenia wt+1 oraz tekst+1 wskazują na znak 'o' przesunięty o jeden względem początku łańcucha. Analogicznie wyrażenia wt+3 oraz tekst+3 wskazują na znak 'j' przesunięty o trzy względem początku łańcucha.

11.3. Metody przekazywania tablic do funkcji

Tablicę można przekazać do funkcji na kilka sposobów.

- Podając jako argument formalny funkcji nazwę tablicy oraz jej rozmiar.

```
typedef int ttab[5];

void pisz1(int tab[5], int zakres)
{
      for (int i = 0; i < zakres; i++)  cout << tab[i] << endl;
}

void pisz2(ttab tab, int zakres)
{
      for (int i = 0; i < zakres; i++) cout << tab[i] << endl;
}
```

W przypadku tablicy int t[ ] = { 0, 1, 2, 3, 4 } wywołania funkcji mają następującą postać: pisz1(t, 5) oraz pisz2(t, 5). Nazwa tablicy jest adresem pierwszego elementu. Dlatego poprawne są wywołania w postaci: pisz1(&t[0], 5) oraz pisz2(&t[0], 5).

- Podając jako argument formalny funkcji nazwę tablicy (bez podawania jej rozmiaru).

```
void pisz3(int tab[ ], int zakres)
{
      for (int i = 0; i < zakres; i++) cout << tab[i] << endl;
}
```
Wywołanie funkcji: pisz3(t, 5).

W języku C mimo iż do funkcji jest przekazywana cała tablica na stos nie są kopiowane poszczególne jej elementy, a jedynie adres pierwszego elementu. Wymienione metody sprowadzają się więc do przekazywania tablicy do funkcji za pomocą wskaźnika do typu elementu tablicy (w tym przypadku wskaźnika do int).

- Przekazując jako parametr funkcji wskaźnik do typu tablicowego

```
void pisz4(int *t, int rozmiar)
{ int *wsk = t;                          // ustawienie wskaźnika
  for (int i=0; i<rozmiar; i++)        // wydruk przez indeks t[i] oraz wsk
  cout << t[i] << " = " << *wsk++ << endl;
}
```
Wywołanie funkcji: pisz4(t, 5).

Jeśli wartość t[i] nie jest wykorzystywana, to można pominąć wskaźnik wsk i odwoływać się tylko do t, modyfikując jego wartość. Zmiany t są aktualne tylko wewnątrz funkcji, nie ma więc obawy, że zmienią adres zawarty we wskaźniku podstawianym za t.

```
void pisz5(int *t, int rozmiar)
{
  for (int i=0; i<rozmiar; i++)        // wydruk przez *t++
  cout << *t++ << "  ";
}
```
Wywołanie funkcji: pisz5(t, 5).

Wyniki dla pisz5: 0 1 2 3 4

11.4. Operacje na tablicach liczbowych

Za pomocą wskaźników można zrealizować dostęp do tablic. Operacje na tablicach często wykorzystują operatory ++, -- oraz *. Operatory ++ oraz -- (poprzednikowe) mają taki sam priorytet jak operator wyłuskania *, ale wiązanie prawostronne (są wykonywane od prawej do lewej). Pozwala to jednoznacznie określić kolejność wykonywania operacji arytmetycznych na wskaźnikach.

```
long c[5] = {0, 1, 2, 3, 4 };          // tablica 5 liczb typu long
long *x = &c[0];                       // x - wskaźnik do long
long d;
printf("Wskaznik x = %p\n", x);      // wskaźnik x = 0012FF48
```

Wartości i adresy poszczególnych elementów tablicy:

```
for (int k=0; k<5; k++)
printf("c[%d] = %ld = *x++ = %ld  -  adres = %p\n", k, c[k], *x++, &c[k]);
```

```
c[0] = 0 = *x++ = 0  -  adres = 0012FF48
c[1] = 1 = *x++ = 1  -  adres = 0012FF4C
c[2] = 2 = *x++ = 2  -  adres = 0012FF50
c[3] = 3 = *x++ = 3  -  adres = 0012FF54
c[4] = 4 = *x++ = 4  -  adres = 0012FF58
```

Operacja d = *x++ powoduje podstawienie za d wartości *x, a następnie zwiększenie x++ (x = x+1). Operator następnikowy ++ jest wykonywany po dokonaniu przypisania i dotyczy, od prawej do lewej, argumentu x, a nie *x. Ostatecznie d == *x == c[0] == 0 i x++. Po operacji wskaźnik x zawiera adres elementu c[1].

```
d = *x++;

printf("x = %p  d = %ld\n", x, d);    // x = 0012FF4C;  x == &c[1];   d == 0
```

W przypadku wątpliwości jak działają operatory należy stosować nawiasy. Wykorzystując nawiasy można otrzymać czytelniejszą postać poprzedniego wyrażenia:

```
d = *(x++);           // d = *x;   d == c[1] == 1;   x++;   x == &c[2]

printf("x = %p  d = %ld\n", x, d);    // x = 0012FF50;  x == &c[2];  d == 1
```

W celu zwiększenia zawartości komórki pamięci wskazywanej przez x należy użyć wyrażenia d = ++*x. Wykonanie operacji prowadzi do (od prawej do lewej) wyłuskania *x, zwiększenia ++(*x) (zwiększenie elementu c[2]=2) i podstawienia wartości wyrażenia do zmiennej d.

```
d = ++*x;           //  ++(*x) == ++c[2] == 3, tj. c[2]=c[2]+1= 2 + 1 = 3

printf("x = %p  d = %ld\n", x, d);    // x == 0012FF50;   d == c[2] == 3
```

W przypadku kolejnego wyrażenia za d jest podstawiana wartość *x, a następnie wartość (*x), czyli element c[2] == 3, jest zwiększana o jeden (c[2] == 4).

```
d = (*x)++;          // d = *x;   d == c[2] == 3;    (*x)++ == c[2]++ == 4

printf("x = %p  d = %ld\n", x, d);    // x == 0012FF50;   d == 3
```

W ostatnim wyrażeniu najpierw zwiększana jest wartość x (ustawienie wskaźnika na adres &c[3]), potem następuje odczytanie zawartości *x i podstawienie jej do zmiennej d.

```
d = *++x;           // ++x == &c[3];  d = c[3];    d == 3

printf("x = %p  d = %ld\n", x, d);    // x == 0012FF54;       d == 3
```

Zawartość tablicy: c[0] == 0; c[1] == 1; c[2] == 4; c[3] == 3; c[4] == 4.

Wartość końcowa x == &c[3] == 0012FF54.

Przykład. 11.1. Wyprowadzanie tablicy liczb całkowitych za pomocą wskaźnika. Elementy są wyprowadzane aż do napotkania pierwszego 0.

```
int tab[ ] = {-4, 1, -5, 2, -3, 0, 7, 8};
int k = sizeof(tab)/sizeof(int);                  // rozmiar tablicy k = 8
i=0;
int *wsk = &tab[0];

while(*wsk!=0 && i<k)          // priorytet wyłuskania (*) jest większy niż !=
{
  printf("*wsk = %2d i = %d\n",*wsk,i);
  wsk++;   i++;
}
```

W wyniku działania programu na ekranie pojawi się:
```
*wsk = -4  i = 0
*wsk =  1  i = 1
*wsk = -5  i = 2
*wsk =  2  i = 3
*wsk = -3  i = 4
```

Wynik wyrażenia (*wsk!=0) jest porównywany z zerem, przy sprawdzaniu (*wsk!=0 && i<k), dlatego można je zastąpić wyrażeniem *wsk.

```
while(*wsk && i<k)
{
  printf("*wsk = %2d i = %d\n",*wsk,i);   wsk++;   i++;
}
```

Przykład. 11.2. Wyprowadzanie za pomocą wskaźnika elementów tablicy typu double różnych od zera.

```
double ftab[ ] = {-1.2, 1, -5.2, -0.7, -3.5, 0, 2.7, 3.8};

k = sizeof(ftab)/sizeof(double);                  // rozmiar tablicy;  k = 8
i=0;
double *fwsk = &ftab[0];

while(i<k) {
  if (*fwsk) printf("*fwsk = %4.1lf i = %d\n",*fwsk,i);
  fwsk++;   i++;   }
```

W wyniku działania programu na ekranie pojawi się:

```
*fwsk = -1.2  i = 0
*fwsk =  1.0  i = 1
*fwsk = -5.2  i = 2
*fwsk = -0.7  i = 3
*fwsk = -3.5  i = 4
*fwsk =  2.7  i = 6
*fwsk =  3.8  i = 7
```

Element ftab[5] == 0 został pominięty.

Dostęp do tablic za pomocą wskaźników jest szybszy niż dostęp realizowany za pomocą indeksów.

W przypadku posługiwania się wyrażeniem tab[i] za każdym razem jest obliczane wyrażenie &tab[0] + i*sizeof(int), które określa adres i-tego elementu tablicy.

```
for (i=0; i<8; i++) printf("tab[%d] = %3d\n", i, tab[i]);
```

Natomiast, w przypadku posługiwania się wskaźnikiem int *wsk=tab realizowana jest tylko operacja wsk++, która powoduje przesunięcie wskaźnika do kolejnej pozycji w tablicy. W przypadku programu:

```
for (i=0, wsk=&tab[0]; i<8; i++) printf("*(wsk+%d) = %3d\n", i, *wsk++);
```

Na ekranie pojawi się:

```
*(wsk+0) =  -4
*(wsk+1) =   1
*(wsk+2) =  -5
*(wsk+3) =   2
*(wsk+4) =  -3
*(wsk+5) =   0
*(wsk+6) =   7
*(wsk+7) =   8
```

11.5. Operacje na tekstach

Wskaźniki pozwalają znacznie skrócić operacje na łańcuchach.

```
char lan[] = "To jest C++";   // 11 znaków + znak '\0' razem 12 znaków
char bufor[20];
unsigned h = 0;
char *src = &lan[0];
char *dst = &bufor[0];

h = strlen(src);       // strlen zwraca liczbę znaków tekstu - bez znaku '\0'
printf("Dlugosc lancucha = %d  %d\n", h, strlen(lan) );    // h == 11
```

Zawartość łańcucha lan można przekopiować do łańcucha bufor na kilka sposobów.

Kopiowanie z wykorzystaniem indeksów tablic

```
for (i=0; i<=h; i++) bufor[i] = lan[i];          // dla i == h kopiowany znak '\0'
```

Nazwa tablicy jest *stałym wskaźnikiem* do jej pierwszego elementu. Można więc wyprowadzić teksty za pomocą

```
printf("lan = %s\nbufor = %s\n", lan, bufor);
```

lub uwzględniając, że src = &lan[0] oraz dst = &bufor[0]

```
printf("lan = %s\nbufor = %s\n", src, dst);
```

Kopiowanie z wykorzystaniem wskaźników

```
char *src = &lan[0];
char *dst = &bufor[0];
```

Wariant 1.

```
while (*src!=0) {
   *dst = *src;              // przypisanie
    src++;  dst++;           // zmiana wskaźników
 }
   *dst = '\0';       // dodanie 0 na końcu
    printf("lan = %s\nbufor = %s\n", lan, bufor);
```

Wariant 2.

```
while (*src) {  *dst++ = *src++;   }
*dst = '\0';       // dodanie 0 na końcu
printf("lan = %s\nbufor = %s\n", lan, bufor);
```

Wariant 3.

W przypadku wyrażenia (*dst++ = *src++), którego wartością jest *src, najpierw realizowane jest podstawienie *dst = *src, a następnie obliczane są wartości wyrażeń (*dst = *src) oraz dst++ i src++. Rozpatrywane wyrażenie może być wykorzystane bezpośrednio do kopiowania tekstów i sterowania pętlą. Umożliwia ono kopiowanie łącznie ze znakiem '\0'.

```
printf("%p  %c   ", src, *src);        // adres pierwszego znaku i znak
printf("%p\n", dst);                   // adres pierwszej komórki bufora

while (*dst++ = *src++)                // drugi znak; jeśli '\0' - koniec
{  printf("%p  %c   ", src, *src ? *src:'0'); // wydruk adresu i znaku
   printf("%p\n", dst); }

printf("Wartości końcowe wskaźników: \n");
printf("\n%p   ", src);              // wartości końcowe wskaźników
printf("%p\n", dst);
printf("lan = %s    bufor = %s\n", lan, bufor);
```

Po zakończeniu pętli wskaźniki zawierają adresy komórek pamięci znajdujących się bezpośrednio za obszarami zajmowanymi przez teksty (adres komórki następującej po komórce zawierającej '\0').
Przykładowe wyniki działania przedstawionego fragmentu programu:

```
00120F80  T   00120F6C
00120F81  o   00120F6D
00120F82      00120F6E
00120F83  j   00120F6F
00120F84  e   00120F70
00120F85  s   00120F71
00120F86  t   00120F72
00120F87      00120F73
00120F88  C   00120F74
00120F89  +   00120F75
00120F8A  +   00120F76
00120F8B  0   00120F77
```

Wartości końcowe wskaźników:

```
00120F8C      00120F78
```

```
lan = To jest C++     bufor = To jest C++
```

W praktyce do kopiowania tekstów wykorzystywana jest standardowa funkcja **char *strcpy(char *dst, const char *src),** która zwraca wskaźnik do bufora przeznaczenia dst: printf("dst = %s\n", strcpy(dst, src)).

11.6. Wskaźniki tablic i tablice wskaźników

Niech double a = 1.0;   double b = 2.0;   double c = 3.0;

Tablicę wskaźników do obiektów typu double definiuje się następująco:

```
double *tfwsk[3] = {&a, &b, &c};
```

Wartość tfwsk[0] jest wskaźnikiem do zmiennej a, tzn. tfwsk[0] == &a. Analogicznie, tfwsk[1] == &b oraz tfwsk[2] == &c. Wynika stąd, że:

```
*tfwsk[0] == a == 1.0
*tfwsk[1] == b == 2.0
*tfwsk[2] == c == 3.0
```

Wydruk zawartości tablicy oraz obiektów wskazywanych realizuje następujący fragment programu.

```
for (int i=0; i<3; i++)
printf("tfwsk[%d] = %p  *tfwsk[%d] = %3.1lf\n", i, tfwsk[i], i, *tfwsk[i]);
```

Przykładowe wyniki:

```
tfwsk[0] = 0012FF60   *tfwsk[0] = 1.0          // sizeof(double) = 8
tfwsk[1] = 0012FF58   *tfwsk[1] = 2.0
tfwsk[2] = 0012FF50   *tfwsk[2] = 3.0
```

Można zdefiniować wskaźnik double* *wf do pierwszego elementu tablicy i za jego pomocą zrealizować dostęp do elementów tablicy oraz obiektów wskazywanych. Wartość *wf jest elementem tablicy, czyli wskaźnikiem double *, a więc wartość **wf jest elementem wskazywanym, czyli liczbą typu double.

```
double * *wf = &tfwsk[0];          // wf jest wskaźnikiem do elementu tablicy
i=0;                               // czyli wskaźnikiem do wskaźnika na double
while (i++<3)  {
  printf("wf = %p  *wf = %p  **wf = %3.1lf\n", wf, *wf, **wf);
  wf++;   }
```

```
wf = 0012FF18          *wf = 0012FF60          **wf = 1.0
wf = 0012FF1C          *wf = 0012FF58          **wf = 2.0
wf = 0012FF20          *wf = 0012FF50          **wf = 3.0
```

Wskaźnik double * *wf wskazuje na wskaźniki liczb typu double, natomiast double *f wskazuje na liczby typu double. Na podstawie przedstawionych definicji (wf + 1) wskazuje o jeden obiekt typu (double*) dalej niż wf, podczas gdy (f + 1) o jeden obiekt typu double dalej niż f.

Wskaźnik do tablicy 3 elementów typu double definiuje się następująco:

```
double (* wA)[3];          // wskaźnik do tablicy 3 elementów typu double
```

Wyrażenie (wA + 1) wskazuje na kolejną tablicę, czyli o 3 elementy typu double dalej niż wA. Wyrażenie *wA jest nazwą tablicy liczb typu double (wskaźnikiem na double - double*), a więc (*wA)[0] jest pierwszym elementem tablicy wskazywanej, czyli liczbą typu double, natomiast (*wA)[1] oraz (*wA)[2] są elementami kolejnymi.

Przykład. 11.3. Dana jest tablica double tab[3] = {0.0, 1.0, 2.0} oraz wskaźnik wA na tablicę 3 elementów typu double (operacja wA++ prowadzi do przesunięcia wskaźnika o 3 elementy typu double). Ustawić wskaźnik wA na adres pierwszego elementu tablicy tab[3]. Wykorzystując wskaźnik wA wyprowadzić zawartość tablicy na ekran.

```
double tab[3] = { 0.0, 1.0, 2.0 };

double (*wA)[3] = ( double (*) [3] ) &tab[0];  // lub (double (*) [3]) tab

// lub (double *) wA = tab;  // wA – zawiera adres początku tablicy
for (i=0; i<3;  i++) printf("tab[%d]= %3.1lf", i, (*wA)[i] );
```

Na ekranie pojawi się:
tab[0]= 0.0
tab[1]= 1.0
tab[2]= 2.0

W analogiczny sposób można zdefiniować tablicę łańcuchów oraz wskaźnik do tablicy łańcuchów.

```
char tb[3][20] = { ”Ala”, ”Ola”, ”Basia” };  // tablica trzech łańcuchów
```

Dla każdego łańcucha rezerwowana jest pamięć o rozmiarze 20 bajtów (dane typu char) mimo iż łańcuchy zajmują mniej pamięci.

```
char (*S)[20] = ( char (*) [20] ) &tb[0][0];  // S - wskaźnik do tablicy
                                              // 20 elementów typu char

for (i=0; i<3; i++) { printf("tb[%d][20] = %s    ", i, S++ ); }
```

Na ekranie pojawi się ”Ala” ”Ola” ”Basia”. To samo można uzyskać za pomocą (*S), czyli wskaźnika do pierwszego znaku tablicy 20 elementów typu char.

```
for (i=0; i<3; i++) {  printf("\ntb[%d][20] = %s", i, (*S) );   S++;  }
```

11.7. Bezpośredni dostęp do pamięci

W trybie adresowania rzeczywistego bezpośredni dostęp do pamięci jest zawsze możliwy (kompilator BC++ 3.1). Nie należy jednak korzystać z pamięci, która nie została przydzielona dla aplikacji, a zwłaszcza wprowadzać dane pod przypadkowe adresy.

W trybie chronionym próba dostępu do komórki pamięci o przypadkowym adresie lub do komórki, która nie została przydzielona dla aplikacji zakończy się błędem ochrony (VC++, BuilderC++).

```
unsigned char *x = new unsigned char;

if (x) *x = ’A’;       // wstawia znak ’A’ do pamięci o adresie x <> NULL
```

Analogicznie, można potraktować x jako wskaźnik tablicy danych typu unsigned char i uzyskać dostęp do pamięci za pomocą x[i], np. x[0]=’C’.

Można opracować funkcje wstawiające i odczytujące dane z komórki pamięci o dowolnym adresie.

Przykład. 11.4. Opracować funkcje:

a) wstawiającą daną typu unsigned char do pamięci wskazywanej przez void *q;
b) odczytującą daną typu unsigned char z pamięci wskazywanej przez void *q.

```
void wstaw(void *q, unsigned char c)
{
  * (unsigned char *) q = c;    }


void czytaj(void *q, unsigned char& c)
{
  c = * (unsigned char *) q;    }
```

Wywołanie w programie głównym:

```
unsigned char z = 0;
wstaw(x, ’C’);      // wstawia znak ’C’ do komórki o adresie x
czytaj(x, z);       // czyta zawartość komórki o adresie x do zmiennej z
cout << z << endl;      // wydruk z = ’C’ na ekran
```

Wykorzystując przedstawione techniki dostępu do pamięci można opracować funkcje umożliwiające skopiowanie zawartości określonego obszaru pamięci do bufora oraz przesłanie zawartości bufora do istniejącego obszaru pamięci. Można do tego celu wykorzystać standardową funkcję memcpy, które kopiuje n bajtów spod adresu src do pamięci o adresie dest.

```
void *memcpy(void *dest, const void *src, size_t n);
```