Wykład 13

13. Zastosowania wskaźników

Za pomocą wskaźników można przekazywać do funkcji adresy zmiennych, tablic i funkcji oraz wykonywać różne operacje na tablicach jedno-indeksowych (jednowymiarowych) i wielo-indeksowych (wielowymiarowych), a także na blokach pamięci.

13.1. Wskaźniki i tablice wielowymiarowe

Tablica jednowymiarowa (jedno-indeksowa) jest ciągiem jednakowych elementów. Tablicę można zainicjować w momencie definicji.

```
int tab[4] = { 1, 2 };   // tablica zewnętrzna (globalna)
```

W przypadku tablic zewnętrznych (globalnych) i statycznych elementy, które nie zostały zainicjowane są równe 0, natomiast dla tablic automatycznych (lokalnych) mogą mieć wartości przypadkowe (zależy od kompilatora).

```
tab[0] == 1;        tab[1] == 2;        tab[2] == 0;        tab[3] == 0;
```

Nazwa tablicy jest stałym wskaźnikiem do pierwszego elementu tablicy.

```
tab == &tab[0];
```

Natomiast, wyrażenie (tab + i) jest adresem i-tego elementu tablicy. Spełnione są zależności:

```
tab + i == &tab[i];                    tab[i] == *(tab + i);
```

Tablica dwuwymiarowa jest tablicą, której elementami są jednowymiarowe tablice. Na przykład, int A[4][3] jest czteroelementową tablicą złożoną z tablic, zawierających po trzy elementy typu int. Składa się ona z 4 wierszy, z których każdy zawiera po 3 elementy typu int.

```
Wiersz nr 0:    A[0][0]    A[0][1]    A[0][2]
Wiersz nr 1:    A[1][0]    A[1][1]    A[1][2]
Wiersz nr 2:    A[2][0]    A[2][1]    A[2][2]
Wiersz nr 3:    A[3][0]    A[3][1]    A[3][2]
```

W momencie definicji można zainicjować tablicę uwzględniając to, iż składa się ona z podtablic lub potraktować ją jako ciągły blok danych.

```
int A[4][3] = { {1,1}, {2,2}, {3,3}, {4,4} };          // 4 podtablice int [3]

for (i=0; i<4; i++)
 {
   for (j=0; j<3; j++)  printf("%3d",A[i][j]);
   printf("\n");
  }
```

```
1     1     0
2     2     0
3     3     0
4     4     0
```

```
int A[4][3] = { 1, 1, 2, 2, 3, 3, 4, 4 };        // ciąg elementów typu int

for (i=0; i<4; i++)
{
   for (j=0; j<3; j++)  printf("%3d",A[i][j]);
   printf("\n");
 }
```

```
1     1     2
2     3     3
4     4     0
0     0     0
```

Reprezentacja tablic dwuwymiarowych

Identyfikator A jest nazwą tablicy int A[4][3], która składa się z czterech tablic typu int [3]. Nazwa tablicy jest wskaźnikiem (stałą wskaźnikową) do jej pierwszego elementu, a więc A jest wskaźnikiem do tablicy trzech elementów typu int. Wynika stąd, że A jest typu int (*) [3].

Elementy tablic wielowymiarowych są umieszczane w pamięci jeden za drugim w kolejności określonej przez zmianę indeksów tablicy zaczynając od skrajnego prawego indeksu (zgodnie z zasadą licznika samochodowego). Wynika stąd, że w przypadku tablic dwuwymiarowych elementy są pamiętane wierszami.

W przypadku tablicy int A[4][3] spełnione są zależności. A jest stałą, która zawiera wskaźnik do pierwszej tablicy (zerowego wiersza) złożonej z trzech elementów typu int (A jest typu int (*) [3]). Wynika stąd, że (*A) = A[0] jest tablicą typu int [3], tj. stałą typu int* zawierającą adres pierwszego elementu tablicy złożonej z trzech liczb całkowitych. Wyrażenie (*A) jest, więc adresem pierwszego elementu w pierwszym wierszu tablicy int A[4][3].

Spełnione są zależności:

```
A[0] == *(A+0) == *A == &A[0][0].   // A wskaźnik wiersza nr 0
```

Analogicznie, int A[i][3] == int (A[i]) [3]. Wynika stąd, że A[i] jest wskaźnikiem (int *) do pierwszego elementu w i-tym wierszu tablicy int A[4][3].

```
A=(A+0)     - wskaźnik tablicy int [3] wiersz_0
(A+1)       - wskaźnik tablicy int [3] wiersz_1
(A+2)       - wskaźnik tablicy int [3] wiersz_2
(A+3)       - wskaźnik tablicy int [3] wiersz_3
```

Tak więc,

```
A[0] == *(A+0) == *A == &A[0][0]  - adres elementu int A[0][0]
A[1] == *(A+1) ==       &A[1][0]  - adres elementu int A[1][0]
A[2] == *(A+2) ==       &A[2][0]  - adres elementu int A[2][0]
A[3] == *(A+3) ==       &A[3][0]  - adres elementu int A[3][0]
```

Wartości pierwszych elementów w kolejnych wierszach tablicy:

```
*A[0] == * *(A+0) == A[0][0]
*A[1] == * *(A+1) == A[1][0]
*A[2] == * *(A+2) == A[2][0]
*A[3] == * *(A+3) == A[3][0]
```

Wydruk elementów tablicy:

```
int A[4][3] = { 1, 1, 2, 2, 3, 3, 4, 4 };        // ciąg elementów typu int

for (i=0; i<4; i++)  printf("%3d",*A[i] );      // wydruk kolumny 0:   1 2 4 0
```

W przypadku tablicy jednowymiarowej: tab[i] == *(tab + i).
Dla tablicy dwuwymiarowej:

A[i][j] == (A[i]) [j] == *(A[i] + j) == * (*(A + i) + j)

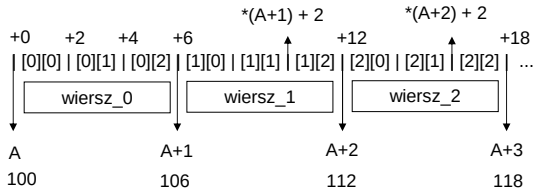
Stała A[i] jest wskaźnikiem (int *) na pierwszy element i-tego wiersza tablicy, zatem A[i] + j wskazuje na element A[i][j]. Ostatecznie,

A[i][j] == *(*(A + i) + j) // przesunięcie do i-tego wiersza
// przesunięcie do j-tej kolumny
W szczególności:

A[0][2] == *(*(A + 0) + 2) == *(*A + 2);
A[1][2] == *(*(A + 1) + 2);
A[2][2] == *(*(A + 2) + 2);

Reprezentacja tablicy int A[4][3] w pamięci

sizeof(int) =2



Dostęp do elementów tablicy

Na tablicach można wykonywać różne operacje wykorzystując zapis indeksowy lub wskaźnikowy.

Przykład 13.1. Dostęp do elementów tablicy dwuwymiarowej.

```
for (i=0; i<4; i++)
{  for (j=0; j<3; j++) printf("%3d",A[i][j]);  printf("\n");  }

for (i=0; i<4; i++)
{  for (j=0; j<3; j++)  printf("%3d", *( *(A + i) + j ) );  printf("\n");  }
```

```
int *wk;          // wskaźnik kolumny
int (*ww)[3];     // wskaźnik wiersza

ww = A;           // wskaźnik do tablicy wiersz_0

for (i=0; i<4; i++) {
  wk = *ww++;   // *ww - wskaźnik do 1-wszego elementu w wierszu
  for (j=0; j<3; j++) printf("%3d",*wk++);
  printf("\n");
 }
```

Tablicę int A[4][3] można potraktować również jako ciągły blok pamięci złożony z elementów typu int. Dostęp do takiego bloku danych można zrealizować za pomocą tablicy jednowymiarowej o odpowiednio wyznaczanych indeksach.

```
int *H = &A[0][0];          // H = A[0]
int nw, nk;

nw = 4;           // liczba wierszy tablicy
nk = 3;           // liczba kolumn tablicy

for (i=0; i < nw; i++) {
 for (j=0; j < nk; j++)
  printf("%3d", H[i*nk + j] );  // przesuwanie o liczbę kolumn, czyli
  printf("\n");                 // o liczbę elementów w wierszu
 }
```

Przekazywanie tablic do funkcji

- Przekazanie przez dokładną definicję tablicy

```
void pisz1(int t[4][3], int w, int k)
{
  int i, j;
  for (i=0; i<w; i++)
  {
   for (j=0; j<k; j++) printf("%3d", t[i][j]); printf("\n");
  }
}
```

Wywołanie: pisz1(A, 4, 3). Dopuszczalne również wywołanie dla tablicy int B[7][3] w postaci pisz1(B, 7, 3).

- Przekazanie przez definicję tablicy z pominięciem 1-wszego wymiaru (można przekazywać tablice o dowolnym pierwszym wymiarze)

```
void pisz2(int t[][3], int w, int k)
{
  int i, j;
  for (i=0; i<w; i++)
  {
   for (j=0; j<k; j++) printf("%3d", t[i][j]);    printf("\n");
  }
}
```

Wywołanie: pisz2(A, 4, 3). Dopuszczalne również wywołanie dla tablicy int B[7][3] w postaci pisz2(B, 7, 3).

- Przekazanie przez wskaźnik do tablicy reprezentującej wiersz

```
void pisz3(int (*t)[3], int w, int k)
{
 int i, j;
 for (i=0; i<w; i++)
 {
  for (j=0; j<k; j++) printf("%3d", t[i][j]);    printf("\n");
 }
}
```

Wywołanie: pisz3(A, 4, 3). Dopuszczalne również wywołanie dla tablicy int B[7][3] w postaci pisz3(B, 7, 3)

- Przekazanie przez wskaźnik do elementu tablicy

```
void pisz4(int *t, int w, int k)
{
 int i, j;
 for (i=0; i<w; i++)   // wydruk tablicy int A[4][3]
 {
  for (j=0; j<k; j++) printf("%3d", t[i*3 + j]);    printf("\n");
 }   // i*3  - przesunięcie indeksu o liczbę kolumn tablicy
}
```

Możliwe wywołania: pisz4(A[0], 4, 3), pisz4(&A[0][0], 4, 3), pisz4((int*)A, 4, 3), pisz4(*A, 4, 3).

Dostęp do tablic wielowymiarowych

Rozważania przedstawione dla tablicy dwuwymiarowej można rozszerzyć na tablice o większej liczbie wymiarów.

Dla tablicy trzywymiarowej int B[i][j][k]:

B[i][j][k] == (B[i][j])[k] == *(B[i][j] + k) == *(*(*(B + i) + j) + k)

Stała B jest stałym wskaźnikiem do tablicy int [j][k]. Wyrażenie B[i] reprezentuje i-tą tablicę int [j][k].

int B[2][4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };

Dla tablicy globalnej B[2][4][3] stała B jest stałą typu int (*)[4][3]. Wyrażenie *B == B[0] jest tablicą int [4][3]. Wykorzystując funkcję pisz1, opracowaną dla tablicy dwuwymiarowej, można wydrukować zawartość tablicy B (elementy 1,2,...,12 – tablica B[0]; elementy 13,14,15 – tablica B[1]).

Przykład 13.2. Wydruk zawartości tablicy trzywymiarowej.

```
void pisz1(int t[4][3], int w, int k)
{
  int i, j;
  for (i=0; i<w; i++)  {
   for (j=0; j<k; j++) printf("%3d", t[i][j]); printf("\n");   }
}
for (i=0; i<2; i++) { pisz1( B[i], 4, 3);   printf("\n"); } // 1...15 , 0, ...
```

Do tablicy int B[2][4][3] można uzyskać dostęp za pomocą wskaźnika do typu int. Wymaga to jednak przeliczenia indeksów. Skrajny lewy indeks [2] powoduje przesunięcie o rozmiar tablicy int [4][3], natomiast środkowy indeks [4] o rozmiar tablicy int [3].

```
int *H = &B[0][0][0];          // lub H = B[0][0]

for (i=0; i<2; i++)  {
  for (j=0; j<4; j++)   {
   for (k=0; k<3; k++) printf("%3d", H[i*4*3 + j*3 + k]);  printf("\n");
                       }
      printf("\n");  }
```

Na ekranie pojawią się tablice (jedna pod drugą):

```
 1  2  3                13 14 15
 4  5  6                 0  0  0
 7  8  9                 0  0  0
10 11 12                 0  0  0
```

W ogólnym przypadku dla tablicy N - wymiarowej:

int $A[w_0][w_1][w_2][...][w_{N-1}]$;

otrzymujemy następujący wzór:

```
int *H = &A[0][0][0][...][0];

for (int i0=0; i0<w0; i0++)    {
  for (int i1=0; i1<w1; i1++)      {
    for (int i2=0; i2<w2; i2++)          { ...  printf("%3d", H[x] );
                                           ...   }
                                   }
                             }
```

gdzie
x – indeks tablicy H określony przez:

$L(k=1,...,N-1)w_k = w_1 * w_2 * ... * w_{N-1}$

oraz

$x = i_0*L(k=1,...,N-1)w_k + i_1*L(k=2,...,N-1)w_k + ... + i_s*L(k=s+1,...,N-1)w_k + ... + i_{N-1}$

Przykład 13.3. Wydruk zawartości tablicy 3-wymiarowej traktowanej jako tablica jednowymiarowa (wzór ogólny).

Dla N = 3 jest $x = i_0* w_1* w_2 + i_1*w_2 + i_2$.

```
int   A[w0][w1][w2];
int *H = &A[0][0][0];

for (int i0=0; i0<w0; i0++)   {
  for (int i1=0; i1<w1; i1++)      {
    for (int i2=0; i2<w2; i2++)    printf("%3d", H[i0* w1* w2 +  i1*w2 + i2]);   }
                                 }
```

13.2. Tablice tworzone dynamicznie

Tablice dwuwymiarowe (wielowymiarowe) mogą być tworzone w sposób dynamiczny na stercie.

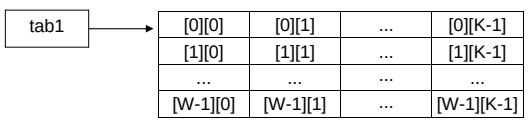
Jeśli oba wymiary tablicy 2-indeksowej są ustalone to można zdefiniować typ tablicowy double s[W][K] reprezentujący tablicę o ustalonych wymiarach, przydzielić pamięć dla tablicy typu s i zapamiętać adres początku obszaru we wskaźniku do wiersza tablicy, czyli we wskaźniku typu double (*) [K]. W ten sposób powstanie na stercie tablica, która rezerwuje ciągły obszar pamięci o rozmiarze W*K*sizeof(double).

Przykład 13.4. Dynamiczna alokacja tablicy 2-wymiarowej o z góry ustalonej liczbie wierszy i kolumn.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>
#include <iostream.h>
#include <alloc.h>
#include <string.h>
// #include <windows.h>  // testowane pod windows - bez coreleft()
                         // kompilator BCW
// przekroczenia zakresu tablic prowadzą do błędu ochrony
// jeśli błąd ochrony zostanie popełniony tuż przed zakończeniem
// programu to może nie zostać zgłoszony przez system

const W=4;        // liczba wierszy tablicy
const K=3;        // liczba kolumn tablicy

typedef double tt[K];      // tt - typ tablica double [K];
                           // sizeof(tt) = K * sizeof(double)

typedef double s[W][K];    // s – typ wskaznik typu double (*) [K]
                           // sizeof(s) = W*K*sizeof(double)
```

```
void main(void)
{  int i,j, l=0;      clrscr();
     // -- Alokacja tablicy dynamicznej o stałych wymiarach  W x K
     tt *tab1;   // tab1 wskaźnik typu tt;

     cout << (unsigned long) coreleft() << endl;   // BC++ 3.1

     tab1 = new s;      // tablica dynamiczna typu s = double  [W]  [K]
                        // W - elementowa tablica złożona z elementów typu
                        // double [K]; new zwraca wskaznik double (*) [K]
                       //  sizeof(s) = W*K*sizeof(double) = 96
     cout << (unsigned long) coreleft() << endl;   // BC++ 3.1.

     if (!tab1)  cout << "Błąd alokacji" << endl;
     else
     {  for (i=0; i<W; i++) {
        for (j=0; j<K; j++)    {
              tab1[i][j] = l++; cout.width(3);  cout << tab1[i][j];
                                }
          cout << endl;
                             }
     delete [ ] tab1;   tab1=NULL;   // zwolnienie pamięci
     }
     cout << (unsigned long) coreleft() << endl;  // BC++ 3.1
     getch();
}
```

Tablica W x K elementów



[0][0]	[0][1]	...	[0][K-1]
[1][0]	[1][1]	...	[1][K-1]
...	...	...	...
[W-1][0]	[W-1][1]	...	[W-1][K-1]

Wskaźnik tab1 zawiera adres początku ciągłego bloku pamięci o rozmiarze W*K*sizeof(double). Jest to wskaźnik typu double (*) [K], tj. wskaźnik, który w wyniku operacji tab1++ pokazuje na adres o K elementów typu double za tab1 (interpretowane jako przesunięcie o wiersz tablicy). Wskaźnik tab1[0] jest typu double* i zawiera adres pierwszego elementu typu double, natomiast tab1[0][0] jest wartością pierwszego elementu, czyli liczbą typu double.

W przypadku, gdy w programie wykorzystywana jest tablica, w której jeden z wymiarów jest znany, natomiast drugi z wymiarów jest zależny od wprowadzanych danych, to można rezerwować w programie tylko tyle danych ile potrzeba korzystając z funkcji alokacji pamięci.

Kolejny przykład ilustruje sposób alokacji tablicy dwuwymiarowej o znanej (stałej) liczbie kolumn K oraz zmiennej (wczytywanej z klawiatury) liczbie wierszy nw. Za pomocą funkcji alokacji pamięci tworzony jest ciągły blok danych złożony z nw tablic typu double (*) [K].

Przykład 13.5. Dynamiczna alokacja tablicy 2-wymiarowej o zmiennej liczbie wierszy (operator new).

```
const K=3;

typedef double tt[K];        // tt typ - tablica double [K];
                             // sizeof(tt) = K * sizeof(double)

void main(void)
{
     int i,j,l=0;

     int nw;    // liczba wierszy  tablicy

// -- Alokacja tablicy 2 wym. o zadawanej liczbie
//    wierszy nw i stałej liczbie kolumn K  - wykorzystanie new

 clrscr();

 cout << "Podaj liczbe wierszy tablicy, np. 4" << endl;

 cin >> nw;   // nw=4

 tt *tab2;    // double (*) [K];  sizeof(tt)=K*sizeof(double)=24

 cout << (unsigned long) coreleft() << endl;  // BC++ 3.1

 tab2 = new tt [nw];     // nw elementów typu tt;  4*sizeof(tt)=96

 cout << (unsigned long) coreleft() << endl;   // BC++ 3.1


if (!tab2)  cout << "Blad alokacji" << endl;
```

```
    else
    { for (i=0; i<nw; i++) {
        for (j=0; j<K; j++) {
          tab2[i][j] = l++;
           cout.width(3);   cout << tab2[i][j];
                            }
           cout << endl;
                        }
    delete [ ] tab2;   tab2=NULL;    // zwolnienie pamięci dla new
    }
    cout << (unsigned long) coreleft() << endl;  // BC++ 3.1
    getch();
}
```

W podobny sposób można utworzyć tablicę 2-wymiarową o jednym stałym wymiarze za pomocą funkcji malloc i calloc.

Przykład 13.6. Dynamiczna alokacja tablicy 2-wymiarowej o zmiennej liczbie wierszy (funkcja malloc).

```
void main()
{
...
// -- Alokacja tablicy 2 wym. o zadawanej liczbie
//    wierszy nw i stałej liczbie kolumn K  - wykorzystanie malloc

 clrscr();  cout << "Podaj liczbe wierszy tablicy, np. 4" << endl;
 cin >> nw;

tt *tab3;    // double (*tab) [K];  sizeof(tt)=K*sizeof(double)=24
cout << (unsigned long) coreleft() << endl;  // BC++ 3.1

tab3 = (tt*) malloc(nw*sizeof(tt));  // tab3 jest wskaznikiem do tt

// tab3 = (tt*) calloc(nw,sizeof(tt));  // tab3 wypelniona zerami
...
// dostęp do tablicy  tab3[i][j]
...
free(tab3);      tab3 = NULL;    // zwolnienie pamięci
cout << (unsigned long) coreleft() << endl;  // BC++ 3.1
getch();
}
```

Jeśli oba wymiary tablicy nie są z góry znane, to można przydzielić pamięć dla dynamicznej tablicy wskaźników na double o nw elementach, a następnie zapamiętać w tej tablicy adresy tablic złożonych z nk elementów typu double.

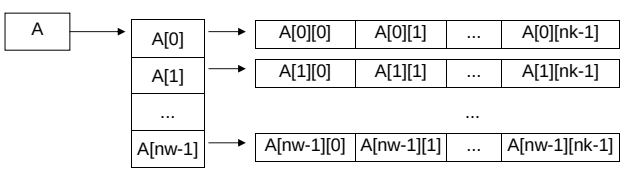
W kolejnym przykładzie adres początku tablicy wskaźników jest pamiętany w zmiennej double * *A. Tablica A składa się z nw elementów typu (double *). Każdy z elementów A[i] zawiera adres tablicy złożonej z nk elementów typu double.

Przykład 13.7. Dynamiczna alokacja tablicy 2-wymiarowej o zmiennej liczbie wierszy i zmiennej liczbie kolumn.

```
// -- Alokacja tablicy 2 wym. o zadawanej liczbie wierszy nw
// -- i zadawanej liczbie kolumn nk - wykorzystanie new
int nw;    // liczba wierszy  tablicy
int nk;    // liczba kolumn   tablicy
double * *A;  // wskaźnik do tablicy wskaźników  na double
                 // reprezentacja tablicy nw wskaźników na tablice nk
                 // elementowe typu double
   cout << "Podaj liczbe wierszy tablicy, np. 4" << endl;
   cin >> nw;        // 4
   cout << "Podaj liczbe kolumn tablicy, np. 3" << endl;
   cin >> nk;        // 3

   cout << endl<< (unsigned long) coreleft();  // BC++ 3.1
   A = new double * [nw];   // tablica nw wskaźników na double
                            // służy do pamiętania adresów
                            // dynamicznych tablic nk elementowych
//  A = (double**) malloc(nw*sizeof(*A));  // wykorzystanie malloc
//  A = (double**) calloc(nw, sizeof(*A));  // wykorzystanie calloc

  cout << endl<< (unsigned long) coreleft();  // BC++ 3.1

  if (!A)  cout << "Blad alokacji" << endl;    else
  {
   for (i=0; i < nw; i++)                  // nw dynamicznych tablic
   {
      if ( (A[i] = new double [nk]) == NULL )   // nk-elementowych
        {  cout << "Blad alokacji" << endl;    exit(0);   }
   }
  }
```

```
  randomize();     // inicjacja generatora liczb losowych

  for (i=0; i<nw; i++)    // losowanie danych
   for (j=0; j<nk; j++)  A[i][j] = random(2);

  for (i=0; i<nw; i++)    // wyprowadzanie danych
  { cout << endl;
   for (j=0; j<nk; j++)  printf("%3.0lf", A[i][j]);
  }
  double  sum = 0.0;     // sumowanie elementów
  for (i=0; i<nw; i++)
   for (j=0; j<nk; j++)  sum+=A[i][j];

   cout << endl << "Suma elementow = " << sum << endl;

   for (i=0; i < nw; i++) delete [ ] A[i];    // usuwanie nw tablic
                                              // nk elementowych
   delete [ ] A;       // usuwanie tablicy wskaźników A
//  free(A);           // usuwanie dla malloc
   cout << endl<< (unsigned long) coreleft();  // BC++ 3.1
   getch();
 }
```

W przedstawionym przykładzie do utworzenia tablicy niezbędna była zmienna A umożliwiająca zapamiętanie adresu dynamicznej tablicy wskaźników o elementach A[0], A[1], ..., A[nw-1]. Każdy z tych wskaźników zawierał z kolei adres tablicy złożonej z nk elementów typu double.



Z przedstawionych rozważań wynika, że tablica A zajmuje nw ciągłych obszarów pamięci o rozmiarze nk*sizeof(double) (wiersze tablicy), jeden ciągły obszar o rozmiarze nw*sizeof(double *) (wskaźniki na pierwsze elementy w wierszach) oraz miejsce w pamięci przeznaczone na zmienną A. W celu zwolnienia pamięci tablicy A należy najpierw zwolnić obszary pamięci przydzielone dla wierszy tablicy A[0], A[1], ..., A[nw-1], a następnie zwolnić pamięć przydzieloną dla tablicy wskaźników A (w przypadku odwrotnej kolejności zwalniania pamięci utracimy wartości zawarte w tablicy wskaźników).

13.3. Wskaźniki a funkcje

W języku C nazwa funkcji jest stałą wskaźnikową, która zawiera adres początku kodu funkcji. Podobnie jak nazwa tablicy jest stałą równą adresowi tablicy.

Na przykład clrscr jest adresem funkcji bibliotecznej (BC++, BuilderC++), która czyści ekran. W celu wykonania fragmentu kodu o adresie clrscr należy umieścić w programie nazwę funkcji uzupełnioną o nawiasy () i odpowiednie argumenty wywołania funkcji. W przypadku funkcji clrscr() prototyp ma postać void clrscr(void), a więc funkcja jest bezparametrowa.

```
clrscr;       // nie jest podejmowane żadne działanie
clrscr();     // wywoływana jest funkcja standardowa clrscr();
```

Jeśli zdefiniuje się wskaźnik do funkcji typu void fun(void)

```
 void (*f)(void) = clrscr,       // f jest wskaźnikiem (*f) funkcji
                          // bezparametrowej, która nie zwraca wyniku void
```

wówczas równoważne są następujące wywołania: (*f)(); oraz f().

Analogicznie można zdefiniować wskaźniki do funkcji innych typów:

```
double (*f)(double);     // f  wskaźnik funkcji o parametrze typu double
                         //    i zwracającej wynik typu double

float * (*g)(int);       // g wskaźnik funkcji o parametrze typu int
                         // i zwracającej wskaźnik do zmiennej typu float
```

Standardowa funkcja sinus ma prototyp w postaci: double sin(double). Po przypisaniu double (*f)(double) = sin możliwe są równoważne wywołania:

```
double x, y;        y = sin(x);          y = (*f)(x);          y = f(x);
```

Można również zdefiniować tablicę wskaźników do funkcji:

```
double ( *tw[ ] ) (double) = { sin, cos, tan };
```

tw[0] – wskaźnik funkcji sin;
tw[1] – wskaźnik funkcji cos;
tw[2] – wskaźnik funkcji tan;

Wówczas, równoważne są następujące wywołania:

```
y = ( *tw[0] )(x);                    y = tw[0](x);
y = ( *tw[1] )(x);                    y = tw[1](x);
y = ( *tw[2] )(x);                    y = tw[2](x);
```

Na wskaźnikach funkcji można wykonywać następujące operacje:
- przekazywać jako argumenty do innych funkcji;
- zwracać jako wynik funkcji;
- porównywać ze wskaźnikiem NULL;
- poddawać operacji wyłuskania;

Uwaga: na wskaźnikach funkcji nie wolno wykonywać operacji arytmetycznych.

Funkcje jako parametry innych funkcji

W języku C funkcje nie mogą zawierać definicji innych funkcji. Można jednak przekazywać do funkcji wskaźniki innych funkcji.

Na przykład można zaprojektować funkcję, która korzysta z rodziny funkcji typu void (*f)(void).

```
void pisz( int n, void (*f) (void) ) {  f();    cout << n << endl;  }
```

Wywołanie funkcji: pisz(5, clrscr);

Przykład. 13.8. Obliczanie sumy wartości funkcji dla kolejnych elementów tablicy, tj. $\sum f(tab[i])$, i=1,...,N.

```
double oblicz(double tab[], int n, double (*f)(double) )
{
  double s = 0.0;
  for (int i=0; i<n; i++) s+= f(tab[i]);
  return s;
}
double fun(double x)
{ return x+1; }

double tab[ ] = { 1, 2, 3, 4 };
```

Wywołanie funkcji: cout << oblicz(tab, 4, fun); // suma = 14

Przykład. 13.9. Zadeklarować typ tablicowy typedef double ttab[N][2]. Opracować funkcję, która wyznacza wartości funkcji double f(double) w przedziale [min, max] dla N punktów postaci xi = min + i*d, gdzie d=(max-min) / (N-1) oraz i=0,1,...,N-1 (N – stała). Obliczone elementy xi, f(xi) powinny być zapamiętane w tablicy T typu ttab w sposób następujący: T[i][0] = xi oraz T[i][1] = f(xi). Prototyp funkcji:

void oblicz(double min, double max, int n, ttab T, wskaźnik_funkcji).

W programie głównym wykonać obliczenia dla funkcji sin w przedziale [-pi/2, pi/2], dla N=10.

```
const int N = 10;                  // rozmiar tablicy
typedef double ttab[N][2];         // typ – tablica double [N][2]
typedef double (*FP)(double);  // typ – wskaźnik do funkcji

void oblicz1(double min, double max, int n, ttab tab, FP fn)
{
   double xi;     double d = (max-min)/(n-1);

   for (int i=0; i<n; i++)  {  xi = min + i*d;    tab[i][0] = xi;   tab[i][1] = fn(xi);  }
}

void main()
{
  ttab T;     // tablica typu ttab

  clrscr();

  oblicz1(-M_PI/2, M_PI/2, N, T, sin);   // M_PI – stała PI

  for(int i=0; i<N; i++)  {
   if (i%3==0 && i>0)   printf("\nf(%7.4lf)=%8.4lf\t",T[i][0],T[i][1]);
   if (i%3 >0 || i==0)  printf("f(%7.4lf)=%8.4lf\t",T[i][0],T[i][1]);  }
  getch();
}
```

Wyniki:

```
f(-1.5708) =  -1.0000      f(-1.2217) = -0.9397      f(-0.8727) =  -0.7660
f(-0.5236) =  -0.5000      f(-0.1745) = -0.1736      f( 0.1745)  =  0.1736
f( 0.5236) =   0.5000      f( 0.8727) =  0.7660      f( 1.2217)  =  0.9397
f( 1.5708)  =  1.0000
```

13.4. Sortowanie tablic

W języku ANSI C dostępna jest standardowa funkcja qsort, która implementuje algorytm sortowania szybkiego (ang. quicksort algorithm).

Algorytm quicksort

Algorytm sortowania szybkiego oparty jest na technice „dziel i rządź". Najpierw w tablicy T[N] wybierany jest losowo pewien element x=T[k], taki że 0 <= k <= n-1. Następnie tablica jest przeglądana od lewej strony, aż do napotkania elementu T[i] >= x, oraz od prawej strony, aż do napotkania elementu T[i] <=x. Znalezione elementy są wymieniane. Proces przeglądania oraz wymiany elementów jest powtarzany, aż do momentu, gdy z obu stron zostanie osiągnięta pozycja elementu x lub (jeśli x został wymieniony), gdy nastąpi spotkanie przy przeglądaniu tablicy od lewej i od prawej. Otrzymuje się w ten sposób tablicę podzieloną na dwie niepuste podtablice: L = {T[0], T[1], ... ,T[q]} oraz P = {T[q+1], T[q+2], ... ,T[n-1]}. W rezultacie tablica T jest podzielona na lewą część L, która zawiera elementy o wartościach nie większych od x oraz prawą część P, która zawiera elementy o wartościach nie mniejszych od x. Faza „rządź" algorytmu sprowadza się do sortowania podtablic L i P za pomocą rekurencyjnych wywołań algorytmu quicksort, tj. dokonywania podziałów i wymiany elementów dla tablic L i P. Proces podziałów jest powtarzany, aż do momentu, gdy pojawią się podtablice zawierające po jednym elemencie.

Czas działania algorytmu quicksort zależy od sposobu podziału tablicy na dwie części. Jeśli tablice otrzymywane w wyniku podziałów są równoliczne (np. w pierwszym kroku są dwie tablice po n/2 elementów, w drugim cztery tablice po n/4 elementów – ogólnie 2^k tablic po $n/(2^k)$ elementów) wówczas efektywność algorytmu jest maksymalna. Proces podziałów kończy się, gdy liczba elementów każdej podtablicy jest równa $n/(2^k) = 1$, tj. gdy k = log(n). Ponieważ dla każdego k=1,2,...,log(n) koszt przestawień 2^k zbiorów $n/(2^k)$ elementowych jest rzędu O(n). Złożoność obliczeniowa algorytmu dla najlepszego przypadku jest O(nlog(n)). Najgorszy przypadek ma miejsce, gdy w każdym kroku procedura dzieląca tworzy jeden obszar złożony z 1 elementu i drugi obszar złożony z (n-1) elementów. Liczba podziałów wynosi wówczas n, a koszt jednego podziału jest rzędu O(n). Stąd złożoność obliczeniowa dla najgorszego przypadku jest rzędu $O(n^2)$.

Na ogół algorytm quicksort niezbyt dobrze działa dla małych wartości n (np. rzędu 10) oraz dla tablic uporządkowanych. Czas działania algorytmu zależy od wyboru ograniczenia x. W celu poprawienia jego efektywności dla najgorszych przypadków wybiera się x jako medianę z kilku wylosowanych elementów tablicy. Na przykład w systemie BC++ 3.1 funkcja qsort wybiera ograniczenie x jako medianę (środkową wartość) z trzech elementów losowo wybranych z podtablicy.

Oprócz dużej szybkości działania dla przeciętnego przypadku zaletą algorytmu quicksort jest również to, że sortuje on „w miejscu", tj. wykonuje operacje bezpośrednio na tablicy wejściowej T[n]. Dlatego po zakończeniu procedury podziałów nie ma potrzeby łączenia podtablic, gdyż cała tablica T jest już posortowana.

Sposób działania algorytmu przy założeniu, że ograniczenie x jest wybierane jako element środkowy w podtablicy ilustruje następujący przykład.

Przykład. 13.10. Niech int T[5] = { 7, 8, 5, 2, 4 }.

W pierwszym kroku element środkowy tablicy x = 5. Pierwszym napotkanym elementem z lewej strony większym lub równym x jest 7, natomiast pierwszym napotkanym elementem z prawej strony mniejszym lub równym x jest 4. Elementy te są wymieniane. Następnie wymieniane są elementy 8 i 2.

Krok A. **7** 8 5 2 **4**

Krok B. **4** 8 5 2 **7**

Krok C. 4 **2** 5 **8** 7

W wyniku podziału podtablica L = { 4, 2 } oraz podtablica P = { 8, 7 }.

Kolejne podziały dotyczą zbioru L = {4, 2 }. Element środkowy x = 4. Wymieniane są elementy 4 i 2.

Krok D. **2 4** 5 8 7

Kolejne zbiory L* = { 2 } oraz P* = { 4 }. Zbiory są jednoelementowe, a więc wymiany kończą się. Można rozpocząć analizę zbioru P = { 8, 7 }. Element środkowy x = 8. Wymieniane są elementy 8 i 7.

Krok E. 2 4 5 **7 8**

Kolejne zbiory L** = { 7 } oraz P** = { 8 }. Zbiory są jednoelementowe, a więc wymiany kończą się. Tablica została posortowana.

Kolejny przykład pokazuje implementację rekurencyjnej wersji algorytmu quicksort.

Przykład. 13.11. Rekurencyjna wersja algorytmu quicksort.

```
const long N=5;  //rozmiar tablicy danych
typedef double tdana;          // typ danej

typedef tdana ttab [N];        // typ tablicy danych

ttab tab = { 7, 8, 5, 2, 4 };     // tablica danych

// ********** quicksort rekurencyjne***********

void r_qsort(ttab t, long a, long b)
{
   // rekurencyjny qsort - wersja z elementem środkowym tablicy
   // t - tablica elementow typu tdana
   // a - nr pierwszego elementu tablicy;
  //  b - nr ostatniego elementu tablicy
  long i, j;            // indeksy elementów tablicy określające
                        // początek i koniec przedziału
  tdana pom, x;   // x - element wybierany względem, którego podział

  i=a;  j=b;            // indeksy przedziału początkowego
  x =  t[(a+b)/2];    // element ze środka tablicy

  do {
  while (t[i] < x) i++;    // poszukiwanie elementu nie mniejszego od x
  while (x < t[j]) j--;    // poszukiwanie elementu nie większego od x

 if (i<=j)   {           // wymiana elementów
   pom = t[i];
   t[i] = t[j];
   t[j] = pom;
   i++;   j--;  }       // j może dojść do -1;  typ j musi być long
   } while (i<=j);

 if (a < j) r_qsort(t, a, j);    // sortowanie lewej podtablicy
 if (i < b) r_qsort(t, i, b);    // sortowanie prawej podtablicy
}

void main()
{
     long i;
     clrscr();
     for (i=0; i<N; i++) cout << tab[i] << "| ";  cout << endl;

     r_qsort(tab, 0, N-1);  // sortowanie tablicy tab

     cout << endl << endl;
     for (i=0; i<N; i++) cout << tab[i] << "| ";     getch();
}
```

Jeśli sortowane tablice są znacznych rozmiarów wywołania rekurencyjne mogą doprowadzić do przepełnienia stosu procesora. Istnieje wersja algorytmu quicksort, w której rekurencja nie jest wykorzystywana. W rozwiązaniu tym kolejne indeksy podtablic są składowane w tablicy dwuwymiarowej reprezentującej stos (indeksy pierwszego rozpatrywanego przedziału są analizowane na końcu).

Przykład. 13.12. Iteracyjna wersja algorytmu quicksort.

```
// ********** quicksort iteracyjne ***********

const long N=5;          // rozmiar tablicy danych

const long M=200;     // rozmiar stosu

typedef double tdana;          // typ danej

typedef tdana ttab [N];       // typ tablicy danych

typedef long tstos[M][2];    // typ stosu kolejnych indeksów przedziałów

tstos stos;                          // stos

ttab tab = { 7, 8, 5, 2, 4 };     // tablica danych

void n_qsort(ttab t, tstos stos, long a, long b)
{
   // nierekurencyjny qsort - wersja z elementem środkowym tablicy
   // t - tablica elementów typu tdana
   // stos - stos indeksów przedziałów a,b
   // a - nr pierwszego elementu tablicy;
   // b - nr ostatniego elementu tablicy

  long i, j;    // indeksy elementów tablicy określające
                // początek i koniec przedziału
  long k;     // indeks pozycji stosu
  tdana pom, x;  // x - element wybierany względem, którego podział

  k=0;                  // pozycja początkowa na stosie
  stos[k][0] = a;    // początek przedziału
  stos[k][1] = b;    // koniec przedziału

 do {          // pobranie indeksów przedziału z wierzchołka stosu
   a = stos[k][0];
   b = stos[k][1];
   k--;
     do {                       // dziel przedział [ t[a], t[b] ]
      i=a; j=b;                 // indeksy przedziału początkowego
      x =  t[(a+b)/2];          // element ze środka tablicy
       do {
         while (t[i] < x) i++;        // poszukiwanie
                                      // elementu nie mniejszego od x
         while (x < t[j]) j--;        // poszukiwanie
                                      // elementu nie większego od x
       if (i<=j)   {                  // wymiana elementów
        pom = t[i];
        t[i] = t[j];
        t[j] = pom;
        i++;   j--; }
          } while (i<=j);             // trzecie do

        if (i < b) {                  // prawy przedział na stos
        k++;
        stos[k][0] = i;
        stos[k][1] = b;
                }
          b=j;                        // nowy przedział lewy
          } while (a < b);      // drugie do
 } while (k>=0);               // pierwsze do
}

void main()
{
     long i;      clrscr();

     for (i=0; i<N; i++) cout << tab[i] << "| ";  cout << endl;

     n_qsort(tab, stos, 0, N-1);  // sortowanie tablicy tab

     cout << endl << endl;

     for (i=0; i<N; i++) cout << tab[i] << "| ";     getch();
}
```

Funkcja qsort

Standardowa funkcja qsort zaimplementowana w języku ANSI C pozwala uporządkować tablicę obiektów dowolnego typu według zadanego kryterium. Sposób porządkowania elementów jest definiowany przez programistę za pomocą funkcji, której wskaźnik jest jednym z parametrów funkcji. Prototyp funkcji qsort ma następującą postać:

```
void qsort (
            void *base,
            size_t nelem,
            size_t width,
            int (*fcmp)(const void *a, const void *b)
           );
```

Wskaźnik *base* zawiera adres początku sortowanego obszaru (adres początku tablicy), *nelem* definiuje liczbę elementów, *width* określa rozmiar pojedynczego elementu tablicy, natomiast *fcmp* jest wskaźnikiem funkcji porządkującej elementy. Funkcja powinna być zaprojektowana przez programistę w taki sposób, aby porównywała elementy i zwracała wynik zależny od wyniku porównania. Jeśli elementy wskazywane są równe (w sensie przyjętego kryterium porządkowania) to funkcja powinna zwracać wartość zero, jeśli element wskazywany przez a jest „większy" niż b, to funkcja powinna zwrócić wartość typu int większą od 0 (*a >*b), w przeciwnym przypadku funkcja powinna zwrócić wartość typu int mniejszą od 0 (*a < *b).

Przykład. 13.13. Sortowanie tablicy liczb typu long za pomocą qsort.

```
const int N=10;

int por(const void *a, const void *b)  // funkcja porównująca
{
   long x = *(long *)a;   // rzutowanie do typu elementu tablicy
   long y = *(long *)b;
   if (x==y) return 0;
   if (x>y)  return 1;
   return -1;
}
long tab[N] = { 7, 3, 0, 1, 3, 0, 5 };          // tablica liczb typu long

void main()
{
    long i;    clrscr();     for (i=0; i<N; i++) cout << tab[i] << "| ";

    qsort(tab, N, sizeof(long), por);         // lub sizeof(tab[0])

    cout << endl << endl;  for (i=0; i<N; i++) cout << tab[i] << "| ";
    getch();   // wynik:    0 | 0 | 0 | 0 | 0 | 1 | 3 | 3 | 5 | 7 |
}
```

W kolejnym przykładzie sortowane będą elementy tablicy typu tdana.

Przykład. 13.14. Sortowanie tablicy liczb typu tdana.

```
const int N=10;

typedef double tdana;
typedef tdana ttab [N];

int por(const void *a, const void *b)
{
   tdana x = *(tdana *)a;  // rzutowanie do typu elementu tablicy
   tdana y = *(tdana *)b;
   return (x>y) - (x<y);                    // x==y , to return 0
}

ttab tab = { 1.0, 1.23, -2.34, 1.5, 1.43, 1.5, 2.2, 3, 2.45, 0};

void main()
{
    long i;    clrscr();      for (i=0; i<N; i++) cout << tab[i] << "| ";

    qsort(tab, N, sizeof(tdana), por);      // lub sizeof(tab[0])

    cout << endl << endl;    for (i=0; i<N; i++) cout << tab[i] << "| ";
    getch();
}
```

Następny przykład ilustruje sposób sortowania tablicy łańcuchów.

Przykład. 13.15. Sortowanie tablicy łańcuchów.

```
int por(const void *a, const void *b)
{
   char *x = (char *)a;     // rzutowanie do wskaźnika na typ char
   char *y = (char *)b;
   return strcmp(x,y);      // porównanie łańcuchów
}

// tablica 6 łańcuchów o rozmiarze 20

char stab[][20] = { "Wanda", "Ola", "Ala", "Basia", "Asia", "Joasia" };
```

```
void main()
{
    long i, N;
    clrscr();     N = sizeof(stab)/sizeof(stab[0]);   // liczba łańcuchów

    for (i=0; i<N; i++) cout << stab[i] << "| ";

    qsort(stab, N, sizeof(stab[0]), por);

    cout << endl << endl;    for (i=0; i<N; i++) cout << stab[i] << "| ";
    getch();
}
```

13.5. Złożone definicje wskaźnikowe

W języku C można definiować wskaźniki do złożonych struktur danych, np. tablic i funkcji. W definicjach wykorzystywane są: operator wyłuskania (*), operator indeksowania [ ], operator wywołania funkcji ( ) oraz nawiasy (...).

Przykłady definicji wskaźników różnych typów.

```
long *i;  // i - wskaźnik zmiennej typu long

long * *p; //  p -wskaźnik na wskaźnik zmiennej typu long

void *q; //  q - wskaźnik na zmienną typu void

void far *z;  //  z - daleki wskaźnik na zmienną typu void

char * ts[5];  // ts - tablica 5 wskaźników zmiennych typu char

float * (*wt) [3];  // wt - wskaźnik tablicy 3 wskaźników
                    //  zmiennych typu float
double (*f) (double);  // f - wskaźnik funkcji o parametrze typu double
                       // i zwracającej wynik typu double
float * (*g)(void);   // g - wskaźnik funkcji bezparametrowej, która
                      //  zwraca wskaźnik zmiennej typu float;
int (* (*h)(void) )[4];  // h - wskaźnik funkcji bezparametrowej
                         // zwracającej wskaźnik tablicy 4 elementów typu int
float ( * (*r)[6] )(int);  // r – wskaźnik tablicy 6 wskaźników do funkcji
                           //  o argumencie typu int oraz wartościach typu float
int (* f( ) ) [3];  // f – funkcja zwracająca wskaźnik do tablicy
                    //  3 elementów typu int
```

Czytając (tworząc) złożone deklaracje wskaźnikowe można posługiwać się następującymi regułami:

1) Odczytujemy nazwę wskaźnika, np. x.

2) Następnie od nazwy przesuwamy się na prawo, gdzie mogą znajdować się operatory o najwyższym priorytecie, takie jak: operator wywołania funkcji ( ) lub operator indeksowania tablicy [ ]; jeśli pojawi się x( ), to x jest funkcją, natomiast jeśli x[ ], to x jest tablicą.

3) Jeśli na prawo od nazwy nic już nie ma, lub pojawi się zamykający nawias ')', to zaczynamy czytanie w lewo; czytanie w lewo kontynuujemy aż do momentu, gdy wszystko przeczytamy lub do napotkania nawiasu zamykającego '('; jeśli podczas czytania w lewo napotkamy '*', to mamy do czynienia z deklaracją wskaźnikową; np. *x() – x jest funkcją, która zwraca wskaźnik; *x[ ] – x jest tablicą wskaźników.

4) Jeśli podczas czytania w lewo pojawi się nawias zamykający, to wychodzimy na zewnątrz nawiasu i ponownie zaczynamy czytanie w prawo, czyli wracamy do punktu 2.

5) Procedurę czytania powtarzamy tak długo, aż przeczytamy całą deklarację.

W przypadku, gdy są wątpliwości jak zdefiniować wskaźnik do złożonej struktury danych można wykorzystać specyfikator typedef.

```
float ( * (*r)[6] )(int);  // r – wskaźnik tablicy 6 wskaźników do funkcji
                           //  o argumencie typu int oraz wartościach typu float
```

Zaczynając od ostatniego elementu definicji - funkcja o argumencie typu int oraz wartościach typu float, lub od razu od wskaźnika funkcji.

```
typedef float (*s) (int);    // s – wskaźnik funkcji o argumencie typu int
                             // i wartościach typu float
typedef s tab[6];   // tablica 6 wskaźników funkcji typu s

tab *wsk;   // wskaźnik tablicy typu tab;
            // (*wsk)[0] – pierwszy wskaźnik funkcji (element tablicy).
```