Wykład 14

14. Struktury i unie

Słowo kluczowe ***struct*** umożliwia zgrupowanie w pojedynczym rekordzie kilku zmiennych różnych typów (składowych - pól struktury). Składowymi struktury mogą być zmienne proste dowolnego typu, tablice, inne struktury, zmienne wskaźnikowe i wskaźniki struktur (w tym wskaźnik do struktury definiowanej) oraz funkcje.

Struktura (zmienna strukturalna) może składać się z kilku pól różnych typów. Każde pole ma zarezerwowane osobne miejsce w pamięci. Pola są umieszczane w pamięci szeregowo zgodnie z kolejnością występowania w strukturze. Struktura przechowuje w danym momencie wartości wszystkich swoich składowych.

Unia jest strukturą, której składowe są umieszczane równolegle w tym samym obszarze pamięci. W danym momencie unia przechowuje wartość tylko jednej składowej - tej, która została zmodyfikowana jako ostatnia.

Struktury i unie reprezentują złożone obiekty danych. Dzięki zgrupowaniu zmiennych różnych typów pod jedną nazwą ułatwiają one wykonywanie na obiektach danych wielu złożonych operacji.

14.1. Deklarowanie i definiowanie struktur

Deklaracja struktury (unii) jest wzorcem, który opisuję budowę struktury. Sama deklaracja typu struktury nie powoduje rezerwacji pamięci, a jedynie określa matrycę według, której będą tworzone zmienne strukturalne. Deklaracja struktury może być połączona z definicją zmiennej strukturalnej (zmiennych strukturalnych). Deklaracja struktury kończy się średnikiem.

Deklaracja struktury (unii) składa się ze słowa kluczowego struct (union) oraz deklaracji zmiennych - składowych struktury, które deklaruje się tak jak inne zmienne. W ogólnym przypadku deklaracja struktury ma następującą postać.

struct *nazwa_typu_strukturalnego* {

```
   typ1 nazwa-zmiennej, zmienne, ...;        // składowe typu 1
   typ2 nazwa-zmiennej, zmienne, ...;        // składowa typu 2
  ...                                        // kolejne składowe
} nazwy_zmiennych_strukturalnych;            //  średnik kończący
```

Przykład 14.1. Deklarowanie struktur i unii.

```
struct tosoba  {
                char naz[19];
                unsigned rok,mies,dzien;
                long id;
                 }  o1, o2;
```

Typ strukturalny tosoba składa się z pięciu pól:
- pola char naz[19];
- trzech pól unsigned rok, mies, dzien;
- pola long id.

Zdefiniowano dwie zmienne strukturalne o1, o2 typu tosoba.

```
union tdana {
                char z[5];
                int     k;
                long p;
                float w;
               } u1, u2;
```

Unia typu tdana składa się z czterech pól:
- pola char z[5];
- pola int k;
- pola long p;
- pola float w.

Zdefiniowano dwie unie u1, u2 typu tdana.

Warianty deklaracji

Nazwa typu strukturalnego definiuje typ struktury. Może ona być wykorzystywana do definiowania zmiennych strukturalnych. W języku C++ definicja zmiennej nie musi być poprzedzona słowem struct lub union i nazwą typu, wystarczy jedynie sama nazwa typu.

```
tosoba stud1, stud2;     // dwie struktury typu tosoba

tdana d1, d2, d3;        // trzy unie typu tdana
```

Nazwę typu strukturalnego można pominąć, ale wówczas aby korzystać ze struktury należy podać nazwę zmiennej strukturalnej.

```
struct  {                           // struktura bez nazwy typu
          char naz[19];
          unsigned rok,mies,dzien;
          long id;
        } o1;
```

Nazwa struktury i nazwy zmiennych strukturalnych mogą pokrywać się z nazwami innych zmiennych niestrukturalnych oraz z nazwami składowych struktury. Jednak nie zaleca się stosowania takich samych nazw dla struktur i innych zmiennych.

```
struct tosoba1  {
                char tosoba1[19];     // nazwa pola tosoba1 == nazwa typu
                unsigned rok,mies,dzien;
                long id;
                 }  o1, rok;     // nazwa pola rok == nazwa zmiennej
```

Struktury i unie można deklarować za pomocą specyfikatora typedef. Etykieta w polu zmiennych służy wówczas jako alternatywna nazwa typu.

```
typedef struct tosoba  {

                char naz[19];
                unsigned rok,mies,dzien;
                long id;

               }  DANA;      // DANA - alternatywna nazwa typu tosoba
```

W języku C i C++ możliwe są deklaracje zmiennych w postaci:

```
struct tosoba w1;   // zmienna typu tosoba

DANA w2;            // zmienna typu tosoba
```

W języku C++ można używać tylko samej nazwy typu bez słowa struct.

```
tosoba w3;      // zmienna typu tosoba

static tosoba w4;   // zmienna statyczna typu tosoba
```

Inicjowanie struktur i unii

W języku ANSI C pola struktur mogą być inicjowane w momencie definicji podobnie jak inicjuje się elementy tablic. Korzysta się w tym celu z nawisów klamrowych, w których zawarta jest lista wartości rozdzielonych przecinkami. Wartości na liście są wstawiane do pól zgodnie z kolejnością ich deklarowania. Na przykład pole naz zmiennej o1 typu tosoba jest inicjowane stałą "Kowal", pole rok stałą 1985, pole mies stałą 11, pole dzien stałą 9, a pole id stałą 123.

```
struct tosoba {
                char naz[19];
                unsigned rok, mies, dzien;
                long id;
               } o1= { "Kowal",1985,11,9, 123 },
                 o2 = { "Kostek",1981,10,9, 234 };

tosoba o3 = { "Kowalski", 1984,6,15, 345 };  // naz = "Kowalski"
```

Jeśli na liście inicjacyjnej zmiennej strukturalnej jest mniej wartości niż składowych, to w przypadku struktur statycznych i zewnętrznych pozostałe składowe są inicjowane zerami, natomiast w przypadku struktur automatycznych zależy to od kompilatora.

```
tosoba o4 = { "Nowak" };    // zm. zewn. - pozostałe składowe są zerami

static tosoba o5 = { "Nowaczek", 1990 };     // pozostałe pola są zerami
```

W przypadku unii wszystkie składowe są pamiętane we wspólnym obszarze pamięci. Dlatego unię inicjuje się *jedną daną* odpowiadającą typowi *pierwszej składowej*. Jeśli dana jest innego typu to następuje automatyczna konwersja (o ile jest to możliwe) do typu pierwszego pola unii. W przypadku unii statycznych (static) lub zewnętrznych pola niezainicjowane są ustawiane na zero.

```
union tdana {
              char z[5];    int k;   long p;    float w;
             } u1 = { 'A', 'B'}, u2 = {"Ala"};     // unie u1, u2

tdana u3 = { 65 };   // z[0] = 'A';  pozostałe pola tablicy - zera

tdana u4 = { 66.42 };  // z[0] = 'B';  rzutowanie (char) 66.42 = 66
```

14.2. Dostęp do pól struktur

Dostęp do składowych struktury (unii) jest realizowany za pomocą operatora wyboru oznaczanego jako . (kropka).

Przykład 14.2. Wydruk zawartości struktur i unii.

```
typedef struct tosoba {
                 char naz[19];
                 unsigned rok, mies, dzien;
                 long id;
                };

typedef union tdana {
               char z[5];
               int    k;
               long p;
               float w;
              };

tosoba z1= { "Kowal",1985,11,9, 123 };    // struktura z1 typu tosoba
tdana u1 = {'A', 'B' };    tdana u2 = {'C'};   // unie u1, u2 typu tdana

void main() {
clrscr();

// wydruk wierszami za pomocą printf
printf("%s\n", z1.naz);      //  "Kowal"
printf("%u\n%u\n%u\n", z1.rok, z1.mies, z1.dzien);
printf("%ld\n", z1.id);
printf("\n");

// wydruk w jednym wierszu za pomocą cout
cout << z1.naz << " " << z1.rok << " ";
cout << z1.mies << " " << z1.dzien << " " << z1.id << endl;
cout << endl;

// wydruk unii wierszami za pomocą printf
printf("%s\n", u1.z);   //   u1  "AB"
printf("%s\n", u2.z);   //   u2   "C"
printf("%d\n", u2.k);   //   u2   67
}
```

Pola struktur mogą być zmieniane w programie i przekazywane do funkcji podobnie jak inne zmienne.

Przykład 14.3. Nadawanie wartości składowym struktur.

```
tosoba z1, z2;    // struktury z1 i z2

void main()
{
   strcpy(z1.naz, "Kos");
    z1.rok = 2000;      z1.mies = 5;       z1.dzien = 11;
    z1.id = 127;

    z2 = z1;    // podstawienie bezpośrednie

    printf("%s", z2.naz);
    printf("%5u %5u %5u", z2.rok, z2.mies, z2.dzien);

    printf("\nPodaj id:  ");
    scanf("%ld", &z2.id);                // wczytanie do pola id
    printf("id = %7ld\n", z2.id);        getch();
}
```

Rezerwacja pamięci

Pola struktury są umieszczane w pamięci zgodnie z kolejnością występowania w deklaracji. Każde pole struktury zajmuje osobny obszar pamięci. Łączny rozmiar struktury jest równy sumie rozmiarów jej składowych.

Przykład 14.4. Rozmieszczenie składowych struktury w pamięci.

```
struct tosoba {
                char naz[19];
                unsigned rok, mies, dzien;
                long id;
               } z2;    // zmienna strukturalna z2
```

Rozmiar struktury jest zawsze większy lub równy sumie rozmiarów jej składowych, tj.

```
sizeof(z2) >= sizeof(z2.naz) + sizeof(z2.rok) + sizeof(z2.mies) +
              + sizeof(z2.dzien) + sizeof(z2.id) = 19 + 2 + 2 + 2 + 4 = 29.
```

Rozmiar struktury zależy od parametrów kompilatora. W przypadku kompilatora BC++3.1 jeśli nie została ustawiona opcja *Options-Compiler-Code_Generation-Word_Alignment* składowe są umieszczane szeregowo jedna za drugą. Wówczas rozmiar struktury jest równy sumie rozmiarów jej składowych, czyli 29. Natomiast jeśli opcja jest wybrana wówczas wszystkie składowe struktury są umieszczane począwszy od adresów parzystych (wyrównywanie do granicy słowa). Wynika stąd, że pomiędzy polem naz i polem rok jest jeden bajt niewykorzystywany, ale uwzględniany w rozmiarze struktury, który jest równy 30.

```
void main()
{
  printf("\nRozmiar struktury %d \n", sizeof(z2) );

 // Word Alignment wyłączony – sizeof(z2)=29
 // Word Alignment włączony –  sizeof(z2)=30

// Adresy składowych struktury szesnastkowo i dziesiętnie

printf("%p = %d\n", &z2.naz[0],  &z2.naz[0] );
printf("%p = %d\n", &z2.rok,  &z2.rok );
printf("%p = %d\n", &z2.mies, &z2.mies);
printf("%p = %d\n", &z2.dzien,&z2.dzien);
printf("%p = %d\n", &z2.id, &z2.id );
getch();
}

// offsety przy  Word Alignment wyłączony

00E4 = 228          &z2.naz[0]
00F7 = 247          &z2.rok
00F9 = 249          &z2.mies
00FB = 251          &z2.dzien
00FD = 253          &z2.id

// offsety przy  Word Alignment włączony

00E6 = 230      =   &z2.naz[0]
00FA = 250      =   &z2.rok
00FC = 252      =   &z2.mies
00FE = 254      =   &z2.dzien
0100 = 256      =   &z2.id
```

Pola unii są umieszczane równolegle w tym samym obszarze pamięci. W związku z tym rozmiar unii jest równy rozmiarowi jej największej składowej, a wszystkie składowe mają ten sam adres.

Przykład 14.5. Rozmieszczenie składowych unii w pamięci.

```
union tdana {
                char z[5];
                int     k;
                long p;
                float w;
               } u2;

Rozmiar unii u2:

sizeof(u2) = max( sizeof(z), sizeof(k), sizeof(p), sizeof(w) ) = 5.

void main()
{
printf("\nRozmiar unii %d \n", sizeof(u2) );

printf("\nRozmiary składowych unii %d %d %d %d\n", sizeof(u2.z),
sizeof(u2.k), sizeof(u2.p), sizeof(u2.w) );

// Adresy składowych unii szesnastkowo i dziesiętnie

printf("%p = %d\n", &u2.z[0],  &u2.z[0] );
printf("%p = %d\n", &u2.k,  &u2.k );
printf("%p = %d\n", &u2.p, &u2.p);
printf("%p = %d\n", &u2.w, &u2.w);

getch();
}

Rozmiar unii 5

Rozmiary składowych unii 5 2 4 4

0106 = 262      =      &u2.z[0]
0106 = 262      =      &u2.k
0106 = 262      =      &u2.p
0106 = 262      =      &u2.w
```

Funkcje jako składowe struktur

W języku C++ składowymi struktur i unii mogą być również funkcje. Na ogół funkcje będące składowymi struktur realizują operacje z wykorzystaniem odpowiednich pól. Rozwiązanie takie prowadzi do lepszego powiązania danych z wykonywanymi na nich operacjami i stanowi podstawę programowania obiektowego.

Przykład 14.6. Wykorzystanie funkcji jako składowych struktur.

```
struct tosoba {
                char naz[19];
                unsigned rok,mies,dzien;
                long id;
                // funkcje inicjujące obiekt typu tosoba
                void ustaw_naz(char nowa[])
                { strcpy(naz, nowa); }
                void ustaw_data(unsigned r, unsigned m, unsigned d)
                {   rok = r;  mies = m; dzien = d; }
                void ustaw_id(long d)
                {  id = d;   }
                //  prototyp funkcji wyprowadzającej obiekt typu tosoba
                void pisz(tosoba);
              } z1 = { "Nowak",1981,10,9,234 },
                z2 = { "Kowal",1981,11,8,123 },
                z3 = { "Lis",1981,12,7,434 };

// definicja funkcji wyprowadzającej obiekt typu tosoba
 void tosoba::pisz(tosoba x)
 {
    printf("%-10s", x.naz);
    printf("%5u %5u %5u", x.rok, x.mies, x.dzien);
    printf("%7ld\n", x.id);
   }

void main()
{
   clrscr();   z1.pisz(z1);     z2.pisz(z2);      z3.pisz(z3);
   z1.ustaw_naz("Dec");  z1.ustaw_data(2002,1,9);  z1.ustaw_id(102);
   z1.pisz(z1);
   getch();
}
```

Struktury zagnieżdżone

Składowymi struktur mogą być również inne struktury. Odwołanie do pola struktury, która sama jest składową innej struktury odbywa się z wykorzystaniem dwóch kropek.

Przykład 14.7. Dostęp do pola struktury tdata, która jest składową struktury tosoba.

```
 struct tdata {
                unsigned rok,mies,dzien;
            };

struct tosoba {                  // typ strukturalny - tosoba
                char naz[19];
                tdata data;     // pole - struktura typu tdata
                long id;
              } z1 = { "Nowak", {1981,10,9}, 234 },
                z2 = { "Kowal",1981,11,8,123 };
                                 // zmienne strukturalne z1, z2
void main()
{
   clrscr();
    z1.data.rok = 2002;   // dostęp do pól struktury zagnieżdżonej
    z1.data.mies= 3;
    z1.data.dzien = 9;

    printf("Pola struktury\n");
    printf("%-10s", z1.naz);
    printf("%5u", z1.data.rok);
    printf("%5u", z1.data.mies);
    printf("%5u", (z1.data).dzien);
    printf("%7ld\n", z1.id);
    getch();
}
```

Konstrukcja z1.data.rok jest interpretowana w kierunku od lewej do prawej jako (z1.data).rok. Najpierw jest odnajdowana struktura z1, jej składnik o nazwie data, a następnie pole składnika data o nazwie rok. W przypadku zagnieżdżenia większej liczby składowych strukturalnych dostęp do składowych jest realizowany w oparciu o odpowiednią liczbę kropek.

Unie bez nazwy

W języku C++ istnieje możliwość deklarowania unii bez nazwy typu i bez definiowania nazwy zmiennej. Pola unii są wówczas traktowane jak zwykłe zmienne, które zajmują ten sam obszar pamięci. W odwołaniach do składowych unii są wykorzystywane jedynie nazwy składowych.

Przykład 14.8. Wykorzystanie unii bez nazwy.

```
static union {              // unia anonimowa
       int    a[3];
       char b[6];
       long c[2];
       double d;
          };
// int d;     // błąd kompilatora

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

     printf("Pola unii\n");
     for (i=0; i<3; i++)
     { a[i]=i;  printf("%4d", a[i]=i); }    // tablica a: 0 1 2
     printf("\n");

     for (i=0; i<6; i++)
     { b[i] = 65+i;   printf("%3c", b[i]); }   // tablica b: A B C D E F
     printf("\n");

     for (i=0; i<2; i++)
     { c[i] = 48+i; printf("%4ld", c[i]); }    // tablica c: 48 49
     printf("\n");

     d = 10.23;                                  // zmienna d = 10.2300
     printf("%9.4lf \n", d);
     getch();
}
```

Dzięki wykorzystaniu unii anonimowej uzyskujemy oszczędność pamięci oraz prostotę notacji. Należy jednak pamiętać, aby nie definiować zmiennych o tych samych nazwach co pola unii bez nazwy.

14.3. Pola bitowe

W przypadku pól typu *całkowitego* można w deklaracji struktury (unii) określić ile bitów będą one zajmowały. Zakres wartości, które można przechowywać w danym polu zależy od liczby bitów przydzielonych polu oraz od tego czy jest to pole ze znakiem, czy bez znaku. Najmniejsza ilość pamięci przeznaczona na pojedyncze pole wynosi 1 bit. Największa wynosi sizeof(int)=16. Jeśli pole 1-bitowe jest typu signed int (char), to może ono przyjmować wartości 0 lub –1. Jeśli pole jest typu unsigned int (char), to może ono przyjmować wartości 0 lub 1. Jeśli nie podamy czy pole typu int (char) jest signed czy unsigned, to sposób interpretacji zależy od kompilatora (na ogół int jest interpretowane jako signed int).

Pole bitowe definiuje się podając typ danej, nazwę pola i po dwukropku liczbę bitów wykorzystywanych przez pole.

Przykład 14.9. Pole bitowe w strukturze.

```
struct  atrybut {
                int bit1       : 1;   // pole jednobitowe int
} A;

struct  atrybut1 {
            unsigned bit1     : 1;  // pole jednobitowe unsigned
} B;

void main()
{
A.bit1 = 1;
printf("%d\n", A.bit1);     // wartość = -1

A.bit1 = 0;
printf("%d\n", A.bit1);      // wartość = 0

B.bit1 = 1;
printf("%u\n", B.bit1);     // wartość =  1

B.bit1 = 0;
printf("%u\n", B.bit1);      // wartość = 0
}
```

Pola struktur są umieszczane w pamięci jedno za drugim zgodnie z kolejnością w jakiej zostały zadeklarowane w strukturze. Sposób rozmieszczenia pól bitowych w pamięci zależy od kompilatora. Na ogół pola są rozmieszczane zgodnie z następującymi regułami:

- na granicy bajtów pola mogą się przełamywać;
- jeśli w kompilatorze włączone jest wyrównywanie do granicy słów (Word alignment), to jedno pole nie może należeć do dwóch różnych słów (jeśli pole nie mieści się w słowie, to jest umieszczane w następnym słowie – wyrównywanie do granicy danych 16-bitowych);
- pola mogą być rozmieszczane począwszy od najmłodszego bitu słowa do najstarszego bitu (najczęściej) lub odwrotnie.

Dostęp do pól bitowych jest realizowany w taki sam sposób jak do zwykłych pól. Kompilator automatycznie dodaje kod umożliwiający odczytanie zawartości pola.

Rozmiar struktury jest zawsze większy lub równy niż suma rozmiarów pól składowych.

Przykład 14.10. Wyznaczanie rozmiaru struktury z polami bitowymi.

```
struct  atrybut {
  int bit1          : 1;
  int bity2_10   : 9;
  int bit11         : 1;
  int bity12_23 : 12;       // rozmiar struktury 23 bity
} A;

void main()
{
A.bit1 = 1;    A.bity2_10  = 32;    A.bit11 = 1;    A.bity12_23 = 1024;

printf("%d\n", A.bit1);                 // -1
printf("%d\n", A.bity2_10);           // 32
printf("%d\n", A.bit11);                // -1
printf("%d\n", A.bity12_23);         // 1024

printf("%d\n", sizeof(A));           // 3 bajty, jeśli Word alignment = off
// printf("%d\n", sizeof(A));        // 4 bajty, jeśli Word alignment = on
}
```

Jeśli opcja Word alignment jest wyłączona, to pola bit1, bity2_10, bit_11 oraz 5 bitów słowa bity12_23 są pamiętane w pierwszych dwóch bajtach (pierwsze słowo). Kolejne 7 bitów słowa bity12_23 jest pamiętane w trzecim bajcie.
Jeśli opcja Word alignment jest włączona, to pola bit1, bity2_10, bit_11 są pamiętane w pierwszych dwóch bajtach (pierwsze słowo), natomiast pole bity12_23 jest pamiętane w trzecim i czwartym bajcie (drugie słowo) - pole nie może należeć do dwóch różnych słów.

W celu dokładnego rozmieszczenia pól bitowych w pamięci można wykorzystać pola bez nazwy oraz pola mające szerokość równą 0. Pole bez nazwy służy jako wypełniacz dla pól nazwanych, natomiast pole o szerokości zero jest sugestią, by kolejne pole zaczynało się w następnym bajcie (lub słowie – jeśli Word alignment = on).

Przykład 14.11. Rozmieszczanie pól bitowych BC++3.1.

```
struct  atrybut {
  int bit1          : 1;      // 1 bajt              // 1 słowo
                    : 5;
  int bity2_10   : 9;      // 2 bajt
                    : 0;
  int bit11         : 1;      // 3  bajt             // 2 słowo
                    : 0;
  int bity12_23  : 12;     // 4 i 5 bajty         // 3 słowo
} A;                  // 28 bitów

void main()
{    printf("%d\n", sizeof(A));      // 5 bajtów, jeśli Word alignment = off
   //  printf("%d\n", sizeof(A));      // 6 bajtów, jeśli Word alignment = on
}
```

Jeśli opcja Word alignment jest wyłączona, to pole bit1 oraz pole nienazwane o szerokości 5 są pamiętane w pierwszym bajcie, pole bity2_10 w bajcie pierwszym i drugim, pole bit_11 zaczyna się od trzeciego bajtu, natomiast pole bity12_23 jest pamiętane w bajtach czwartym i piątym.

Jeśli opcja Word alignment jest włączona, to pole bit1, pole nienazwane o szerokości 5 oraz pole bity2_10 są pamiętane w pierwszym słowie, pole bit_11 zaczyna się w drugim słowie, natomiast pole bity12_23 jest pamiętane w trzecim słowie.

W przypadku unii z polami bitowymi wszystkie dane są pamiętane we wspólnym obszarze pamięci, którego rozmiar jest równym rozmiarowi najszerszego pola bitowego.

Struktury i unie z polami bitowymi ułatwiają wykonywanie operacji na bitach danych. Mogą one być wykorzystywane w zastępstwie operatorów bitowych (np. &, ^, |).

14.4. Pola wskaźnikowe

Struktury mogą zawierać pola typu wskaźnikowego. Reguły inicjowania i modyfikowania składowych wskaźnikowych są takie same jak w przypadku innych zmiennych wskaźnikowych. W szczególności należy pamiętać, aby nie wstawiać danych do obszaru pamięci, który nie został przydzielony, np. poprzez użycie wskaźnika o przypadkowej wartości.

```
struct tosoba {
                char naz[19];      // tablica znakowa
                char *imie;        // wskaznik do char
                int far * pit;     // daleki wskaźnik do int
                unsigned r, m, d;
                long id;           // dla kompilatorów 32-bit. bez far
      }  z1 = { "Kowal","Adam", (int far *)0xB8000000, 1981,11,8, 123},
         z2 = { "Nowak", NULL, NULL, 1981,10,9, 234 };
```

Szczególną uwagę należy zwrócić podczas używania składowych, które są *wskaźnikami do typu char*. Składowe takie można inicjować stałymi łańcuchowymi w momencie definicji oraz podstawiać pod nie inne stałe łańcuchowe w programie. Zawierają one wówczas adresy miejsc pamięci, w których kompilator przechowuje stałe łańcuchowe. W szczególności pole z1.imie zawiera adres miejsca w pamięci, w którym znajduje się stała ”Adam”. Pod to pole można podstawiać inne stałe łańcuchowe, ale tylko o rozmiarze nie większym niż sizeof(”Adam”), np. z1.imie =”Olek”. Dane o większym rozmiarze mogą skasować stałe znajdujące się w pamięci za stałą ”Adam”. Natomiast, do pola z2.imie nie można podstawiać stałych łańcuchowych, gdyż z polem tym nie jest skojarzony żaden obszar pamięci (wskaźnik z2.imie nie wskazuje na zarezerwowany obszar pamięci). Można więc wczytywać dane do pól z1.naz i z1.imie (ale tak, aby nie przekroczyć zakresu pól, tj. 19 znaków dla naz i 5 znaków dla imie), np.

```
gets(z1.naz);    // 18 znaków + ‘0’          gets(z1.imie).  //    4 znaki + ‘0’
```

Nie wolno jednak wczytywać danych do pamięci wskazywanej przez niezainicjowane pola wskaźnikowe, np.

```
gets(z2.imie);     // nie wolno wczytywać pod adres z2.imie
                   // bo z polem z2.imie
                   // nie jest skojarzony żaden obszar pamięci
```

Podobnie nie są poprawne operacje wczytywania do pola pit.

```
scanf("%d", z1.pit);    // z1.pit zawiera adres pamięci, ale pamięć
                        // nie była rezerwowana
                        // nie wolno wczytywać pod ten adres

scanf("%d", z2.pit);   // z2.pit nie wiadomo na co wskazuje
                       // i pamięć nie była rezerwowana
                       // nie wolno wczytywać pod ten adres
```

W wymienionych przykładach kompilacja przebiegnie poprawnie, ale działanie programu jest błędne i może prowadzić do z góry nie przewidywalnych skutków.

Do przechowywania danych łańcuchowych w strukturach lepiej wykorzystywać składowe będące tablicami znaków, gdyż jest wtedy mniejsze ryzyko popełnienia błędu. Jeśli natomiast wykorzystuje się pola wskaźnikowe, to należy pamiętać, aby przechowywać w nich adresy stałych łańcuchowych lub istniejących obszarów pamięci, np. tablic znakowych lub obszarów pamięci przydzielonych dynamicznie na stercie.

Przykład 14.12. Wykorzystanie pól wskaźnikowych w strukturach.

```
struct tosoba {
                char naz[19];   // tablica znakowa
                char *imie;     // wskaznik do char
                int far * pit;  // wskaznik do int
                unsigned r, m, d;
                long id;
                   // dla kompilatorów 32-bitowych bez far
}   z1 = { "Kowal","Adam", (int far *)0xA1000000, 1981,11,8, 123 },
    z2 = { "Nowak", NULL, NULL, 1981,10,9, 234 };
```

```
void main()
{
    int k;
    clrscr();

     printf("Pola struktury\n");
     printf("%-10s", z1.naz);      printf("%-10s", z1.imie);
     printf("%Fp ", z1.pit);
     printf("%5u", z1.r);  printf("%5u", z1.m);   printf("%5u", z1.d);
     printf("%7ld\n", z1.id);
     for (k=0; k<4; k++)
     printf("%c", *(z1.imie + k)); // Adam wydruk zawart. pamieci z1.imie
     printf("\n");

        //  z1.pit = (int far *)0xB8000000;
         // adres pamięci ekranu w BC++3.1– tu można wpisywać dane
         //*z1.pit = 0x0741;     // wpisanie do komórki pamięci z1.pit
                                 // 0xB800:0x0000 = 0x41 = 65 = 'A'
                                 // 0xB800:0x0001 = 0x07 = 7
    k = sizeof("Jan");           // rozmiar stalej "Jan" = 4
    z2.imie = new char [k];   // przydzial: bufor 4 znakowy na imie
    z2.pit = new int [10];  /    / przydzial: bufor na 10 liczb typu int

    strcpy(z2.naz, "Kosowski"); // wpisanie do bufora naz = "Kosowski"
    if (z2.imie)
    strcpy(z2.imie, "Jan");         // wpisanie do bufora imie = "Jan"

   // z2.imie = "Jan";    // blad ! wpisanie do wskaznika imie adresu "Jan"
                          // skasowanie starego wskaznika
                          // blad przy zwolnieniu pamieci
    if (z2.pit)
    {
        for (k=0; k<10; k++) *(z2.pit + k) = k;  //wpisanie do obszaru pit
    // lub
    //   for (k=0; k<10; k++) z2.pit[k] = 2*k;
    }

    printf("Pola struktury\n");
    printf("%-10s", z2.naz);      printf("%-10s", z2.imie);
    printf("%Fp ", z2.pit);
    printf("%5u", z2.r);   printf("%5u", z2.m);   printf("%5u", z2.d);
    printf("%7ld\n", z2.id);
```

```
    printf("Zawartosc obszaru pit\n");

    if (z2.pit)
    for (k=0; k<10; k++) printf("%5d", z2.pit[k]);   / 0 1 2 3 4 5 6 7 8 9

    delete [] z2.pit;       // zwolnienie obszaru pit
    delete [] z2.imie;    // zwolnienie obszaru imie

getch();
}
```

14.5. Wskaźniki do struktur

W programach można wykorzystywać wskaźniki do struktur. Do wskaźników do struktur można wpisywać adresy istniejących zmiennych lub stałych strukturalnych. Mogą to być zmienne statyczne lub zmienne tworzone dynamicznie. Wskaźnik na strukturę definiuje się tak jak wskaźniki do innych typów danych. Jeśli s jest wskaźnikiem do struktury typu tosoba, to dostęp do składowych można zrealizować na dwa sposoby: za pomocą operatora wyłuskania (*) oraz operatora wyboru (.) lub za pomocą operatora wyboru (->) – myślnik i znak większości.

Przykład 14.13. Wykorzystanie wskaźników do struktur.

```
struct tosoba {
                char naz[19];
                unsigned rok, mies, dzien;
                long id;
               } z1 = { "Kowal",1981,11,8,123 };

tosoba z2 = z1;          // z1,  z2 = z1 - zmienne strukturalne typu tosoba

tosoba *s = &z1;         //  s - wskaźnik na strukturę typu tosoba
                         //  zainicjowany na adres z1
                  // dostęp (*s).rok;    *s.rok - błąd

void main(void)
{
   tosoba *wa = new tosoba(z1);   // wa - wskazanie na zmienną
                                  // dynamiczną typu tosoba inicjacja
                                  // pamięci zawartością zmiennej z1;
                                  // na stercie powstaje kopia z1
```

```
 // lub
 //   tosoba *wa = new tosoba;      //alokacja pamięci bez inicjacji
 //   *wa = z1;          // wpisanie z1 do obszaru o adresie wa

 clrscr();

  printf("Pola struktury z. \n");
  printf("%-10s", z2.naz);
  printf("%5u", z2.rok);   printf("%5u", z2.mies);  printf("%5u", z2.dzien);
  printf("%7ld\n", z2.id);  printf("\n");

  printf("Pola struktury (*s). \n");
  printf("%-10s",  (*s).naz);
  printf("%5u",    (*s).rok);
  printf("%5u",    (*s).mies);
  printf("%5u",    (*s).dzien);
  printf("%7ld\n", (*s).id);   printf("\n");

  printf("Pola struktury s-> \n");
  printf("%-10s", s->naz);
  printf("%5u",   s->rok);
  printf("%5u",   s->mies);
  printf("%5u",   s->dzien);
  printf("%7ld\n",s->id);     printf("\n");

  printf("Wczytanie nowych danych do struktury *wa \n");
  printf("naz: ");      scanf("%s",  wa->naz);        // lub &wa->naz
  printf("rok: ");      scanf("%u",  &wa->rok);
  printf("mies: ");     scanf("%u",  &wa->mies);
  printf("dzien: ");    scanf("%u",  &wa->dzien);
  printf("id: ");       scanf("%ld", &wa->id);   printf("\n");

  printf("Pola struktury wa-> \n");
  printf("%-10s", wa->naz);
  printf("%5u",   wa->rok);
  printf("%5u",   wa->mies);
  printf("%5u",   wa->dzien);
  printf("%7ld\n",wa->id);     printf("\n");

  delete wa;            // zwolnienie pamięci
  getch();
}
```

14.6. Przekazywanie struktur do funkcji

W języku C struktura nie może być argumentem ani wynikiem funkcji. Argumentami i wynikami funkcji mogą być natomiast wskaźniki do struktur. W języku C++ nie ma takich ograniczeń. Struktury mogą być przekazywane do funkcji przez:

- wartości,
- wskaźniki,
- referencje.

W ten sam sposób mogą być również zwracane.

Przykład 14.14. Wykorzystanie struktur jako argumentów i wyników funkcji.

```
struct tosoba {
                  char naz[19];
                  unsigned rok,mies,dzien;
                  long id;
                 } z1 = { "Kowal",1981,11,9,123 };

// wyprowadzanie struktur tosoba

void pisz1(tosoba s)    // przekazanie struktury przez wartość
{
  printf("%20s",s.naz);
  printf("%5u",s.rok);  printf("%5u",s.mies);   printf("%5u",s.dzien);
  printf("%7ld",s.id);
  printf("\n");
}

void pisz2(tosoba *s)   // przekazanie struktury przez wskaźnik
{
  printf("%20s",s->naz);      // lub (*s).naz
  printf("%5u",s->rok);      //       (*s).rok
  printf("%5u",s->mies);    //       (*s).mies
  printf("%5u",s->dzien);   //       (*s).dzien
  printf("%7ld",s->id);       //       (*s).id
  printf("\n");
}
```

```
void pisz3(tosoba& s)    // przekazanie struktury przez referencję
{
  printf("%20s",s.naz);
  printf("%5u",s.rok);  printf("%5u",s.mies);   printf("%5u",s.dzien);
  printf("%7ld",s.id);
  printf("\n");
}

// inicjowanie struktur tosoba

tosoba ustaw1(char naz[19], unsigned rok)
{
   tosoba p;
   strcpy(p.naz, naz);    p.rok = rok;
   return p;
}

tosoba *ustaw2(char naz[19], unsigned rok)
 // zwracany wskaźnik do utworzonego obiektu tosoba
{
   tosoba *p = new tosoba;    // nowy obiekt tosoba
   strcpy(p->naz, naz);      p->rok = rok;
   return p;
}

tosoba& ustaw3(char naz[19], unsigned rok)
{
    tosoba s;
    tosoba &p = s;

    strcpy(p.naz, naz);      p.rok = rok;
    return p;
}

tosoba z2 = z1;     // inicjacja z2 zawartością z1

void main()
{
   tosoba *wb, z4;

   clrscr();     pisz1(z2);      pisz2(&z2);       pisz3(z2);
```

```
   z2 = ustaw1("Ala", 1);  // nowa zawartość z2;
                           // tylko pola naz i rok są ustawione;
                           // pozostałe są przypadkowe

   wb = ustaw2("Ola", 2);  // dynamiczne utworzenie obiektu;
                           //  tylko pola naz i rok są ustawione;
                           // pozostałe są przypadkowe

   z4 = ustaw3("Ela", 3);   // z4 - zmodyfikowane naz i rok
                            // pozostałe pola są przypadkowe
   delete wb;     // zwolnienie pamięci
}
```

14.7. Tablice struktur i tablice wskaźników na struktury

W programach można wykorzystywać tablice struktur oraz tablice wskaźników na struktury. Tablice struktur definiuje się podobnie jak tablice innych typów danych.

```
const N = 7;

typedef  struct tosoba {
                  char naz[19];
                  unsigned rok,mies,dzien;
                  long id;
                     };

tosoba tab[N];          // tablica struktur typu tosoba

tosoba* wtab[N]       // tablica wskaźników na struktury typu tosoba
```

Możliwe jest połączenie definicji tablicy struktur z jej inicjacją. Każdy element tablicy jest struktura, którą należy zainicjować zgodnie z zasadami inicjowania struktur.

```
tosoba zbior[ ] ={  {"LechK",  1980, 11, 3, 1},    // tablica zbior
                    {"AlicjaZ", 1981, 5, 12, 2 },
                    {"JanL",    1979, 12, 7, 3 },
                    {"JanL",    1983,  2, 10, 4},
                    {"OlekW", 1984, 11, 28, 5},
                    {"AniaT",   1982, 9, 23, 6},
                    {"ZenonJ",1979, 6, 15, 7}   };
```

Wydruk zawartości tablicy struktur można zrealizować za pomocą funkcji umożliwiających wyprowadzenie struktury oraz funkcji umożliwiającej wyprowadzenie struktury poprzez jej wskaźnik.

```
int N = sizeof(zbior) / sizeof(tosoba);  // rozmiar tablicy o nazwie zbior = 7

for (i=0; i<N; i++) pisz1( tab[i] );    // wydruk tablicy struktur

for (i=0; i<N; i++) pisz2( wtab[i] );  // wydruk tablicy struktur poprzez
                                       // wskaźniki do struktur
```

Przykład 14.15. Wydruk zawartości tablicy struktur.

```
void main(void)
{
  clrscr();
  int N = sizeof(zbior)/sizeof(tosoba);   // rozmiar tablicy zbior

  for (int i=0; i<N; i++) pisz1( zbior[i] );   // wydruk poprzez funkcję
  printf("\n");

  for (i=0; i<N; i++) printf("%20s\n", zbior[i].naz);
  // wydruk bezpośredni nazwisk
}
```

Przykład 14.16. Wydruk zawartości tablicy struktur poprzez tablicę wskaźników na struktury.

```
const N=7;
tosoba* wtab[N]       // tablica wskaźników na struktury typu tosoba

void main(void)
{
  clrscr();

  for (int i=0; i<N; i++)   // inicjacja tablicy wtab adresami struktur zbior
  wtab[i] = &zbior[i];

  for (i=0; i<N; i++) printf("%20s\n", wtab[i]->naz);
    // wydruk tablicy zbior za pomocą tablicy wskaźników
getch();
}
```

Na tablicy struktur można wykonywać różne operacje. W szczególności modyfikację i wydruk zawartości tablic można zrealizować przekazując do funkcji całe tablice. Tablica struktur może być traktowana jako reprezentant bazy danych.

Przykład 14.17. Wyznaczanie liczby osób, których rok urodzenia jest w przedziale [1980, 1982].

```
void zmien1(tosoba t[], int ile)     // przekazanie tablicy struktur
{
   for (int i=0; i<ile; i++) t[i].id = 2*i;   // modyfikacja indeksu id
 }

void zmien2(tosoba* t[], int ile)   // przekazanie tablicy wskaźników
{
 for (int i=0; i<ile; i++) t[i]->id = i;   // modyfikacja indeksu id
}

void main(void)
{
  clrscr();
  int ile=0;

  for (int i=0; i<N; i++)
  if (1980<=zbior[i].rok && zbior[i].rok<=1982)
  {  ile++;      printf("%20s\n", zbior[i].naz);  }

  printf("Liczba osób = %d\n", ile);

  zmien1(zbior, N);     // zmiana indeksów
  for (i=0; i<N; i++) cout << zbior[i].id << endl;

  for (i=0; i<N; i++) wzbior[i]=&zbior[i];    // inicjacja wtab

  cout << endl;

  zmien2(wzbior, N);     // zmiana indeksów
  for (i=0; i<N; i++) cout << wzbior[i]->id << endl;
  getch();
}
```

W kolejnym przykładzie pokazano sposób realizacji różnych operacji na tablicy struktur typu tosoba, w tym:

- inicjację wybranego elementu tablicy losowymi danymi,
- wyszukiwanie elementów według ustalonego kryterium,
- wyprowadzanie zawartości tablicy począwszy od zadanego numeru,
- sortowanie tablicy według wybranego pola.

Przykład 14.18. Zdefiniować zainicjowaną wartościami początkowymi tablicę struktur typu tosoba o rozmiarze N, gdzie N – stała.

```
typedef struct tosoba  {
                char naz[19];
                unsigned rok,mies,dzien;
                long id;
                        };
```

Opracować funkcje:

a) wyprowadzającą na ekran element tablicy o numerze k, gdzie 0 <= k < N;

b) znajdującą pierwszy element tablicy o podanej wartości pola naz, począwszy od pozycji k do pozycji n (0 <= k < n < N), i zwracającą numer pozycji elementu;

c) inicjującą element na pozycji k (0 <= k < N) losowymi wartościami w sposób następujący: do pola naz[20] wstawiany jest łańcuch ‘naz’ zakończony losową liczbą należącą do przedziału [1, 100), np. ‘naz21’, ‘naz3’, itd.; do pozostałych pól wstawiane są losowe liczby należące, odpowiednio, do przedziałów: rok - [1900, 1981], mies: [1, 12], dzien: [1, 31], id: [1, 1000];

d) znajdującą pierwszy element tablicy dla którego data (rok, mies, dzien) jest w przedziale [a, b], gdzie a jest datą początkową, natomiast b – datą końcową, począwszy od pozycji k do pozycji n (0 <= k < n < N), i zwracającą numer pozycji elementu;

e) sortującą elementy tablicy według nazwisk.

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

const N = 8;

typedef struct tdata { unsigned r, m, d; };

typedef  struct tosoba {
                char naz[19];
                unsigned rok,mies,dzien;
                long id;
                  };

tosoba tab[]=  { {"LechK", 1981,11,5,100},  {"WandaZ",1984,2,14,101},
                 {"JanL", 1983,7, 22,102},  {"AniaT",  1981,9,27,103},
                 {"OlekW", 1981,9,18,104},  {"JanL", 1983,7,15,105},
                 {"ZenonJ",1985,1,12,106},  {"AsiaS", 1986,12,7,107}  };

void pisz(tosoba t[], int k)   // wyprowadza element z pozycji k
{
  printf("%20s",t[k].naz);
  printf("%5u", t[k].rok);
  printf("%5u", t[k].mies);
  printf("%5u", t[k].dzien);
  printf("%7ld", t[k].id);
  printf("\n");
}

int szukaj_naz(tosoba t[], char *nz, int k, int n)
{
  int i, p=-1;

  for(i=k; i<n; i++)
  if ( !strcmp(t[i].naz, nz) ) { p=i; break; }
  return p;
}
```

```
void ustaw(tosoba t[], int k)
{
  sprintf(t[k].naz,"Naz%d",random(100));
  t[k].rok    =  random(82)+1900;
  t[k].mies  =  random(11)+1;
  t[k].dzien =  random(31)+1;
  t[k].id      =  random(1000)+1;
}

int szukaj_data(tosoba t[], tdata a, tdata b, int k, int n)
{
  int i, p=-1;
  double x,y, w;

  x=a.r*10000.0 + a.m*100 + a.d;   // data poczatkowa
  y=b.r*10000.0 + b.m*100 + b.d;   // data koncowa

  for(i=k; i<n; i++)
  {
    w=t[i].rok*10000.0 + t[i].mies*100 + t[i].dzien;   // data
    if (x<=w && w<=y) { p=i; break; }
  }
  return p;
}

int por(const void *a, const void *b)
{
    tosoba *x = (tosoba *) a;
    tosoba *y = (tosoba *) b;

    return strcmp(x->naz, y->naz);
}

void sortuj_naz(tosoba t[], int n)   // sortuje n elementow
{
     qsort(t, n, sizeof(t[0]), por);
}
```

```
void main(void)
{
  int i,j;

  clrscr();    randomize();

 // wydruk zawartości tablicy struktur tab
  printf("Tablica: \n");   for (i=0; i<N; i++) pisz(tab,i);   getch();

// modyfikacja elementu tab[0]
  printf("Zmiana tab[0]: \n");    ustaw(tab,0);
  for (i=0; i<N; i++) pisz(tab,i);   printf("\n");   getch();

// poszukiwanie nazwiska JanL
  printf("Szukanie JanL i AAA: \n");

  i=szukaj_naz(tab,"JanL", 0, N);    // i=-1 jeśli nie znaleziono
  printf("\n");    if (i>-1) pisz(tab,i);     printf("\n");

// poszukiwanie nazwiska AAA
  i=szukaj_naz(tab,"AAA", 0, N);   // i=-1 jeśli nie znaleziono
  if (i>-1) pisz(tab,i);    printf("\n");     getch();

// poszukiwanie danych z zakresu dat
  printf("Szukanie wg. dat: od 1981-01-01 do 1981-09-27 \n");
  tdata a = {1981, 1, 1};
  tdata b = {1981, 9, 27};

  for (i=0; i<N; i++)
  {
     i=szukaj_data(tab, a, b, i, N);
     if (i==-1) break;     pisz(tab,i);
   }
  printf("\n");    getch();

// sortowanie elementów wg. nazwisk
  printf("Sortowanie \n");
  sortuj_naz(tab, N);
  for (i=0; i<N; i++) pisz(tab,i);

  getch();
}
```