

Tablice i struktury

czyli złożone typy danych.

TABLICA

przechowuje elementy tego samego typu

struktura jednorodna, homogeniczna

Elementy identyfikowane liczbami (indeksem).

8	1	-6	3	5	7	4	9	2	10	-4	88	6	3	1	3	332	2
---	---	----	---	---	---	---	---	---	----	----	----	---	---	---	---	-----	---

STRUKTURA

przechowuje elementy dowolnego typu

struktura niejednorodna, heterogeniczna

Elementy identyfikowane przez nazwy.

"Hans"		
"Kloss"		
4	11	2013
'M'		
181.5		

- wszystkie elementy są tego samego typu
- elementy identyfikowane przez liczbę całkowitą (indeks)
- tablice jednowymiarowe
- rozmiar musi być znany w momencie kompilacji
tablice statyczne
- swobodny dostęp (*random acces*) do elementów
- operator dostępu []
- w C tablice są indeksowane od 0

0	1	2	3	4	5	6	7	8	9
8	1	-6	3	5	7	4	9	2	10

DEKLARACJA TABLICY

`typ identyfikator[rozmiar];`

PRZYKŁAD

```
#define MAX 1000
const int n=200;

int a[10];
float tablica[MAX];
char napis[n];
```

- operator `[]` daje dostęp do *i*-tego elementu
- indeksowanie wartościami całkowitymi
- brak kontroli zakresu tablicy podczas kompilacji

PRZYKŁADY

```
tablica[0] = 1.3;  
napis[3] = 'x';  
i = a[i];  
a[i] = a[i] + 5;  
tablica[i-1] = tablica[i];  
tablica[maxind(x)] = tablica[0];
```

```
float t[10];
```

t[0]	t[1]	t[2]	t[2]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]
------	------	------	------	------	------	------	------	------	------

WCZYTYWANIE WARTOŚCI

```
float t[10];
int i=0;

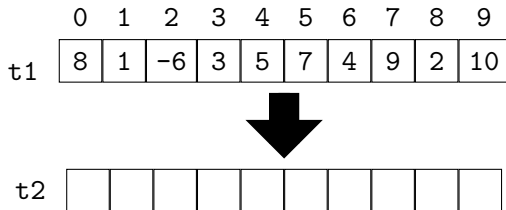
while(i<10)
{
    printf("t[%d]= ", i);
    scanf("%f", &t[i]);
    i = i + 1;
}
```

ZEROWANIE WARTOŚCI

```
float t[10];
int i;

for(i=0; i<10; i++)
{
    t[i] = 0;
}
```

KOPIOWANIE TABLIC

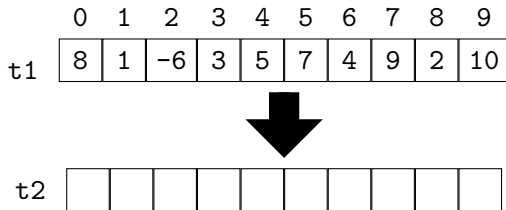


```
int t1[10];
int t2[10];
```

```
t1 = t2;
```

Źle !

KOPIOWANIE TABLIC



```
int i;
int t1[10];
int t2[10];

i = 0;
while( i < 10 )
{
    t2[i] = t1[i];
    i++;
}
```


TABLICE JAKO PARAMETRY FUNKCJI

Tablica jednowymiarowa w argumentach funkcji podawana bez rozmiaru (informacja o rozmiarze jest ignorowana i nie jest dostępna wewnątrz funkcji).

```
1 float max(float t[], int n)
2 {
3     float m = t[0];
4     while( n > 1 )
5     {
6         n = n - 1;
7         if( m < t[n] ) m = t[n];
8     }
9     return m;
10 }
```

 max.c

PRZYKŁADOWE DEKLARACJE FUNKCJI

```
void wczytaj(float tab[], int n);  
float max(float t[], int n);
```

Dobrze

```
float max(float t[10]);  
float max(float t, int n);  
float max(float t[]);
```

Źle

TABLICE JAKO PARAMETRY FUNKCJI

Zawartość tablicy przekazana do funkcji nie jest kopiowana, zaś funkcja może dowolnie modyfikować elementy przekazanej tablicy.

```
1  #include<stdio.h>
2
3  void funkcja(int tab[], int x)
4  {
5      tab[0]++;
6      x++;
7  }
8
9  int main()
10 {
11     int tab[10], x = 5;
12
13     tab[0] = x;
14     funkcja(tab, x);
15     printf("tab[0]=%d, x=%d\n", tab[0], x);
16 }
```

TABLICE JAKO PARAMETRY FUNKCJI

Modyfikator `const` pozwala zaznaczyć, że funkcja nie zmienia wartości elementów tablicy.

PRZYKŁAD:

```
float max(const float t[], int n);
```

```
void funkcja(const int tab[], int x)
{
    tab[0]++;
    x++;
}
```

Błąd! Nie można modyfikować.

TABLICE JAKO PARAMETRY FUNKCJI

Parametrem aktualnym funkcji (w momencie wywołania) jest nazwa tablicy

```
float max(const float t[], int n);  
void wczytaj(float t[], int n);
```

```
int main()  
{  
    float t[10], x;  
    int n=10;  
  
    wczytaj(t,n);  
    x = max(t,n);  
  
    x = max(t[10], 10);  
    x = max(t[], 10);  
    x = max(float t[], int n);  
}
```

Dobrze

Źle

Problem: w zbiorze zawierającym n elementów odnajdź element x .

Algorytm 1 Przeszukiwanie liniowe

Dane wejściowe: ciąg $\{t_0, t_1, \dots, t_{n-1}\}$ zawierający n elementów,
szukany element x , pozycja początku przeszukiwania i

Wynik: pozycja pierwszego znalezionej elementu x w ciągu lub
wartość -1 jeśli nie znaleziono

- 1: **dopóki** $i < n$ **wykonuj**
 - 2: **jeżeli** $t_i = x$ **wykonaj**
 - 3: **zwróć** i
 - 4: $i \leftarrow i + 1$
 - 5: **zwróć** -1
-

PRZYKŁAD W C: PRZESZUKIWANIE LINIOWE

```
1 int szukaj(const int t[], int n, int x, int i)
2 {
3     while( i < n )
4     {
5         if( t[i]== x ) return i;
6         i = i + 1;
7     }
8     return -1;
9 }
```

 przeszukiwanie.c

- Ile porównań należy wykonać w najgorszym przypadku?
- Czy istnieje szybszy sposób przeszukania ciągu elementów?

Algorytm 2 Przeszukiwanie liniowe z wartownikiem

Dane wejściowe: ciąg $\{t_0, t_1, \dots, t_{n-1}\}$ zawierający n elementów, szukany element x , pozycja początku przeszukiwania i

Wynik: pozycja pierwszego znalezionej elementu x w ciągu lub wartość -1 jeśli nie znaleziono

- 1: $t_n \leftarrow x$
 - 2: **dopóki** $t_i \neq x$ **wykonuj**
 - 3: $i \leftarrow i + 1$
 - 4: **jeżeli** $i = n$ **wykonaj**
 - 5: **zwróć** -1
 - 6: **w przeciwnym wypadku**
 - 7: **zwróć** i
-

PRZYKŁAD W C: PRZESZUKIWANIE LINIOWE Z WARTOWNIKIEM

```
1 int szukaj2(int t[], int n, int x, int i)
2 {
3     t[n] = x;                /* ustawienie wartownika */
4     while( t[i] != x ) i = i + 1;
5     if( i != n ) return i;
6     return -1;
7 }
```

 przeszukiwanie2.c

Typ złożony struct

- przechowuje zmienne dowolnego typu
- **pole** struktury to pojedynczy element składowy
- pola są identyfikowane nazwami
- operator dostępu bezpośredniego .
- struktury ułatwiają organizację danych → załączek obiektowości
- struktura może być argumentem funkcji oraz wartością zwracaną z funkcji

"Bond"
"James"
007
3.2

DEKLARACJA STRUKTURY

```
struct nazwa  
{  
    typ pole1;  
    typ pole2;  
    ...  
};
```

```
struct student  
{  
    char nazwisko[30];  
    char imie[30];  
    int indeks;  
    float srednia;  
};
```

UTWORZENIE ZMIENNEJ (DEFINICJA)

```
struct nazwa identyfikator;
```

```
struct student s;
```

DOSTĘP DO PÓL STRUKTURY

```
identyfikator.pole
```

```
s.srednia = 5.0;
```

```
#include <stdio.h>

struct zespolona
{
    float re;
    float im;
};

int main()
{
    struct zespolona z1 ,z2;
    z1.re = 2.5;
    z1.im = -2.2;

    z2 = z1;
}
```

Poprawne kopiowanie

DEKLARACJE FUNKCJI

```
struct zespolona iloczyn(struct zespolona z1, struct
    zespolona z2);
void wyswietl(struct student s);

int main()
{
    struct zespolona z1, z2, z3;
    struct student s;

    z3 = iloczyn(z1, z2);
    wyswietl(s);

    return 0;
}
```

Inicjalizacja elementów tablicy

```
int tab[10] = { 5, 3, 7};
```

tab	5	3	7	?	?	?	?	?	?
-----	---	---	---	---	---	---	---	---	---

Gdy pominiemy rozmiar tablicy to jest on wyznaczany automatycznie

```
int tab[] = { 5, 3, 7};
```

tab	5	3	7
-----	---	---	---

Tablica znaków

```
int tab[] = { 'A', 'B', 'C' };
```

tab

A	B	C
---	---	---

Napis (łańcuch znakowy)

```
int tab[] = "ABC";
```

tab

A	B	C	\0
---	---	---	----

Inicjalizacja wartości struktur

```
struct student
{
    int numer;
    char nazwisko[5];
};
```

```
struct student janek = { 13, "ABC"};
struct student franek = { 5, { 'A', 'B', 'C' } };
```

janek

13				
A	B	C	\0	?

franek

5				
A	B	C	?	?

- tablice wielowymiarowe, macierze, tablice tablic
`t[10][2][3]`
- struktury zawierające struktury
`s.data.dzien = 1`
- tablice struktur
`s[1].wiek = 31`
- struktury zawierające tablice
`punkt.wsp[1] = 1.2`

Problem: wyznaczyć położenie środka masy dla n punktów materialnych.

PUNKT MATERIALNY

$$p_i = \{m_i, \vec{r}_i\}, \quad \vec{r}_i = [x_i, y_i, z_i]$$

ŚRODEK MASY DWÓCH PUNKTÓW

$$\vec{r}_{12} = \frac{m_1 \vec{r}_1 + m_2 \vec{r}_2}{m_1 + m_2}, \quad m_{12} = m_1 + m_2$$

ŚRODEK MASY n PUNKTÓW

$$\vec{r}_0 = \frac{\sum_{i=1}^n m_i \vec{r}_i}{\sum_{i=1}^n m_i}, \quad m_0 = \sum_{i=1}^n m_i$$

TABLICA 4 ELEMENTOWA

konwencja: 0,1,2 - współrzędne kartezjańskie, 3 - masa

```
float punkt[4];
```

STRUKTURA

```
struct punkt
{
    float m;
    float x;
    float y;
    float z;
};
```

```
struct punkt
{
    float m;
    float wsp[3];
};
```

```
float* srodek(const float p1[], const float p2[])
{
    float sm[4];
    int i=0;

    sm[3] = p1[3] + p2[3];

    for(i=0; i<3; i++)
        sm[i] = (p1[3] * p1[i] + p2[3] * p2[i])/sm[3];

    return sm;
}
```

```
float* srodek(const float p1[], const float p2[])
{
    float sm[4];
    int i=0;

    sm[3] = p1[3] + p2[3];

    for(i=0; i<3; i++)
        sm[i] = (p1[3] * p1[i] + p2[3] * p2[i])/sm[3];

    return sm;
}
```

zmienna lokalna

brak kopiowania tablic

Źle!

```
1 void srodek(const float p1[], const float p2[], float sm[])
2 {
3     int i=0;
4
5     sm[3] = p1[3] + p2[3];
6
7     for(i=0; i<3; i++)
8         sm[i] = (p1[3] * p1[i] + p2[3] * p2[i]) / sm[3];
9 }
```

 sm1.c

```
1 struct punkt
2 {
3     float x, y, z;
4     float m;
5 };
6
7 struct punkt srodek(struct punkt p1, struct punkt p2)
8 {
9     struct punkt sm;
10    sm.m = p1.m + p2.m;
11    sm.x = ( p1.m * p1.x + p2.m * p2.x ) / sm.m;
12    sm.y = ( p1.m * p1.y + p2.m * p2.y ) / sm.m;
13    sm.z = ( p1.m * p1.z + p2.m * p2.z ) / sm.m;
14    return sm;
15 }
```


ŚRODEK MASY 2 PUNKTÓW

STRUKTURY C.D.

```
1 struct punkt
2 {
3     float wsp[3];
4     float m;
5 };
6
7 struct punkt srodek(struct punkt p1, struct punkt p2)
8 {
9     struct punkt sm;
10    int i=0;
11    sm.m=p1.m+p2.m;
12    while(i<3)
13    {
14        sm.wsp[i]=(p1.m*p1.wsp[i]+p2.m*p2.wsp[i])/sm.m;
15        i = i + 1;
16    }
17    return sm;
18 }
```

Algorytm 3 Środek masy n punktów materialnych

Dane wejściowe: zestaw punktów $\{p_1, p_2, \dots, p_n\}$ określonych przez masę i współrzędne kartezjańskie $p_i = \{x_i, y_i, z_i, m_i\}$

Wynik: $p_0 = \{x_0, y_0, z_0, m_0\}$ położenie środka masy i masa całkowita układu

- 1: $p_0 \leftarrow \{0, 0, 0, 0\}$
 - 2: $i \leftarrow 0$
 - 3: **dopóki** $i < n$ **wykonuj**
 - 4: **wczytaj** p_i
 - 5: $p_0 \leftarrow \text{srodek}(p_i, p_0)$
 - 6: $i \leftarrow i + 1$
 - 7: **wypisz** p_0
-

ŚRODEK MASY n PUNKTÓW

```
1  int main()
2  {
3      struct punkt p, p1;
4      char dalej='t';
5
6      p.m=0.0;  p.x = 0.0;  p.y = 0.0;  p.z = 0.0;
7
8      do
9      {
10         p1 = wczytaj();
11         p = srodek(p1,p);
12
13         printf("Czy dodac kolejny punkt [t/n] ? ");
14         scanf(" %c",&dalej);
15     }while(dalej != 'n' );
16
17     printf("Srodek masy:\n");
18     wypisz(p);
19
20     return 0;
21 }
```

TABLICA TABLIC

```
float chmura[1000][4];
```

```
m42 = chmura[42][3]
```

TABLICA STRUKTUR

```
struct punkt chmura[1000];
```

```
m42 = chmura[42].m
```

STRUKTURA Z TABLICAMI

```
struct chmura {  
    int n;  
    float x[1000];  
    float y[1000];  
    float z[1000];  
    float m[1000];  
};
```

```
struct chmura c;  
m42 = c.m[42]
```

```
struct chmura {  
    int n;  
    struct punkt p[1000];  
};
```

```
struct chmura c;  
m42 = c.p[42].m
```

```
1  struct punkt srodek(struct punkt p[], int n)
2  {
3      struct punkt sm;
4      int i=0;
5
6      sm.m = 0.0; sm.x = 0.0; sm.y = 0.0; sm.z = 0.0;
7
8      while(i<n)
9      {
10         sm.m = sm.m + p[i].m;
11         sm.x = sm.x + p[i].x * p[i].m;
12         sm.y = sm.y + p[i].y * p[i].m;
13         sm.z = sm.z + p[i].z * p[i].m;
14         i = i + 1;
15     }
16     sm.x = sm.x/sm.m;
17     sm.y = sm.y/sm.m;
18     sm.z = sm.z/sm.m;
19     return sm;
20 }
```

```
1  int main()  
2  {  
3      struct punkt chmura[MAX];  
4      int i=0;  
5  
6      do  
7      {  
8          chmura[i] = wczytaj();  
9          i = i + 1;  
10     }while(czy_dalej() == 1 && i < MAX );  
11  
12     printf("Srodek masy:\n");  
13     wypisz(srodek(chmura,i));  
14  
15     return 0;  
16 }
```

ŚRODEK MASY n PUNKTÓW

STRUKTURA Z TABLICĄ PUNKTÓW

```
1  int main()
2  {
3      struct chmura c;
4      int i=0;
5
6      c.n = 0;
7
8      do
9      {
10         c = dodaj(c, wczytaj());
11         i = i + 1;
12     }while( czy_dalej() && i < MAX );
13
14     printf("Aktualny zbior punktow:\n");
15     wypisz_chmure(c);
16     printf("Srodek masy:\n");
17     wypisz_punkt(srodek(c));
18
19     return 0;
20 }
```

- **Tablica** - zbiór elementów tego samego typu indeksowanych od 0
- **Struktura** - zbiór elementów różnych typów o nazwanych polach
- Tablice trzeba kopiować „ręcznie” element po elemencie
- Właściwie dobrana reprezentacja danych może istotnie ułatwić realizację rozwiązania. Reprezentacja danych ma wpływ na czytelność kodu i efektywność programu
- Dobra zasada: twórz funkcje do manipulowania złożonymi typami danych
- Inne typy złożone: unie, pola bitowe, typ wyliczeniowy (zob. wykład 11)

-  Maciej M. Sysło, „*Algorytmy*”, WSiP, Warszawa, 2002.
-  David Griffiths, Dawn Griffiths „*Rusz głową! C.*”, Helion, Gliwice, 2013.
-  „Kurs programowania w C”, WikiBooks,
<http://pl.wikibooks.org/wiki/C>