Programowanie imperatywne

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 22.03.2020

5. Deklaracje i typy

Wprowadzenie

- W języku C przy deklaracji obiektów (zmiennych, funkcji) wymagane jest podanie ich typów:

```
type-specifier var;
type-specifier foo(/*lista parametrów */)
```

- Wyrażenie `type-specifier` może być:
  - Typem wbudowanym `char`, `short`, `int`, `long`, `double`, `signed`, `unsigned` (lub `void` dla funkcji)
  - Pełną deklaracją nowego typu lub
  - Identyfikatorem typu zadeklarowanego wcześniej

Rodzaje typów 1

- **Typy wbudowane**

  Zmienne mogą on przechowywać pojedyncze wartości całkowite lub zmiennoprzecinkowe.

  Reprezentacja danych (liczba bajtów, ich kolejność, kolejność bitów) jest uzależniona od architektury sprzętowej.

  Zazwyczaj typy całkowitoliczbowe ze znakiem (np.: `int`) reprezentowane w kodzie uzupełnień do 2.

- **Tablice**

  Są to ciągi elementów tego samego typu.

  Brak kontroli rozmiarów tablicy.

Rodzaje typów 2

- **Wskaźniki**

  Wartościami zmiennych wskaźnikowych są adresy innych zmiennych. Za ich pomocą można modyfikować wartości wskazywanych zmiennych.

  Wskaźniki są typem całkowitoliczbowym (zazwyczaj długość `long`, czyli 4b na platformie 32bitowej, 8b na platformie 64-bitowej).

- **Typy wyliczeniowe**

  Zmienne wyliczeniowe są zawsze typu całkowitego. Przechowują one wartość będącą elementem pewnego zbioru. Poszczególne elementy zbioru są identyfikowane przez nazwę

Rodzaje typów 3

- **Struktury**

  Typy złożone, będące zgrupowaniem zmiennych innych (prostszych) typów.

  Zmienne składowe (pola) mogą być typami wbudowanymi, tablicami lub typami zdefiniowanymi przez użytkownika.

- **Unie**

  Mogą przechowywać wartości różnych typów. Pamięć dla pól składowych pokrywa się.

  Unie oszczędnie korzystają z pamięci, ale są kłopotliwe w użyciu.

  W językach obiektowych unii praktycznie nie stosuje się.

Typy wyliczeniowe 1

- Typy wyliczeniowe definiują zbiory wartości. Każdy z elementów zbioru jest jednoznacznie identyfikowany przez nazwę.
- Użycie typów wyliczeniowych jest alternatywą do wykorzystania preprocesora.

```
#define BOOL      int

#define FALSE     0
#define TRUE      1

BOOL positive(int i)
{
      if(i<=0)return FALSE;
      return TRUE;
}
```

Typy wyliczeniowe 2

Specyfikacje typu wyliczeniowego mają postać:

```
enum [tag] {enumerator-list}
```

deklaruje nowy typ

```
[enum] tag
```

odwołanie do zadeklarowanego wcześniej typu o nazwie „enum tag”

```
enumerator-list
```

jest listą identyfikatorów oddzielonych przecinkami, każdemu elementowi listy można nadawać unikalne stałe wartości całkowitoliczbowe

Typy wyliczeniowe 3 - przykład

```
enum BOOL {FALSE=0, TRUE};  // deklaruje typ BOOL

BOOL positive(int i)
{
      if(i<=0)return FALSE;
      return TRUE;
}

enum color {red,green,blue} _blue=blue;
enum color _red=red;
```

Typy wyliczeniowe 4 - przykład

```
enum {false=0,bad=0,fail=0,true=1,good=1,ok=1};

int main() {

  printf("%d\n",false);
  printf("%d\n",bad);
  printf("%d\n",fail);
  printf("%d\n",true);
  printf("%d\n",good);
  printf("%d\n",ok);

  return 0;
}
```

Wartości przypisane stałym mogą powtarzać się.

Typy wyliczeniowe 5 - przykład

```
enum {ok=0,err1,err2=1,err3,err4};

#define str(A) #A

int main() {

  printf("%s = %d\n",str(ok),ok);
  printf("%s = %d\n",str(err1),err1);
  printf("%s = %d\n",str(err2),err2);
  printf("%s = %d\n",str(err3),err3);
  printf("%s = %d\n",str(err1),err4);

  return 0;
}
```

`#define str(A) #A`

Makro preprocesora zamieniające argument na tekst…

```
ok = 0
err1 = 1
err2 = 1
err3 = 2
err1 = 3
```

Kompilator przydziela kolejne wartości, nie dbając o unikalność

Typy wyliczeniowe 6 - przykład

```
enum Direction{north=0,south,west,east};

void printDir(enum Direction dir)
{
        switch(dir){
        case north: printf("north");break;
        case south: printf("south");break;
        case west:  printf("west");break;
        case east:  printf("east");break;
        }
}
int main() {
        for(int i=0;i<4;i++){
                printDir((Direction)i);
                // wymagane rzutowanie
                printf("\n");
        }
}
```

deklaruje typ 'enum Direction' = Direction

Tablice 1

- Tablice są to ciągi danych tego samego typu zajmujące ciągły obszar pamięci.
- Zazwyczaj tablice mają stały rozmiar określony w momencie ich deklaracji (wyjątek VLA C99).
- Dostęp do wybranych elementów tablic realizowany jest za pośrednictwem operatora **[ ]**. Wewnątrz nawiasów **[ ]** podawany jest indeks elementu.
- Elementy tablic są indeksowane od 0 do $size - 1$, gdzie $size$ jest rozmiarem tablicy

Tablice 2

Deklaracja tablicy ma postać:

```
type-specifier name[const-expression]
```

lub

```
type-specifier name[]
```

`type-specifier`

definiuje typ elementów

`name`

nazwa tablicy

`const-expression`

określa rozmiar tablicy, musi być dodatnią stałą całkowitą (wyjątek C99).

Tablice 3 - przykłady

```
int a[10];

#define SIZE 100
double tab[100];

#define X 100
#define Y 50
double tab[X*Y];
```

- Jeżeli definiuje się tablice o stałych rozmiarach dobrym zwyczajem jest użycie symbolu stałej preprocesora. Upraszcza to ewentualne zmiany.
- Rozmiary tablic mogą być zdefiniowane jako wyrażenia obliczane w czasie kompilacji.

Tablice 4

Deklaracja tablicy z pominięciem rozmiaru może się pojawić jako:

- formalny parametr funkcji,

```
void foo(int a[],int size)
{
}
```

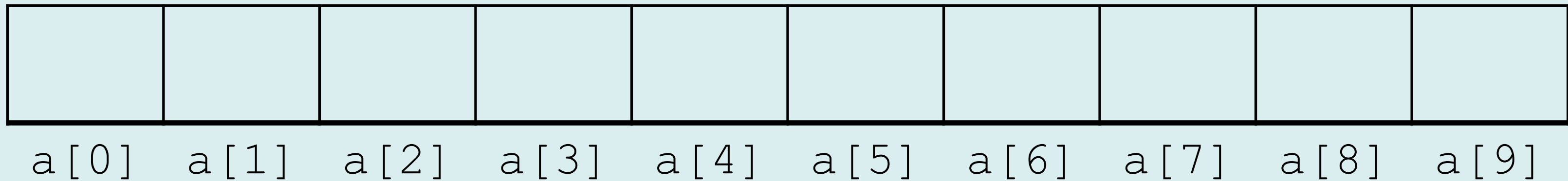
- odwołanie do tablicy zdefiniowanej w innym miejscu programu,

```
extern int b[];
```

- deklaracja połączona z inicjalizacją.

```
int a[]={0,1,2,3,4,5,6,7,8,9};
```

Tablice 5 - elementy



- Przykład: `int a[10];`
- Elementy tablic indeksowane są od 0
- Wyrażenie `name[integral-expression]`, np.: `a[7]` identyfikuje element tablicy
- Element tablicy jest traktowany jak zmienna typu użytego w deklaracji tablicy. Może być ona czytana, modyfikowana. Np.:
  - `printf("a[%d]=%d", i, a[i]);`
  - `a[i]++;`

Tablice 6

- Indeks tablicy `integral-expression` może być dowolnym wyrażeniem typu całkowitoliczbowego.
- Podczas kompilacji i **wykonania** nie jest sprawdzana poprawność zakresu indeksów tablicy. Odpowiedzialność spoczywa na programiście.
- Wskazane jest stosowanie typowych wzorców iteracji z pętlą for:

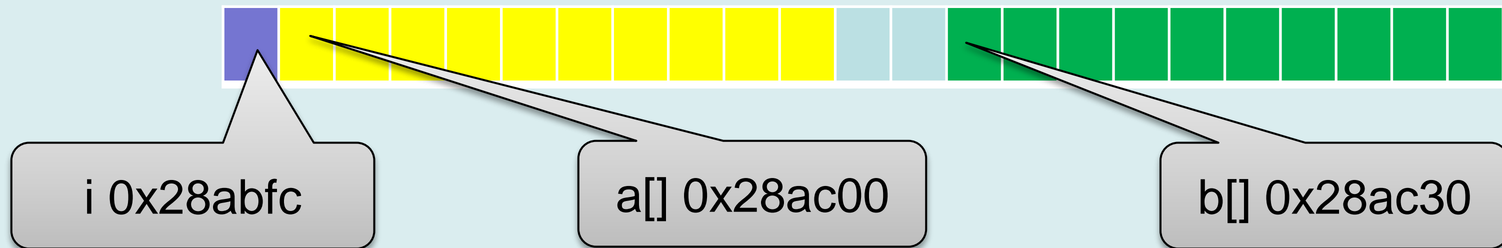
```
#define SIZE 10
int a[SIZE];
int i;

for(i=0;i<SIZE;i++)
   a[i]=2*i+1;
for(i=0;i<SIZE;i++)
   printf("%d ",a[i]);
```

Tablice 7

- W przypadku przekroczenia zakresu, program może zachowywać się w sposób losowy, np. nadpisywać wartości innych zmiennych…

```
int main() {
  int b[10],a[10],i;

  for(i=0;i<10;i++)b[i]=0;

  for(i=0;i<20;i++)a[i]=2*i+1; // błędny zakres

  for(i=0;i<10;i++)printf("b[%d]=%d ",i,b[i]);

  return 0;
}
```

Tablice 8 - analiza




```
int main() {
  int b[10],a[10],i;

  printf("b=%p a=%p &i=%p\n",&b[0],&a[0],&i);
  for(i=0;i<10;i++)b[i]=0;

  for(i=0;i<20;i++)a[i]=2*i+1;

  for(i=0;i<10;i++)printf("b[%d]=%d ",i,b[i]);
  return 0;
}
```

Drukuje adresy

Nadpisuje część tablicy b[]

```
b=0x28ac30 a=0x28ac00 &i=0x28abfc
b[0]=25 b[1]=27 b[2]=29 b[3]=31 b[4]=33 b[5]=35 b[6]=37
b[7]=39 b[8]=0 b[9]=0
```

Tablice - wyrażenia indeks. 1

- Indeks tablicy może być dowolnym wyrażeniem

```
#define ROWS 4
#define COLS 5

int main() {

  int tab[ROWS*COLS];
  int i,j;

  srand(time(NULL)); // stdlib.h i time.h
  for(i=0;i<ROWS;i++)
       for(j=0;j<COLS;j++)
              tab[i*COLS+j]=rand()%100;

  for(i=0;i<ROWS;i++){
       for(j=0;j<COLS;j++)
              printf("%d\t",tab[i*COLS+j]);
       printf("\n");
  }
return 0;
}
```

```
29      30      85      20      72
24      61      42      62      8
24      82      74      54      24
27      67      37      11      2
```

Tablice - wyrażenia indeks. 2

Raczej należy unikać efektów ubocznych…

```
#define SIZE 10
int main() {

  int i=0, a[SIZE];
  while(i<SIZE)a[i++]=2*i+1;
  while(i<SIZE)a[i]=2*++i+1;
  while(i<SIZE)a[i]=2*i+++1;
  return 0;
}
```

Lepiej…

```
while(i<SIZE){
    a[i]=2*i+1;
    i++;
}
```

Tablice – inicjalizacja 1

- Podczas deklaracji zmiennej typu tablicowego możliwe jest nadanie wartości początkowych.
- Wyrażenie inicjujące tablicy ma postać listy stałych wyrażeń oddzielonych przecinkami

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

int b[10]={1,2,3,4};

int c[]={1,2,2*2,2*2*2,2*2*2*2,};
```

Można umieścić nadmiarowy przecinek na końcu

Reszta tablicy zostanie uzupełniona zerami

Kompilator obliczy rozmiary tablicy

Tablice - inicjalizacja 2

- W C99 możliwe jest wskazanie wybranych elementów i nadanie im wartości (ang. *designated initializer*).
- Pozostałe elementy będą miały wartość 0

```
#define SIZE 10

int main() {

  int i=0, tab[SIZE]={[0] = 1,[5]=21,[9] = 10};
  for(i=0;i<SIZE;i++)printf("%d ",tab[i]);
  return 0;
}
```

```
1 0 0 0 0 21 0 0 0 10
```

Tablice - sizeof

- Operator **`sizeof`** zwraca rozmiar typu danych lub zmiennej określonego typu.
- W przypadku zadeklarowanej tablicy, dla której przydzielona została pamięć, za pomocą operatora **`sizeof`** można określić ile bajtów zajmuje tablica. **Deklaracja tablicy musi być widoczna!!!.**
- Dla deklaracji: `type tab[SIZE];`

  operator zwróci `sizeof(type)*SIZE`
- Jeżeli rozmiar tablicy nie jest jawnie podany, wyrażenie `sizeof tab /sizeof tab[0]` pozwala na obliczenie liczby jej elementów.

```
int main() {

  int i=0, tab[]={[0] = 1,[5]=21,[6] = 10};
  printf("Liczba elementow tablicy %d ",
              sizeof tab/sizeof tab[0]);
  return 0;
}
```

```
Liczba elementow tablicy 7
```

Tablice i funkcje 1

- Deklarowanie tablic, jako parametrów funkcji i przekazywanie ich, jako argumentów jest kłopotliwe. Kompilator przekazuje do funkcji jedynie adres początku tablicy.
- Brak wbudowanego mechanizmu, który pozwoliłby przekazać informacje, gdzie tablica kończy się.

```
#define SIZE 10

void printT(int tab[SIZE]) {
    int i;
    for (i=0;i<SIZE;i++)printf("%d ",tab[i]);
}

int main() {

  int tab[SIZE]={1,2,3,4};
  printT(tab);
  return 0;
}
```

Umawiamy się: tablice mają 10 elementów

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

Tablice i funkcje 2

- Wewnątrz funkcji **nie można obliczyć rozmiaru tablicy** za pomocą operatora `sizeof`.

```
#define SIZE 10

void printT(int tab[]) {
    int i;
    for (i=0;i<sizeof tab/sizeof tab[0];i++)
    printf("%d ",tab[i]);
}

int main() {

  int tab[]={1,2,3,4,[10]=-1};
  printT(tab);
  return 0;
}
```

Do funkcji przekazywany jest adres pierwszego elementu tablicy. Rozmiar wskaźnika (adresu) to 4 lub 8

```
1 (32bit)
```

```
1 2 (64bit)
```

Tablice i funkcje 3

- Wybieramy specjalną wartość (ang. *sentinel*) oznaczającą koniec tablicy (nie jest ona znaczącym elementem tablicy).
- Kłopotliwe, trzeba zawsze pamiętać o zwiększeniu rozmiaru…

```
void printToSentinel(int tab[])
{
    int i;
    for (i=0;tab[i]>=0;i++)
        printf("%d ",tab[i]);
}

int main() {

  int tab[]={1,2,3,4,[10]=-1};
  printToSentinel(tab);
  return 0;
}
```

sentinel = strażnik, wartownik. Poza specjalnymi przypadkami, jak 0 dla tablic znaków lub 0/NULL dla wskaźników – nie polecam.

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

Tablice i funkcje 4

- Najczęściej do funkcji przekazywany jest rozmiar tablicy.
- Przykład

```
int maxElement(int t[],int size)
{
      int max;
      int i;
      max = t[0];
      for(i=1;i<size;i++){
            if(max<t[i])max = t[i];
      }
      return max;
}
```

Tablice i funkcje 5

- Inna wersja

```
#include <limits.h>

int maxElement2(int t[],int size)
{
	int max=INT_MIN; //=(-2147483647 - 1)
	int i;
	for(i=0;i<size;i++){
		if(max<t[i])max = t[i];
	}
	return max;
}
```

Tablice i funkcje 6

- Funkcja nie może zwrócić tablicy (ciągłego obszaru pamięci zawierającego elementy).
- Funkcja może zwrócić adres tablicy, ale pojawia się problem przydziału pamięci.

```
int *returnTable(void){
    int tab[]={1,2,3,4};
    return tab;
}
```

```
main.c: In function `returnTable':
main.c:20: warning: function returns address of local
variable
```

Tablice i funkcje 7

W typowych implementacjach, funkcja rzadko zwraca tablicę. Najczęściej przekazuje się ją z zewnątrz, a funkcja wypełnia ją wartościami.

```
void sum(double a[], double b[], double r[], int size){
    int i;
    for(i=0;i<size;i++)r[i]=a[i]+b[i];
}
```

Jak odróżnić parametr obliczany od wejściowego? Dodając **`const`**.

```
void sum(   const double a[],  // tylko do odczytu
            const double b[],  // tylko do odczytu
            double r[],        // obliczane
            int size){
    int i;
    for(i=0;i<size;i++)r[i]=a[i]+b[i];
}
```

Tablice wielowymiarowe 1

Tablice wielowymiarowe deklarowane są jako

```
type-specifier name[const-expr][const-expr]...
```

W przypadku dwuwymiarowym jest to deklaracja postaci

```
TYPE name[row-count][col-count]
```

Elementy tablic dwuwymiarowych są rozmieszczane kolejno wierszami.

Tablice wielowymiarowe 2 - przykład

Deklaracja:

```
int a[2][3]
```

- Logiczne rozmieszczenie elementów tablicy:

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

- Fizyczne rozmieszczenie elementów tablicy:

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

Tablice wielowymiarowe 3 - przykład

```
#define ROWS 2
#define COLS 3

int main()
{
 int a[ROWS][COLS];
 int i,j;
 for(i=0;i< ROWS;i++)
   for(j=0;j<COLS;j++)
     a[i][j]=COLS*i+j;
 for(i=0;i<ROWS;i++)
   for(j=0;j<COLS;j++)
     printf("a[%d][%d]=%d address = %d\n",
             i,j,  a[i][j], &a[i][j]);
      return 0;
}
```

```
a[0][0]=0 address = 2686752
a[0][1]=1 address = 2686756
a[0][2]=2 address = 2686760
a[1][0]=3 address = 2686764
a[1][1]=4 address = 2686768
a[1][2]=5 address = 2686772
```

Tablice wielowymiarowe 4

- Jeżeli parametrem funkcji jest tablica, przekazywany jest wyłącznie adres początku tablicy.
- W przypadku tablic dwuwymiarowych, kompilator nie wie, gdzie należy podzielić ciąg danych na wiersze.

```
void zero(int tab[][], int rows, int cols){
    int i,j;
    for(i=0;i<rows;i++)
        for(j=0;j<cols;j++)
            tab[i][j]=0;
}
```

```
main.c: In function `zero':
main.c:22: error: invalid use of array with unspecified
bounds
```

Tablice wielowymiarowe 5

Dwa możliwe rozwiązania (trzecie wykorzystujące VLA: dalej)

```
void zero_3(int tab[][3], int rows, int cols){
    int i,j;
    for(i=0;i<rows;i++)
        for(j=0;j<cols;j++)
            tab[i][j]=0;
}


void zero_1D(int tab[], int rows, int cols){
    int i,j;
    for(i=0;i<rows;i++)
        for(j=0;j<cols;j++)
            tab[i*cols+j]=0;
}
```

Deklarujemy jawnie: tablica ma 3 kolumny

Stosujemy tablicę jednowymiarową i przeliczamy indeksy,

Tablice wielowymiarowe 6 - inicjalizacja

- Inicjalizując tablice wielowymiarowe podaje się kolejne wiersze.
- Wewnętrzne nawiasy separujące wiersze można pominąć.
- Można również stosować desygnatory

```
int days_in_month[2][12] = {
      {31,28,31,30,31,30,31,31,30,31,30,31},
      {31,29,31,30,31,30,31,31,30,31,30,31}
};

char* text_matrix[2][2]
={[0][0]="ala",[0][1]="ma",[1][0]="kota"};

void p(){
    int i,j;
    for(i=0;i<2;i++){
        for(j=0;j<2;j++)
            if(text_matrix[i][j])
               printf("%s ",text_matrix[i][j]);
}
```

Tablice o zmiennych rozmiarach 1

- Standard C99 wprowadził nową konstrukcję – tablice o zmiennych rozmiarach (VLA – *Variable Length Arrays*).
- Rozmiary tablic są określane w trakcie wykonania programu (są one zmiennymi lub wyrażeniami, których wartości nie da się obliczyć w trakcie kompilacji).
- Pamięć dla tablic o zmiennych rozmiarach przydzielana jest na stosie. Nie da się zdeklarować jako VLA tablic statycznych (globalnych i z przypisanym modyfikatorem `static`)

Tablice o zmiennych rozmiarach 2 – przykład 1

```
void printFibo(int n){
    int i;

    int tab[n];

    tab[0]=1;
    tab[1]=1;
    for(i=2;i<n;i++)tab[i]=tab[i-1]+tab[i-2];
    for(i=0;i<n;i++)printf("%d ",tab[i]);
}

int main()
{
    printFibo(10);
    return 0;
}
```

`int tab[n];`
Rozmiar tablicy jest określony przez formalny parametr funkcji.

```
1 1 2 3 5 8 13 21 34 55
```

Tablice o zmiennych rozmiarach 3 – przykład 2

```
int main()
{
    int n,i,sum=0;
    printf("\tsp:%u\n",get_sp());

    printf("Podaj liczbe elementow:");
    scanf("%d",&n);

    int tab[n];

    printf("\tsp:%u\n",get_sp());
    printf("\nPodaj kolejne elementy:");
    for(i=0;i<n;i++)scanf("%d",&tab[i]);

    for(i=0;i<n;i++)sum=sum+tab[i];

    printf("Suma wynosi %d\n",sum);
    return 0;
}
```

Przed i po zadeklarowaniu tablicy drukowany jest wskaźnik stosu.

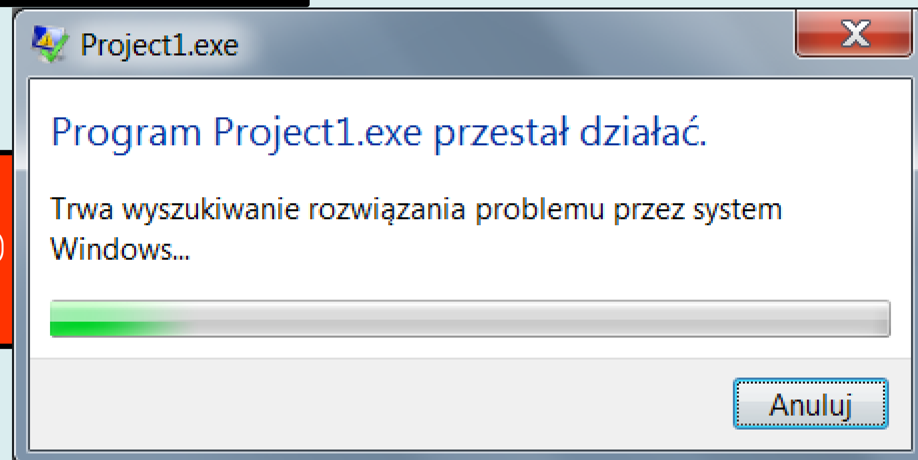
Rozmiar tablicy jest odczytywany ze standardowego wejścia.

Tablice o zmiennych rozmiarach 4 – przykład 2

```
// odczyt wkaźnika stosu
unsigned get_sp() {
  register unsigned esp __asm__ ("esp");
  return esp+8;               // korekta
}
```

```
    sp:2686736
Podaj liczbe elementow:4
    sp:2686704

Podaj kolejne elementy: 1 2 3 4
Suma wynosi 10
```

```
    sp:2686736
Podaj liczbe elementow:10000000
```



Tablice o zmiennych rozmiarach 5

Wątpliwości

- Co stanie się, jeżeli wywołamy funkcję `printFibo()` z niewłaściwymi argumentami : -5, 0, 1, 1000000
- Zmienne rozmiary tablic mogą pochodzić z zewnątrz. Konieczna dodatkowa kontrola.
- Znacznie bezpieczniejszym rozwiązaniem jest dynamiczna alokacja pamięci – dostęp do GB, a nie MB
- Wiele kompilatorów (w tym Visual Studio) nie implementuje tego mechanizmu. Kompilacja przykładu z funkcją `printFibo()`.

```
1>main.c (7): error C2057: expected constant expression
1>main.c (7): error C2466: cannot allocate an array of constant size 0
1>main.c (7): error C2133: 'tab' : unknown size
1>
1>Build FAILED.
```

Tablice o zmiennych rozmiarach 6 ☺

```
void mul(int rows,int cols, double r[rows],
const double a[rows][cols], const double v[cols]){
    int i,j;
    for(i=0;i<rows;i++){
        r[i]=0;
        for(j=0;j<cols;j++)
            r[i]=r[i]+a[i][j]*v[j];
    }
}

int main() {
  double a[4][4]=
  { [0][0]=1, [1][1]=1, [2][2]=1, [3][3]=1,};
  double r[4],   v[4]={1,2,3,4};
  int i;
  mul(4,4,r,a,v);
  for(i=0;i<4;i++)printf("%.1f ",r[i]);
  return 0;
}
```

Rozmiary tablic `r`, `a` i `v` są określone przez formalne parametry `rows` i `cols`

```
1.0 2.0 3.0 4.0
```

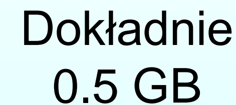
Czym zastąpić VLA?

Bezpieczniejszym rozwiązaniem jest **dynamiczna alokacja pamięci**.

Kod	Opis
`int *tab;`	Deklaracja zmiennej
`tab=malloc(sizeof(int)*n);`	Przydział pamięci
`if(tab==0){ /* obsługa błędów */ }`	Sprawdzenie, czy udało się przydzielić pamięć (coraz rzadziej stosowane…)
`tab[n-1]=0;`	Przykładowe użycie
`free(tab);`	Zwolnienie pamięci

Czym zastąpić VLA? Przykład

```
int main(){
    int *tab;
    int i;
    int size;
    printf("Podaj rozmiar tablicy:");
    scanf("%d",&size);
    if(size<=0)return -1;
    tab=malloc(sizeof(int)*size);
    if(tab==0)return -1;
    for(i=0;i<size;i++){
        tab[i]=-i*i+7*i*456;
    }
    printf("max element %d",maxElement(tab,size));
    free(tab);
    return 0;
}
```




```
Podaj rozmiar tablicy:134217728
max element 2147483628
Process returned 0 (0x0)   execution time : 6.066 s
Press any key to continue.
```

Co należy zapamiętać

- Deklaracja tablic
- Indeksowanie od 0 do *rozmiar-1*
- Pętle `for(i=0;i<size;i++)tab[i]`…
- Inicjalizacja tablic
- Wyrażenie `sizeof tab/sizeof tab[0]`
  Kiedy można je stosować!
- Organizacja pamięci dla tablic dwuwymiarowych
- Blaski i cienie tablic o zmiennych rozmiarach

Struktury 1

Struktury są konstrukcją umożliwiającą grupowanie zmiennych, które razem opisują pewien obiekt modelowany w programie. Zmienne te nazywane są polami (ang. *field*, *member*).

```
struct osoba
{
	char imie[32];
	char nazwisko[32];
	int wiek;
};

struct complex
{
	double re,im;
} solution;
```

Struktury 2

Specyfikacje typu strukturalnego mają postać:

`struct [tag] { struct-declaration-list }`

Deklaracja nowego typu

`struct tag`

Odwołanie do zadeklarowanego wcześniej typu o nazwie „struct tag".

`struct-declaration-list`

Jest listą deklaracji pól struktury. Wewnątrz struktury można deklarować pola dowolnego typu poza typem `void`.

Można także zadeklarować zmienną typu strukturalnego (podstrukturę).

Struktury 3

Deklaracja zmiennych typu strukturalnego

Deklarując pola typu strukturalnego nie możemy ich inicjować. Poniższa deklaracja jest więc nieprawidłowa:

```
struct complex {
    double re=0;
    double im =0;
}
```

W języku C++ począwszy od standardu C++11 możemy. Deklaracja struktury zostanie prawidłowo skompilowana.

Struktury 4

- Deklarując zmienną typu strukturalnego możemy inicjować jej pola podając listę stałych odpowiedniego typu

```
struct complex
{
      double re;
      double im ;
} solution = {0.0,0.0} ;

struct complex vector = {1.0,0.0};
```

Struktury 5

```
enum Color {red,green,blue};

struct circle
{
        double xcenter, ycenter, radius;
        enum Color color;
} aCircle = {0.0, 0.0, 10.0, red};
```

- Uwaga: nazwą typu strukturalnego jest 'struct tag' a nie 'tag'.

```
struct circle _circle; // deklaracja poprawna
circle _circle; // deklaracja niepoprawna
```

Struktury 6

Dostęp do pól struktury

- Pola struktury są identyfikowane poprzez podanie ich nazwy (identyfikatora). Nazwy te muszą być unikalne w kontekście struktury, natomiast mogą być użyte w innych strukturach lub jako nazwy zmiennych lub funkcji.

- Dostęp do pól zmiennej typu strukturalnego realizowany jest za pomocą **operatora kropkowego** (ang. *dot operator*).

Struktury 7

```
void foo()
{
      struct circle aCircle;
      aCircle.xcenter = 0.0;
      aCircle.ycenter = 0.0;
      aCircle.radius = 10.0;
      aCircle.color = red;
      …
      printf("Circle (%f %f) %f %d",
            aCircle.xcenter,
            aCircle.ycenter,
            aCircle.radius,
            aCircle.color) ;
}
```

Wyrażenie `aCircle.xcenter` identyfikuje pole `xcenter` wewnątrz pamięci przydzielonej dla zmiennej `aCircle`.

Struktury 8

Przydział pamięci dla zmiennych strukturalnych

Jeżeli deklarujemy zmienną typu strukturalnego, wówczas

- możemy spodziewać się, że pamięć dla poszczególnych pól zostanie przydzielona w kolejności zgodnej z deklaracją typu
- nie możemy oczekiwać, że pola struktury zajmą ciągły obszar pamięci.

Sposób przydziału pamięci jest powiązany ściśle z architekturą sprzętową.

Standardowo, rozmiar pamięci komputera podawany jest w **bajtach**.

Pojęciem związanym z architekturą sprzętową jest długość **słowa maszynowego** odpowiadająca szerokości magistrali danych.

Struktury 9

Długość słowa to:

- 1 bajt dla procesorów 8-bitowych
- 2 bajty dla procesorów 16-bitowych
- 4 bajty dla procesorów 32-bitowych
- 8 bajtów dla procesorów 64-bitowych

Jeżeli zmienne (w tym pola struktur) są umieszczane w pamięci pod adresami wewnątrz słów maszynowych, wówczas dostęp do nich jest znacznie szybszy, ponieważ jest dokonywany w jednym cyklu pamięci. W typowym przypadku kompilator C/C++ będzie rozmieszczał pola struktur na granicy słów maszynowych Inny sposób rozmieszczenia pól struktur wymaga ustawienia specyficznych opcji kompilatora.

Struktury 10

- Czy suma rozmiarów pól jest równa rozmiarowi struktury? Z reguły nie…

```
int main()
{
        struct {
                char c;
                short i;
                double d;
        } v;
        printf("%lu %lu\n",
                sizeof(v),

        sizeof(char)+sizeof(short)+sizeof(double));
        return 0;
}
```

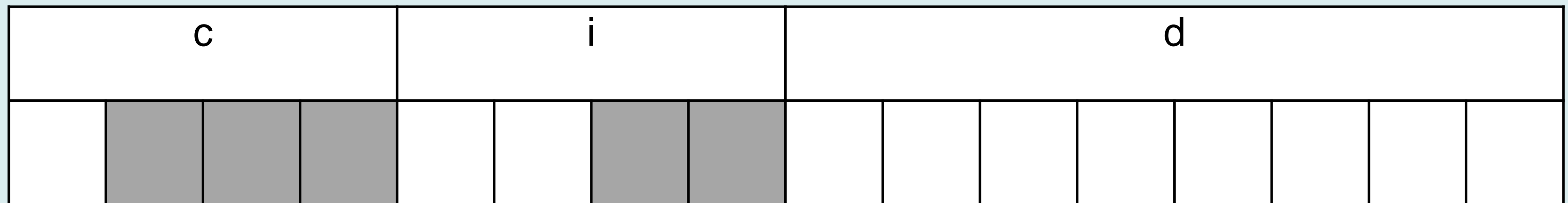
```
16 11
```

Struktury 11

Prawdopodobne rozmieszczenie pól struktury.

Szare prostokąty oznaczają nieużywane bajty.

```
struct {
        char c;
        short i;
        double d;
};
```




```
struct  __attribute__((__packed__))
{
    char c;
    short i;
    double d;
};
```

Deklaracja spakowanej struktury (kompilator GNU). Jej rozmiar to 11 bajtów.

Struktury 12

```
struct  test{
    char c;
    double d;
    short i;
};

struct  __attribute__((__packed__))
test_packed{
    char c;
    double d;
    short i;
};

#define NUM 1000000000
struct test s1;
struct test_packed s2;
```

```
int main() {
    printf("%d %d\n",sizeof(s1),sizeof(s2));
    clock_t t1 = clock();
    for(int i=0;i<NUM;i++){
        s1.c='a'; s1.i=i; s1.c++; s1.i++;
    }
    clock_t t2 = clock();
    for(int i=0;i<NUM;i++){
        s2.c='a'; s2.i=i; s2.c++; s2.i++;
    }
    clock_t t3 = clock();
    printf("not packed:%f packed: %f\n",
            (double)(t2-t1)/CLOCKS_PER_SEC,
           (double)(t3-t2)/CLOCKS_PER_SEC) ;
    return 0;
}
```

Wynik:
24 11
not packed:2.109000 packed: 2.797000

Po zmianie kolejności – znaczna różnica rozmiarów.
Wbrew pozorom zysk czasowy nie jest wielki: 0.7 sec dla 10^9 operacji

Pola bitowe 1

- Pola bitowe są to całkowitoliczbowe pola struktur (także unii), których wielkość jest ograniczona do zadanej liczby bitów.
- Użycie pól bitowych jest bardziej oszczędne niż stosowanie zmiennych całkowitych (`int`, `short`, `long`).
- Składnia:

```
type-specifier tag : constant-expression
```

`type-specifier:`

```
    [signed | unsigned] int | short | long
```

`tag:`

nazwa pola (opcjonalna)

`constant-expression:`

określa liczbę bitów

Pola bitowe 2

- Przykład

```
struct
{
    unsigned short icon : 8;
    unsigned short color : 4;
    unsigned short underline : 1;
    unsigned short blink : 1;
}screen[25][80]
```

- `icon` – znak do wyświetlenia
- `color` – 16 kolorów
- `underline, blink` – dodatkowe atrybuty znaku.

Wszystkie pola mieszczą się w 16 bitach.

Pola bitowe 3

- Wielkość pól bitowych nie może przekraczać wielkości typu podstawowego:
- Pole bitowe bez nazwy jest używane dla wyrównania bitów
- Pole bitowe zerowej wielkości bez nazwy zapewnia wyrównanie do granic typu całkowitego `int`.
- Dostęp do pol bitowych realizuje się za pomocą operatora kropkowego (.) .

Unie 1

Unie stanowią zgrupowanie zmiennych.
W odróżnieniu od struktur, pola unii zajmują ten sam obszar pamięci.

Zasady deklaracji są analogiczne, jak dla struktur.

Składnia

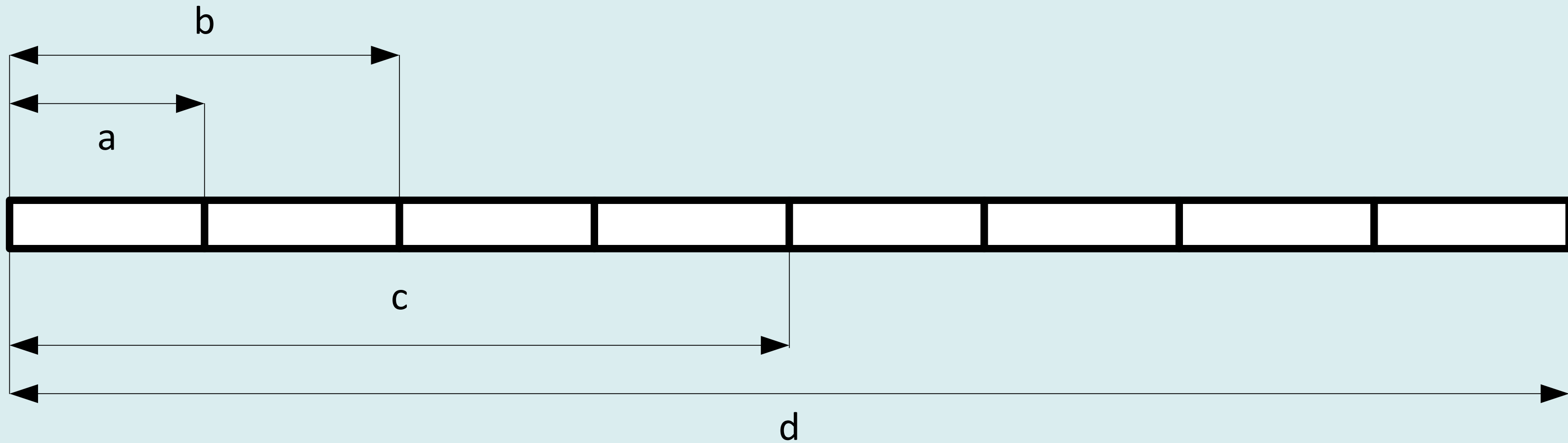
```
union [tag] { union-declaration-list }
```

```
union-declaration-list
```

jest listą pól unii

Unie 2 – organizacja pamięci

```
union
{
        char a;
        short b;
        int c;
        double d;
} x;
```



Unie 3

- Dostęp do pól unii realizowany jest za pomocą operatora kropkowego.
- Programista korzystający z unii jest odpowiedzialny za poprawną realizację dostępu do pól unii. Zazwyczaj wiąże się to z koniecznością zapisania dodatkowej informacji o typie elementów składowanych w unii.
- Poniższy kod jest raczej błędny:

```
x.a='a';
printf("%f", x.d);
```

```
union
{
        char a;
        short b;
        int c;
        double d;
} x;
```

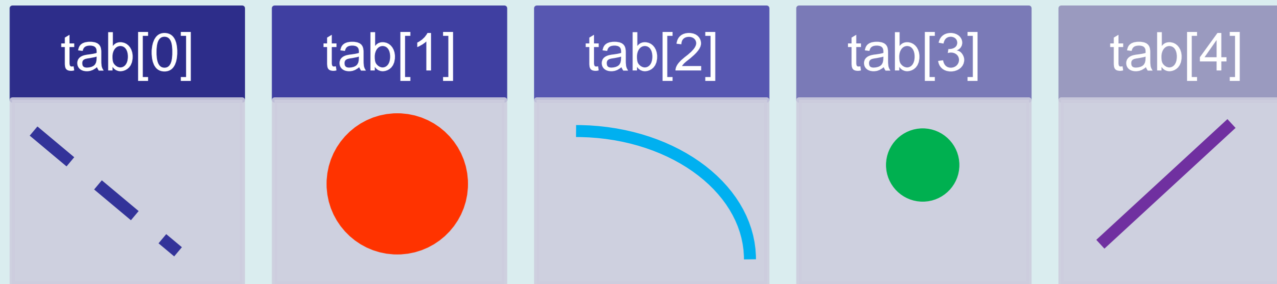
Unie 4 - przykład

Typowym zastosowaniem unii jest spłaszczenie hierarchii obiektów.

```
struct Line
{
        double x1,y1,x2,y2;
};

struct Circle
{
        double centerx,centery;
        double radius ;
};

struct Arc
{
        double centerx,centery;
        double radius ;
        double startAngle, endAngle ;
};
```

Unie 5 - przykład

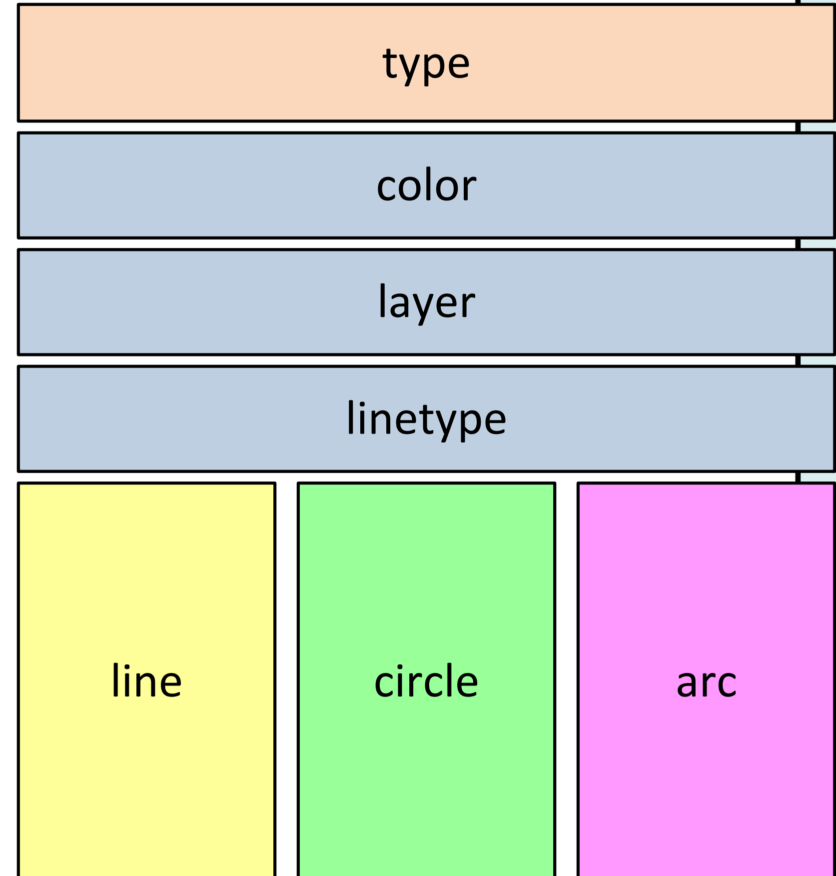
- Jak umieścić obiekty `Line`, `Circle` i `Arc` np.: w tablicy w dowolnej kolejności?



- Struktura `Entity`, która może być linią, kołem albo łukiem…
- `struc Entity tab[1000];` -- tablica pozwalająca na umieszczenie w niej 1000 wektorów…

Unie 5 - przykład

```
enum EntityType {eline,ecircle, earc};
struct Entity
{
        // selektor typu unii
        enum EntityType  type ;

        // wspólne pola
        int color;
        int layer;
        int linetype;

        // elementy składowe
        union
        {
        struct Line line;
        struct Circle circle;
        struct Arc arc;
        };
};
```



Unie 6 - przykład

Funkcje do „rysowania”

```
void drawLine( double x1,double y1, // pierwszy punkt
               double x2,double y2  // drugi punkt
){
    printf("line[%f,%f,%f,%f]\n",x1,y1,x2,y2);
}

void drawCircle(double xc,double yc,double radius)
{
    printf("circle[(%f,%f),%f]\n",xc,yc,radius);
}

void drawArc(  double xc,double yc,double radius,
               double sa,double ea)
{
    printf(„arc[(%f,%f),%f,%f>%f]\n",xc,yc,radius,sa,ea);
}
```

```
// w przyszłości lepiej...
void drawLine(      const struct Line*line){
    printf("line[%f,%f,%f,%f]\n",
          line->x1, line->y1,
          line->> x2, line-> y2);
}
```

Unie 7 - przykład

Funkcja do „rysowania” struct Entity

```
void draw(struct Entity entity)
{
// setColor(entity.color);
// setLayer(entity.layer);
// setLinetype(entity.linetype);
 switch(entity.type){
     case eline:
       drawLine(entity.line.x1, entity.line.y1,
                entity.line.x2, entity.line.y2);
       break;
     case ecircle:
       drawCircle(entity.circle.centerx,entity.circle.centery,
                  entity.circle.radius);
       break;
     case earc:
       drawArc(entity.arc.centerx,entity.arc.centery,
               entity.arc.radius,
               entity.arc.startAngle,entity.arc.endAngle);
       break;
  }
}
```

draw line accessing `entity.line`

draw cicrcle accessing `entity.circle`;

draw arc accessing `entity.arc`

Unie 8 - przykład

Wypełnianie tablicy wektorami

```
int main(){
    struct Entity tab[1000];
    int count=0;
    int i;
    // umieszczenie linii
    tab[0].type=eline;
    tab[0].line.x1=10;
    tab[0].line.y1=10;
    tab[0].line.x2=100;
    tab[0].line.y2=100;
    count++;
     // umieszczenie okręgu
    tab[1].type=ecircle;
    tab[1].circle.centerx=100;
    tab[1].circle.centery=100;
    tab[1].circle.radius=20;
    count++;
```

Unie 8 - przykład

Wypełnianie tablicy wektorami

```
    // ...

    // umieszczenie łuku
    tab[2].type=earc;
    tab[2].arc.centerx=200;
    tab[2].arc.centery=200;
    tab[2].arc.radius=40;
    tab[2].arc.startAngle=0;
    tab[2].arc.endAngle=90;
    count++;

    // rysowanie
    for(i=0;i<count;i++)draw(tab[i]);

    return 0;

}   // koniec funkcji main()
```

```
line[10.000000,10.000000,100.000000,100.000000]
circle[(100.000000,100.000000),20.000000]
arc[(200.000000,200.000000),40.000000,0.000000>90.000000]
```

Co należy zapamiętać

- Deklaracja typu wyliczeniowego enum

```
enum color {red,green,blue};
```

- Deklaracja struktur

```
struct complex
{
    double re;
    double im ;
}
```

- Deklaracja zmiennych wraz z nadaniem wartości początkowych.

```
struct complex x={1.0,1.0};
```

Co należy zapamiętać

- Pola bitowe

```
struct bitfields{
    int a:1;
    int b:1;
    int c:1;
};
```

- Deklaracja unii, praktyka umieszczania unii wewnątrz struktury razem z selektorem unii.
- Przydział pamięci dla struktur i unii
- Dostęp do pól struktur lub unii (operator kropkowy)

```
e.type=earc;
e.arc.centerx=200;
```