

# Informatyka 2

Politechnika Białostocka - Wydział Elektryczny  
Elektrotechnika, semestr III, studia stacjonarne I stopnia  
Rok akademicki 2018/2019

## Wykład nr 3 (16.10.2018)

dr inż. Jarosław Forenc

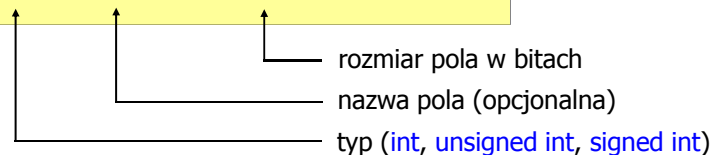
## Plan wykładu nr 3

- Pola bitowe, unie
- Wskaźniki
  - deklaracja
  - przypisanie wartości
  - związek z tablicami
  - operacje
- Dynamiczny przydział pamięci
  - funkcje calloc, malloc, free
  - przydział pamięci na wektor

## Pola bitowe

- Umożliwiają dostęp do pojedynczych bitów oraz przechowywanie małych wartości zajmujących pojedyncze bity
- Pola bitowe deklarowane są wewnątrz struktur

```
typ id_pola : wielkość_pola;
```



- Wartości zapisane w polach traktowane są jak liczby całkowite
- Zakres wartości pól wynika z **wielkości\_pola**

## Pola bitowe

```
struct Bits  
{  
    unsigned int a : 4;    /* zakres: 0...15 */  
    unsigned int b : 2;    /* zakres: 0...3 */  
    unsigned int  : 4;  
    unsigned int c : 6;    /* zakres: 0...63 */  
};
```

- Dostęp do pól bitowych odbywa się na takiej samej zasadzie jak do normalnych pól struktury

```
struct Bits dane;  
dane.a = 10;  
dane.b = 3;
```

## Pola bitowe

```
struct Bits
{
    unsigned int a : 4;    /* zakres: 0...15 */
    unsigned int b : 2;    /* zakres: 0...3 */
    unsigned int c : 4;
    unsigned int d : 6;    /* zakres: 0...63 */
};
```

- Jeśli pole nie ma nazwy, to nie można się do niego odwoływać
- Pola bitowe nie mają adresów
  - nie można wobec pola bitowego stosować operatora & (adres)
  - nie można polu bitowemu nadać wartości funkcją scanf()

## Pola bitowe - przykład

```
struct Flags_8086
{
    unsigned int CF : 1;    /* Carry Flag */
    unsigned int PF : 1;    /* Parity Flag */
    unsigned int AF : 1;    /* Auxiliary - Carry Flag */
    unsigned int ZF : 1;    /* Zero Flag */
    unsigned int SF : 1;    /* Signum Flag */
    unsigned int TF : 1;    /* Trap Flag */
    unsigned int IF : 1;    /* Interrupt Flag */
    unsigned int DF : 1;    /* Direction Flag */
    unsigned int OF : 1;    /* Overflow Flag */
};
```

## Unie

- Specjalny rodzaj struktury umożliwiający przechowywanie danych różnych typów w tym samym obszarze pamięci

```
union zbior
{
    char znak;
    int liczba1;
    double liczba2;
};
```

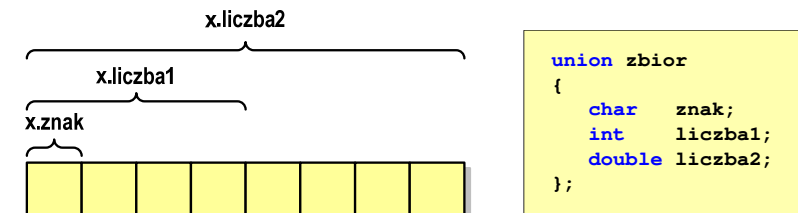
- Do przechowywania wartości w unii należy zadeklarować zmienną

```
union zbior x;
```

## Unie

```
union zbior x;
```

- Zmienna **x** może przechowywać wartość typu **char** lub typu **int** lub typu **double**, ale tylko jedną z nich w danym momencie



- Rozmiar unii wyznaczany jest przez rozmiar największego jej pola

## Unie

```
union zbior x;
```

- Dostęp do pól unii jest taki sam jak do pól struktury

```
x.znak = 'a';  
x.liczba2 = 12.15;
```

- Unię można zainicjować jedynie wartością o typie jej pierwszej składowej

```
union zbior x = {'a'};
```

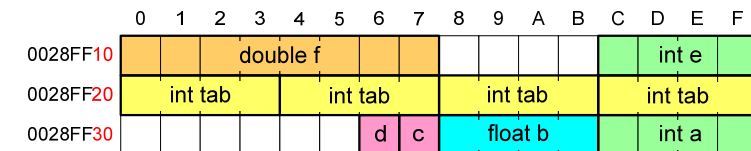
- Unie tego samego typu można sobie przypisywać

## Co to jest wskaźnik?

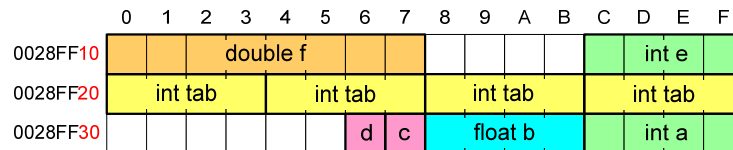
- **Wskaźnik** - zmienna mogąca zawierać adres obszaru pamięci - najczęściej adres innej zmiennej (obiektu)

```
int a;  
float b;  
char c, d;  
int tab[4], e;  
double f;
```

- Zmienne przechowywane są w pamięci komputera



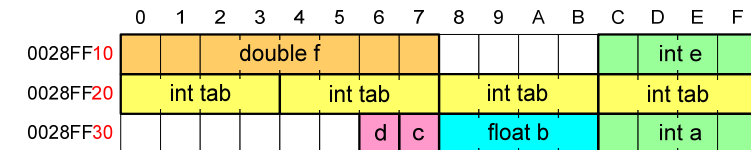
## Co to jest wskaźnik?



- Każda zmienna znajduje się pod konkretnym adresem i zależnie od typu zajmuje określoną liczbę bajtów
- Podczas kompilacji wszystkie nazwy zmiennych zastępowane są ich adresami
- Wyświetlenie adresu zmiennej:

```
printf("Adres zmiennej a: %p\n", &a);  
printf("Adres tablicy tab: %p\n", tab);
```

## Co to jest wskaźnik?



- Każda zmienna znajduje się pod konkretnym adresem i zależnie od typu zajmuje określoną liczbę bajtów
- Podczas kompilacji wszystkie nazwy zmiennych zastępowane są ich adresami
- Wyświetlenie adresu zmiennej:

```
printf("Adres zmiennej a: %p\n", &a);  
printf("Adres tablicy tab: %p\n", tab);
```

```
Adres zmiennej a: 0028FF3C  
Adres tablicy tab: 0028FF20
```

## Deklaracja wskaźnika

- Deklarując wskaźnik (zmienną wskazującą) należy podać **typ** obiektu na jaki on wskazuje
- Deklaracja wskaźnika wygląda tak samo jak każdej innej zmiennej, tylko że jego **nazwa** poprzedzona jest symbolem gwiazdki (\*)

```
typ *nazwa_zmiennej;
```

lub

```
typ* nazwa_zmiennej;
```

lub

```
typ * nazwa_zmiennej;
```

lub

```
typ*nazwa_zmiennej;
```

## Deklaracja wskaźnika

- Deklaracja zmiennej wskaźnikowej do typu **int**

```
int *ptr;
```

- Mówimy, że zmienna **ptr** jest typu: **wskaźnik do zmiennej typu int**

- Do przechowywania adresu zmiennej typu **double** trzeba zadeklarować zmienną typu: **wskaźnik do zmiennej typu double**

```
double *ptrd;
```

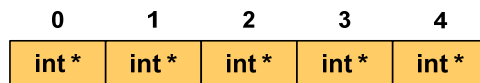
- Można konstruować wskaźniki do danych dowolnego typu łącznie z typami **wskaźnik do wskaźnika do...**

```
char **wsk;
```

## Deklaracja wskaźnika

- Można deklarować tablice wskaźników - zmienna **tab\_ptr** jest tablicą zawierającą **5 wskaźników do typu int**

```
int *tab_ptr[5];
```



- Natomiast zmienna **ptr\_tab** jest **wskaźnikiem do 5-elementowej tablicy liczb int**

```
int (*ptr_tab)[5];
```

## Deklaracja wskaźnika

- W deklaracji wskaźnika lepiej jest pisać **\*** przy zmiennej, a nie przy typie:

```
int *ptr1; /* lepiej */  
int* ptr2; /* gorzej */
```

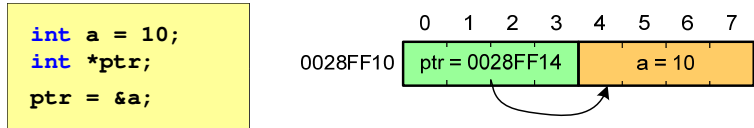
gdyż trudniej jest popełnić błąd przy deklaracji dwóch wskaźników:

```
int *p1, *p2;  
int* p3, p4;
```

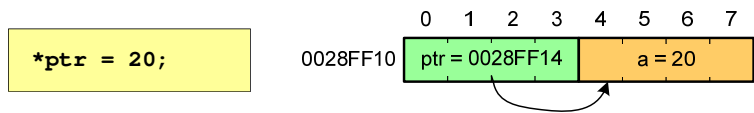
- W powyższym przykładzie zmienne **p1**, **p2** i **p3** są **wskaźnikami do typu int**, zaś zmienna **p4** jest „zwykłą” zmienną typu **int**

## Przypisywanie wartości wskaźnikom

- Wskaźnikom można przypisywać adresy zmiennych
- Adresy takie tworzy się za pomocą operatora pobierania adresu &



- Mając adres zmiennej można „dostać się” do jej wartości używając tzw. operatora wyłuskania (odwołania pośredniego) - gwiazdki (\*)



## Wskaźnik pusty

- **Wskaźnik pusty** to specjalna wartość, odróżnialna od wszystkich innych wartości wskaźnikowych, dla której gwarantuje się nierówność ze wskaźnikiem do dowolnego obiektu
- Do zapisu wskaźnika pustego stosuje się wyrażenie całkowite o wartości **zero (0)**

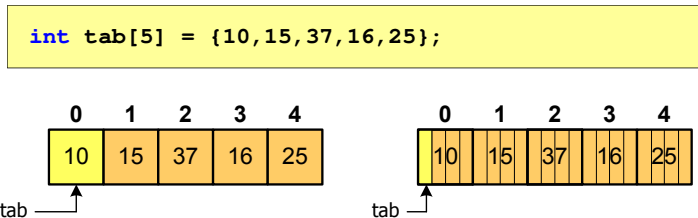
```
int *ptr = 0;
```

- Zamiast wartości **0** można stosować makrodefinicję preprocesora **NULL**, która podczas kompilacji programu zamieniana jest na **0**

```
int *ptr = NULL;
```

## Wskaźniki a tablice

- Nazwa tablicy jest jej adresem (dokładniej - adresem elementu o indeksie **0**)

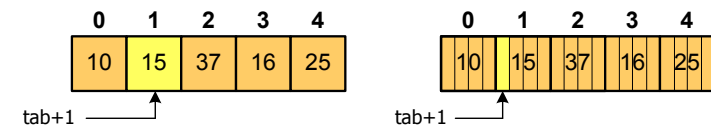


- Zastosowanie operatora **\*** przed nazwą tablicy pozwala „dostać się” do zawartości elementu o indeksie **0**

**\*tab** jest równoważne **tab[0]**

## Wskaźniki a tablice

- Dodanie **1** do adresu tablicy przenosi nas do elementu tablicy o indeksie **1**



zatem: **\*(tab+1)** jest równoważne **tab[1]**

ogólnie: **\*(tab+i)** jest równoważne **tab[i]**

- W zapisie **\*(tab+i)** nawiasy są konieczne, gdyż operator **\*** ma bardzo wysoki priorytet

## Wskaźniki a tablice

- Brak nawiasów powoduje błędne odwołania do elementów tablicy

```
int tab[5] = {10,15,37,16,25};
int x;

x = *(tab+2);
printf("x = %d",x);      /* x = 37 */

x = *tab+2;
printf("x = %d",x);      /* x = 12 */
```

$x = *(tab+2);$  jest równoważne  $x = tab[2];$

$x = *tab+2;$  jest równoważne  $x = tab[0]+2;$

## Operacje na wskaźnikach (1)

- **Przypisanie** - wskaźnikowi można przypisać:
  - adres zmiennej (nazwa zmiennej poprzedzona znakiem &)
  - inny wskaźnik
  - tablicę (nazwa to jej adres)

```
int tab[3] = {1, 2, 3};
int x = 10, *ptr1, *ptr2, *ptr3;

ptr1 = &x;
ptr2 = ptr1;
ptr3 = tab;
```

- Typ adresu i wskaźnika muszą być zgodne

## Operacje na wskaźnikach (2)

- **Pobranie wartości (dereferencja)**
  - otrzymanie wartości przechowywanej w pamięci, w miejscu wskazywanym przez wskaźnik
  - operator pobrania wartości (dereferencji, wyluskania): \*

```
int x = 10, *ptr, y;

ptr = &x;
y = *ptr;
printf("Wartosc x i y: %d\n",y);
```

Wartosc x i y: 10

## Operacje na wskaźnikach (3)

- **Pobranie adresu wskaźnika**
  - tak jak inne zmienne, także wskaźniki posiadają wartość i adres

```
int x = 10, *ptr;

ptr = &x;
printf("Adres zmiennej x: %p\n",ptr);
printf("Adres wskaźnika ptr: %p\n",&ptr);
```

Adres zmiennej x: 002CF920  
Adres wskaźnika ptr: 002CF914

## Operacje na wskaźnikach (4)

- Dodanie liczby całkowitej do wskaźnika
  - przed dodaniem liczby całkowitej jest ona mnożona przez liczbę bajtów zajmowanych przez wartość wskazywanego typu

```
int tab[5] = {0,1,2,3,4};  
  
printf("Adres tab: %p\n", tab);  
printf("Adres tab+2: %p\n", (tab+2));  
printf("tab[0]: %d\n", *tab);  
printf("tab[2]: %d\n", *(tab+2));
```

```
Adres tab: 002CFC60  
Adres tab+2: 002CFC68  
tab[0]: 0  
tab[2]: 2
```

## Operacje na wskaźnikach (5)

- Zwiększenie wskaźnika (inkrementacja)
  - do wskaźnika można dodać 1 lub zastosować operator ++
  - wskaźnik będzie pokazywał na kolejny element tablicy

```
int tab[5] = {0,1,2,3,4}, *ptr;  
  
ptr = tab;  
printf("tab[0]: %d\n", *ptr);  
ptr++;  
printf("tab[1]: %d\n", *ptr);  
ptr = ptr + 1;  
printf("tab[2]: %d\n", *ptr);
```

```
tab[0]: 0  
tab[1]: 1  
tab[2]: 2
```

## Operacje na wskaźnikach (5)

- Zwiększenie wskaźnika (inkrementacja)
  - do wskaźnika można dodać 1 lub zastosować operator ++
  - wskaźnik będzie pokazywał na kolejny element tablicy

```
int tab[5] = {0,1,2,3,4};  
  
printf("tab[0]: %d\n", *tab);  
tab++;  
printf("tab[1]: %d\n", *tab);
```

error C2105: '+' needs l-value

## Operacje na wskaźnikach (6/7)

- Odjęcie liczby całkowitej od wskaźnika
  - działa analogicznie jak dodanie liczby całkowitej do wskaźnika, ale wskaźnik musi być lewym operandem odejmowania
- Zmniejszenie wskaźnika (dekrementacja)
  - działa analogicznie jak inkrementacja

## Operacje na wskaźnikach (8)

- **Odejmowanie wskaźników**
  - różnicę między dwoma wskaźnikami oblicza się najczęściej dla wskaźników należących do tej samej tablicy
  - różnica ta określa jak daleko od siebie znajdują się elementy tablicy

```
int tab[5] = {0,1,2,3,4}, *ptr;  
ptr = tab + 3;  
printf("Roznica: %d\n", ptr-tab);
```

```
Roznica: 3
```

- różnica wskaźników należących do dwóch różnych tablic może spowodować błąd w programie

## Operacje na wskaźnikach (9)

- **Porównanie wskaźników**
  - porównanie może dotyczyć tylko wskaźników tego samego typu
  - w porównaniach stosowane są standardowe operatory:  
<, >, <=, >=, ==, !=

```
int tab[5] = {0,1,2,3,4}, *ptr;  
ptr = tab + 2;  
ptr--;  
--ptr;  
if (tab == ptr)  
    printf("Ten sam wskaznik\n");  
else  
    printf("Inny wskaznik\n");
```

```
Ten sam wskaznik
```

## Dynamiczny przydział pamięci w języku C

- Kiedy stosuje się dynamiczny przydział pamięci?
  - gdy rozmiar tablicy będzie znany dopiero podczas wykonania programu a nie podczas jego kompilacji
  - gdy rozmiar tablicy jest bardzo duży (np. największy rozmiar tablicy elementów typu `char` w języku C wynosi ok. 1 000 000)
- Do dynamicznego przydziału pamięci stosowane są funkcje:
  - `calloc()`
  - `malloc()`
- Przydział pamięci następuje w obszarze **sterty** (stosu zmiennych dynamicznych)
- Przydzieloną pamięć należy zwolnić wywołując funkcję:
  - `free()`

## Dynamiczny przydział pamięci w języku C

```
CALLOC stdlib.h  
void *calloc(size_t num, size_t size);
```

- Przydziela blok pamięci o rozmiarze `num*size` (mogący pomieścić tablicę `num`-elementów, każdy rozmiaru `size`)
- Zwraca wskaźnik do przydzielonego bloku pamięci
- Jeśli pamięci nie można przydzielić, to zwraca wartość `NULL`
- Przydzielona pamięć jest inicjowana zerami (bitowo)
- Zwracaną wartość wskaźnika należy rzutować na właściwy typ

```
int *tab;  
tab = (int *) calloc(10, sizeof(int));
```



## Dynamiczny przydział pamięci w języku C

**MALLOC** stdlib.h  
`void *malloc(size_t size);`

- Przydziela blok pamięci o rozmiarze określonym parametrem `size`
- Zwraca wskaźnik do przydzielonego bloku pamięci
- Jeśli pamięci nie można przydzielić, to zwraca wartość `NULL`
- Przydzielona pamięć nie jest inicjowana
- Zwracaną wartość wskaźnika należy rzutować na właściwy typ

```
int *tab;  
tab = (int *) malloc(10*sizeof(int));
```

## Dynamiczny przydział pamięci w języku C

**FREE** stdlib.h  
`void *free(void *ptr);`

- Zwalnia blok pamięci wskazywany parametrem `ptr`
- Wartość `ptr` musi być wynikiem wywołania funkcji `calloc()` lub `malloc()`

```
int *tab;  
tab = (int *) calloc(10, sizeof(int));  
/* ... */  
free(tab);
```

## Dynamiczny przydział pamięci na wektor

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    int *tab, i, n, x;  
    float suma = 0.0;  
  
    printf("Podaj ilosc liczb: ");  
    scanf("%d", &n);  
  
    tab = (int *) calloc(n, sizeof(int));  
    if (tab == NULL)  
    {  
        printf("Nie mozna przydzielic pamieci.\n");  
        exit(-1);  
    }  
}
```

## Dynamiczny przydział pamięci na wektor

```
for (i=0; i<n; i++) /* wczytanie liczb */  
{  
    printf("Podaj liczbe nr %d: ", i+1);  
    scanf("%d", &x);  
    tab[i] = x;  
}  
  
for (i=0; i<n; i++)  
    suma = suma + tab[i];  
  
printf("Srednia %d liczb wynosi %f\n", n, suma/n);  
  
free(tab);  
  
return 0;  
}
```

## Dynamiczny przydział pamięci na wektor

```
for (i=0; i<n; i++)  
{  
    printf("Podaj lic  
    scanf("%d",&x);  
    tab[i] = x;  
}
```

```
Podaj ilosc liczb: 5  
Podaj liczbe nr 1: 1  
Podaj liczbe nr 2: 2  
Podaj liczbe nr 3: 3  
Podaj liczbe nr 4: 4  
Podaj liczbe nr 5: 5  
Srednia 5 liczb wynosi 3.000000
```

```
for (i=0; i<n; i++)  
    suma = suma + tab[i];  
  
printf("Srednia %d liczb wynosi %f\n",n,suma/n);  
  
free(tab);  
  
return 0;  
}
```

## Dynamiczny przydział pamięci na wektor

- Wczytanie liczb bezpośrednio do wektora `tab`

```
for (i=0; i<n; i++) /* wczytanie liczb */  
{  
    printf("Podaj liczbe nr %d: ",i+1);  
    scanf("%d",&tab[i]);  
}
```

- Inny sposób odwołania do elementów wektora `tab`

```
for (i=0; i<n; i++) /* wczytanie liczb */  
{  
    printf("Podaj liczbe nr %d: ",i+1);  
    scanf("%d", (tab+i));  
}
```

## Koniec wykładu nr 3

Dziękuję za uwagę!