



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Wydział Elektryczny
Katedra Elektrotechniki, Energoelektroniki i Elektroenergetyki

Materiały do wykładu z przedmiotu:
Informatyka
Kod: EDS1B1007

WYKŁAD NR 4

Opracował: dr inż. Jarosław Forenc

Białystok 2021

Materiały zostały opracowane w ramach projektu „PB2020 - Zintegrowany Program Rozwoju Politechniki Białostockiej” realizowanego w ramach Działania 3.5 Programu Operacyjnego Wiedza, Edukacja, Rozwój 2014-2020 współfinansowanego ze środków Europejskiego Funduszu Społecznego.

Plan wykładu nr 4

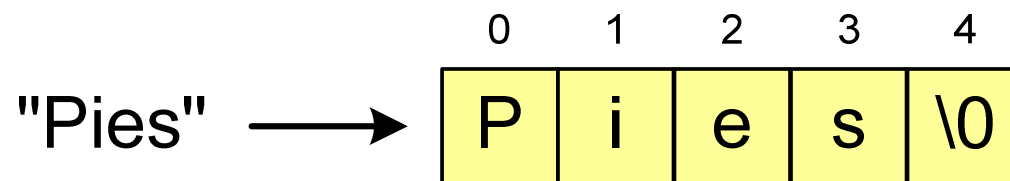
- Łańcuchy znaków w języku C
- Struktury, pola bitowe, unie
- Wskaźniki

Język C - łańcuchy znaków

- **łańcuch znaków** (ciąg znaków, napis, literał łańcuchowy, stała łańcuchowa, C-string) - ciąg złożony z zera lub większej liczby znaków zawartych między znakami cudzysłowu

"Pies"

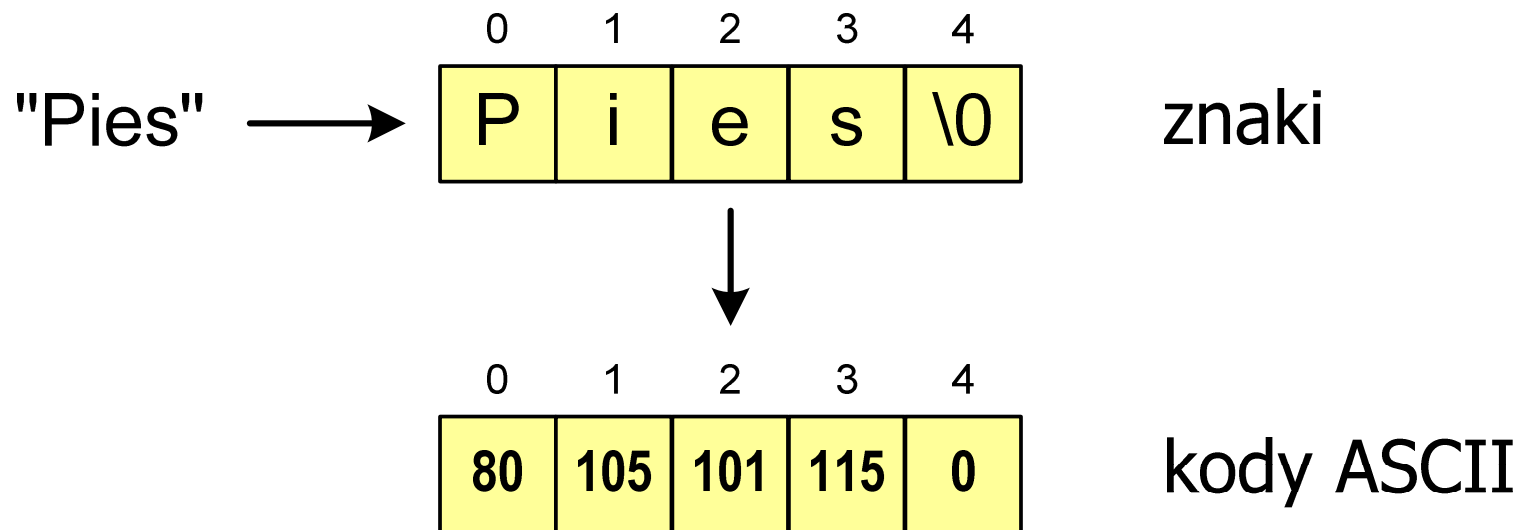
- Implementacja - tablica, której elementami są pojedyncze znaki (typ **char**)



- Ostatni znak (**\0**, liczba **zero**, znak zerowy) oznacza koniec napisu

Język C - łańcuchy znaków

- W rzeczywistości w tablicy zamiast znaków przechowywane są odpowiadające im kody ASCII (czyli liczby)



Język C - deklaracja łańcucha znaków

- Deklaracja zmiennej przechowującej łańcuch znaków

```
char nazwa_zmiennej[rozmiar];
```

Przykład:

```
char txt[10];
```

- Tablica `txt` może przechowywać napisy o maksymalnej długości do 9 znaków

Język C - inicjalizacja łańcucha znaków

- Inicjalizacja łańcucha znaków

```
char txt1[10] = "Pies";  
char txt2[10] = {'P', 'i', 'e', 's'};  
char txt3[10] = {80, 105, 101, 115};
```

- Pozostałe elementy tablicy otrzymują wartość zero

P	i	e	s	\0	\0	\0	\0	\0	\0
---	---	---	---	----	----	----	----	----	----

```
char txt4[] = "Pies";  
char *txt5 = "Pies";
```

Język C - inicjalizacja łańcucha znaków

- Inicjalizacja możliwa jest tylko przy deklaracji

```
char txt[10];  
txt = "Pies";    /* BŁĄD!!! */
```

- Przypisanie zmiennej `txt` wartości `"Pies"` wymaga zastosowania funkcji `strcpy()` z pliku nagłówkowego `string.h`

```
char txt[10];  
strcpy(txt, "Pies");
```

Język C - stała znakowa

- **Stałą znakową** tworzy jeden znak ujęty w apostrofy

```
char zn = 'x';
```

- W rzeczywistości stała znakowa jest to liczba całkowita, której wartość odpowiada wartości kodu ASCII reprezentowanego znaku
- Zamiast powyższego kodu można napisać:

```
char zn = 120;
```

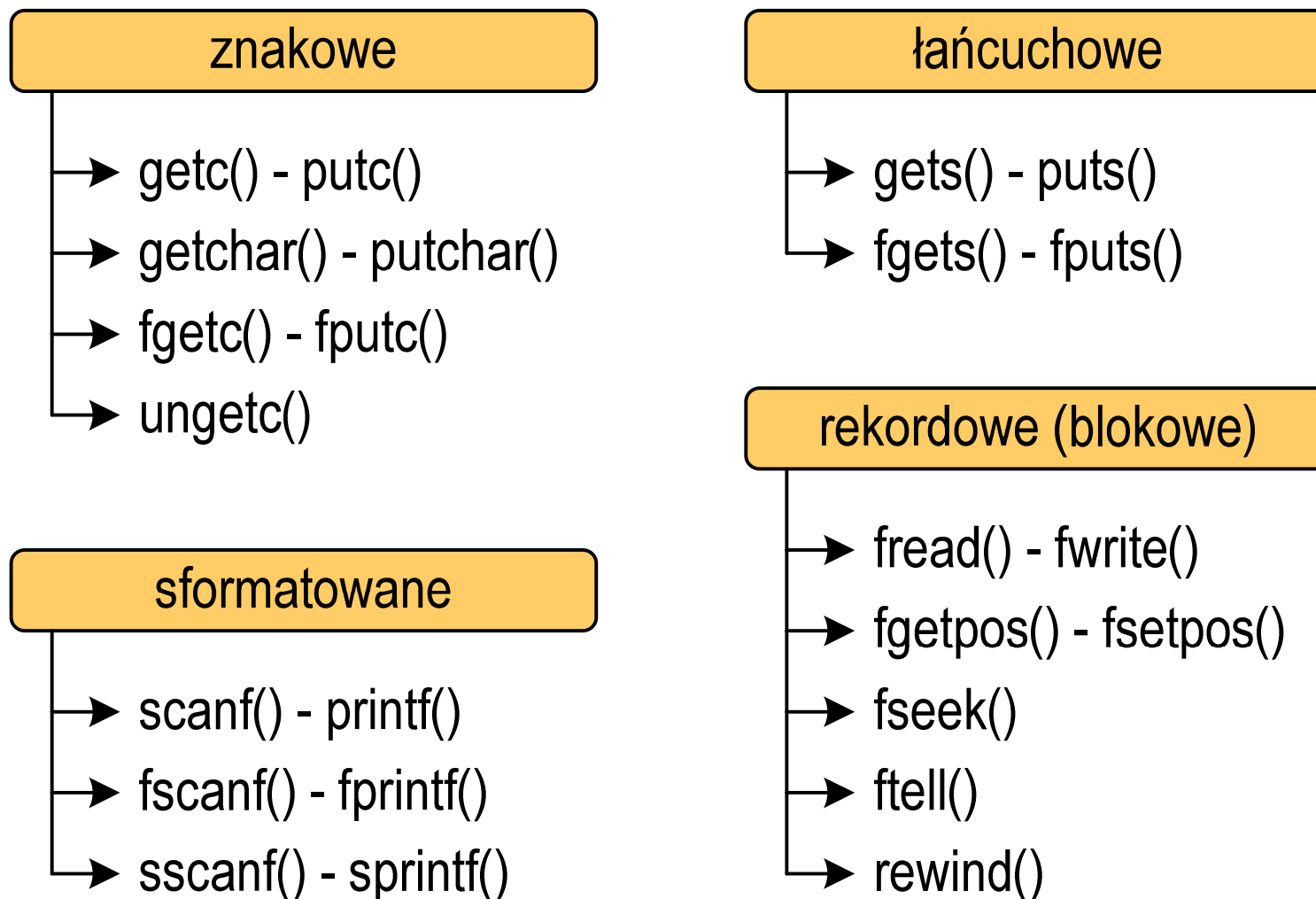
- Uwaga:
 - **'x'** - stała znakowa (jeden znak)
 - **"x"** - łańcuch znaków (dwa znaki: x oraz \0)

Język C - stała znakowa

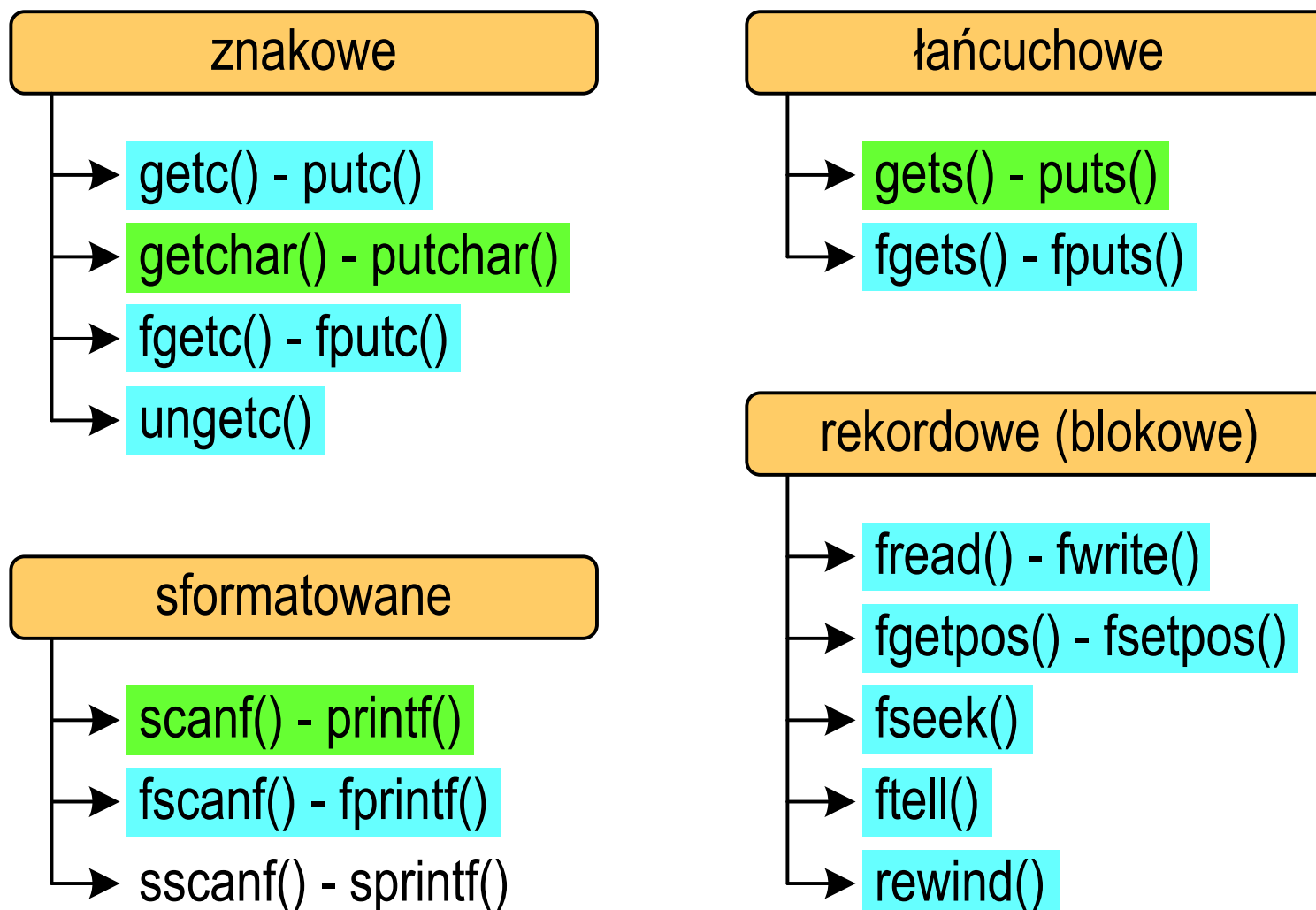
- Niektóre znaki mogą być reprezentowane w stałych znakowych przez sekwencje specjalne, które wyglądają jak dwa znaki, ale reprezentują tylko jeden znak

' \n ' - nowy wiersz	' \\ ' - \ (ang. backslash)
' \t ' - tabulator poziomy	' \' ' - apostrof
' \v ' - tabulator pionowy	' \" ' - cudzysłów
' \a ' - alarm	' \? ' - znak zapytania

Język C - standardowe funkcje wejścia-wyjścia



Język C - standardowe funkcje wejścia-wyjścia



Język C - wyświetlenie tekstu

- Wyświetlenie tekstu funkcją `printf()` wymaga specyfikatora `%s`

```
char napis[15] = "Jan Kowalski";  
printf("Osoba: [%s]\n", napis);
```

```
Osoba: [Jan Kowalski]
```

- W specyfikatorze `%s`: szerokość określa szerokość pola, zaś precyzja - liczbę pierwszych znaków z łańcucha

```
char napis[15] = "Jan Kowalski";  
printf("[%10.6s]\n", napis);
```

```
[    Jan Ko]
```

Język C - wyświetlenie tekstu

- Do wyświetlenia tekstu można zastosować funkcję `puts()`

```
puts()
```

```
int puts(const char *s);
```

- Funkcja `puts()` wypisuje na `stdout` (ekran) zawartość łańcucha znakowego (ciąg znaków zakończony znakiem `'\0'`), zastępując znak `'\0'` znakiem `'\n'`

```
char napis[15] = "Jan Kowalski";  
puts(napis);
```

```
Jan Kowalski
```

Język C - wyświetlenie tekstu

- Wyświetlenie znaku funkcją `printf()` wymaga specyfikatora `%c`

```
char zn = 'x';  
printf("Znak to: [%c]\n", zn);
```

```
Znak to: [x]
```

- Do wyświetlenia znaku można zastosować także funkcję `putchar()`

```
putchar()      int putchar(int znak);
```

```
putchar('K'); putchar(111); putchar(0x74);
```

```
Kot
```

Język C - wyświetlenie tekstu

- Łańcuch znaków jest zwykłą tablicą - można więc odwoływać się do jej pojedynczych elementów

```
char txt[15] = "Ola ma laptopa";  
printf("Znaki: ");  
for (int i=0; i<15; i++) printf("%c ",txt[i]);
```

```
Znaki: O l a   m a   l a p t o p a
```

```
printf("Kody: ");  
for (int i=0; i<15; i++) printf("%d ",txt[i]);
```

```
Kody:  79 108 97 32 109 97 32 108 97 112 116 111 112 97 0
```

Język C - wczytanie tekstu

- Do wczytania tekstu funkcją `scanf()` stosowany jest specyfikator `%s`

```
char napis[15];  
scanf("%s", napis);
```

brak znaku `&`

- W specyfikatorze formatu `%s` można podać szerokość

```
char napis[15];  
scanf("%10s", napis);
```

- W powyższym przykładzie `scanf()` zakończy wczytywanie tekstu po pierwszym białym znaku (spacja, tabulacja, enter) lub w momencie pobrania 10 znaków

Język C - wczytanie tekstu

- W przypadku wprowadzenia tekstu "To jest napis", funkcja `scanf()` zapamięta tylko wyraz "To"
- Zapamiętanie całego wiersza tekstu (do naciśnięcia klawisza `Enter`) wymaga użycia funkcji `gets()`

```
gets ( )
```

```
char *gets (char *s) ;
```

- Funkcja `gets()` wprowadza wiersz (ciąg znaków zakończony `'\n'`) ze strumienia `stdin` (klawiatura) i umieszcza w obszarze pamięci wskazywanym przez wskaźnik `s` zastępując `'\n'` znakiem `'\0'`

```
char napis [15] ;  
gets (napis) ;
```

Język C - wczytanie znaku

- Wczytanie jednego znaku funkcją `scanf()` wymaga specyfikatora formatu `%c` (przed zmienną `znak` musi wystąpić operator `&`)

```
int znak;  
scanf ("%c", &znak);
```

- Do wczytania znaku można zastosować także funkcję `getchar()`

<code>getchar()</code>	<code>int getchar(void);</code>
------------------------	---------------------------------

```
int znak;  
znak = getchar();
```

Język C - plik nagłówkowy string.h

`strcpy()`

```
char *strcpy(char *s1, const char *s2);
```

- Kopiuje łańcuch `s2` do łańcucha `s1`

`strlen()`

```
size_t strlen(const char *s);
```

- Zwraca długość łańcucha znaków, nie uwzględnia znaku `'\0'`

`strcat()`

```
char *strcat(char *s1, const char *s2);
```

- Dołącza do łańcucha `s1` łańcuch `s2`

Język C - plik nagłówkowy string.h

`strcmp()`

```
int strcmp(const char *s1, const char *s2);
```

- Porównuje łańcuchy `s1` i `s2` z rozróżnianiem wielkości liter

`strncmp()`

```
int strncmp(const char *s1, const char *s2);
```

- Porównuje łańcuchy `s1` i `s2` bez rozróżniania wielkości liter

`strchr()`

```
char *strchr(const char *s, int c);
```

- Szuka w łańcuchu `s` znaku `c`

Język C - plik nagłówkowy string.h

`strlwr()`

`char *strlwr(char *s);`

- Zamienia w łańcuchu **s** wielkie litery na małe

`strupr()`

`char *strupr(char *s);`

- Zamienia w łańcuchu **s** małe litery na wielkie

`strrev()`

`char *strrev(char *s);`

- Odwraca kolejność znaków w łańcuchu **s**

Język C - plik nagłówkowy string.h (przykład)

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char napis1[] = "Tekst w buforze", napis2[20];

    printf("napis1: %s \n", napis1);
    int dlugosc = strlen(napis1);
    printf("liczba znakow w napis1: %d \n", dlugosc);
    strcpy(napis2, napis1);
    printf("napis2: %s \n", napis2);
    strrev(napis2);
    printf("napis2 (odwr): %s \n", napis2);

    return 0;
}
```

Język C - plik nagłówkowy string.h (przykład)

```
#include <stdio.h>
#include <string.h>
```

```
int main(void)
{
```

```
    char napis1[] = "T
```

```
    printf("napis1: %s \n", napis1);
```

```
    int dlugosc = strlen(napis1);
```

```
    printf("liczba znakow w napis1: %d \n", dlugosc);
```

```
    strcpy(napis2, napis1);
```

```
    printf("napis2: %s \n", napis2);
```

```
    strrev(napis2);
```

```
    printf("napis2 (odwr): %s \n", napis2);
```

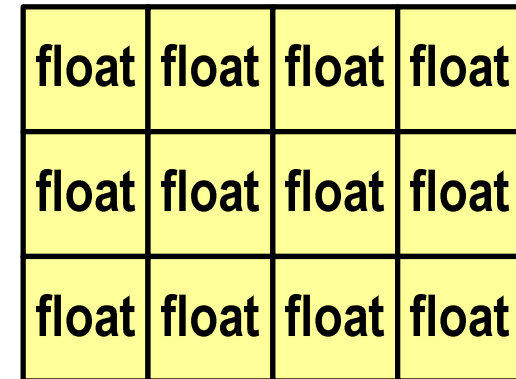
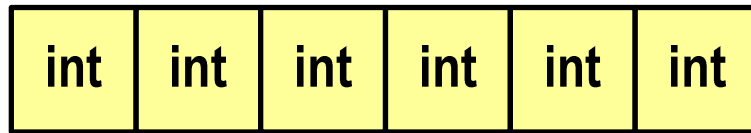
```
    return 0;
```

```
}
```

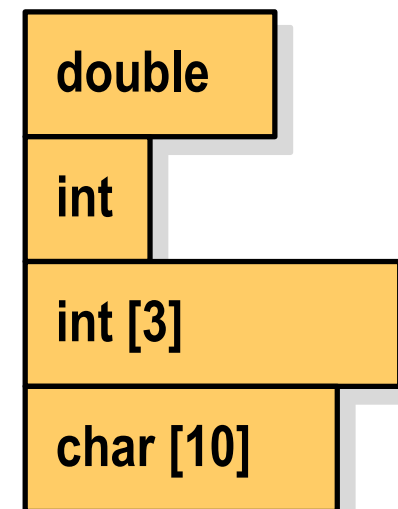
```
napis1: Tekst w buforze
liczba znakow w napis1: 15
napis2: Tekst w buforze
napis2 (odwr): ezrofub w tskeT
```


Struktury w języku C

- **Tablica** - ciągły obszar pamięci zawierający elementy tego samego typu



- **Struktura** - zestaw elementów różnych typów, zgrupowanych pod jedną nazwą



Deklaracja struktury

```
struct nazwa
{
    opis_pola_1;
    opis_pola_2;
    ...
    opis_pola_n;
};
```

```
struct punkt
{
    int x;
    int y;
};
```

- Elementy struktury to **pola** (dane, komponenty, składowe) struktury
- Deklaracje pól mają taką samą postać jak deklaracje zmiennych
- Deklarując strukturę tworzymy nowy typ danych (**struct punkt**), którym można posługiwać się tak samo jak każdym innym typem standardowym

Deklaracja struktury

```
struct osoba
{
    char imie[15];
    char nazwisko[20];
    int wiek, waga;
};
```

```
struct zesp
{
    float Re, Im;
};
```

- Deklaracja struktury nie tworzy obiektu (nie przydziela pamięci na pola struktury)
- Zapisanie danych do struktury wymaga zdefiniowania **zmiennej strukturalnej**

Deklaracja zmiennej strukturalnej

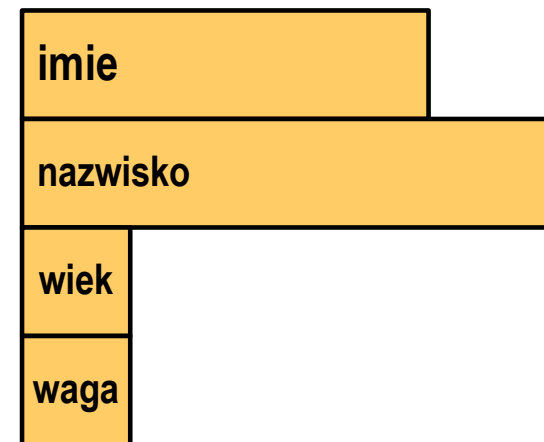
```
#include <stdio.h>

struct osoba
{
    char imie[15];
    char nazwisko[20];
    int wiek, waga;
} Kowal ;

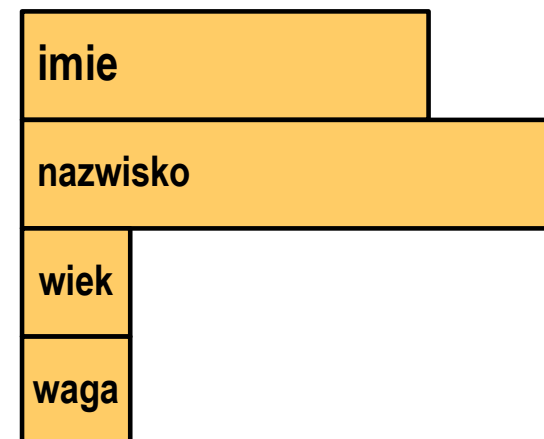
int main(void)
{
    struct osoba Nowak ;
    ...
}
```

- **Kowal, Nowak** - zmienne typu **struct osoba**

Kowal



Nowak



Odwołania do pól struktury

- Dostęp do pól struktury możliwy jest dzięki konstrukcji typu:

```
nazwa_struktury.nazwa_pola
```

- Operator `.` nazywany jest **operatorem bezpośredniego wyboru pola**
- Zapisanie wartości **25** do pola **wiek** zmiennej **Nowak** ma postać

```
Nowak.wiek = 25;
```

- Wyrażenie **Nowak.wiek** traktowane jest jak zmienna typu **int**

```
printf("Wiek: %d\n", Nowak.wiek);  
scanf("%d", &Nowak.wiek);
```

Odwołania do pól struktury

- Dostęp do pól struktury możliwy jest dzięki konstrukcji typu:

```
nazwa_struktury.nazwa_pola
```

- Operator `.` nazywany jest **operatorem bezpośredniego wyboru pola**
- Zapisanie wartości **Jan** do pola **imie** zmiennej **Nowak** ma postać

```
strcpy(Nowak.imie, "Jan");
```

- Wyrażenie **Nowak.imie** traktowane jest jak łańcuch znaków

```
printf("Imie: %s\n", Nowak.imie);  
gets(Nowak.imie);
```


Struktury - przykład

```
#include <stdio.h>

struct osoba
{
    char imie[15];
    char nazwisko[20];
    int  wiek;
};

int main(void)
{
    struct osoba Nowak;
```

Struktury - przykład

```
printf("Imie:      ");  
gets(Nowak.imie);  
  
printf("Nazwisko: ");  
gets(Nowak.nazwisko);  
  
printf("Wiek:      ");  
scanf("%d", &Nowak.wiek);  
  
printf("%s %s, wiek: %d\n", Nowak.imie,  
      Nowak.nazwisko, Nowak.wiek);  
  
return 0;  
}
```

```
Imie:      Jan  
Nazwisko:  Nowak  
Wiek:      22  
Jan Nowak, wiek: 22
```

Struktury w języku C

- **Inicjalizacja** może dotyczyć tylko zmiennych strukturalnych, nie można inicjalizować pól w deklaracji struktury

```
struct osoba
{
    char imie[15], nazwisko[20];
    int wiek, waga;
};
```

```
struct osoba Nowak = {"Jan", "Nowak", 25, 74};
```

- Do zmiennych strukturalnych można stosować **operator =**

```
struct osoba Kowal = {"Ewa", "Kowal", 21, 54};
struct osoba Kowal1;
Kowal1 = Kowal;
```

Struktury w języku C

```
#include <stdio.h>

struct date
{
    int day;
    int month;
    int year;
} day1;

int main(void)
{
    struct date day2 = {19, 11, 2018};
}
```

day1

day	?
month	?
year	?

day2

day	19
month	11
year	2018

Struktury w języku C

```
    day1.day = 1;
    day1.month = 9;
    day1.year = 2018;

    printf("Date1: %02d-%02d-%4d\n",
           day1.day, day1.month, day1.year);
    printf("Date2: %02d-%02d-%4d\n",
           day2.day, day2.month, day2.year);

    return 0;
}
```

```
Date1: 01-09-2018
Date2: 19-11-2018
```

day1

day	1
month	9
year	2018

day2

day	19
month	11
year	2018

Złożone deklaracje struktur

```
struct punkt  
{  
    int x;  
    int y;  
} tab[3];
```

tab

0	x	y
1	x	y
2	x	y

```
tab[0].x = 10;  
tab[0].y = 20;  
tab[1].x = 15;  
...
```

```
struct trojkat  
{  
    int nr;  
    struct punkt A, B, C;  
} Tr1;
```

Tr1

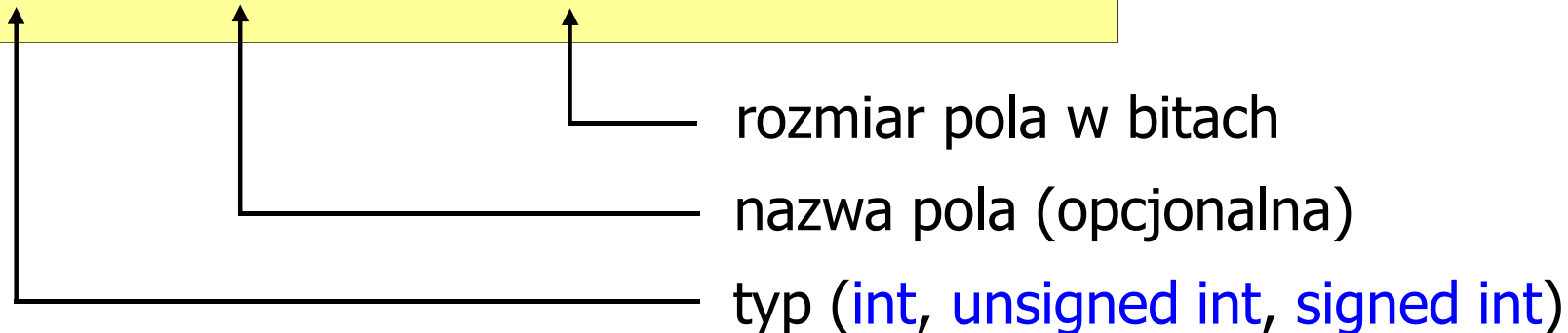
nr		
A	x	y
B	x	y
C	x	y

```
Tr1.nr = 1;  
Tr1.A.x = 10;  
Tr1.A.y = 20;  
Tr1.B.x = 15;  
...
```

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   : 4;
    unsigned int c : 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()**

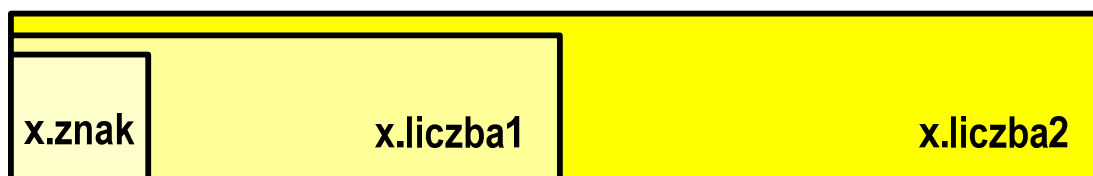
Unie

- Specjalny rodzaj struktury umożliwiający przechowywanie danych różnych typów w **tym samym obszarze pamięci**
- Do przechowywania wartości w unii należy zadeklarować zmienną

```
union zbior x;
```

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

- 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

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

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

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

- Unię można zainicjować jedynie wartością o typie jej pierwszej składowej
- Unie tego samego typu można sobie przypisywać

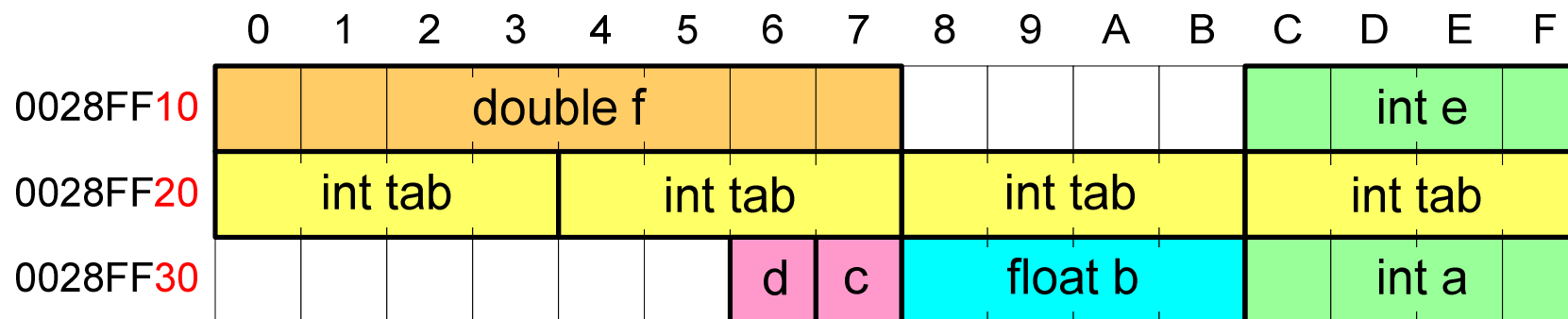
```
union zbior x = {'a'};  
union zbior y;  
y = x;
```

Co to jest wskaźnik?

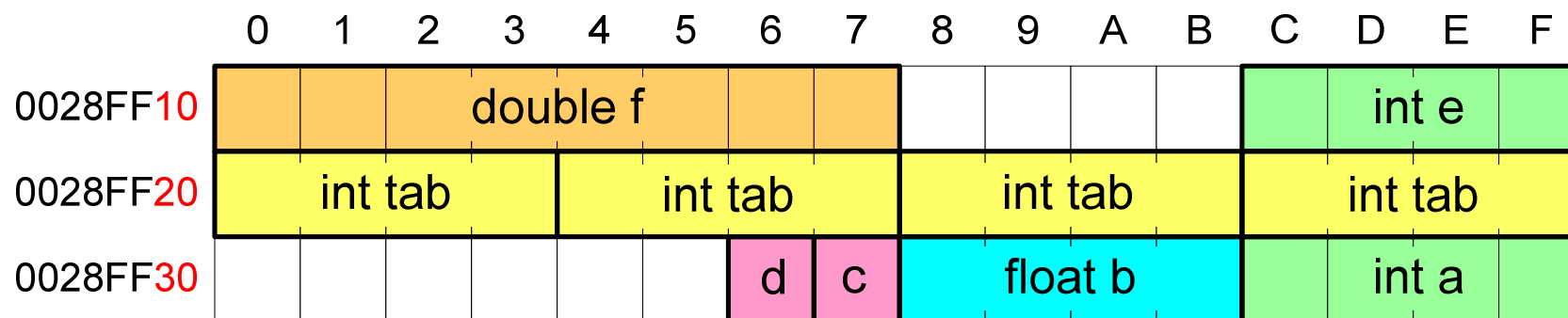
- **Wskaźnik** - zmienna mogąca zawierać adres obszaru pamięci
- najczęściej adres innej zmiennej (obiektu)

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

- Zmienne przechowywane są w pamięci komputera



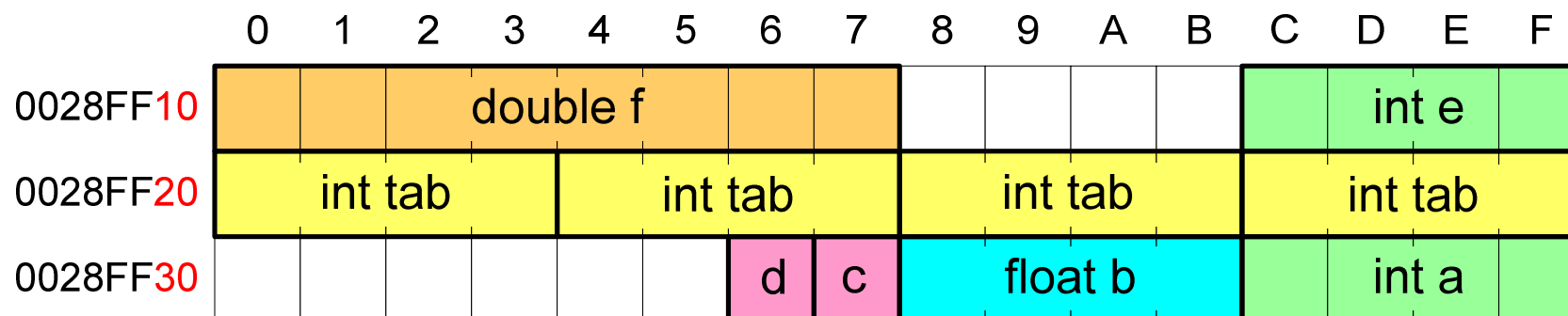
Co to jest wskaźnik?



- 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:

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

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

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;
```

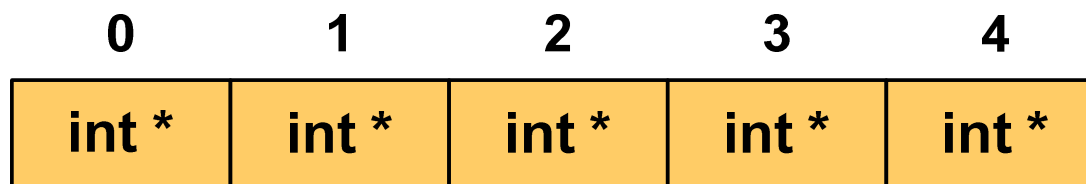
- Można konstruować wskaźniki do danych dowolnego typu łącznie z typami **wskaźnik do wskaźnika do...**

```
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:

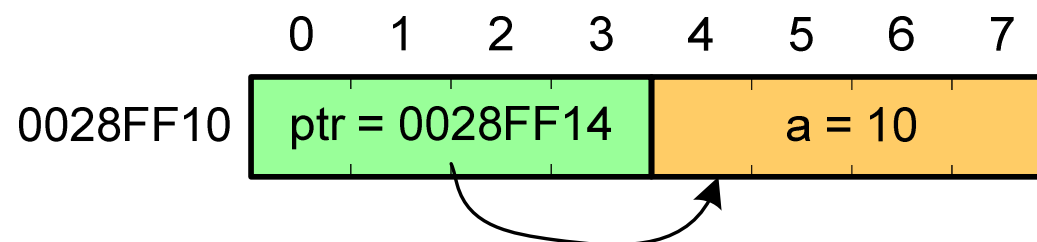
```
int *p1, *p2;  
int* p3, p4;
```

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

Przypisywanie wartości wskaźnikom

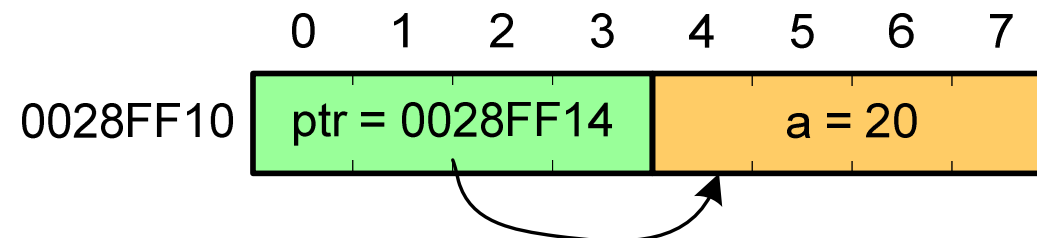
- Wskaźnikom można przypisywać adresy zmiennych
- Adresy takie tworzy się za pomocą operatora pobierania adresu **&**

```
int a = 10;  
int *ptr;  
ptr = &a;
```



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

```
*ptr = 20;
```



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;
```

Przykład: przypisywanie wartości wskaźnikom

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int x = 15;
```

```
    int *ptri = NULL;
```

```
    printf("x =      %d\n", x);
```

```
    printf("ptri = %p\n", ptri);
```

```
    ptri = &x;                // przypisanie adresu
```

```
    printf("ptri = %p\n", ptri);
```

```
    *ptri = *ptri + 10;      // x = x + 10
```

```
    printf("x =      %d\n", x);
```

```
    printf("x =      %d\n", *ptri);
```

```
    return 0;
```

```
}
```

```
x =      15
```

```
ptri = 0000000000000000
```

```
ptri = 00000000010FF960
```

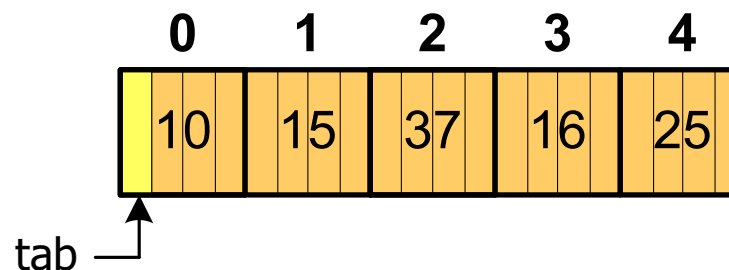
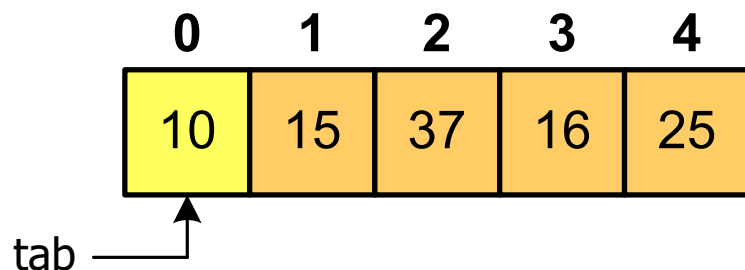
```
x =      25
```

```
x =      25
```

Wskaźniki a tablice

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

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

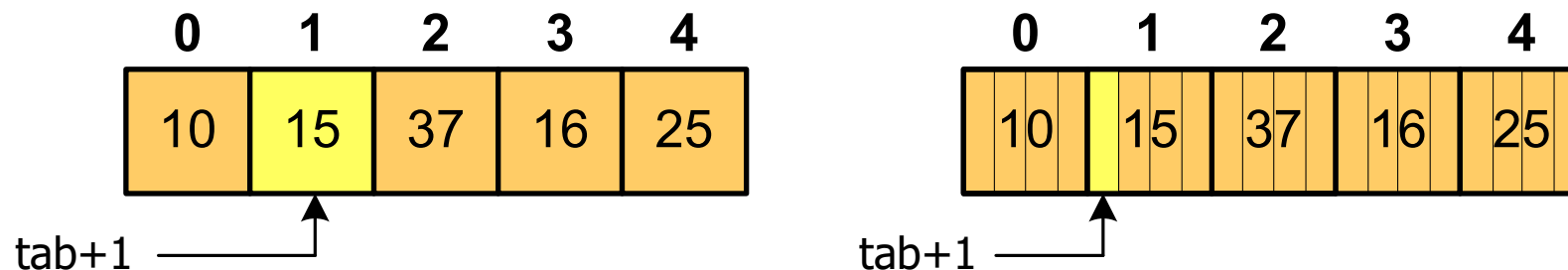


- Zastosowanie operatora `*` przed nazwą tablicy pozwala „dostać się” do zawartości elementu o indeksie 0

`*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 $x = tab[0]+2;$

Koniec wykładu nr 4

Dziękuję za uwagę!