

# Informatyka 2

Politechnika Białostocka - Wydział Elektryczny  
Elektrotechnika, semestr III, studia stacjonarne I stopnia  
Rok akademicki 2017/2018

## Wykład nr 2 (09.10.2017)

dr inż. Jarosław Forenc

## Plan wykładu nr 2

- Tablice o zmiennym rozmiarze (VLA)
- Struktury, pola bitowe, unie
- Wskaźniki
- Dynamiczny przydział pamięci
- Dynamiczne struktury danych

## Tablice o zmiennym rozmiarze (VLA)

- VLA (ang. variable length array) - tablice, których rozmiar określany jest na etapie wykonywania programu (np. jako rozmiar może wystąpić nazwa zmiennej)

```
int n;  
n = 10;  
int T[n];
```

```
int n;  
scanf("%d", &n);  
int T[n];
```

- Rozmiar tablicy, a standardy języka C:
  - do standardu C99 rozmiar tablicy musiał być stałym wyrażeniem całkowitym (stała liczbowa: 5, #define N 5, const int n = 5;)
  - w standardzie C99 wprowadzono tablice o zmiennym rozmiarze
  - w standardzie C11 tablice o zmiennym rozmiarze określone są jako opcjonalne dla implementacji

## Tablice VLA (VC++ 2008)

```
#include <stdio.h>  
#include <math.h>  
  
int main(void)  
{  
    int n, i;  
  
    printf("Rozmiar wektora: ");  
    scanf("%d", &n);  
  
    float T[n];  
  
    for (i=0; i<n; i++)  
        T[i] = sqrt((float)i);  
  
    for (i=0; i<n; i++)  
        printf("T[%d] = %f\n", i, T[i]);  
  
    return 0;  
}
```

## Tablice VLA (VC++ 2008)

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

int main(void)
{
    int n, i;

    printf("Rozmiar wektora: ");
    scanf("%d", &n);

    float T[n];

    for (i=0; i<n; i++)
        T[i] = sqrt((float)i);

    for (i=0; i<n; i++)
        printf("T[%d] = %f\n", i, T[i]);

    return 0;
}
```

error C2057: expected constant expression  
error C2466: cannot allocate an array of constant size 0  
error C2133: 'T' : unknown size

## Tablice VLA (Dev-C++, Code::Blocks)

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

int main(void)
{
    int n, i;

    printf("Rozmiar wektora: ");
    scanf("%d", &n);

    float T[n];

    for (i=0; i<n; i++)
        T[i] = sqrt((float)i);

    for (i=0; i<n; i++)
        printf("T[%d] = %f\n", i, T[i]);

    return 0;
}
```

Rozmiar wektora: 8  
T[0] = 0.000000  
T[1] = 1.000000  
T[2] = 1.414214  
T[3] = 1.732051  
T[4] = 2.000000  
T[5] = 2.236068  
T[6] = 2.449490  
T[7] = 2.645751

## Tablice VLA

- Tablica VLA może być także tablicą dwu- lub wielowymiarową

```
int n = 5, m = 6;
int T1[n][m], T2[n][m][n];
```

- Nie można modyfikować rozmiaru tablic VLA po deklaracji
- Tablice VLA nie mogą być inicjalizowane podczas deklaracji
  - błędy i ostrzeżenia w Code::Blocks

```
error: variable-sized object may not be initialized
warning: excess elements in array initializer
warning: (near initialization for 'T')
```

- w Dev-C++ inicjalizacja jest dopuszczalna!

## Modularność programu

- Program komputerowy powinien być podzielony na osobne **jednostki**, z których każda wykonuje jedno zadanie
- Moduły (jednostki) to najczęściej **funkcje** języka C (ale mogą to być też oddzielne pętle)
- Zalety budowy modularnej programu:
  - większa czytelność kodu programu
  - prostsza modyfikacja programu

## Modularność programu

### ■ Przykład

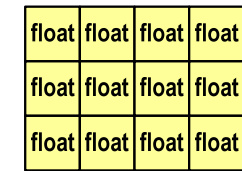
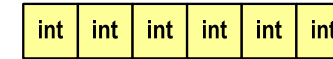
```
int T[10], i, s = 0;
srand(time(NULL));
for(i=0; i<10; i++)
{
    T[i] = rand()%100;
    printf("%4d",T[i]);
    s = s + T[i];
}
```

```
int T[10], i, s = 0;
srand(time(NULL));
for(i=0; i<10; i++)
    T[i] = rand()%100;
for(i=0; i<10; i++)
    printf("%4d",T[i]);
for(i=0; i<10; i++)
    s = s + T[i];
```

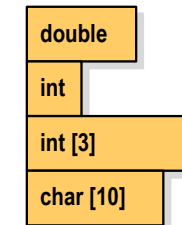
- Zamiast jednej pętli **for** stosowane są trzy pętle

## 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;
};
```

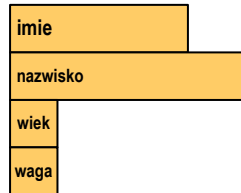
- Deklaracja struktury nie tworzy obiektu (nie przydziela pamięci na pola struktury)
- Zapisanie danych do struktury wymaga zdefiniowania **zmiennnej strukturalnej**

## Deklaracja zmiennej strukturalnej

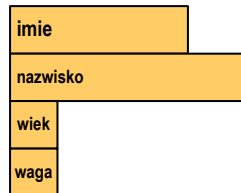
```
struct osoba  
{  
    char imie[15];  
    char nazwisko[20];  
    int wiek, waga;  
} Kowal, Nowak;
```

- Kowal, Nowak  
- zmienne strukturalne  
typu `struct osoba`

Kowal



Nowak



## Deklaracja zmiennej strukturalnej

```
#include <stdio.h>  
  
struct osoba  
{  
    char imie[15];  
    char nazwisko[20];  
    int wiek, waga;  
};  
  
int main(void)  
{  
    struct osoba Kowal;  
    struct osoba Nowak;  
    ...  
    return 0;  
}
```

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);
```

## Odwołania do pól struktury

- Gdy zmienna strukturalna jest wskaźnikiem, to do odwołania do pola struktury używamy **operatora pośredniego wyboru pola (->)**

```
wskaźnik_do_struktury -> nazwa_pola
```

```
struct osoba Nowak, *Nowak1;  
Nowak1 = &Nowak;  
Nowak1 -> wiek = 25;  
  
/* lub */  
(*Nowak1).wiek = 25;
```

- W ostatnim zapisie nawiasy są konieczne, gdyż operator **.** ma wyższy priorytet niż operator **\***

## 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
```

## Inicjalizacja zmiennej strukturalnej

- Inicjalizowane mogą być tylko zmienne strukturalne, nie można inicjalizować pól w deklaracji struktury

```
struct osoba  
{  
    char imie[15];  
    char nazwisko[20];  
    int wiek, waga;  
};  
  
int main(void)  
{  
    struct osoba Nowak1 = {"Jan", "Nowak", 25, 74};  
    ...  
}
```

## Struktury a operator przypisania (=)

- Struktury tego samego typu można sobie przypisywać (nawet jeśli zawierają tablice)

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

int main(void)
{
    struct osoba Nowak1 = {"Jan", "Nowak", 25, 74};
    struct osoba Nowak2;

    Nowak2 = Nowak1;
}
```

operator przypisania

## 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;
```

rozmiar pola w bitach

nazwa pola (opcjonalna)

typ (int, unsigned int, signed int)

- 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 c : 4;
    unsigned int d : 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;
};
```

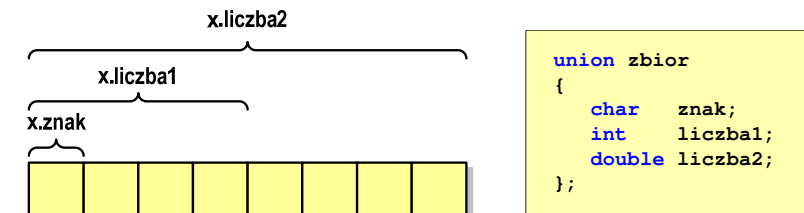
- Do przechowywania wartości w unii należy zadeklarować zmienną

```
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'};
```

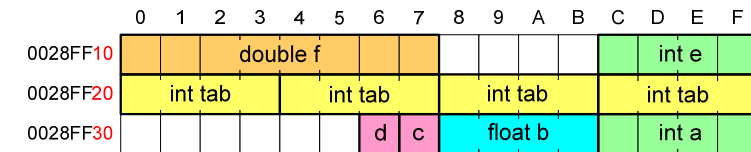
- Unie tego samego typu można sobie przypisywać

## Co to jest wskaźnik?

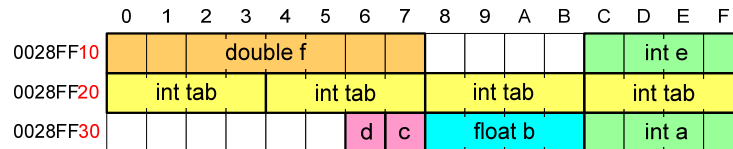
- **Wskaźnik** - zmienna mogąca zawierać adres obszaru pamięci - najczęściej adres innej zmiennej (obiektu)

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

- Zmienne przechowywane są w pamięci komputera



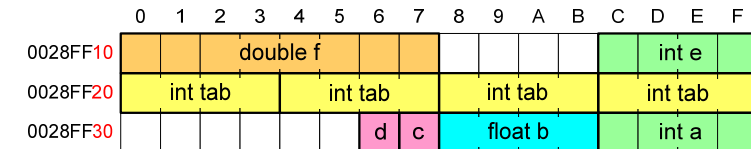
## Co to jest wskaźnik?



- 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:

```
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;
```

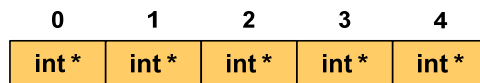
- Można konstruować wskaźniki do danych dowolnego typu łącznie z typami **wskaźnik do wskaźnika do...**

```
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:

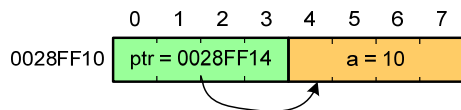
```
int *p1, *p2;  
int* p3, p4;
```

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

## Przypisywanie wartości wskaźnikom

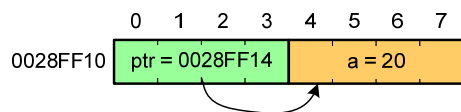
- Wskaźnikom można przypisywać adresy zmiennych
- Adresy takie tworzy się za pomocą operatora pobierania adresu &

```
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;
```

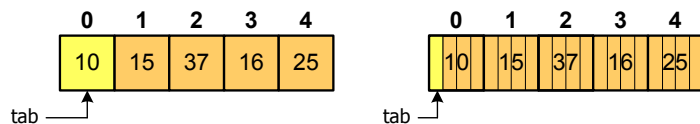
- Zamiast wartości **0** można stosować makrodefinicję preprocesora **NULL**, która podczas kompilacji programu zamieniana jest na **0**

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

## Wskaźniki a tablice

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

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

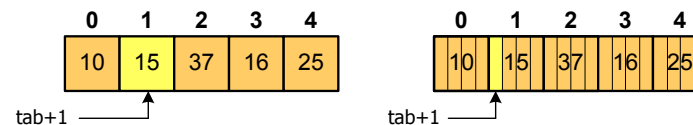


- Zastosowanie operatora **\*** przed nazwą tablicy pozwala „dostać się” do zawartości elementu o indeksie **0**

**\*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;`

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

```
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
```

## Dynamiczny przydział pamięci na wektor

- Wczytanie liczb bezpośrednio do wektora `tab`

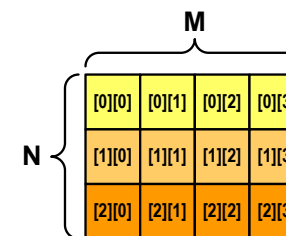
```
for (i=0; i<n; i++) /* wczytanie liczb */
{
    printf("Podaj liczbę nr %d: ", i+1);
    scanf("%d", &tab[i]);
}
```

- Inny sposób odwołania do elementów wektora `tab`

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

## Dynamiczny przydział pamięci na macierz

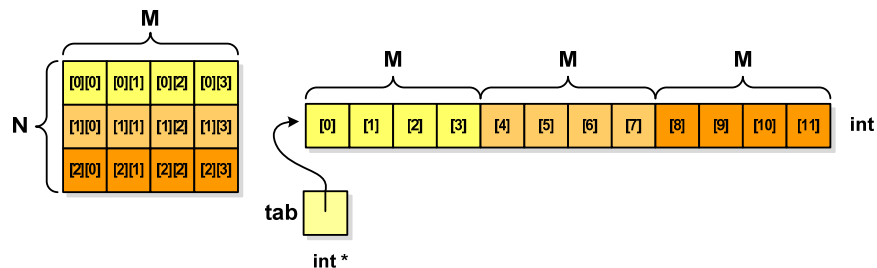
- Funkcje `calloc()` i `malloc()` umożliwiają bezpośrednio przydział pamięci tylko na wektor elementów
- Dynamiczny przydział pamięci na macierz wymaga zastosowania specjalnych metod
- Przydzielamy pamięć na macierz zawierającą **N-wierszy** i **M-kolumn**



## Dynamiczny przydział pamięci na macierz (1)

- Wektor N×M-elementowy
- Przydział pamięci:

```
int *tab = (int *) calloc(N*M, sizeof(int));
```



## Dynamiczny przydział pamięci na macierz (1)

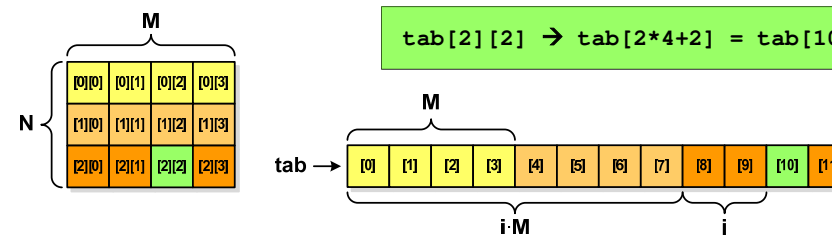
- Odwołanie do elementów macierzy:

`tab[i*M+j]`

lub

`*(tab+i*M+j)`

`tab[2][2] → tab[2*4+2] = tab[10]`



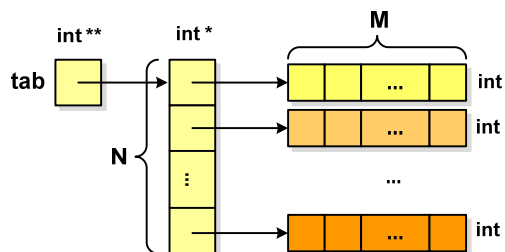
- Zwolnienie pamięci:

```
free(tab);
```

## Dynamiczny przydział pamięci na macierz (2)

- N-elementowy wektor wskaźników + N-wektorów M-elementowych
- Przydział pamięci:

```
int **tab = (int **) calloc(N, sizeof(int *));  
for (i=0; i<N; i++)  
    tab[i] = (int *) calloc(M, sizeof(int));
```

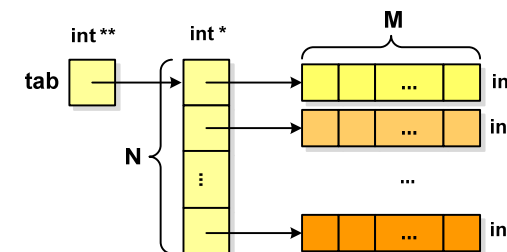


## Dynamiczny przydział pamięci na macierz (2)

- Odwołania do elementów macierzy:
- Zwolnienie pamięci:

tab[i][j]

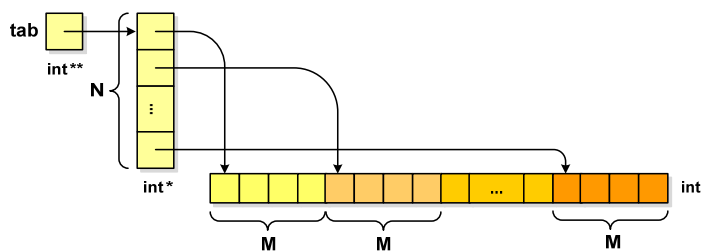
```
for (i=0; i<N; i++)  
    free(tab[i]);  
free(tab);
```



## Dynamiczny przydział pamięci na macierz (3)

- N-elementowy wektor wskaźników + wektor N×M-elementowy
- Przydział pamięci:

```
int **tab = (int **) malloc(N*sizeof(int *));  
tab[0] = (int *) malloc(N*M*sizeof(int));  
for (i=1; i<N; i++)  
    tab[i] = tab[0]+i*M;
```

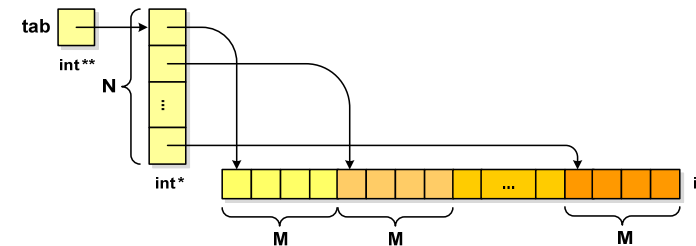


## Dynamiczny przydział pamięci na macierz (3)

- Odwołania do elementów macierzy:
- Zwolnienie pamięci:

tab[i][j]

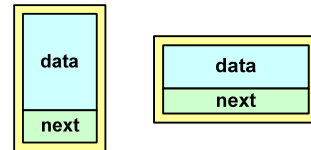
```
free(tab[0]);  
free(tab);
```



## Dynamiczne struktury danych

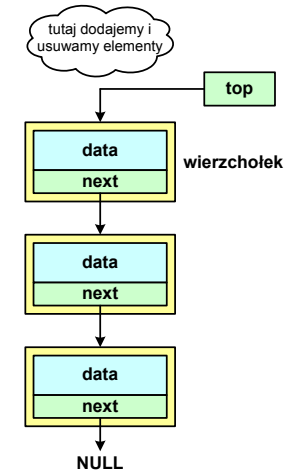
- **Dynamiczne struktury danych** - struktury danych, którym pamięć jest przydzielana i zwalniana w trakcie wykonywania programu
  - stos, kolejka
  - lista (jednokierunkowa, dwukierunkowa, cykliczna)
  - drzewo
- Elementy w dynamicznych strukturach danych są strukturami składającymi się z „użytecznych” danych (**data**) oraz z jednego lub kilku wskaźników (**next**) zawierających adresy innych elementów

```
struct element  
{  
    typ data;  
    struct element *next;  
};
```



## Stos

- **stos** (ang. stack) - struktur składająca się z elementów, z których każdy posiada tylko adres następnika
- dostęp do danych przechowywanych na stosie jest możliwy tylko w miejscu określanym mianem **wierzchołka** stosu (ang. top)
- wierzchołek stosu jest jedynym miejscem, do którego można dołączać lub z którego można usuwać elementy
- każdy składnik stosu posiada wyróżniony element (**next**) zawierający adres następnego elementu
- wskaźnik ostatniego elementu stosu wskazuje na adres pustej (**NULL**)
- podstawowe operacje na stosie to:
  - dodanie elementu do stosu - funkcja **push()**
  - zdjęcie elementu ze stosu - funkcja **pop()**



## Notacja polska

- **Notacja polska** (zapis przedrostkowy, Notacja Łukasiewicza) jest to sposób zapisu wyrażeń arytmetycznych, podający najpierw operator, a następnie argumenty
- Wyrażenie arytmetyczne:

4 / (1 + 3)

ma w notacji polskiej postać:

/ 4 + 1 3

- Wyrażenie powyższe nie wymaga nawiasów, ponieważ przypisanie argumentów do operatorów wynika wprost z ich kolejności w zapisie
- Notacja ta była podstawą opracowania tzw. **odwrotnej notacji polskiej**

## Odwrotna notacja polska

- **Odwrotna Notacja Polska** - ONP (ang. Reverse Polish Notation, RPN) jest sposobem zapisu wyrażeń arytmetycznych, w którym operator umieszczany jest **po** argumentach
- Wyrażenie arytmetyczne:

(1 + 3) / 2

ma w odwrotnej notacji polskiej postać:

1 3 + 2 /

- Odwrotna notacja polska została opracowana przez australijskiego naukowca **Charlesa Hamblina**

## Odwrotna notacja polska

- Obliczenie wartości wyrażenia przy zastosowaniu ONP wymaga:
  - zamiany notacji konwencjonalnej (nawiasowej) na ONP (algorytm Dijkstry nazywany stacją rozrządową)
  - obliczenia wartości wyrażenia arytmetycznego zapisanego w ONP

■ W obu powyższych algorytmach wykorzystywany jest stos

■ Przykład:

- wyrażenie arytmetyczne:

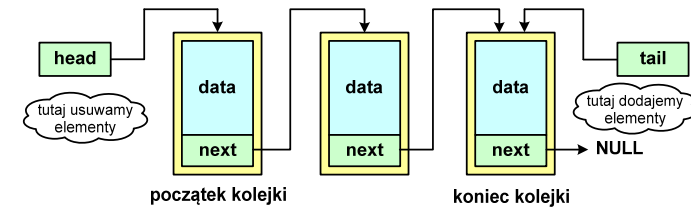
$(2 + 1) * 3 - 4 * (7 + 4)$

- ma w odwrotnej notacji polskiej postać:

$2 1 + 3 * 4 7 4 + * -$

## Kolejka

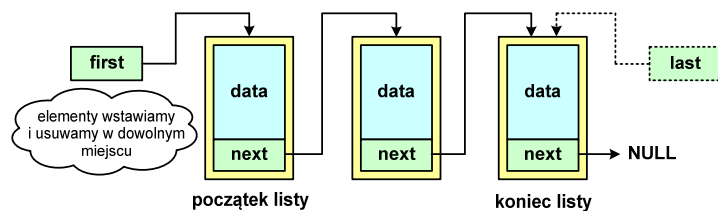
- **Kolejka** - składa się z liniowo uporządkowanych elementów
- Elementy dołączane są tylko na końcu kolejki (wskaźnik **tail**)
- Elementy usuwane są tylko z początku kolejki (wskaźnik **head**)



- Powiązanie między elementami kolejki jest takie samo, jak w stosie
- Kolejka nazywana jest stosem **FIFO** (ang. **First In First Out**)

## Lista jednokierunkowa

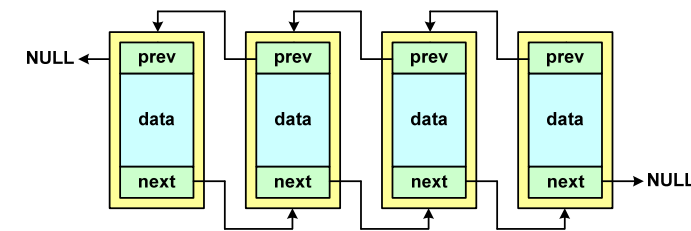
- Organizacja listy jednokierunkowej podobna jest do organizacji stosu i kolejki
- Dla każdego składnika (poza ostatnim) jest określony następny składnik (lub poprzedni - zależnie od implementacji)



- Zapamiętywany jest wskaźnik tylko na pierwszy element listy (**first**) lub wskaźniki na pierwszy (**first**) i ostatni element listy (**last**)
- Elementy listy można dołączać/usuwać w dowolnym miejscu listy

## Lista dwukierunkowa

- Każdy węzeł posiada adres następnika, jak i poprzednika
- W strukturze tego typu wygodne jest przechodzenie pomiędzy elementami w obu kierunkach (od początku do końca i odwrotnie)

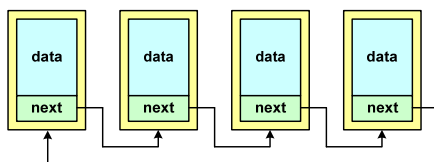




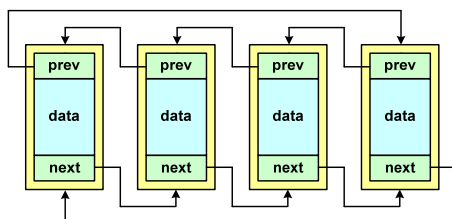
## Lista cykliczna

- Powstaje z listy jednokierunkowej lub dwukierunkowej, poprzez połączenie ostatniego element z pierwszym

Jednokierunkowa:

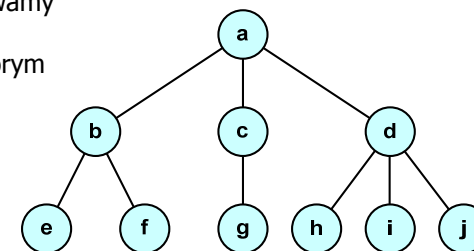


Dwukierunkowa:



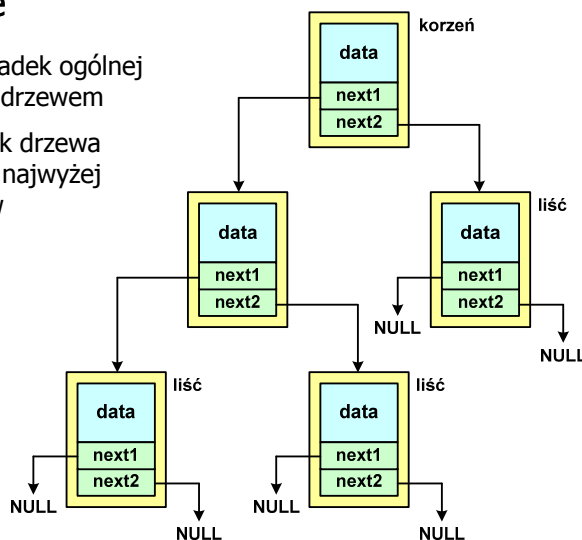
## Drzewo

- Najbardziej ogólna dynamiczna struktura danych, może być reprezentowane graficznie na różne sposoby
- Na górze znajduje się **korzeń drzewa** (a)
- Skojarzone z korzeniem poddrzewa połączone są z nim liniami zwanymi **gałęziami drzewa**
- Potomkiem węzła **w** nazywamy każdy, różny od **w**, węzeł należący do drzewa, w którym **w** jest korzeniem
- Węzeł, który nie ma potomków, to **liść drzewa**



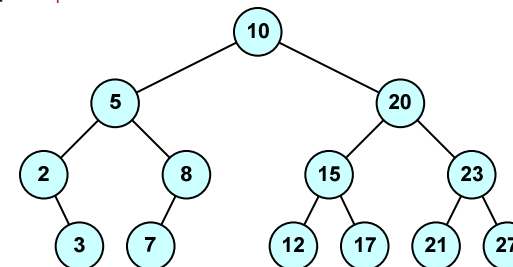
## Drzewo binarne

- Szczególny przypadek ogólnej struktury zwanej drzewem
- Każdy wierzchołek drzewa binarnego ma co najwyżej dwóch potomków



## Binarne drzewo wyszukiwawcze

- Drzewo binarne, w którym dla każdego węzła  $w_i$ :
  - wszystkie klucze w lewym poddrzewie węzła  $w_i$  są mniejsze od klucza w węźle  $w_i$
  - wszystkie klucze w prawym poddrzewie węzła  $w_i$  są większe od klucza w węźle  $w_i$



- Zaleta: szybkość wyszukiwania informacji

Koniec wykładu nr 2

**Dziękuję za uwagę!**  
**(Następny wykład: 23.10.2017)**