

Informatyka 2 (ES1E3017)

Politechnika Białostocka - Wydział Elektryczny
Elektrotechnika, semestr III, studia stacjonarne I stopnia
Rok akademicki 2020/2021

Wykład nr 2 (13.10.2020)

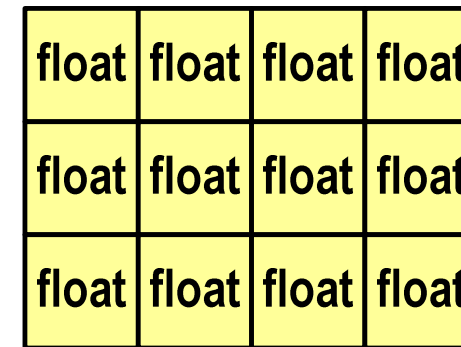
dr inż. Jarosław Forenc

Plan wykładu nr 2

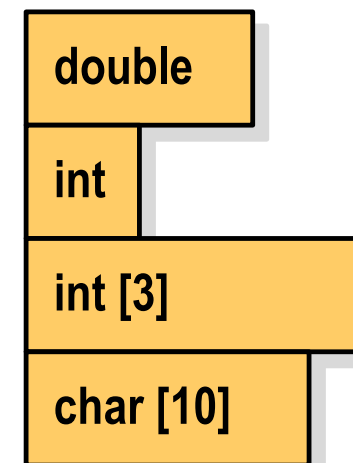
- Struktury, pola bitowe, unie
 - deklaracja struktury i zmiennej strukturalnej
 - odwołania do pól struktury
 - inicjalizacja zmiennej strukturalnej
 - złożone deklaracje struktur
- Wskaźniki
 - deklaracja, przypisanie wartości
 - związek z tablicami, operacje na wskaźnikach
- Dynamiczny przydział pamięci
 - funkcje calloc, malloc, free
 - przydział pamięci na wektor i macierz

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;
};
```

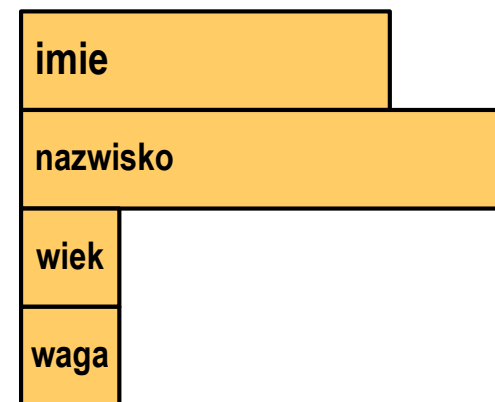
- Deklaracja struktury nie tworzy obiektu (nie przydziela pamięci na pola struktury)
- Zapisanie danych do struktury wymaga zdefiniowania **zmiennej strukturalnej**

Deklaracja zmiennej strukturalnej

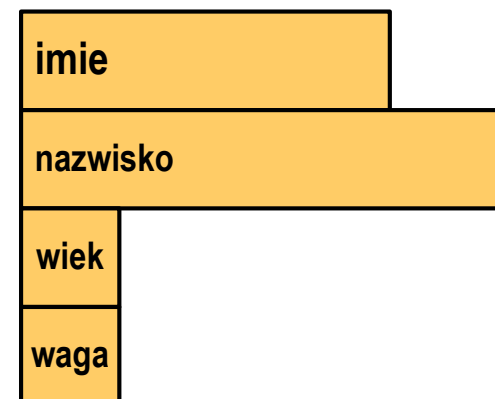
```
struct osoba
{
    char imie[15];
    char nazwisko[20];
    int wiek, waga;
} Kowal, Nowak;
```

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

Kowal



Nowak



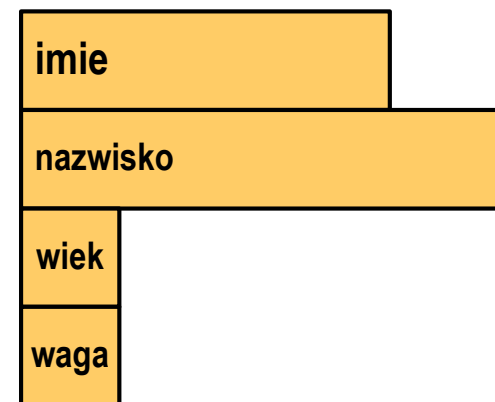
Deklaracja zmiennej strukturalnej

```
#include <stdio.h>

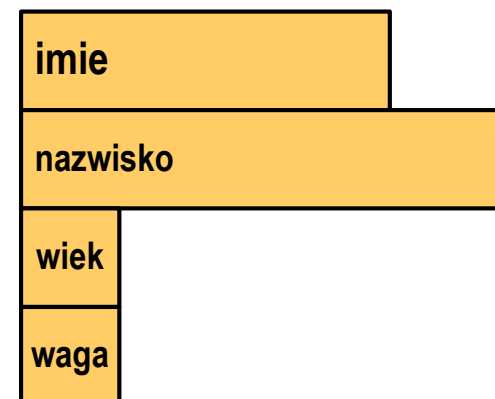
struct osoba
{
    char imie[15];
    char nazwisko[20];
    int wiek, waga;
};

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 (osoba)

```
#include <stdio.h>

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

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

Struktury - przykład (osoba)

```
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 - przykład (miernik)

```
#include <stdio.h>

struct miernik
{
    double k;    // klasa dokładności
    int d;      // liczba działek podziałki
    double Zp;  // zakres pomiarowy
};

int main(void)
{
    // Amperomierz LE-3P
    struct miernik LE3P = {0.5, 75, 12.0};
    double Dpm, p;
```

Struktury - przykład (miernik)

```
printf("Amperomierz analogowy LE-3P\n");  
printf("Zakres pomiarowy: %g A\n", LE3P.Zp);  
printf("Liczba dzialek podzialki: %d\n", LE3P.d);  
printf("Klasa dokladnosci: %g\n", LE3P.k);  
printf("-----\n");  
  
printf("Bezwzglyedny maksymalny blad pomiaru:\n");  
p = 0.2;  
Dpm = LE3P.Zp*(LE3P.k/100+p/LE3P.d);  
printf("* dla p = %g, Dpm = %g A\n", p, Dpm);  
  
p = 0.5;  
Dpm = LE3P.Zp*(LE3P.k/100+p/LE3P.d);  
printf("* dla p = %g, Dpm = %g A\n", p, Dpm);  
  
return 0;  
}
```

Struktury - przykład (miernik)

```
printf("Amperomi  
printf("Zakres p  
printf("Liczba d  
printf("Klasa do  
printf("-----  
printf("Bezwzgle  
p = 0.2;  
Dpm = LE3P.Zp*(L  
printf("* dla p = %g, Dpm = %g A\n", p, Dpm);  
  
p = 0.5;  
Dpm = LE3P.Zp*(LE3P.k/100+p/LE3P.d);  
printf("* dla p = %g, Dpm = %g A\n", p, Dpm);  
return 0;  
}
```

```
Amperomierz analogowy LE-3P  
Zakres pomiarowy: 12 A  
Liczba dzialek podzialki: 75  
Klasa dokladnosci: 0.5  
-----  
Bezwzglezny maksymalny blad pomiaru:  
* dla p = 0.2, Dpm = 0.092 A  
* dla p = 0.5, Dpm = 0.14 A
```

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};
    ...
}
```

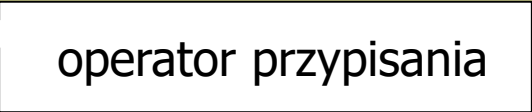

Struktury a operator przypisania (=)

- 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

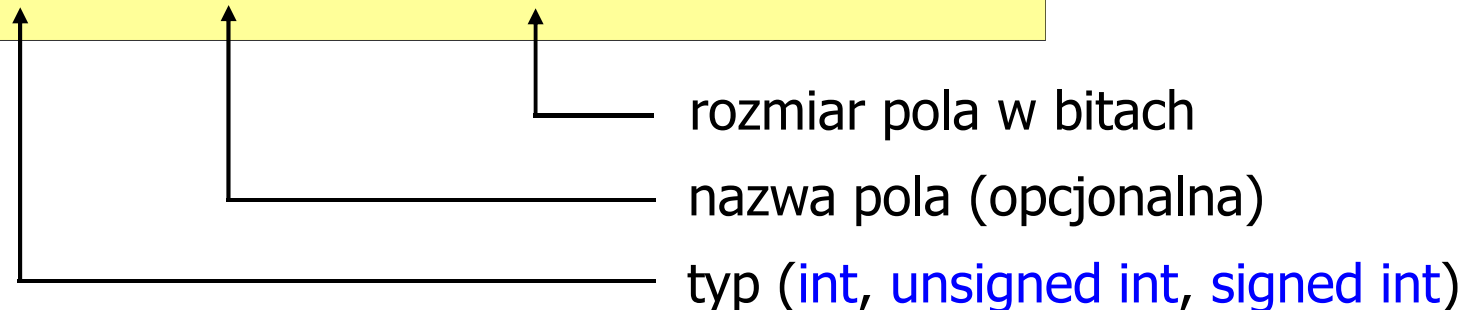
nr		
A	x	y
B	x	y
C	x	y

```
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;
```



- 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()**

Pola bitowe - przykład

```
struct Flags_8086
{
    unsigned int CF : 1;    /* Carry Flag */
    unsigned int   : 1;
    unsigned int PF : 1;    /* Parity Flag */
    unsigned int   : 1;
    unsigned int AF : 1;    /* Auxiliary - Carry Flag */
    unsigned int   : 1;
    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;
};
```

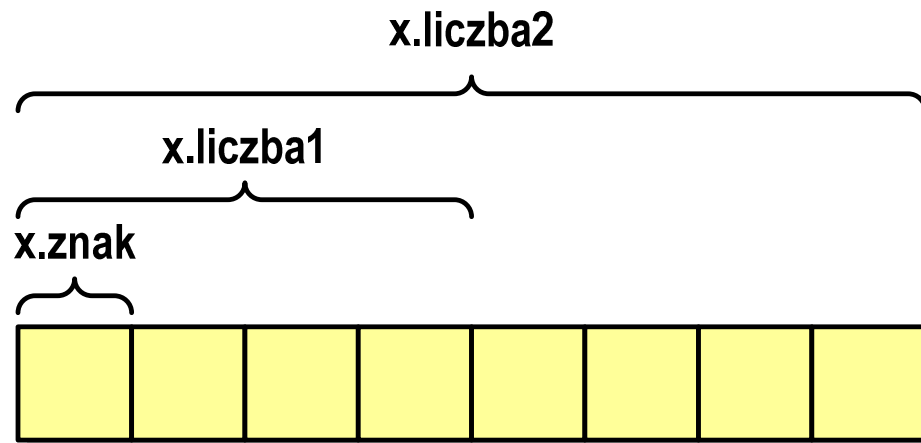
- Do przechowywania wartości w unii należy zadeklarować zmienną

```
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



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

- 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'};
```

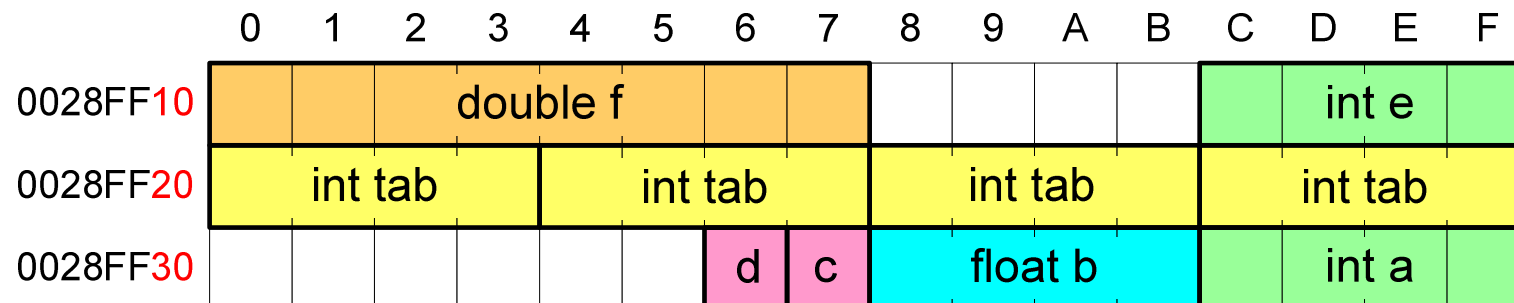
- Unie tego samego typu można sobie przypisywać

Co to jest wskaźnik?

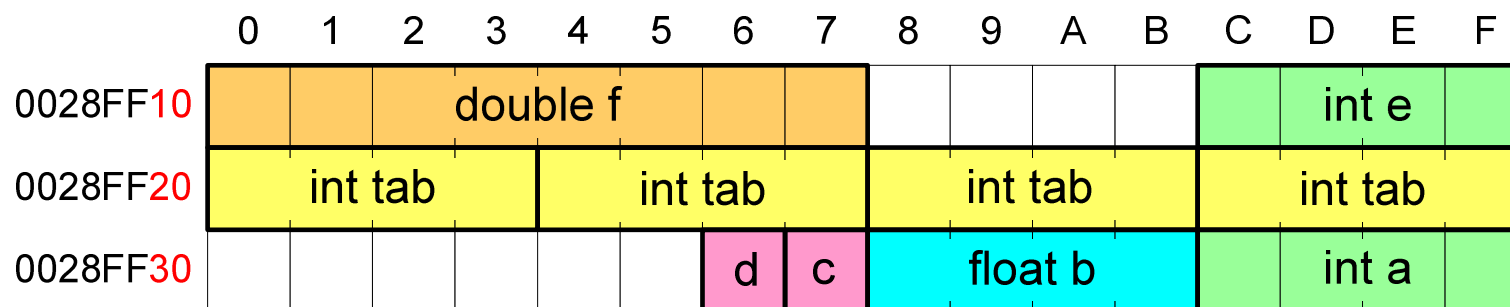
- **Wskaźnik** - zmienna mogąca zawierać adres obszaru pamięci
- najczęściej adres innej zmiennej (obiektu)

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

- Zmienne przechowywane są w pamięci komputera



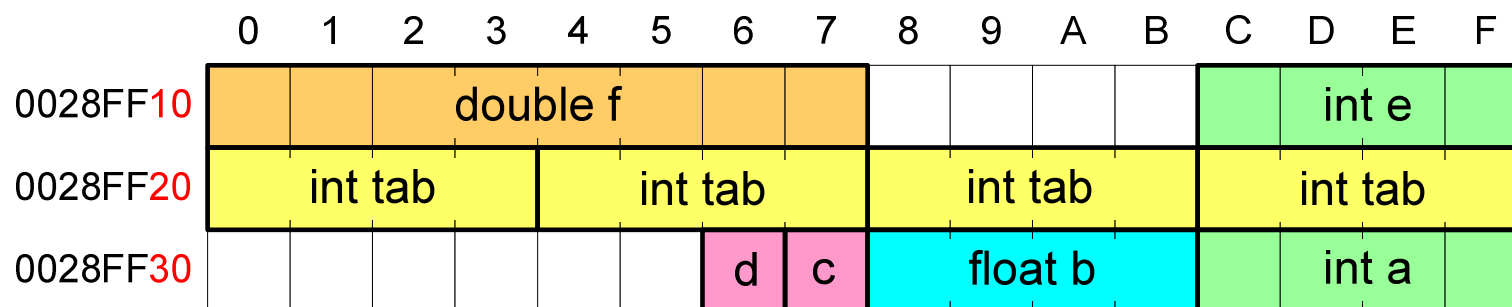
Co to jest wskaźnik?



- 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;
```

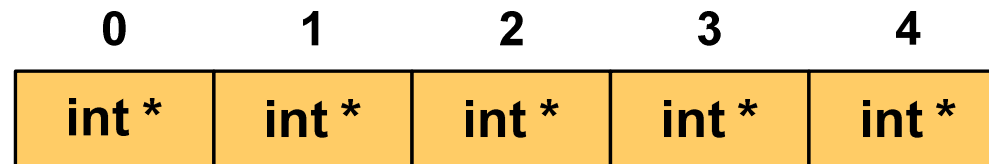
- Można konstruować wskaźniki do danych dowolnego typu łącznie z typami **wskaźnik do wskaźnika do...**

```
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:

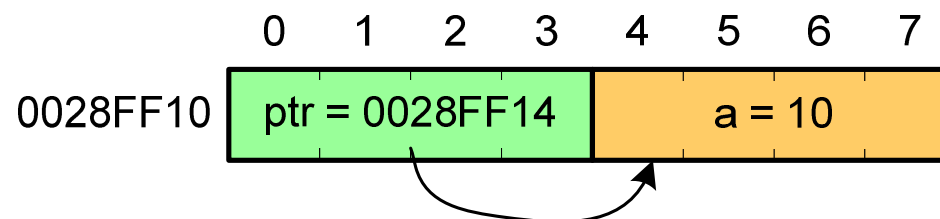
```
int *p1, *p2;  
int* p3, p4;
```

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

Przypisywanie wartości wskaźnikom

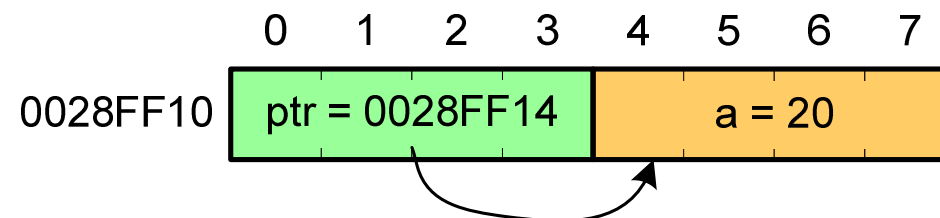
- Wskaźnikom można przypisywać adresy zmiennych
- Adresy takie tworzy się za pomocą operatora pobierania adresu **&**

```
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;
```

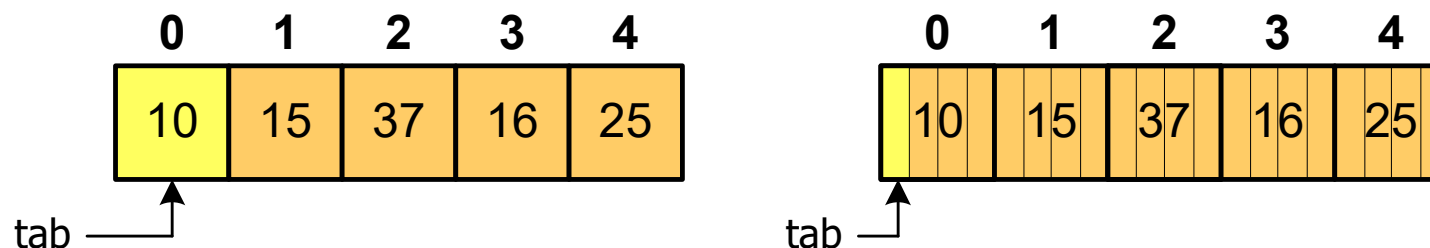
- Zamiast wartości **0** można stosować makrodefinicję preprocesora **NULL**, która podczas kompilacji programu zamieniana jest na **0**

```
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};
```

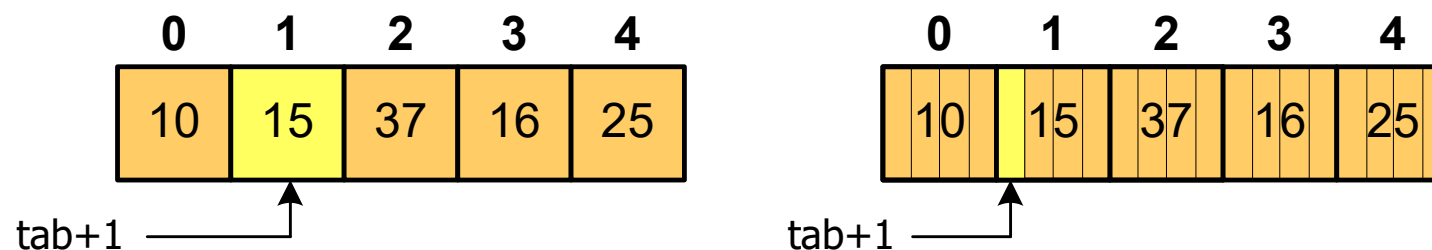


- Zastosowanie operatora `*` przed nazwą tablicy pozwala „dostać się” do zawartości elementu o indeksie 0

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

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, wyłuskania): *

```
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 liczbę nr %d: ", i+1);
    scanf("%d", &x);
    tab[i] = x;
}

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

printf("Średnia %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 liczbę 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;  
}
```

```
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**

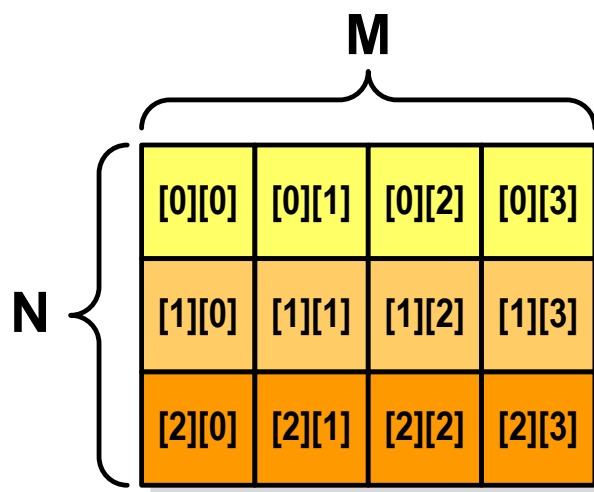
```
for (i=0; i<n; i++)    /* wczytanie liczb */
{
    printf("Podaj liczbę nr %d: ", i+1);
    scanf("%d", &tab[i]);
}
```

- 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

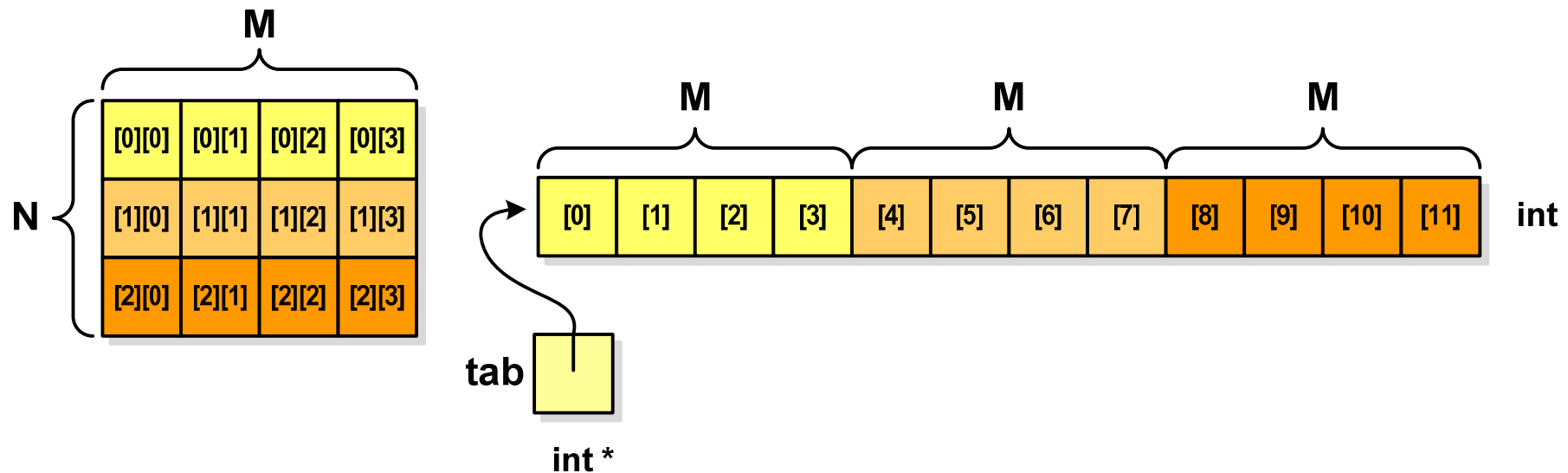
- Funkcje `calloc()` i `malloc()` umożliwiają bezpośrednio przydział pamięci tylko na wektor elementów
- Dynamiczny przydział pamięci na macierz wymaga zastosowania specjalnych metod
- Przydzielamy pamięć na macierz zawierającą **N-wierszy** i **M-kolumn**



Dynamiczny przydział pamięci na macierz (1)

- Wektor $N \times M$ -elementowy
- Przydział pamięci:

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



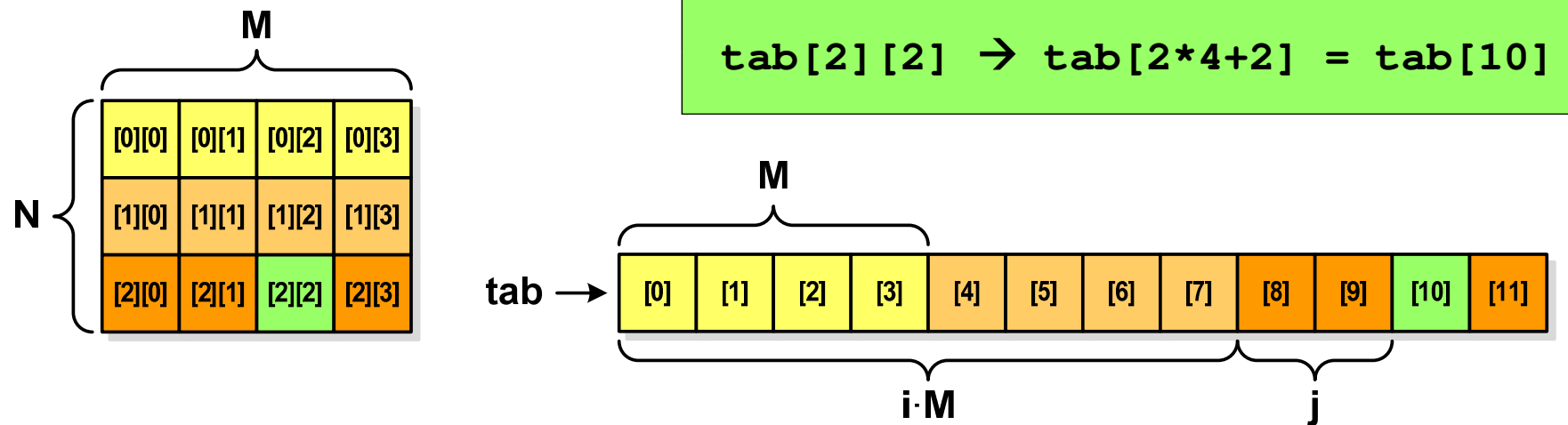
Dynamiczny przydział pamięci na macierz (1)

- Odwołanie do elementów macierzy:

`tab[i*M+j]`

lub

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



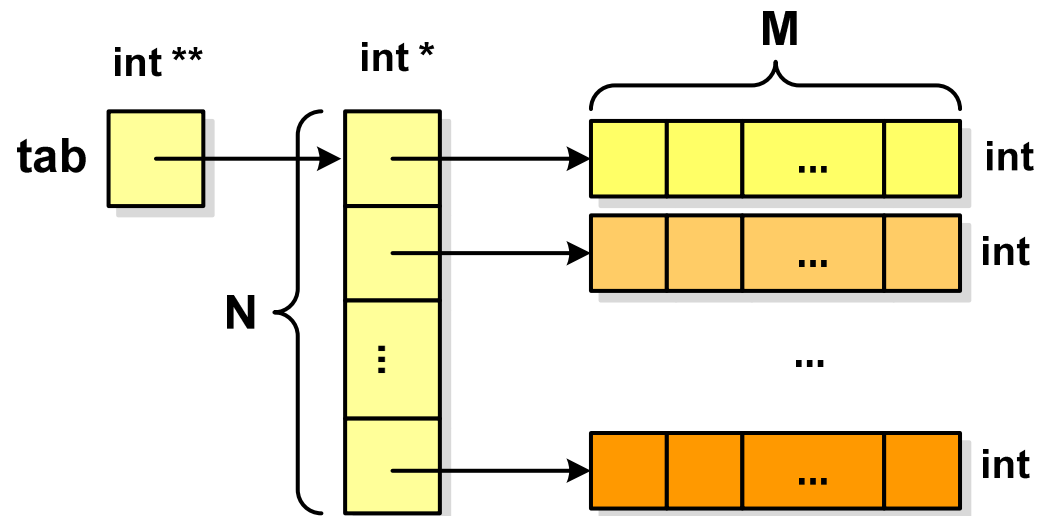
- Zwolnienie pamięci:

`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));
```

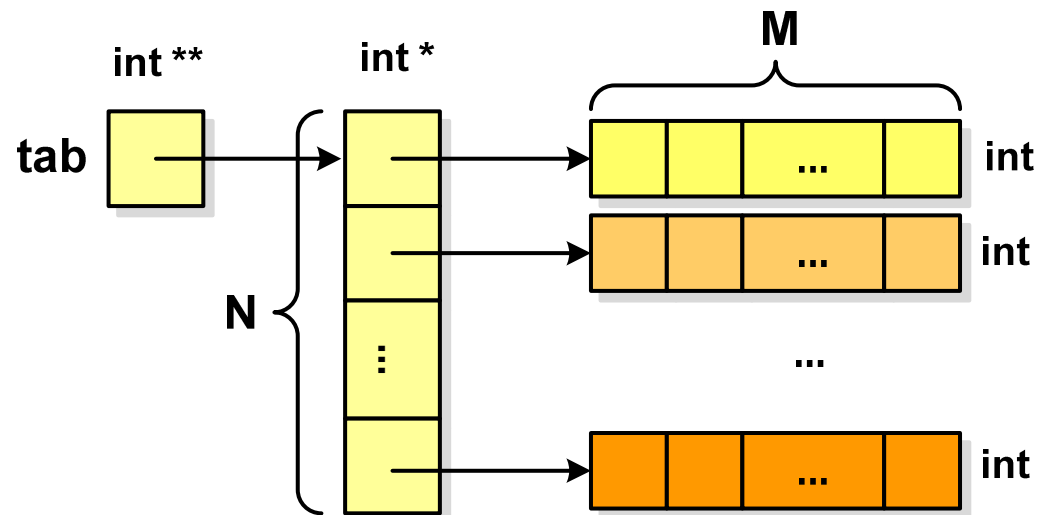


Dynamiczny przydział pamięci na macierz (2)

- Odwołania do elementów macierzy:
- Zwolnienie pamięci:

`tab[i][j]`

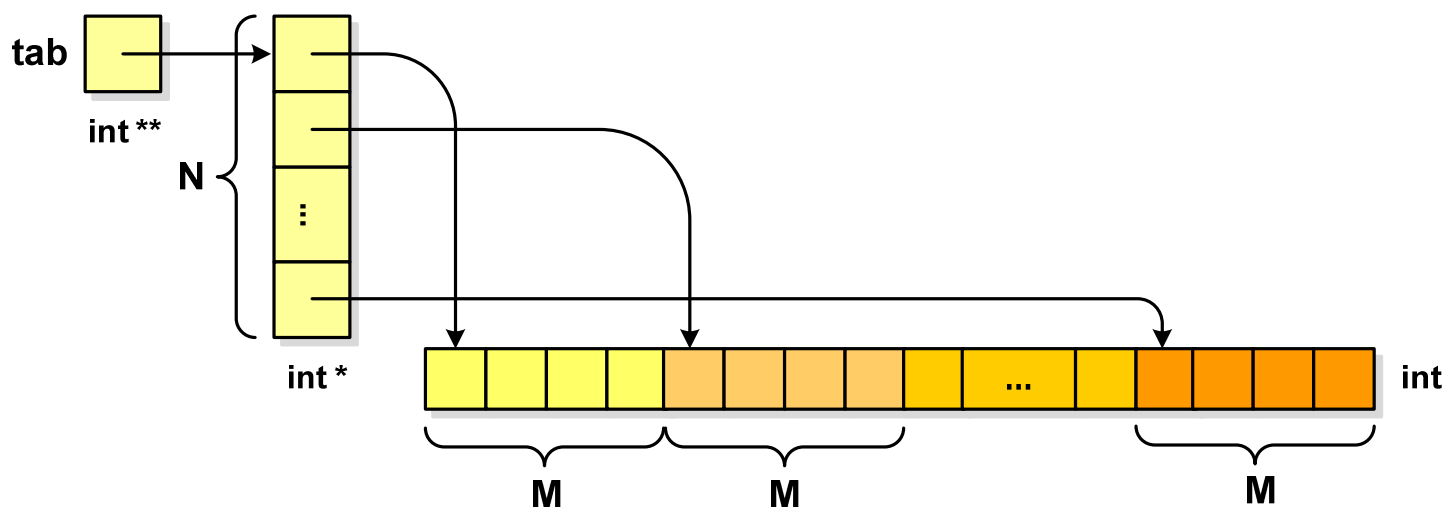
```
for (i=0; i<N; i++)  
    free(tab[i]);  
free(tab);
```



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;
```

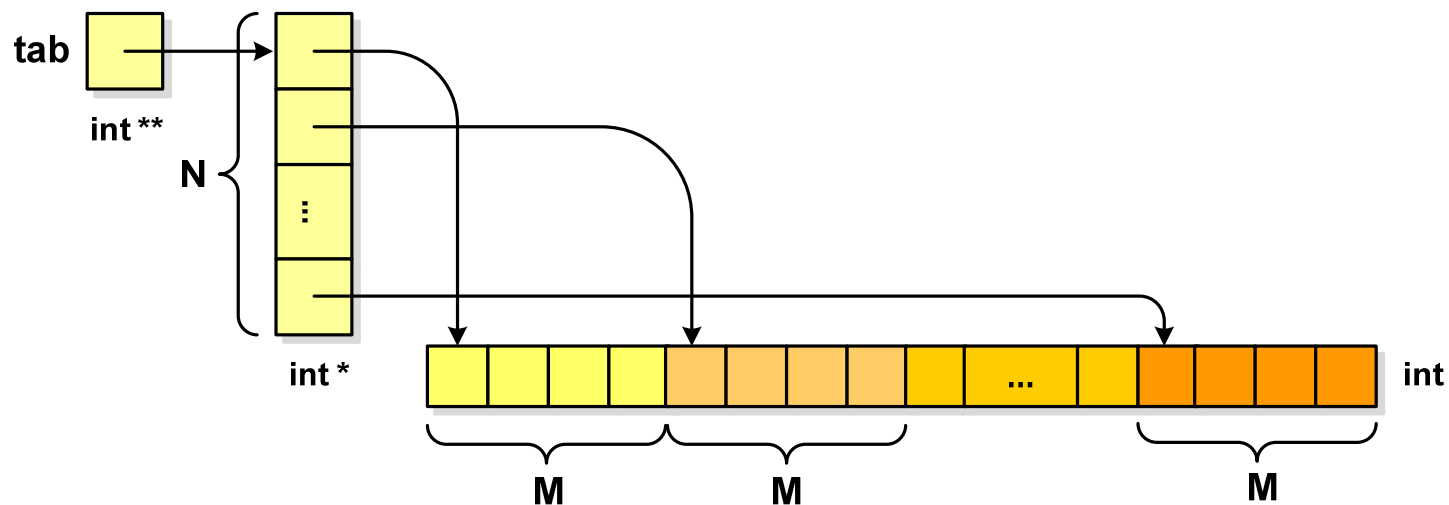


Dynamiczny przydział pamięci na macierz (3)

- Odwołania do elementów macierzy:
- Zwolnienie pamięci:

`tab[i][j]`

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



Koniec wykładu nr 2

Dziękuję za uwagę!