Informatyka 1 (ES1F1002)

Politechnika Białostocka - Wydział Elektryczny

Elektrotechnika, semestr II, studia stacjonarne I stopnia

Rok akademicki 2022/2023

Wykład nr 10 (12.12.2022)

dr inż. Jarosław Forenc

Plan wykładu nr 10

- Pola bitowe, unie
- Wskaźniki
  - deklaracja, przypisanie wartości
  - związek z tablicami, operacje na wskaźnikach
- Dynamiczny przydział pamięci
  - funkcje calloc, malloc, free
  - przydział pamięci na strukturę, wektor i macierz
- Dynamiczne struktury danych
  - stos, kolejka, lista, drzewo

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   : 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;
};
```

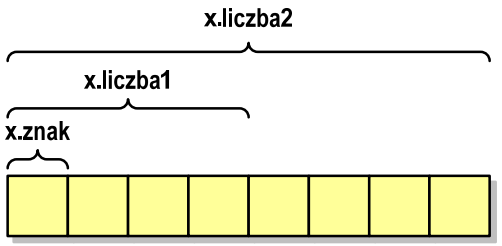
- Do przechowywania wartości w unii należy zadeklarować zmienną

```
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'};
```

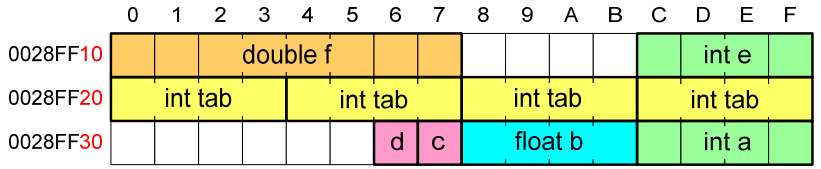
- Unie tego samego typu można sobie przypisywać

Co to jest wskaźnik?

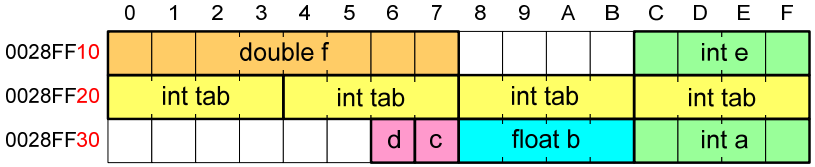
- Wskaźnik - zmienna mogącą zawierać adres obszaru pamięci - najczęściej adres innej zmiennej (obiektu)

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

- Zmienne przechowywane są w pamięci komputera



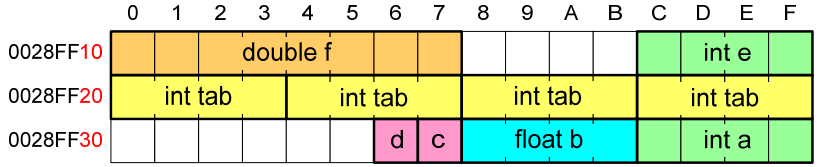
Co to jest wskaźnik?



- 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;
```

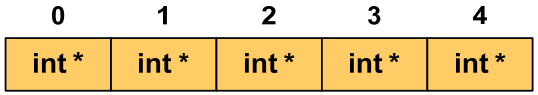
- Można konstruować wskaźniki do danych dowolnego typu łącznie z typami wskaźnik do wskaźnika do...

```
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:

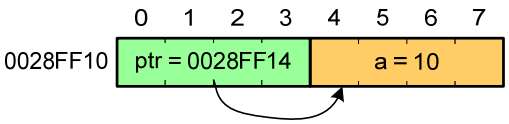
```
int *p1, *p2;
int* p3, p4;
```

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

Przypisywanie wartości wskaźnikom

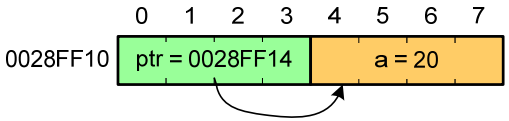
- Wskaźnikom można przypisywać adresy zmiennych
- Adresy takie tworzy się za pomocą operatora pobierania adresu &

```
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

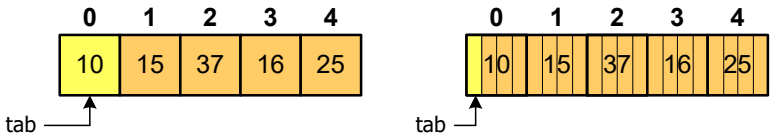
```
#include <stdio.h>

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};
```

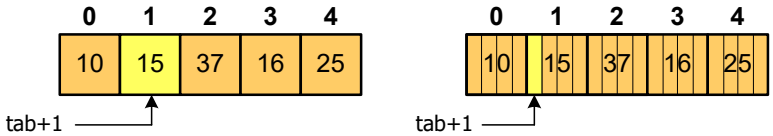


- Zastosowanie operatora * przed nazwą tablicy pozwala „dostać się" do zawartości elementu o indeksie 0

*tab jest równoważne tab[0]

Wskaźniki a tablice

- Dodanie 1 do adresu tablicy przenosi nas do elementu tablicy o indeksie 1 (przesunięcie o 4 bajty, gdyż int zajmuje 4 bajty)



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

x = *tab+1; jest równoważne x = tab[0]+1;

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 wskaznika ptr: %p\n",&ptr);
```

```
Adres zmiennej x:     002CF920
Adres wskaznika 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);
```


Przykład: przydział pamięci na jedną zmienną

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

int main(void)
{
   float *wsk;

   wsk = (float *) calloc(1,sizeof(float));
   if (wsk == NULL)
   {
      printf("Blad przydzialu pamieci\n");
      return 0;
   }

   *wsk = 123.45f;
   printf("wartosc = %g\n",*wsk);

   free(wsk);
   return 0;
}
```

```
wartosc = 123.45
```

Przykład: przydział pamięci na strukturę

```
10,20 - 30,40
```

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

struct punkt
{
   int x, y;
};

int main(void)
{
   struct punkt p, *wsk_p;

   wsk_p = (struct punkt*) malloc(sizeof(struct punkt));

   p.x = 10; p.y = 20;
   wsk_p->x = 30; wsk_p->y = 40;
   printf("%d,%d - %d,%d\n",p.x,p.y,wsk_p->x,wsk_p->y);

   free(wsk_p);
   return 0;
}
```

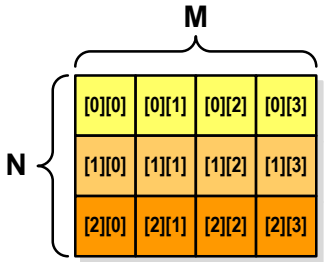

Przykład: przydział pamięci na wektor

```
tab[0] = 0
tab[1] = 1
tab[2] = 4
tab[3] = 9
tab[4] = 16
tab[5] = 25
tab[6] = 36
tab[7] = 49
tab[8] = 64
tab[9] = 81
```

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

int main(void)
{
   int *tab, n = 10;

   tab = (int *) calloc(n,sizeof(int));

   for (int i=0; i<n; i++)
   {
       tab[i] = i*i;
       printf("tab[%d] = %d\n",i,tab[i]);
   }

   free(tab);

   return 0;
}
```


Dynamiczny przydział pamięci na macierz

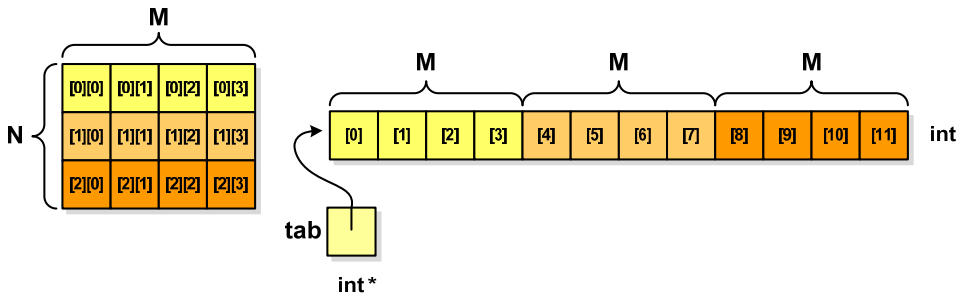
- Funkcje calloc() i malloc() umożliwiają bezpośrednio przydział pamięci tylko na wektor elementów
- Dynamiczny przydział pamięci na macierz wymaga zastosowania specjalnych metod
- Przydzielamy pamięć na macierz zawierającą N-wierszy i M-kolumn



Dynamiczny przydział pamięci na macierz (1)

- Wektor N×M-elementowy
- Przydział pamięci:

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



Dynamiczny przydział pamięci na macierz (1)

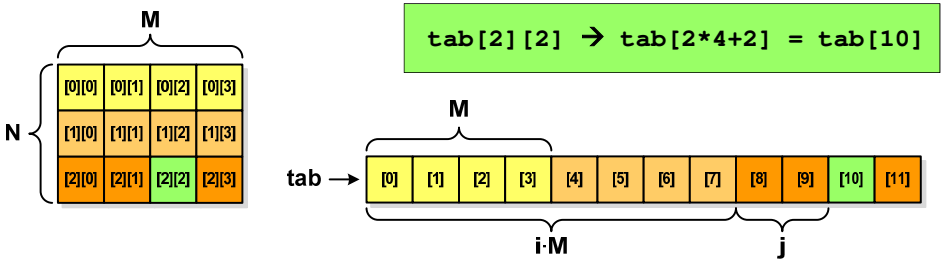
- Odwołanie do elementów macierzy:

```
tab[i*M+j]
```

lub

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

```
tab[2][2] → tab[2*4+2] = tab[10]
```



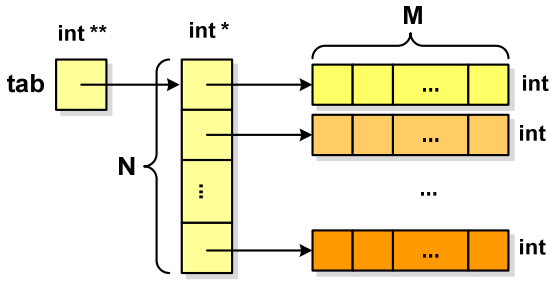
- Zwolnienie pamięci:

```
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));
```



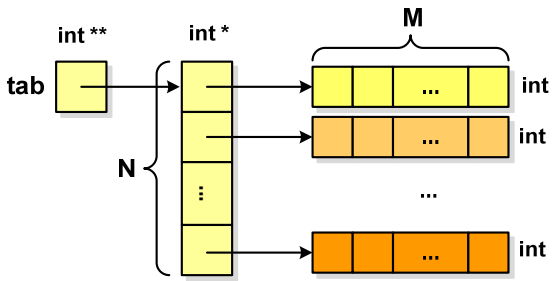
Dynamiczny przydział pamięci na macierz (2)

- Odwołania do elementów macierzy:

```
tab[i][j]
```

- Zwolnienie pamięci:

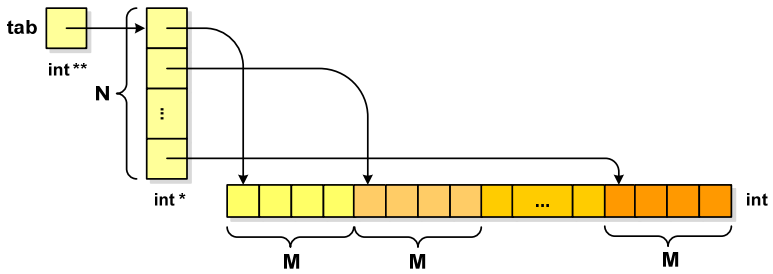
```
for (i=0; i<N; i++)
    free(tab[i]);
free(tab);
```



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;
```



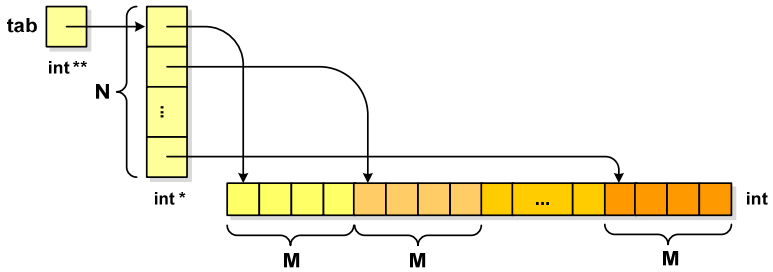
Dynamiczny przydział pamięci na macierz (3)

- Odwołania do elementów macierzy:

```
tab[i][j]
```

- Zwolnienie pamięci:

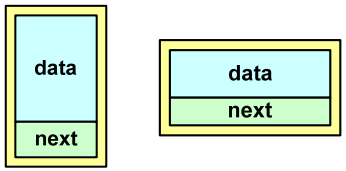
```
free(tab[0]);
free(tab);
```



Dynamiczne struktury danych

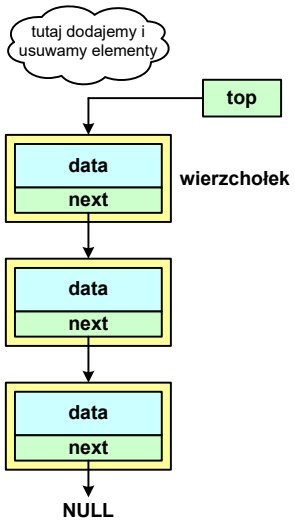
- Dynamiczne struktury danych - struktury danych, którym pamięć jest przydzielana i zwalniana w trakcie wykonywania programu
  - stos, kolejka
  - lista (jednokierunkowa, dwukierunkowa, cykliczna)
  - drzewo
- Elementy w dynamicznych strukturach danych są strukturami składającymi się z „użytecznych” 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 pusty (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:

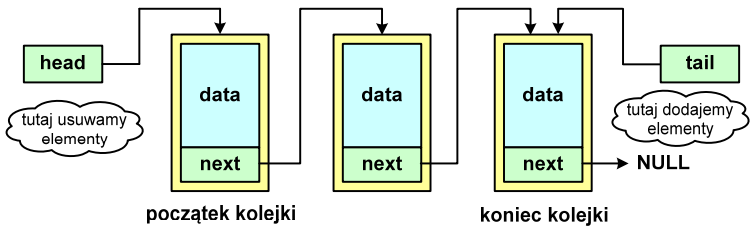
```
(2 + 1) * 3 – 4 * (7 + 4)
```

  - ma w odwrotnej notacji polskiej postać:

```
2 1 + 3 * 4 7 4 + * –
```


Kolejka

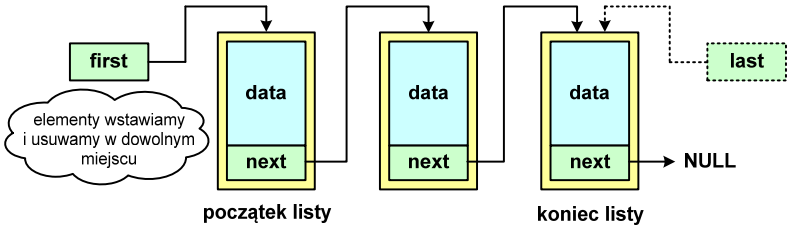
- Kolejka - składa się z liniowo uporządkowanych elementów
- Elementy dołączane są tylko na końcu kolejki (wskaźnik tail)
- Elementy usuwane są tylko z początku kolejki (wskaźnik head)



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

Lista jednokierunkowa

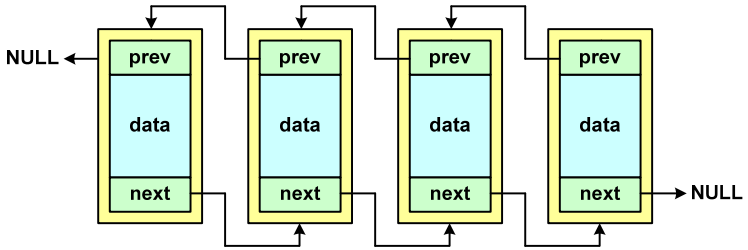
- Organizacja listy jednokierunkowej podobna jest do organizacji stosu i kolejki
- Dla każdego składnika (poza ostatnim) jest określony następny składnik (lub poprzedni - zależnie od implementacji)



- 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

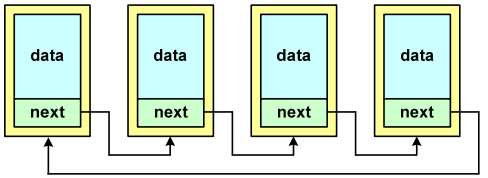
- Każdy węzeł posiada adres następnika, jak i poprzednika
- W strukturze tego typu wygodne jest przechodzenie pomiędzy elementami w obu kierunkach (od początku do końca i odwrotnie)



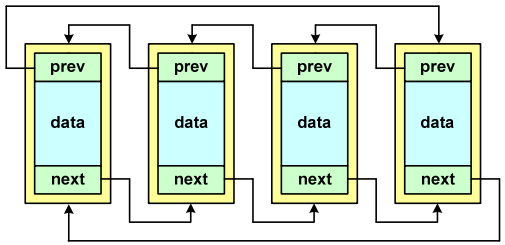
Lista cykliczna

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

Jednokierunkowa:

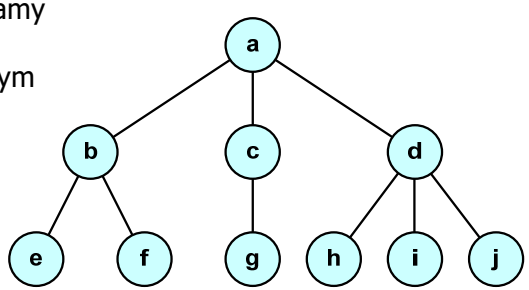


Dwukierunkowa:



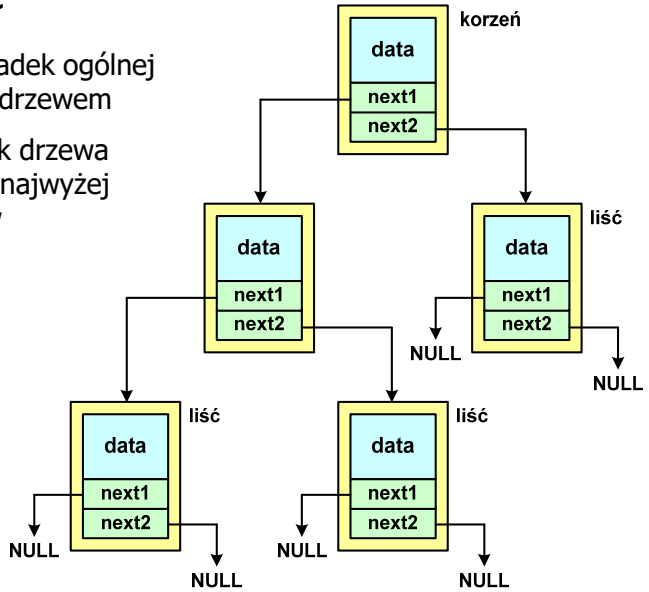
Drzewo

- 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



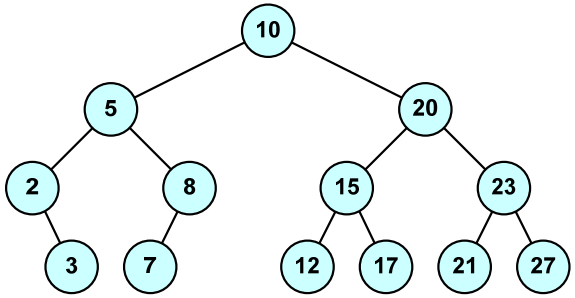
Drzewo binarne

- 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 10

Dziękuję za uwagę!