

Wydział Elektryczny
Katedra Elektrotechniki, Energoelektroniki i Elektroenergetyki

Materiały do wykładu z przedmiotu:
Informatyka
Kod: EDS1B1007

WYKŁAD NR 5

Opracował: **dr inż. Jarosław Forenc**
Białystok 2022

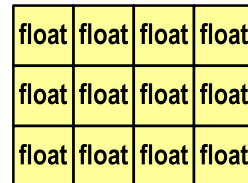
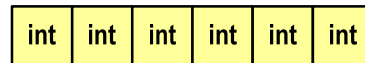
Materiały zostały opracowane w ramach projektu „PB2020 - Zintegrowany Program Rozwoju Politechniki Białostockiej” realizowanego w ramach Działania 3.5 Programu Operacyjnego Wiedza, Edukacja, Rozwój 2014-2020 współfinansowanego ze środków Europejskiego Funduszu Społecznego.

Plan wykładu nr 5

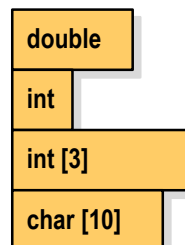
- Język C
 - struktury
 - pola bitowe
 - unie
 - wskaźniki
 - dynamiczny przydział pamięci
- Architektura von Neumanna i architektura harwardzka
- Struktura i funkcjonowanie komputera
 - procesor, rozkazy, przerwania, magistrala
 - pamięć komputerowa, pamięć podręczna

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;
};
```

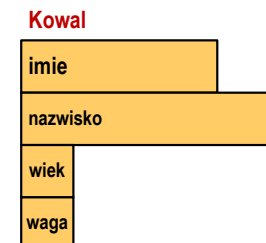
- Deklaracja struktury nie tworzy obiektu (nie przydziela pamięci na pola struktury)
- Zapisanie danych do struktury wymaga zdefiniowania **zmiennej strukturalnej**

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**

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;
}
    
```

day1

day	1
month	9
year	2018

day2

day	19
month	11
year	2018

```

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

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;
    
```

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   : 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()

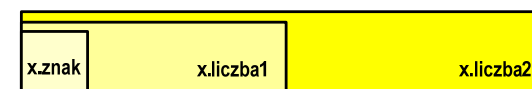
Unie

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

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

```
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

- 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

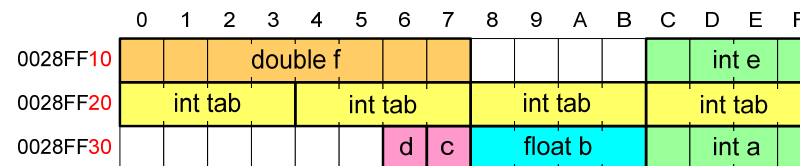
```
union zbior x = {'a'};
union zbior y;
y = x;
```

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?

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0028FF10																
0028FF20																
0028FF30																

- 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);
```

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;
```

Co to jest wskaźnik?

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0028FF10																
0028FF20																
0028FF30																

- 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);
```

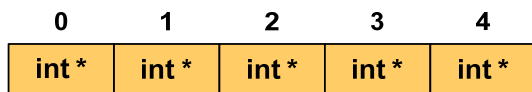
Deklaracja wskaźnika

- 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 deklorować tablice wskaźników - zmienna `tab_ptr` jest tablicą zawierającą 5 wskaźników do typu `int`

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



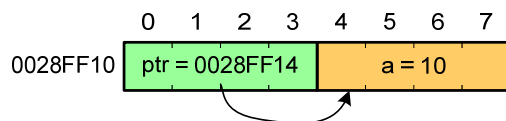
- Natomiast zmienna `ptr_tab` jest wskaźnikiem do 5-elementowej tablicy liczb `int`

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

## Przypisywanie wartości wskaźnikom

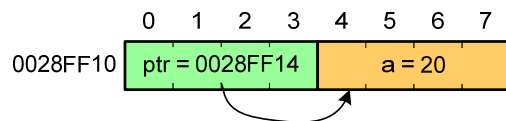
- Wskaźnikom można przypisywać adresy zmiennych
- Adresy takie tworzy się za pomocą operatora pobierania adresu `&`

```
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;
```

## 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`

## 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;
```

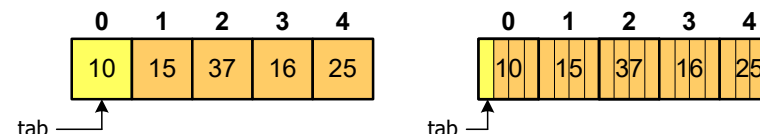
```
}
```

```
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};
```

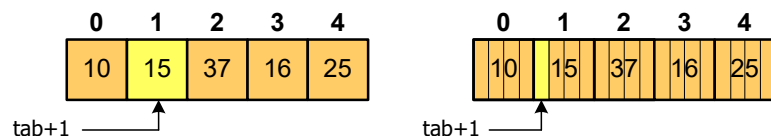


- Zastosowanie operatora \* przed nazwą tablicy pozwala „dostać się” do zawartości elementu o indeksie 0

`*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;`



## 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
- 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

**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

**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

**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("Bład przydziału pamięci\n");
 return 0;
 }

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

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

wartosc = 123.45

## Przykład: przydział pamięci na strukturę

```
#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;
}
```

10, 20 - 30, 40

## Przykład: przydział pamięci na wektor

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

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

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

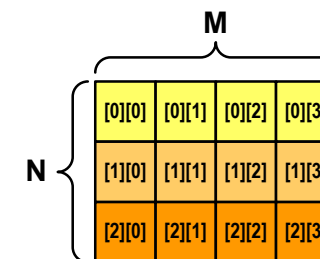
 for (int i=0; i<n; i++)
 {
 tab[i] = i*i;
 printf("tab[%d] = %d\n", i, tab[i]);
 }

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

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

## Dynamiczny przydział pamięci na macierz

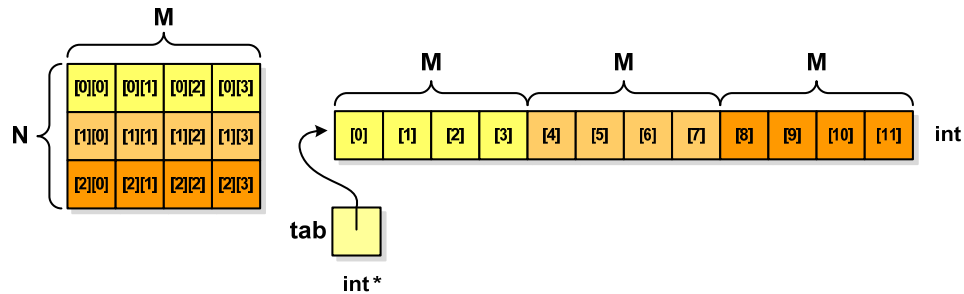
- Funkcje `calloc()` i `malloc()` umożliwiają bezpośrednio przydział pamięci tylko na wektor elementów
- Dynamiczny przydział pamięci na macierz wymaga zastosowania specjalnych metod
- Przydzielamy pamięć na macierz zawierającą **N-wierszy** i **M-kolumn**



## Dynamiczny przydział pamięci na macierz (1)

- Wektor N×M-elementowy
- Przydział pamięci:

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



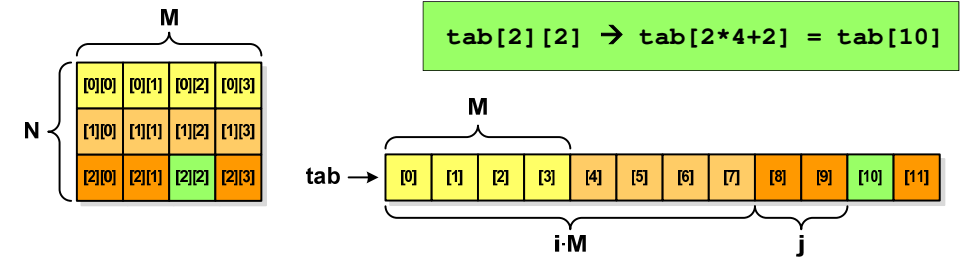
## Dynamiczny przydział pamięci na macierz (1)

- Odwołanie do elementów macierzy:

`tab[i*M+j]`

lub

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



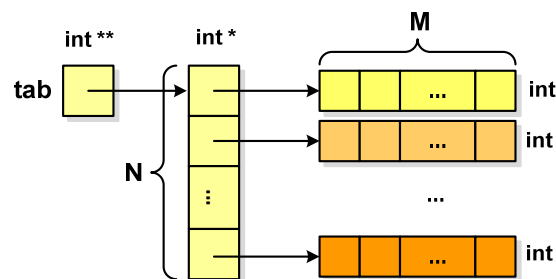
- Zwolnienie pamięci:

```
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));
```

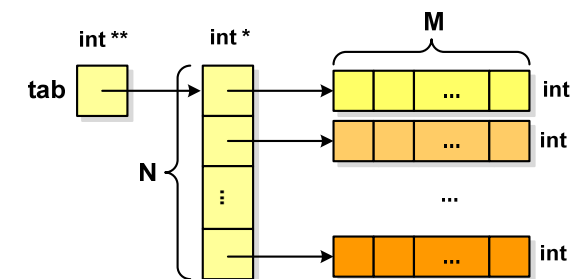


## Dynamiczny przydział pamięci na macierz (2)

- Odwołania do elementów macierzy:
- Zwolnienie pamięci:

`tab[i][j]`

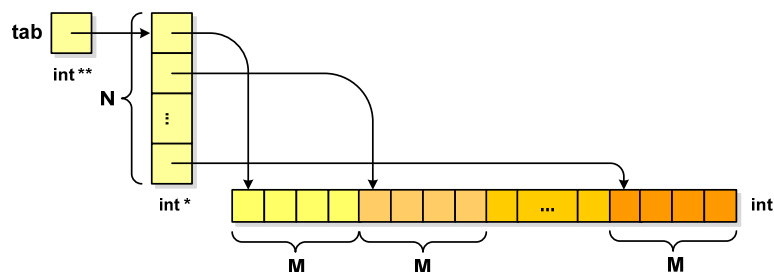
```
for (i=0; i<N; i++)
 free(tab[i]);
free(tab);
```



## Dynamiczny przydział pamięci na macierz (3)

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

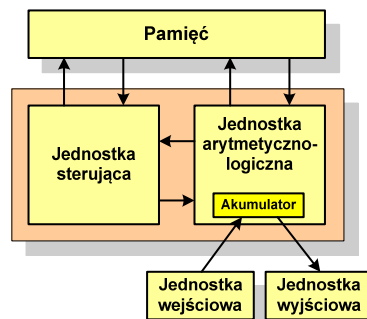
```
int **tab = (int **) malloc(N*sizeof(int *));
tab[0] = (int *) malloc(N*M*sizeof(int));
for (i=1; i<N; i++)
 tab[i] = tab[0]+i*M;
```



## Architektura von Neumanna

- Rodzaj architektury komputera, opisanej w 1945 roku przez matematyka Johna von Neumanna
- Inne nazwy: architektura z Princeton, store-program computer (koncepcja przechowywanego programu)
- Zakłada podział komputera na kilka części:

- jednostka sterująca (CU - Control Unit)
- jednostka arytmetyczno-logiczna (ALU - Arithmetic Logic Unit)
- pamięć główna (memory)
- urządzenia wejścia-wyjścia (input/output)



## Architektura von Neumanna - podstawowe cechy

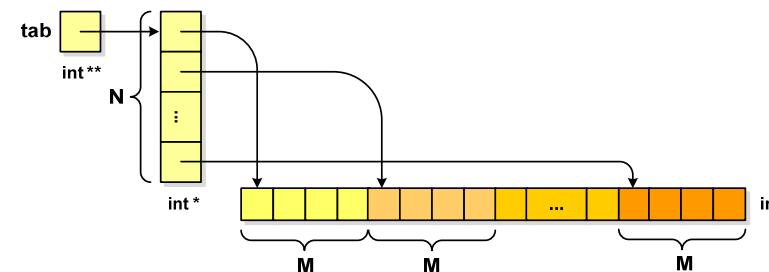
- Informacje przechowywane są w komórkach pamięci (cell) o jednakowym rozmiarze, każda komórka ma numer - adres
- Dane oraz instrukcje programu (rozkazy) zakodowane są za pomocą liczb i przechowywane w tej samej pamięci
- Dane i instrukcje czytane są przy wykorzystaniu tej samej magistrali
- Praca komputera to sekwencyjne odczytywanie instrukcji z pamięci komputera i ich wykonywanie w procesorze
- Wykonanie rozkazu:
  - pobranie z pamięci słowa będącego kodem instrukcji
  - pobranie z pamięci danych
  - wykonanie instrukcji
  - zapisanie wyników do pamięci

## Dynamiczny przydział pamięci na macierz (3)

- Odwołania do elementów macierzy:
- Zwolnienie pamięci:

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

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

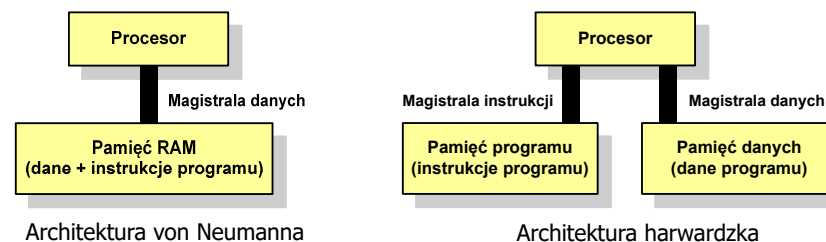


## Architektura harwardzka

- Nazwa architektury pochodzi od komputera **Harward Mark I**:
  - zaprojektowany przez Howarda Aikena
  - pamięć instrukcji - taśma dziurkowana, pamięć danych - elektromechaniczne liczniki
- Architektura komputera, w której **pamięć danych jest oddzielona od pamięci instrukcji**
- Pamięci danych i instrukcji mogą różnić się:
  - technologią wykonania
  - strukturą adresowania
  - długością słowa
- **Procesor może w tym samym czasie czytać instrukcje oraz uzyskiwać dostęp do danych**

## Architektura harwardzka i von Neumanna

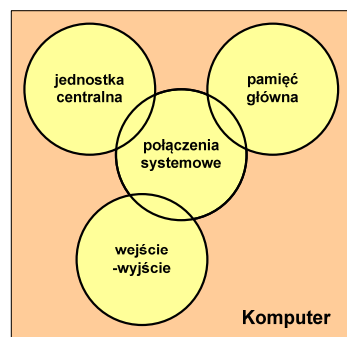
- W architekturze harwardzkiej pamięć instrukcji i pamięć danych:
  - zajmują różne przestrzenie adresowe
  - mają oddzielne szyny (magistrale) do procesora
  - zaimplementowane są w inny sposób



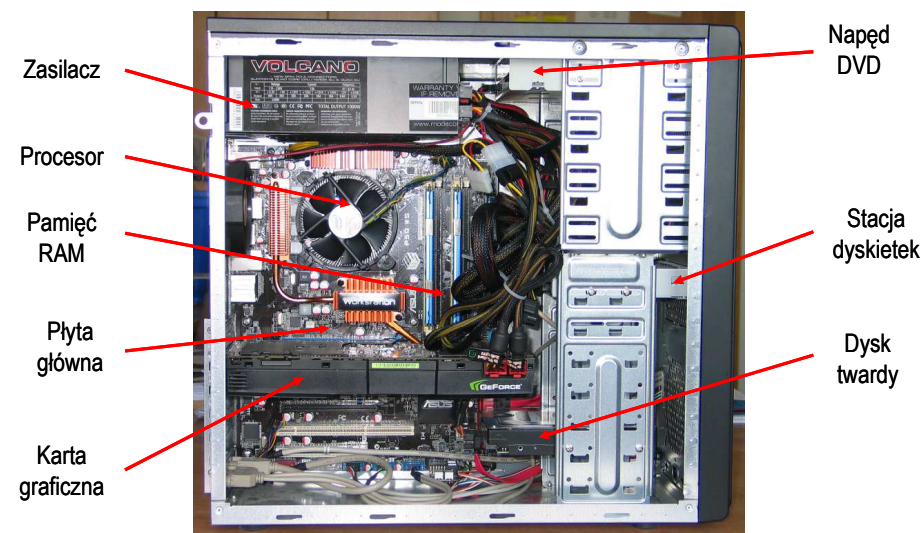
- Zmodyfikowana architektura harwardzka:
  - oddzielone pamięci danych i rozkazów, lecz wykorzystujące wspólną magistralę

## Ogólna struktura systemu komputerowego

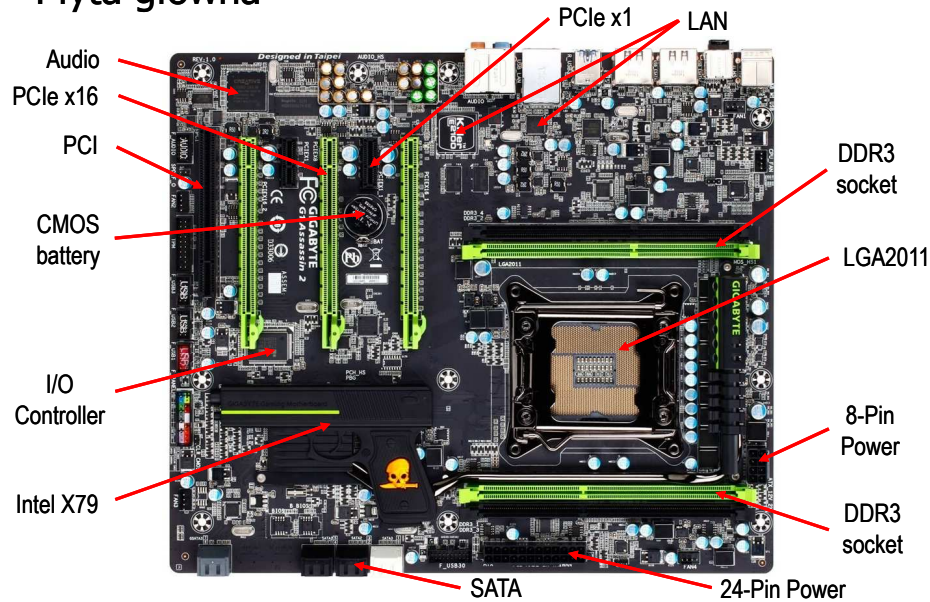
- Komputer tworzą cztery główne składniki:
  - **procesor** (jednostka centralna, CPU) - steruje działaniem komputera i realizuje przetwarzanie danych
  - **pamięć główna** - przechowuje dane
  - **wejście-wyjście** - przenosi dane między komputerem a jego otoczeniem zewnętrznym
  - **połączenia systemu** - mechanizmy zapewniające komunikację między składnikami systemu



## Jednostka centralna



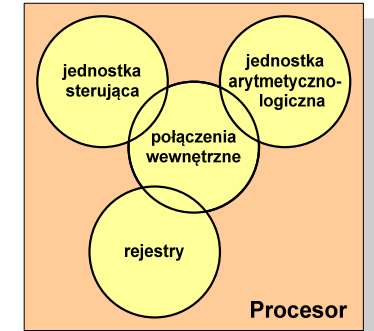
## Płyta główna



## Ogólna struktura procesora

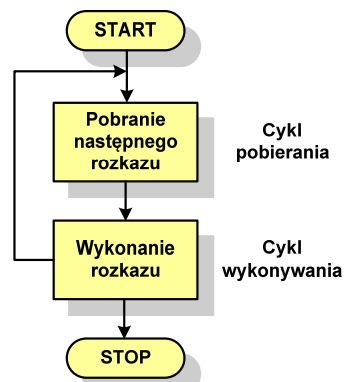
### ■ Główne składniki strukturalne procesora to:

- **jednostka sterująca** - steruje działaniem procesora i pośrednio całego komputera
- **jednostka arytmetyczno-logiczna (ALU)** - realizuje przetwarzanie danych przez komputer
- **rejstry** - realizują wewnętrzne przechowywanie danych w procesorze
- **połączenia procesora** - wszystkie mechanizmy zapewniające komunikację między jednostką sterującą, ALU i rejestrami.



## Działanie komputera

- Podstawowe zadanie komputera to wykonywanie **programu**
- Program składa się z **rozkazów** przechowywanych w pamięci
- Rozkazy są przetwarzane w dwu krokach:

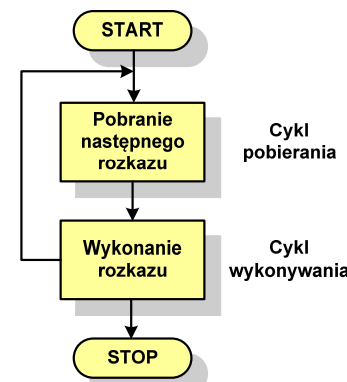


### ■ Cykl pobierania (ang. fetch):

- odczytanie rozkazu z pamięci
- **licznik rozkazów (PC)** lub **wskaźnik instrukcji (IP)** określa, który rozkaz ma być pobrany
- jeśli procesor nie otrzyma innego polecenia, to inkrementuje licznik **PC** po każdym pobraniu rozkazu.

## Działanie komputera

- Podstawowe zadanie komputera to wykonywanie **programu**
- Program składa się z **rozkazów** przechowywanych w pamięci
- Rozkazy są przetwarzane w dwu krokach:

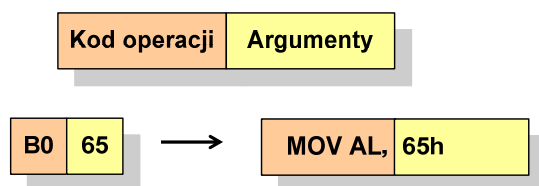


### ■ Cykl wykonywania (ang. execution):

- pobrany rozkaz jest umieszczany w **rejestrze rozkazu (IR)**
- rozkaz określa działania, które ma podjąć procesor
- procesor interpretuje rozkaz i przeprowadza wymagane operacje.

## Działanie komputera

- **Rozkaz:**
  - przechowywany jest w postaci **binarnej**
  - ma określony **format**
  - używa określonego **trybu adresowania**
- **Format** - sposób rozmieszczenia informacji w kodzie rozkazu
- Rozkaz zawiera:
  - **kod operacji** (rodzaj wykonywanej operacji)
  - **argumenty** (lub adresy argumentów) wykonywanych operacji



## Program w assemblerze

```
.model SMALL
.286
.stack 100h
.code
start:
 jmp begin

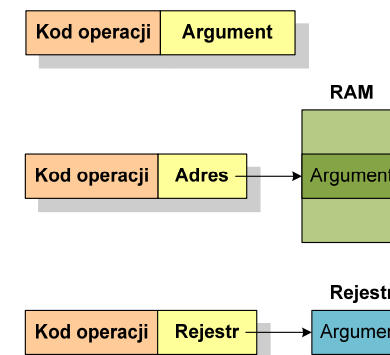
handler:
 pusha
 push ds
 pop ds
 popa
 iret

begin:
 mov ax, 0000h
 mov ds, ax
 mov di, 0070h
 lea ax, handler
```

```
cli
mov [di], ax
mov [di+2], cs
sti
mov ax, 3100h
mov dx, (offset begin - offset handler)
inc dx
int 21h
end
start
```

## Działanie komputera

- **Tryb adresowania** - sposób określania miejsca przechowywania argumentów rozkazu (operandów)
- Przykładowe rodzaje adresowania:
  - **natychmiastowe** - argument znajduje się w kodzie rozkazu
  - **bezpośrednie** - kod rozkazu zawiera adres komórki pamięci, w której znajduje się argument
  - **rejestrowe** - kod rozkazu zawiera oznaczenie rejestru, w którym znajduje się argument

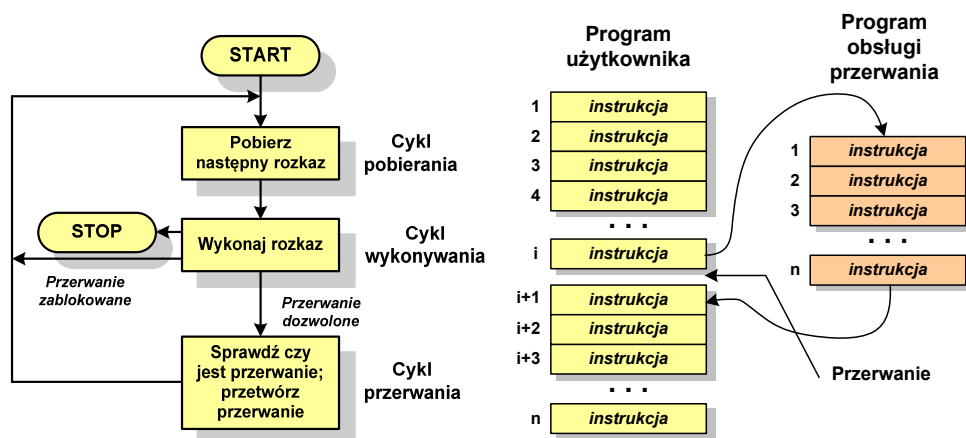


## Działanie komputera - przerwania

- Wykonywanie kolejnych rozkazów przez procesor może zostać przerwane poprzez wystąpienie tzw. **przerwania (interrupt)**
- Przerwanie jest to **sygnał** pochodzący od sprzętu lub oprogramowania informujący procesor o wystąpieniu jakiegoś zdarzenia (np. wciśnięcie klawisza na klawiaturze)
- Bez przerwania procesor musiałby ciągle kontrolować wszystkie urządzenia zewnętrzne, np. klawiatura, port szeregowy
- Każde przerwanie posiada procedurę obsługi przerwania, która jest wykonywana w momencie jego wystąpienia
- Adresy procedur obsługi przerwania zapisane są w tablicy wektorów przerwania

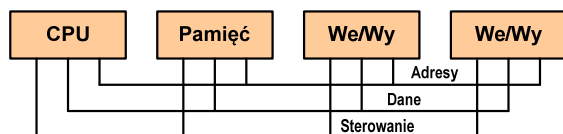
## Działanie komputera - przerwania

- Implementacja przerwania wymaga dodania cyklu przerwania do cyklu rozkazu



## Magistrala

- Najczęściej stosowana struktura połączeń to **magistrala**, składająca się z wielu linii komunikacyjnych, którym przypisane jest określone znaczenie i określona funkcja



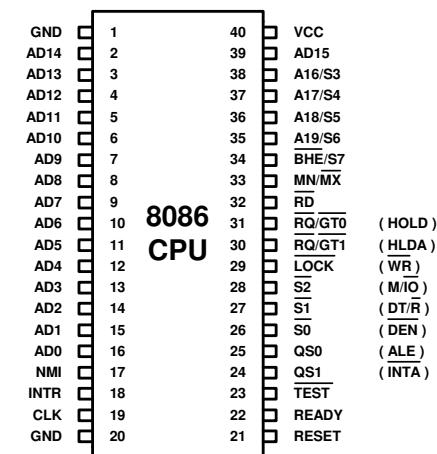
- **linie danych (szyna danych)** - przenoszą dane między modułami systemu, liczba linii określa szerokość szyny danych (8, 16, 32, 64 bity)
- **linie adresowe** - służą do określania źródła i miejsca przeznaczenia danych przesyłanych magistralą; liczba linii adresowych określa maksymalną możliwą pojemność pamięci systemu
- **linie sterowania** - służą do sterowania dostępem do linii danych i linii adresowych

## Rodzaje przerwania

- **Sprzętowe**
  - **zewnętrzne** - sygnały pochodzące z urządzeń zewnętrznych i służące do komunikacji z nimi, np. 08H - zegar, 09h - klawiatura
  - **wewnętrzne** - wywoływane przez procesor w celu zasygnalizowania sytuacji wyjątkowych (faults, traps, aborts)
- **Programowe**
  - instrukcje programu wywołują przerwanie - tym samym wykonywana jest procedura obsługi przerwania
  - służą głównie do komunikacji z systemem operacyjnym (DOS - 21h, Windows - 2h, Linux - 80h)

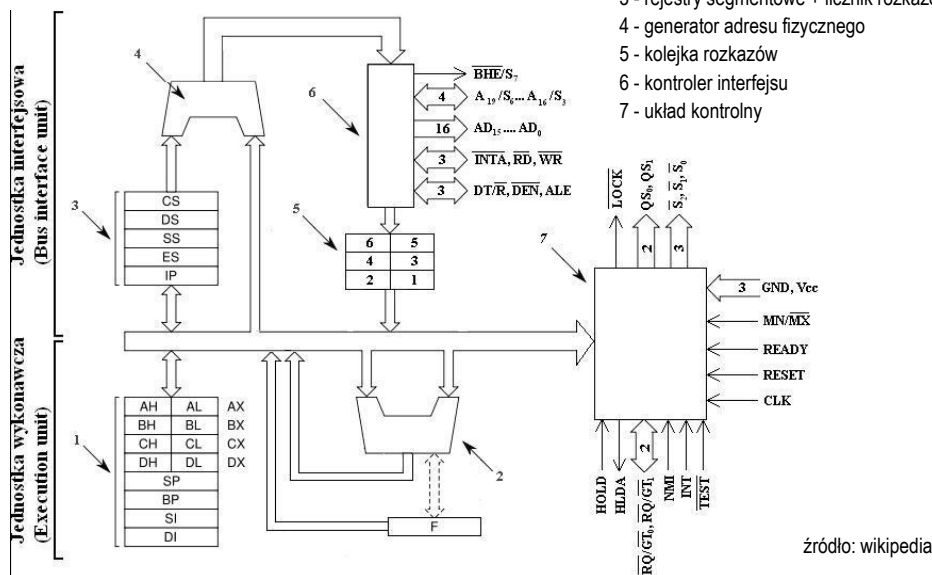
## Intel 8086

- 1978 rok
- Procesor 16-bitowy
- 16-bitowa magistrala danych
- 20-bitowa magistrala adresowa
- Adresowanie do 1 MB pamięci
- Częstotliwość: 10 MHz
- Multipleksowane magistrale: danych i adresowa
- Litografia: 3  $\mu$ m





## Intel 8086



## Półprzewodnikowa pamięć główna

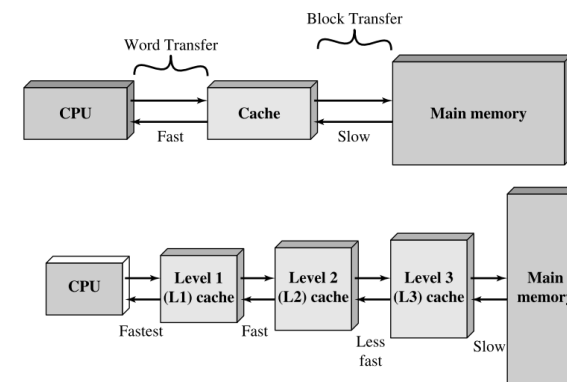
- **RAM** (Random Access Memory) - pamięć o dostępie swobodnym
  - odczyt i zapis następuje za pomocą sygnałów elektrycznych
  - pamięć ulotna - po odłączeniu zasilania dane są tracone
  - **DRAM** - pamięć dynamiczna:
    - przechowuje dane podobnie jak kondensator ładunek elektryczny
    - wymaga operacji odświeżania
    - jest mniejsza, gęściej upakowana i tańsza niż pamięć statyczna
    - stosowana jest do budowy głównej pamięci operacyjnej komputera
  - **SRAM** - pamięć statyczna:
    - przechowuje dane za pomocą przerzutnikowych konfiguracji bramek logicznych
    - nie wymaga operacji odświeżania
    - jest szybsza i droższa od pamięci dynamicznej
    - stosowana jest do budowy pamięci podręcznej

## Półprzewodnikowa pamięć główna

- **ROM** (ang. Read-Only Memory) - pamięć stała
  - pamięć o dostępie swobodnym przeznaczona tylko do odczytu
  - dane są zapisywane podczas procesu wytwarzania, pamięć nieulotna
- **PROM** (ang. Programmable ROM) - programowalna pamięć ROM
  - pamięć nieulotna, może być zapisywana tylko jeden raz
  - zapis jest realizowany elektrycznie po wyprodukowaniu
- **EPROM** - pamięć wielokrotnie programowalna, kasowanie następuje przez naświetlanie promieniami UV
- **EEPROM** - pamięć kasowana i programowana na drodze elektrycznej
- **Flash** - rozwinięcie koncepcji pamięci EEPROM, możliwe kasowanie i programowanie bez wymontowywania pamięci z urządzenia

## Pamięć podręczna (cache)

- Dodatkowa, szybka pamięć (SRAM) umieszczana pomiędzy procesorem a pamięcią główną
- Zastosowanie pamięci podręcznej ma na celu przyspieszenie dostępu procesora do pamięci głównej



Koniec wykładu nr 5

Dziękuję za uwagę!