



Wydział Elektryczny
Katedra Elektrotechniki Teoretycznej i Metrologii

Materiały do wykładu z przedmiotu:
Informatyka
Kod: EDS1A1 007

WYKŁAD NR 4

Opracował: dr inż. Jarosław Forenc
Białystok 2018

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

Plan wykładu nr 4

- Struktury, pola bitowe, unie
- Wskaźniki
- Dynamiczny przydział pamięci
- Funkcje w języku C
- Prototypy funkcji, typy funkcji
- Przekazywanie argumentów do funkcji przez wartość i wskaźnik

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

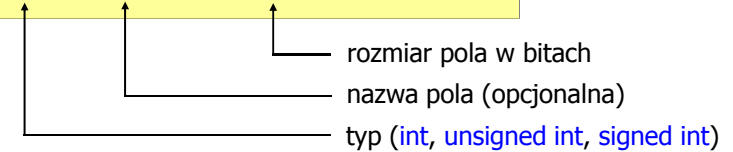
nr		
A	x	y
B	x	y
C	x	y

```
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;
```



- 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; /* zakres: 0...15 */  
    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; /* zakres: 0...15 */  
    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   : 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;
};
```

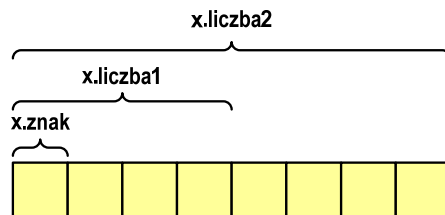
- Do przechowywania wartości w unii należy zadeklarować zmienną

```
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'};
```

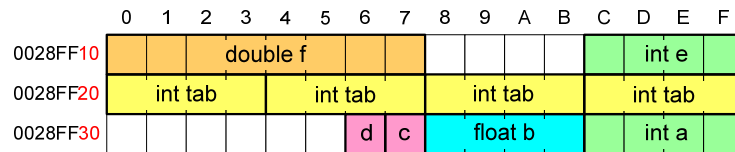
- Unie tego samego typu można sobie przypisywać

Co to jest wskaźnik?

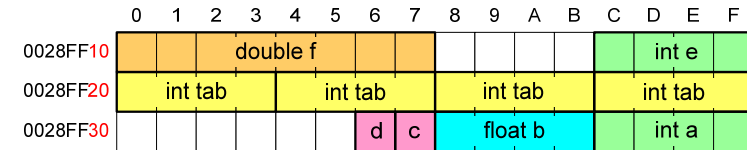
- **Wskaźnik** - zmienna mogąca zawierać adres obszaru pamięci
- najczęściej adres innej zmiennej (obiektu)

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

- Zmienne przechowywane są w pamięci komputera



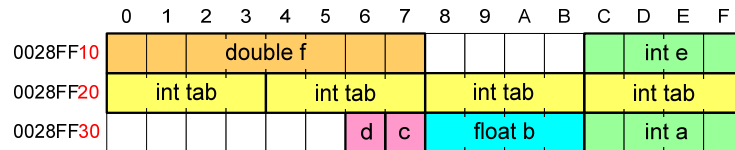
Co to jest wskaźnik?



- 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;
```

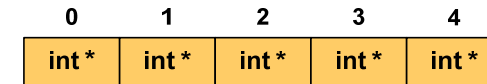
- Można konstruować wskaźniki do danych dowolnego typu łącznie z typami **wskaźnik do wskaźnika do...**

```
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:

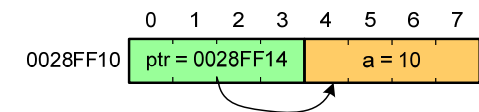
```
int *p1, *p2;  
int* p3, p4;
```

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

Przypisywanie wartości wskaźnikom

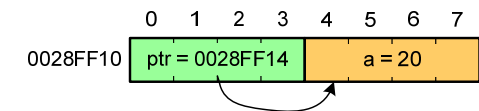
- Wskaźnikom można przypisywać adresy zmiennych
- Adresy takie tworzy się za pomocą operatora pobierania adresu `&`

```
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;
```

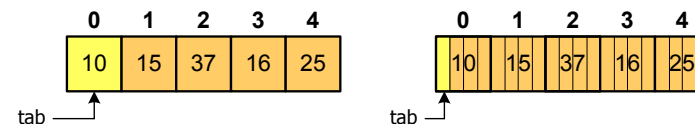
- Zamiast wartości **0** można stosować makrodefinicję preprocesora **NULL**, która podczas kompilacji programu zamieniana jest na **0**

```
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};
```

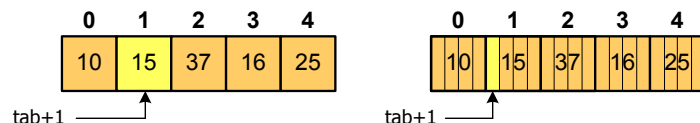


- Zastosowanie operatora ***** przed nazwą tablicy pozwala „dostać się” do zawartości elementu o indeksie **0**

***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;**

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 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);

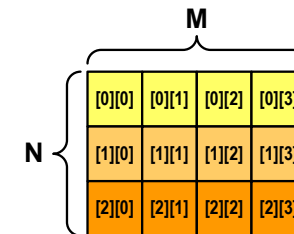
free(tab);

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 macierz

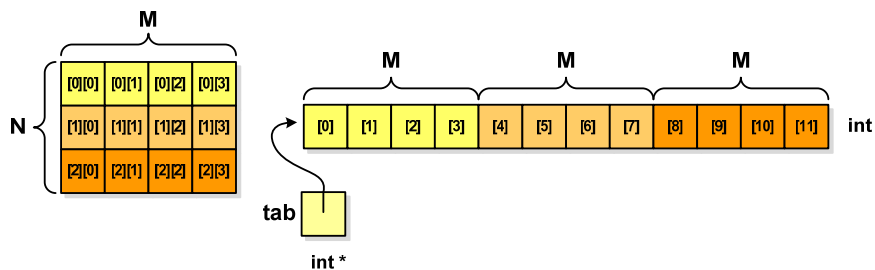
- Funkcje `calloc()` i `malloc()` umożliwiają bezpośrednio przydział pamięci tylko na wektor elementów
- Dynamiczny przydział pamięci na macierz wymaga zastosowania specjalnych metod
- Przydzielamy pamięć na macierz zawierającą **N-wierszy** i **M-kolumn**



Dynamiczny przydział pamięci na macierz (1)

- Wektor N×M-elementowy
- Przydział pamięci:

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



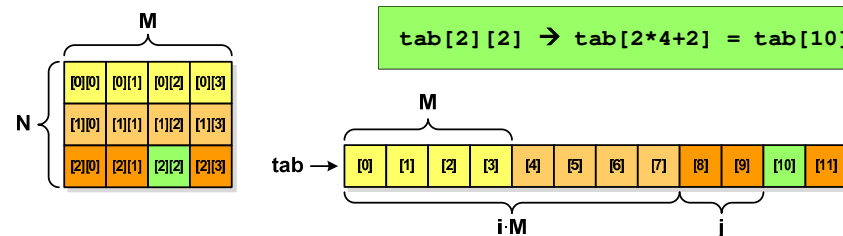
Dynamiczny przydział pamięci na macierz (1)

- Odwwołanie do elementów macierzy:

`tab[i*M+j]`

lub

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



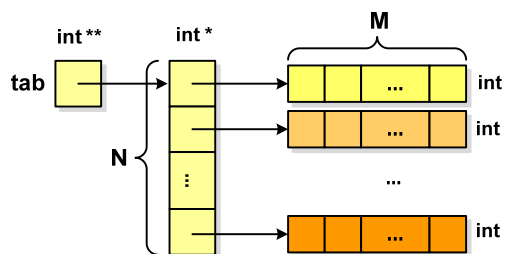
- Zwolnienie pamięci:

```
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));
```



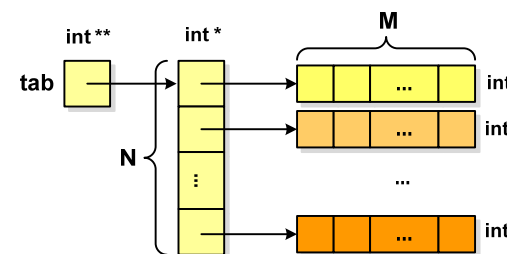
Dynamiczny przydział pamięci na macierz (2)

- Odwwołania do elementów macierzy:

`tab[i][j]`

- Zwolnienie pamięci:

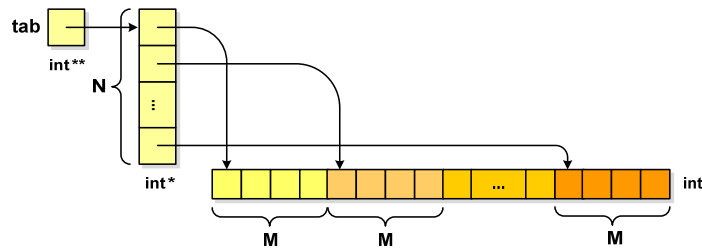
```
for (i=0; i<N; i++)  
    free(tab[i]);  
free(tab);
```



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;
```

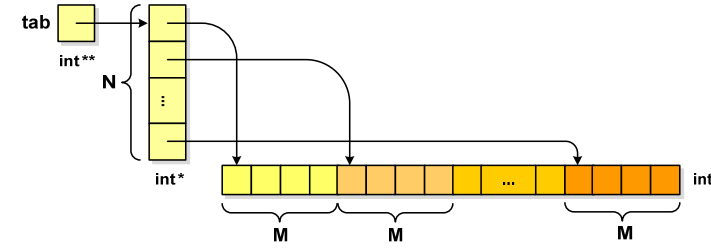


Dynamiczny przydział pamięci na macierz (3)

- Odwołania do elementów macierzy:
- Zwolnienie pamięci:

tab[i][j]

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



Program w języku C

- Program w języku C składa się z **funkcji** i **zmiennych**
 - funkcje zawierają instrukcje wykonujące operacje
 - zmienne przechowują wartości

```
#include <stdio.h> /* przekatna kwadratu */  
#include <math.h>  
  
int main(void)  
{  
    float a = 10.0f, d;  
    d = a * sqrt(2.0f);  
    printf("Bok = %g, przekatna = %g\n", a, d);  
    return 0;  
}
```

Bok = 10, przekatna = 14.1421

Program w języku C

- Program w języku C składa się z **funkcji** i **zmiennych**
 - funkcje zawierają instrukcje wykonujące operacje
 - zmienne przechowują wartości

```
#include <stdio.h> /* przekatna kwadratu */  
#include <math.h>  
  
int main(void)  
{  
    float a = 10.0f, d;  
    d = a * sqrt(2.0f);  
    printf("Bok = %g, przekatna = %g\n", a, d);  
    return 0;  
}
```

definicja funkcji

Program w języku C

- Program w języku C składa się z **funkcji** i **zmiennych**
 - funkcje zawierają instrukcje wykonujące operacje
 - zmiennie przechowują wartości

```
#include <stdio.h> /* przekatna kwadratu */
#include <math.h>

int main(void)
{
    float a = 10.0f, d;
    d = a * sqrt(2.0f);
    printf("Bok = %g, przekatna = %g\n", a, d);
    return 0;
}
```

wywołania funkcji

Funkcje w języku C

```
#include <stdio.h> /* przekatna kwadratu */
#include <math.h>

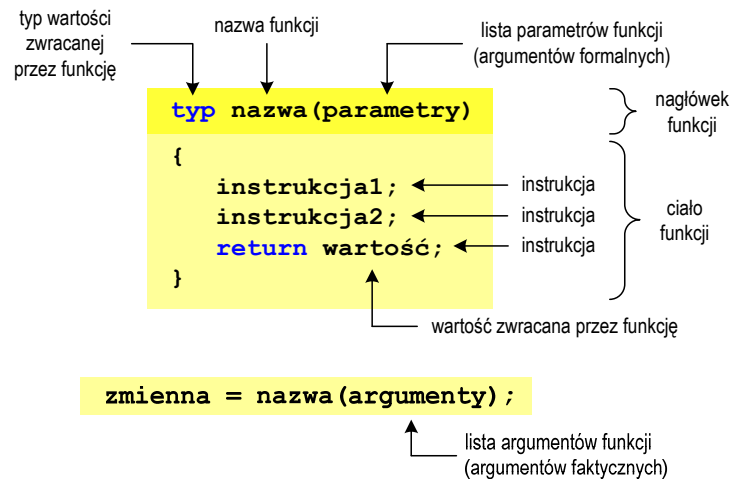
float przekatna(float bok)
{
    float wynik;
    wynik = bok * sqrt(2.0f);
    return wynik;
}

int main(void)
{
    float a = 10.0f, d;
    d = przekatna(a);
    printf("Bok = %g, przekatna = %g\n", a, d);
    return 0;
}
```

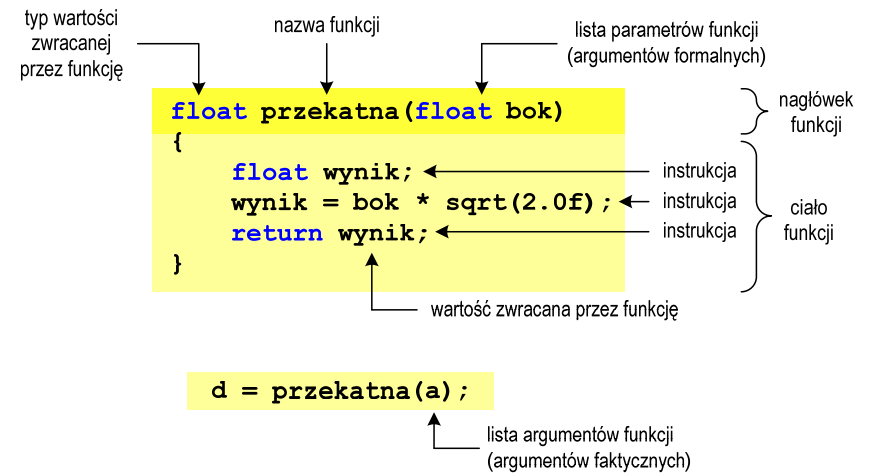
definicja funkcji

definicja funkcji

Ogólna struktura funkcji w języku C



Ogólna struktura funkcji w języku C



Argumenty funkcji

- Argumentami funkcji mogą być stałe liczbowe, zmienne, wyrażenia arytmetyczne, wywołania innych funkcji

```
d = przekatna(a);  
d = przekatna(10);  
d = przekatna(2*a+5);  
d = przekatna(sqrt(a)+15);
```

- Wywołanie funkcji może być argumentem innej funkcji

```
printf("Bok = %g, przekatna = %g\n",  
      a, przekatna(a));
```

Parametry funkcji

- Parametry funkcji traktowane są tak samo jak zmienne zadeklarowane w tej funkcji i zainicjalizowane wartościami argumentów wywołania

```
float przekatna(float bok)  
{  
    float wynik;  
    wynik = bok * sqrt(2.0f);  
    return wynik;  
}
```

- Funkcję `przekatna()` można zapisać w prostszej postaci:

```
float przekatna(float bok)  
{  
    return bok * sqrt(2.0f);  
}
```

Parametry funkcji

- Jeśli funkcja ma kilka parametrów, to dla każdego z nich podaje się:
 - typ parametru
 - nazwę parametru
- Parametry oddzielane są od siebie przecinkami

```
/* przekatna prostokąta */  
float przekatna(float a, float b)  
{  
    return sqrt(a*a+b*b);  
}
```

Parametry funkcji

- W różnych funkcjach zmienne mogą mieć takie same nazwy

```
#include <stdio.h> /* przekatna prostokąta */  
#include <math.h>  
  
float przekatna(float a, float b)  
{  
    return sqrt(a*a+b*b);  
}  
  
int main(void)  
{  
    float a = 10.0f, b = 5.5f, d;  
    d = przekatna(a,b);  
    printf("Przekatna prostokata = %g\n",d);  
    return 0;  
}
```

Domyślne wartości parametrów funkcji

- W definicji funkcji można jej parametrom nadać domyślne wartości

```
float przekatna(float a = 10, float b = 5.5f)
{
    return sqrt(a*a+b*b);
}
```

- W takim przypadku funkcję można wywołać z dwoma, jednym lub bez żadnych argumentów

```
d = przekatna(a,b);
```

```
d = przekatna(a);
```

```
d = przekatna();
```

- Brakujące argumenty zostaną zastąpione wartościami domyślnymi

Domyślne wartości parametrów funkcji

- Nie wszystkie parametry muszą mieć podane domyślne wartości
- Wartości muszą być podawane od prawej strony listy parametrów

```
float przekatna(float a, float b = 5.5f)
{
    return sqrt(a*a+b*b);
}
```

- Powyższa funkcja może być wywołana z jednym lub dwoma argumentami

```
d = przekatna(a,b);
```

```
d = przekatna(a);
```

- Domyślne wartości parametrów mogą być podane w deklaracji lub w definicji funkcji

Wartość zwracana przez funkcję

- Słowo kluczowe `return` może wystąpić w funkcji wiele razy

```
float ocena(int pkt)
{
    if (pkt>90) return 5.0f;
    if (pkt>80 && pkt<91) return 4.5f;
    if (pkt>70 && pkt<81) return 4.0f;
    if (pkt>60 && pkt<71) return 3.5f;
    if (pkt>50 && pkt<61) return 3.0f;
    if (pkt<51) return 2.0f;
}
```

91-100 pkt. → 5,0

71-80 pkt. → 4,0

51-60 pkt. → 3,0

81-90 pkt. → 4,5

61-70 pkt. → 3,5

0-50 pkt. → 2,0

Prototyp funkcji

- Czy można zmienić kolejność definicji funkcji w kodzie programu?

```
#include <stdio.h> /* przekatna prostokata */
#include <math.h>
```

```
float przekatna(float a, float b)
{
    return sqrt(a*a+b*b);
}
```

```
int main(void)
{
    float a = 10.0f, b = 5.5f, d;
    d = przekatna(a,b);
    printf("Przekatna prostokata = %g\n",d);
    return 0;
}
```

definicja funkcji

definicja funkcji

Prototyp funkcji

- Czy można zmienić kolejność definicji funkcji w kodzie programu?

```
#include <stdio.h> /* przekątna prostokąta */
#include <math.h>

int main(void)
{
    float a = 10.0f, b = 5.5f, d;
    d = przekatna(a,b);
    printf("Przekatna prostokata = %g\n",d);
    return 0;
}

float przekatna(float a, float b)
{
    return sqrt(a*a+b*b);
}
```

Prototyp funkcji

- Czy można zmienić kolejność definicji funkcji w kodzie programu?

```
#include <stdio.h> /* przekątna prostokąta */
#include <math.h>

int main(void)
{
    float a = 10.0f, b = 5.5f, d;
    d = przekatna(a,b);
    printf("Przekatna prostokata = %g\n",d);
    return 0;
}

float przekatna(float a, float b)
{
    return sqrt(a*a+b*b);
}
```

error C3861: 'przekatna': identifier not found

Prototyp funkcji

```
#include <stdio.h> /* przekątna prostokąta */
#include <math.h>

float przekatna(float a, float b);

int main(void)
{
    float a = 10.0f, b = 5.5f, d;
    d = przekatna(a,b);
    printf("Przekatna prostokata = %g\n",d);
    return 0;
}

float przekatna(float a, float b)
{
    return sqrt(a*a+b*b);
}
```

Prototyp funkcji

- Prototyp funkcji jest to jej nagłówek zakończony średnikiem
- Inne określenia prototypu funkcji:
 - deklaracja funkcji
 - zapowiedź funkcji
- Dzięki prototypowi kompilator sprawdza w wywołaniu funkcji:
 - nazwę funkcji
 - liczbę i typ argumentów
 - typ zwracanej wartości
- Nazwy parametrów nie mają znaczenia i mogą być pominięte:

```
float przekatna(float a, float b);
```

```
d = przekatna(a,b);
```

```
float przekatna(float, float);
```

Prototyp funkcji

- W przypadku umieszczenia prototypu funkcji i pominięcia jej definicji błąd wystąpi nie na etapie kompilacji, ale łączenia (linkowania)

```
#include <stdio.h>    /* przekatna prostokata */  
#include <math.h>
```

```
float przekatna(float a, float b);
```

prototyp funkcji

```
int main(void)  
{  
    float a = 10.0f, b = 5.5f, d;  
    d = przekatna(a,b);  
    printf("Przekatna prostokata = %g\n",d);  
    return 0;  
}
```

definicja funkcji

Prototyp funkcji

- W przypadku umieszczenia prototypu funkcji i pominięcia jej definicji błąd wystąpi nie na etapie kompilacji, ale łączenia (linkowania)

```
1>Compiling...  
1>test.cpp  
1>Compiling manifest to resources...  
1>Microsoft (R) Windows (R) Resource Compiler Version 6.0.5724.0  
1>Copyright (C) Microsoft Corporation. All rights reserved.  
1>Linking...  
1>test.obj : error LNK2019: unresolved external symbol "float __cdecl  
przekatna(float,float)" (?przekatna@@@YAMMM@Z) referenced in function _main  
1>D:\test\Debug\test.exe : fatal error LNK1120: 1 unresolved externals
```

Typy funkcji (1)

- Dotychczas prezentowane funkcje miały argumenty i zwracały wartości
- Struktura i wywołanie takiej funkcji ma następującą postać

```
typ nazwa (parametry)  
{  
    instrukcje;  
    return wartosc;  
}
```

```
typ zm;  
zm = nazwa(argumenty);
```

- Można zdefiniować także funkcje, które nie mają argumentów i/lub nie zwracają żadnej wartości

Typy funkcji (2)

- Funkcja bez argumentów i nie zwracająca wartości:
 - w nagłówku funkcji, typ zwracanej wartości to **void**
 - zamiast parametrów, podaje się słowo **void** lub nie wpisuje się nic
 - jeśli występuje **return**, to nie może po nim znajdować się żadna wartość
 - jeśli **return** nie występuje, to funkcja kończy się po wykonaniu wszystkich instrukcji
- Struktura funkcji:

```
void nazwa(void)  
{  
    instrukcje;  
    return;  
}
```

```
void nazwa()  
{  
    instrukcje;  
    return;  
}
```

Typy funkcji (2)

- Funkcja bez argumentów i nie zwracająca wartości:
 - w nagłówku funkcji, typ zwracanej wartości to **void**
 - zamiast parametrów, podaje się słowo **void** lub nie wpisuje się nic
 - jeśli występuje **return**, to nie może po nim znajdować się żadna wartość
 - jeśli **return** nie występuje, to funkcja kończy się po wykonaniu wszystkich instrukcji

- Struktura funkcji:

```
void nazwa(void)
{
    instrukcje;
}
```

```
void nazwa ()
{
    instrukcje;
}
```

- Wywołanie funkcji: `nazwa ();`

Typy funkcji (3)

- Funkcja z argumentami i nie zwracająca wartości:
 - w nagłówku funkcji, typ zwracanej wartości to **void**
 - jeśli występuje **return**, to nie może po nim znajdować się żadna wartość
 - jeśli **return** nie występuje, to funkcja kończy się po wykonaniu wszystkich instrukcji

- Struktura funkcji:

```
void nazwa(parametry)
{
    instrukcje;
    return;
}
```

```
void nazwa(parametry)
{
    instrukcje;
}
```

- Wywołanie funkcji: `nazwa(argumenty);`

Typy funkcji (2) - przykład

```
#include <stdio.h>

void drukuj_linie(void)
{
    printf("-----\n");
}

int main(void)
{
    drukuj_linie();
    printf("Funkcje nie sa trudne!\n");
    drukuj_linie();

    return 0;
}
```

```
-----
Funkcje nie sa trudne!
-----
```

Typy funkcji (3) - przykład

```
#include <stdio.h>

void drukuj_dane(char *imie, char *nazwisko, int wiek)
{
    printf("Imie:           %s\n", imie);
    printf("Nazwisko:        %s\n", nazwisko);
    printf("Wiek:              %d\n", wiek);
    printf("Rok urodzenia:     %d\n", 2018-wiek);
}

int main(void)
{
    drukuj_dane("Jan", "Kowalski", 23);
    drukuj_dane("Barbara", "Nowak", 28);

    return 0;
}
```


Typy funkcji (3) - przykład

```
#include <stdio.h>

void drukuj_dane(char *imie,
{
    printf("Imie:
    printf("Nazwisko:
    printf("Wiek:
    printf("Rok urodzenia:
}

int main(void)
{
    drukuj_dane("Jan", "Kowalski", 23);
    drukuj_dane("Barbara", "Nowak", 28);

    return 0;
}
```

```
Imie:           Jan
Nazwisko:       Kowalski
Wiek:           23
Rok urodzenia: 1995

Imie:           Barbara
Nazwisko:       Nowak
Wiek:           28
Rok urodzenia: 1990
```

Typy funkcji (4)

- Funkcja bez argumentów i zwracająca wartość:
 - zamiast parametrów, podaje się słowo **void** lub nie wpisuje się nic
 - typ zwracanej wartości musi być zgodny z typem w nagłówku funkcji

- Struktura funkcji:

```
typ nazwa(void)
{
    instrukcje;
    return wartość;
}
```

```
typ nazwa()
{
    instrukcje;
    return wartość;
}
```

- Wywołanie funkcji:

```
typ zm;
zm = nazwa();
```

Typy funkcji (4) - przykład

```
#include <stdio.h>

int liczba_sekund_rok(void)
{
    return (365 * 24 * 60 * 60);
}

int main(void)
{
    int wynik;

    wynik = liczba_sekund_rok();
    printf("W roku jest: %d sekund\n", wynik);

    return 0;
}
```

```
W roku jest: 31536000 sekund
```

Przekazywanie argumentów do funkcji

- Przekazywanie argumentów przez **wartość**:
 - po wywołaniu funkcji tworzone są lokalne kopie zmiennych skojarzonych z jej argumentami
 - w funkcji widoczne są one pod postacią parametrów funkcji
 - parametry te mogą być traktowane jak lokalne zmienne, którym przypisano początkową wartość
- Przekazywanie argumentów przez **wskaźnik**:
 - do funkcji przekazywane są adresy zmiennych będących jej argumentami
 - wszystkie operacje wykonywane w funkcji na takich argumentach będą odnosiły się do zmiennych z funkcji wywołującej

Przekazywanie argumentów przez wartość

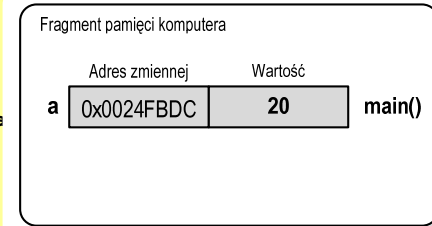
```
#include <stdio.h>

void fun(int a)
{
    a = 10;
    printf("fun: a = %d\n", a);
}

int main(void)
{
    int a = 20;

    fun(a);
    printf("main: a = %d\n", a);

    return 0;
}
```



Przekazywanie argumentów przez wartość

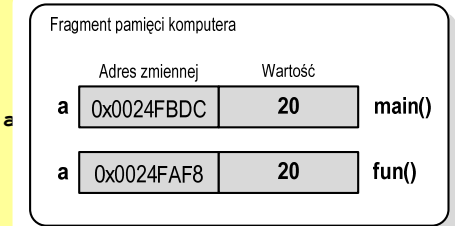
```
#include <stdio.h>

void fun(int a)
{
    a = 10;
    printf("fun: a = %d\n", a);
}

int main(void)
{
    int a = 20;

    fun(a);
    printf("main: a = %d\n", a);

    return 0;
}
```



Przekazywanie argumentów przez wartość

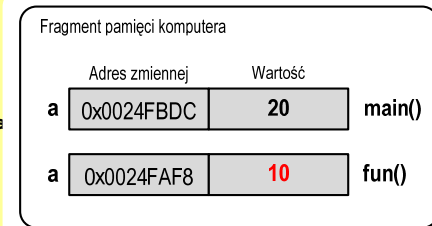
```
#include <stdio.h>

void fun(int a)
{
    a = 10;
    printf("fun: a = %d\n", a);
}

int main(void)
{
    int a = 20;

    fun(a);
    printf("main: a = %d\n", a);

    return 0;
}
```



fun: a = 10

Przekazywanie argumentów przez wartość

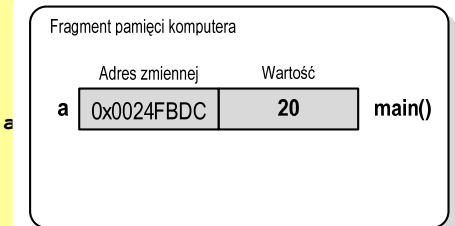
```
#include <stdio.h>

void fun(int a)
{
    a = 10;
    printf("fun: a = %d\n", a);
}

int main(void)
{
    int a = 20;

    fun(a);
    printf("main: a = %d\n", a);

    return 0;
}
```



fun: a = 10
main: a = 20

Przekazywanie argumentów przez wskaźnik

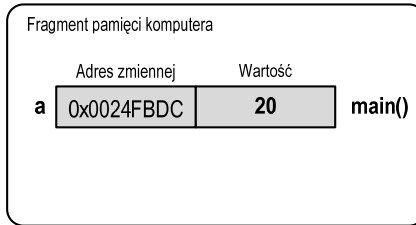
```
#include <stdio.h>

void fun(int *a)
{
    *a = 10;
    printf("fun: a = %d\n", *a)
}

int main(void)
{
    int a = 20;

    fun(&a);
    printf("main: a = %d\n", a);

    return 0;
}
```



Przekazywanie argumentów przez wskaźnik

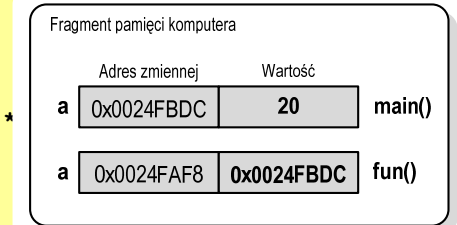
```
#include <stdio.h>

void fun(int *a)
{
    *a = 10;
    printf("fun: a = %d\n", *a)
}

int main(void)
{
    int a = 20;

    fun(&a);
    printf("main: a = %d\n", a);

    return 0;
}
```



Przekazywanie argumentów przez wskaźnik

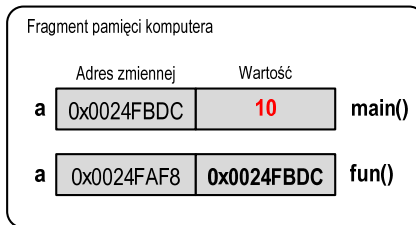
```
#include <stdio.h>

void fun(int *a)
{
    *a = 10;
    printf("fun: a = %d\n", *a)
}

int main(void)
{
    int a = 20;

    fun(&a);
    printf("main: a = %d\n", a);

    return 0;
}
```



fun: a = 10

Przekazywanie argumentów przez wskaźnik

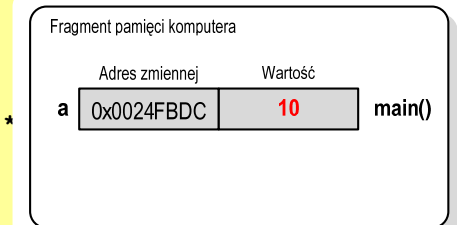
```
#include <stdio.h>

void fun(int *a)
{
    *a = 10;
    printf("fun: a = %d\n", *a)
}

int main(void)
{
    int a = 20;

    fun(&a);
    printf("main: a = %d\n", a);

    return 0;
}
```



fun: a = 10
main: a = 10

Koniec wykładu nr 4

Dziękuję za uwagę!