

# Wykład 10: Funkcje.

dr inż. Andrzej Stafiniak

*Wrocław 2023*



HR EXCELLENCE IN RESEARCH



Politechnika Wroclawska

# Podstawowe informacje o funkcjach

**Czym jest funkcja ?**

# Podstawowe informacje o funkcjach

**Funkcja** - Funkcja to wydzielony kodu programu (podprogram), który ma ściśle zdefiniowane zadanie, wykonanie instrukcji z ciała funkcji, np. różnego rodzaju obliczenia czy operacje we-wy.

Funkcja może zwracać jakąś wartość, ale może też modyfikować wartość zmiennych.

Do funkcji możemy przekazać wartości za pomocą listy argumentów podczas wywołania funkcji.

# Podstawowe informacje o funkcjach

## Co dają nam funkcję ?



**Funkcja** - Funkcja to wydzielony kodu programu (podprogram), który ma ściśle zdefiniowane zadanie, wykonanie instrukcji z ciała funkcji, np. różnego rodzaju obliczenia czy operacje we-wy.

Funkcja może zwracać jakąś wartość, ale może też modyfikować wartość zmiennych.

Do funkcji możemy przekazać wartości za pomocą listy argumentów podczas wywołania funkcji.

# Podstawowe informacje o funkcjach

Dzięki **funkcjom** mamy:

- Logiczny podział programu na podprogramy – programowanie **proceduralne**.
- Umożliwienie pracy zespołowej przy tworzeniu rozbudowanego oprogramowania.
- Zabezpieczenie przed niepotrzebnym duplikowaniem fragmentów kodów źródłowych.
- Zwiększenie czytelności kodu źródłowego.
- Możliwość wykorzystania mechanizmów **rekurencji**.

# Podstawowe informacje o funkcjach

W języku C/C++ wyróżniamy:

- funkcję `main()` – **główną funkcję** aplikacji konsolowej;
- funkcje **predefiniowane** w bibliotekach – np. `strlen()`;
- funkcje definiowane przez nas.

## Podstawowe informacje o funkcjach

## Ogólna składnia funkcji:

## Nagłówek funkcji

## Sygnatura funkcji

## Lista argumentów funkcji

```
typZwracany nazwaFunkcji (argument1, argument2, ..) {
```

```
// "Ciało funkcji"
```

}

# Przykłady funkcji

Funkcja, która nic nie pobiera oraz nic nie zwraca:

```
void showInfo() { // void showInfo(void)
    char ch = '!';
    for(int i=0; i<3; ++i) {
        printf("No more war %c%c%c", 33, '!', ch);
    }
    // return;
}
```

Wywołanie funkcji:      showInfo();



# Przykłady funkcji

Funkcja, która pobiera dwa argumenty oraz nic nie zwraca:

Jeśli nie określimy typu zwracanego przez funkcję, kompilator języka C przyjmie domyślnie typ zwracany jako int.

```
void showDivResult(int x, int y) {  
    float result;  
    result = (float) x / y;  
    printf("Division result: %f", result);  
    // return;  
}
```

Argumenty formalne,  
parametry

Wywołanie funkcji:

```
showDivResult(2, 4);
```

Argumenty aktualne

# Argument vs. Parametr

**Argumenty (argumenty aktualne)** to wartości stałe lub nazwy zmiennych przekazywanych do funkcji w czasie wywołania funkcji.

```
returnDivResult(2, 4)
```

**Parametry (argumenty formalne)** to wartości, które funkcja oczekuje do otrzymania.

```
void showDivResult(int x, int y)
```

# Przykłady funkcji

Funkcja, która pobiera dwa argumenty oraz zwraca wartość:

```
float returnDivResult(int x, int y) {  
    float result;  
    result = (float) x / y;  
    return result;    //return (float) x/y;  
}
```

Słowo kluczowe **return**

Wywołanie funkcji: **float** z = returnDivResult(2, 4);

# Instrukcja `return`

Instrukcja `return` (słowo kluczowe):

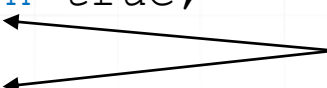
- kończy wykonywanie funkcji,
- zwraca, w miejscu wywołania funkcji, wartość wyrażenia stojącego po słowie kluczowym `return`,
- jeśli wyrażenie zostanie pominięte (w przypadku funkcji innej niż typu `void`) wartość zwracana funkcji jest niezdefiniowana,
- wyrażenie po słowie `return` jest obliczane a następnie konwertowane na typ zwracany przez funkcję,
- każda funkcja, która ma typ zwracany inny niż `void` musi zawierać słowo kluczowe `return`.

# Przykłady funkcji

Funkcja, która pobiera argument oraz zwraca wartość:

```
bool isOdd(int x) {    //<stdbool.h>
    if (x%2)
        return true;
    else
        return false;
}
```

W zależności od potrzeby



Wywołanie funkcji `isOdd()` w ciele innej funkcji (funkcja zagnieżdżona w funkcji)

```
void printParity(int a) {    //<stdbool.h>
    if (!isOdd(a))
        printf("Even number");
    else
        printf("Odd number");
}
```

# Przykłady funkcji

Wywołanie funkcji w funkcji:

```
bool isOdd(int x) {  
    if (x%2)  
        return true;  
    else  
        return false;  
}
```

```
void printParity(int a) {  
    if (!isOdd(a))  
        printf("Even number");  
    else  
        printf("Odd number");  
}
```

Wykonywanie funkcji `printParity()` jest chwilowo zawieszone, gdy następuje wywołanie funkcji `isOdd()`.

Instrukcja `return` kończy wykonywanie funkcji `isOdd()` i zwracane jest sterowanie do funkcji zewnętrznej w punkcie bezpośrednio po wywołaniu.

# Deklaracja, definicja oraz wywołanie funkcji

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

```
//Deklaracja funkcji - prototyp funkcji
```

```
float returnAddResult(int x, int y);    //<plik_nagłówekowy.h>
```

**Prototyp funkcji** zapowiada typ zwracany przez funkcję oraz typy argumentów funkcji.

```
//Funkcja główna main()
```

```
int main(){
```

```
    int a = 2, b = 3;
```

```
    printf("Addition result: %f", returnAddResult(a, b));
```

```
    return 0;
```

```
}
```

```
//Definicja funkcji
```

```
float returnAddResult(int x, int y){
```

```
    return (float) x + y;
```

```
}
```

Niedozwolone jest definiowanie nowej funkcji w ciele innej funkcji.

# Funkcje – przekazywanie argumentów

Przekazywanie argumentów do funkcji może odbywać się przez:

- wartości (język C/C++) ,
- wskaźniki (język C/C++),
- referencje (język C++)



# Przekazywanie argumentów przez wartości

Przekazywanie argumentów przez **wartości** charakteryzuje się:

- tym, że na stosie tworzone są nowe zmienne lokalne - kopie argumentów przekazanych do funkcji,
- ich czas życia związany jest z czasem wykonania bloku funkcyjnego,
- operowanie na kopiach argumentów, nie ma możliwości modyfikowania oryginalnych zmiennych,
- w takiej sytuacji aby móc wyciągnąć z funkcji efekt operacji należy posłużyć się instrukcją `return`.

```
int x=1, y=2;
```

```
float z = returnDivResult(x, y);
```

# Przekazywanie argumentów przez wartości

```
#include <stdio.h>

int podwajanie( int dl){
    dl = dl*2;
    printf("\nWywołanie funkcji:\n");
    printf("Adres kopi zmiennej dl: %#x\n", &dl);
    printf("Wartosc kopi zmiennej dl: %d\n", dl);
    return dl;
}

int main(){
    int dl = 10;

    printf("Adres oryginalnej zmiennej dl: %#x\n", &dl);
    printf("Wartosc oryginalnej zmiennej dl: %d\n", dl);

    podwajanie(dl);

    printf("\nWartosc oryginalnej zmiennej dl: %d\n", dl);

    dl = podwajanie(dl);

    printf("\nNowa wartosc oryginalnej zmiennej dl: %d\n", dl);
    printf("Adres oryginalnej zmiennej dl: %#x\n", &dl);

    return 0;
}
```

Adres oryginalnej zmiennej dl: 0x61ff1c  
Wartosc oryginalnej zmiennej dl: 10

Wywołanie funkcji:

Adres kopi zmiennej dl: 0x61ff00  
Wartosc kopi zmiennej dl: 20

Wartosc oryginalnej zmiennej dl: 10

Wywołanie funkcji:

Adres kopi zmiennej dl: 0x61ff00  
Wartosc kopi zmiennej dl: 20

Nowa wartosc oryginalnej zmiennej dl: 20  
Adres oryginalnej zmiennej dl: 0x61ff1c

# Przekazywanie argumentów przez wskaźniki

Przekazywanie argumentów przez **wskaźniki** polega na przekazywaniu do funkcji adresów zmiennych, na których funkcja wykonuje operacje:

- może odczytać wartość,
- znając adres zasobu może go nadpisać, zmodyfikować,
- wykorzystując wskaźniki, istnieje możliwość przekazania do funkcji złożonych struktur danych typu tablica,

```
float returnDivResult(int * x, int * y);
```

```
    int x=1, y=3;  
    int * ptrX = &x, ptrY = &y;  
    float z = returnDivResult(ptrX, ptrY);
```

# Przekazywanie argumentów przez wskaźniki

```
#include <stdio.h>

void podwajanie( int *dl){
    *dl = *dl * 2;
    printf("\nWywołanie funkcji:\n");
    printf("Adres, na którym operuje funkcja: %#x\n", dl);
}

int main(){
    int dl = 10;
    int * ptrDl = &dl;

    printf("Adres zmiennej dl: %#x\n", &dl);
    printf("Wartosc zmiennej dl: %d\n", dl);

    podwajanie(ptrDl);

    printf("\nNowa wartosc zmiennej dl: %d\n", dl);

    podwajanie(&dl);

    printf("\nNowa wartosc zmiennej dl: %d\n", dl);

    return 0;
}
```

```
Adres zmiennej dl: 0x61ff18
Wartosc zmiennej dl: 10

Wywołanie funkcji:
Adres, na którym operuje funkcja: 0x61ff18

Nowa wartosc zmiennej dl: 20

Wywołanie funkcji:
Adres, na którym operuje funkcja: 0x61ff18

Nowa wartosc zmiennej dl: 40
```

# Przekazywanie argumentów przez referencje

W języku C++ mamy jeszcze jeden mechanizm przekazywania argumentów do funkcji, przekazywanie przez **referencję**. W nagłówku funkcji stosujemy w trakcie deklarowania parametrów funkcji dodatkowa znak **&**.

```
float returnDivResult(int & x, int & y);
```

- Przekazywanie argumentów przez referencję w porównaniu do metody ze wskaźnikami jest łatwiejsze, ponieważ nie trzeba operować na **\*** i **&**.
- Zmienne wewnątrz funkcji nie są kopią, oznacza to, że operując na zmiennych referencyjnych operujemy także na zmiennej oryginalnej.

# Przekazywanie argumentów przez referencje

```
#include <iostream>

void podwajanie( int &dl){
    dl = dl * 2;
}

int main(){
    int dl = 10;

    printf("Wartosc zmiennej dl: %d\n", dl);

    podwajanie(dl);

    printf("\nNowa wartosc zmiennej dl: %d\n", dl);

    podwajanie(dl);

    printf("\nNowa wartosc zmiennej dl: %d\n", dl);

    return 0;
}
```

Wartosc zmiennej dl: 10

Nowa wartosc zmiennej dl: 20

Nowa wartosc zmiennej dl: 40

# Przeciążanie funkcji

W języku C nie ma możliwości zdefiniowania dwóch funkcji o tej samej nazwie.

Język C++ wprowadził **mechanizm przeciążania funkcji** oraz operatorów.

**Mechanizm przeciążania funkcji** (polimorfizm, wielopostaciowość) umożliwia stosowanie tych samych nazw funkcji, ale różniących się listą argumentów.

Mechanizm ten polega na tym, że w zależności od kontekstu wywołania funkcji (dla liczby oraz typu argumentów) następuje odwołanie do odpowiedniej definicji/implementacji funkcji.

Odnalezienie odpowiedniej definicji odbywa się na etapie kompilacji (polimorfizm statyczny, polimorfizm czasu kompilacji)

# Przeciążanie funkcji (polimorfizm, wielopostaciowość)

```
#include <iostream>

void sum( int x, int y){
    std::cout << "Sum of integers: " << x+y << std::endl;
}

void sum( int x=1){
    std::cout << "Sum of default argument: " << x+x << std::endl;
}

void sum( int x, int y, int z){
    std::cout << "Sum of 3 integers: " << x+y+z<< std::endl;
}

void sum( float x, float y){
    std::cout << "Sum of floats: " << x+y << std::endl;
}

void sum( double x, double y){
    std::cout << "Sum of doubles: " << x+y << std::endl;
}

int main(){
    sum(1, 2);
    sum(1, 2, 3);
    sum(1.1f, 2.1f);
    sum(1.2, 2.2);
    sum(3);
    sum();
    return 0;
}
```

```
Sum of integers: 3
Sum of 3 integers: 6
Sum of floats: 3.2
Sum of doubles: 3.4
Sum of default argument: 6
Sum of default argument: 2
```

```
0040a004 D  tls used
00401581 T  __Z3sumdd
0040153a T  __Z3sumff
004014a8 T  __Z3sumi
00401460 T  __Z3sumii
004014ed T  __Z3sumiii
00401673 t  __Z41__static_initiali
004016f0 T  __ZNSolsEd
004016e8 T  __ZNSolsEf
```



# Przeciążanie funkcji (argument domyślny)

```
#include <iostream>

void sum( int x, int y){
    std::cout << "Sum of integers: " << x+y << std::endl;
}

void sum( int x=1){
    std::cout << "Sum of default argument: " << x+x << std::endl;
}

void sum( int x, int y, int z){
    std::cout << "Sum of 3 integers: " << x+y+z << std::endl;
}

void sum( float x, float y){
    std::cout << "Sum of floats: " << x+y << std::endl;
}

void sum( double x, double y){
    std::cout << "Sum of doubles: " << x+y << std::endl;
}

int main(){
    sum(1, 2);
    sum(1, 2, 3);
    sum(1.1f, 2.1f);
    sum(1.2, 2.2);
    sum(3);
    sum();
    return 0;
}
```

```
Sum of integers: 3
Sum of 3 integers: 6
Sum of floats: 3.2
Sum of doubles: 3.4
Sum of default argument: 6
Sum of default argument: 2
```

Wykorzystując mechanizm przeciążanie nazw funkcji można definiować funkcje z domyślnymi argumentami.

Przez domyślny argument określa się argument aktualny, z którym zostanie wywołana funkcja, jeżeli nie podamy jawnie innego.

Wartość domyślą przypisujemy do parametru funkcji za pomocą operatora przypisania.

Każdy następujący argument po argumencie z wartością domyślną też musi posiadać wartość domyślną.

# Wskaźniki funkcyjne

Funkcje, tak samo jak zmienne również posiadają swój adres w pamięci komputera (segmencie text (code)).

Dzięki temu mamy możliwość stosowania również **wskaźników funkcyjnych**, jako zmiennych przechowujących adresy funkcji.

Składnia wskaźnika funkcyjnego:

**typZwracany** (\*nazwaWskaznika) (listaArgumentów)

Przypisanie do **wskaźnika funkcyjnego** adresu funkcji odbywa się bez wykorzystania operatora przypisania adresu & jak sytuacji zmiennych. Nazwa funkcji jest wskaźnikiem adresu na nią samą, podobnie jak nazwa tablicy stanowi wskaźnik na jej pierwszy element.

```
nazwaWskaznika = nazwaFunkcji;
```

# Wskaźniki funkcyjne

```
#include <stdio.h>

int add( int x, int y){
    return x+y;
}

int mul( int x, int y){
    return x*y;
}

void showResultOperation(int x, int y, int (* funPtr) (int, int)){
    printf("Operation: %d\n", funPtr(x,y));
}

int main(){
    int a = 10, b = 20;

    // Deklaracja i inicjalizacja wskaźnika funkcyjnego
    int (* funPtr) (int, int)=add;
    //funPtr = add;

    printf("Add: %d\n", funPtr(a,b));

    showResultOperation(a,b,add);
    showResultOperation(a,b,mul);

    return 0;
}
```

```
Add: 30
Operation: 30
Operation: 200
```

```

1 // Wskaznik na typ calkowity
2 int * ptr;
3 // Tablica liczb calkowitych
4 int tab[];
5 // Tablica wskaznikow na typ calkowity
6 int * tab[];
7 // Wskaznik na wskaznik na typ calkowity
8 int ** ptr;
9 // Funkcja pobierajaca wskaznik na typ calkowity i
   niezwracajaca nic
10 void func(int *);
11 // Wskaznik na funkcje pobierajaca wskaznik na typ
   calkowity i niezwracajaca nic
12 void (*ptr)(int *);
13 // Funkcja pobierajaca liczbe calkowita i zwracajaca
   wskaznik na typ zmiennoprzecinkowy
14 double * func(int);
15 // Wskaznik na funkcje pobierajaca liczbe calkowita
   i zwracajaca wskaznik na typ zmiennoprzecinkowy
16 double * (*ptr)(int);
17 // Funkcja niepobierajaca nic i zwracajaca wskaznik
   na funkcje pobierajaca liczbe calkowita i
   zwracajaca wskaznik na typ zmiennoprzecinkowy
18 double * (*func())(int)
19 // Tablica wskaznikow na funkcje pobierajacych
   liczbe calkowita i zwracajacych wskaznik na typ
   zmiennoprzecinkowy
20 double * (*ptr[])(int);

```

# Makrodefinicje

- Ze słowem **makro**, czy **makrodefinicja** mieliśmy już kontakt, np.:

`getc()`, `putc()`, `va_start()`, `va_arg()` ...

Czym są **makrodefinicje** ?

# Makrodefinicje

- **Makrodefinicje** są szczególną konstrukcją stosowaną w języku C – coś na kształt funkcji, ale nie do końca. Tworzone są za pomocą dyrektywy preprocesora `#define`, podobnie jak stałe:

```
#define PI 3.1415
```

w kodzie źródłowy, wszędzie gdzie znajduje się identyfikator `PI`, preprocesor wstawi wartość 3.1415.

- Ale makrodefinicje są czymś więcej niż stałą, mogą realizować pewne zadanie. Konstrukcja makrodefinicji:

```
#define NAZWA (lista parametrów) instrukcje
```

- Generalnie, dyrektywa `#define` nakazuje preprocesorowi zamianę ciągu znaków `NAZWA` na ciąg instrukcji znajdujący się po opcjonalnej liście parametrów.

# Makrodefinicje

- **Makrodefinicje** stosowane są często w celu ułatwienia i skrócenia czasu związanego z pisanem kodu źródłowego :

```
#define ERROR printf("Failed to close the file \n")
```

w kodzie źródłowy, wszędzie gdzie znajduje się identyfikator `ERROR`, preprocesor wstawi funkcję `printf`.

- **Makrodefinicje** mogą być definiowane z wykorzystaniem parametrów:

```
#define MIN (x, y) (x)<(y)?(x):(y)
```

wywołanie w kodzie źródłowy makra `zmienna = MIN(a, b) ;`, spowoduje, że preprocesor podstawí pod `MIN(a, b)` wyrażenie zdefiniowane w makrze -  
`zmienna = (a)<(b)?(a):(b) ;`.

# Makrodefinicje

- **Makrodefinicje** stosowane są często w celu ułatwienia i skrócenia czasu związanego z pisanem kodu źródłowego :

```
#define ERROR printf("Failed to close the file \n")
```

w kodzie źródłowy, wszędzie gdzie znajduje się identyfikator `ERROR`, preprocesor wstawi instrukcję `printf`.

- **Makrodefinicje** mogą być definiowane z wykorzystaniem parametrów:

```
#define MIN (x, y) ((x)<(y)?(x):(y))
```

Warto zastosować dodatkowe nawiasy grupujące? **Dlaczego?**

wywołanie w kodzie źródłowy makra `zmienna = MIN(a, b) ;` spowoduje, że preprocesor podstawí pod `MIN(a, b)` wyrażenie zdefiniowane w makrze -  
`zmienna = (a)<(b)?(a):(b) ;`.



# Makrodefinicje

## ➤ Przykład

```
#define SUM (x, y) x+y
```

wywołanie w kodzie źródłowy

```
float sredniaWartosc = SUM(3.0f, 7.0f) / 2.0f;
```

```
printf("Srednia wartosc - %f", sredniaWartosc);
```

Jaka wartość zostanie wyświetlona?

# Makrodefinicje

➤ Kolejny przykład:

```
#define STOPNIE_F (x) ((5.0/9.0)*(x-32))  
  
#define STOPNIE_K (x) (x+272.15)
```

Wywołanie makropoleczeń:

```
float tempC = 36.6;
```

```
printf("Wartosc temperatury w stopniach Farenheita to %f, w  
stopniach Kelwina to %f", STOPNIE_F(tempC),  
STOPNIE_K(tempC));
```

# Makrodefinicje

## ➤ Kolejny przykład:

```
#define ERROR printf("Failed to close the file \n"); \
        exit ( EXIT_FAILURE );
```

Znak '\\' oznacza kontynuację makrodefinicji.

## Wywołanie makropolecień:

```
if (fPtr == NULL) {
    ERROR
}
```

```
if (fPtr == NULL) {
    ERROR;
}
```

?

# Makrodefinicje

- Jakie są wady, a jakie zalety **makrodefinicji, makra** w porównaniu do funkcji?

# Makrodefinicje

- Jakie są wady, a jakie zalety **makrodefinicji**, **makra** w porównaniu do funkcji?
- Makrodefinicje będą powiększać kod źródłowy, a zatem kod maszynowy pliku wykonywalnego naszej aplikacji.
- Zastosowanie **makra** spowoduje wklejenie fragmentu kodu we wskazane miejsce, dzięki czemu nie musimy wykonywać skoków do funkcji. Oznacza to że wykonanie jakiegoś prostego zadania, wykorzystując **makrodefinicję**, będzie szybsze niż w przypadku funkcji.

# Specyfikator inline – funkcje rozwijane (C++)

- W sytuacji prostych funkcji, może wystąpić sytuacja, że czas przełączania kontekstu (skok do kodu funkcji i powrót do miejsca wywołania) jest dłuższy niż czas wykonania samego ciała funkcji.
- W takiej sytuacji w języku C++ wprowadzono możliwość optymalizacji takich funkcji przez zastosowanie specyfikatora `inline`.
- W trakcie kompilacji, w miejsce wywołania funkcji zdefiniowanej ze specyfikatorem `inline`, może zostać przekopiowany kod ciała funkcji (podobnie jak w przypadku makra).
- Może, ponieważ specyfikator ten stanowi tylko sugestię dla kompilatora. To, że określimy funkcję jako funkcję rozwijaną (`inline`) nie oznacza, że kompilator przeprowadzi operacje podstawienia, jeśli uzna ją za nieoptymalną.

# Specyfikator inline – funkcje rozwijane

- Funkcję rozwijalną oznaczmy specyfikatorem `inline` w miejscu definicji oraz musi zostać spełniony warunek o nierozdzielaniu definicji i deklaracji, czyli definicja funkcji jest za razem jej deklaracją.
- Przykład:

```
#include <iostream>
```

```
inline int warAbsolutna (int x) {  
    return (x<0 ? -x : x);  
}
```

```
int main() {  
    int x = -5;  
    printf("Wartosc absolutna x to %d", warAbsolutna(x));  
    return 0;  
}
```

# Specyfikator inline – funkcje rozwijane

- Specyfikator `inline` stanowi tylko sugestię dla kompilatora. To, że określimy funkcję jako funkcję rozwijaną nie oznacza, że kompilator przeprowadzi operacje podstawienia, jeśli uzna ją za nieoptymalną. Kompilator nie uwzględni tej sugestii gdy:
  - funkcja jest funkcją rekurencyjną,
  - funkcja zawiera pętlę lub instrukcję `switch`,
  - funkcja jest wywoływana przed jej definicją,
  - funkcja zawiera zbyt wiele instrukcji.



# Funkcje o nieokreślonej liczbie argumentów

- Funkcje o **nieokreślonej liczbie argumentów** (ang. variadic functions) to funkcje, które posiadają zmienna lista argumentów.
- W czasie wywołania funkcje mogą przyjąć zero, jeden lub dowolną liczbę argumentów. Powszechnie znane nam funkcje takie jak: `printf()` i `scanf()`, to funkcje o nieograniczonej liczbie argumentów. W poniżej przedstawionych nagłówkach charakteryzują cię trójkropkiem na końcu listy argumentów.
- Nagłówek funkcji:  

```
int printf(const char *format,...)
int scanf(const char *format,...)
```
- W pliku nagłówkowym `<stdarg.h>` biblioteki standardowej zdefiniowane są makra: `va_start`, `va_arg`, `va_copy`, `va_end`, dzięki którym mamy możliwość zaimplementowania **zmiennej listy argumentów**.
- Zmienna lista argumentów rozpoczyna się od wartości całkowitej stanowiącej liczbę `count` argumentów z jaka będzie wywoływana funkcja. Składnia nagłówka funkcji z **nieokreśloną liczbą argumentów**:

```
typ varFun(unsigned int count,...)
```

# Funkcje o nieograniczonej liczbie argumentów

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

double average (unsigned int count, ...) {

    // va_list to typ danych używanych przez makra va_start , va_end , va_arg
    va_list args ;

    // Przekazanie argumentu count poprzedzającego zmienną listę argumentów
    va_start (args , count);

    // Inicjalizacja zmiennej lokalnej pomocniczej
    double sum = 0;

    // Pobranie kolejnej wartości ze zmiennej listy argumentów
    //(oczekiwany typ zmiennych to double)
    for ( unsigned int i = 0; i < count ; ++i)
        sum += va_arg (args , double );

    // Zakonczono iterowanie po zmiennej liście argumentów
    va_end ( args );
    return sum / count ;
}

int main () {
    const unsigned int count = 5;
    printf ("%f", average (count, 1.3, 2.5, 20.7, 3.4, -1.6));
    return 0;
}
```

Makra: `va_start`, `va_arg`, `va_copy`, `va_end`

Wartości całkowitej stanowiącej liczbę argumentów z jaka będzie wywoływana funkcja.

Tworzymy zmienną `args`, która będzie przechowywać listę argumentów.

`va_start` inicjalizuje zmienne typu `va_list`.

`va_arg` pobiera kolejne argumenty z listy zapisanej w zainicjalizowanej zmiennej typu `va_list`, w naszym przypadku `args` oraz potrzebuje informację jak ma rozumieć daną wartość czyli deskryptor typu (tu: `double`).

Ze względu że kopie argumentów odłożone są kolejno na stosie, ważne jest aby podać poprawny deskryptor typu i zachować poprawną arytmetykę wskaźników/adresów.

5.260000