Funkcje

- Funkcja jest wydzielonym fragmentem programu, który realizuje określone zadanie. Dzięki nim program może być podzielony na mniejsze, niezależne jednostki.
- Funkcję reprezentuje w programie jej *nazwa*.
- Każdy program w języku C++ ma co najmniej jedną funkcję. Musi mieć również dokładnie jedną funkcję o nazwie `main`, od której rozpoczyna się wykonywanie programu.
- Funkcja `main()` wywołuje inne funkcje, aby wykonały odpowiednie zadania programu.
- Każda funkcja składa się z czterech elementów: typu przekazywanej (zwracanej) przez funkcję wartości, nazwy funkcji, listy argumentów formalnych i treści funkcji.
- Listę argumentów formalnych ujmuje się w nawiasy `()`. Może ona być pusta lub zawierać pewną liczbę argumentów oddzielonych przecinkami.
- Treść funkcji jest ciągiem instrukcji ujętych w parę nawiasów klamrowych.
- Wykonywanie funkcji rozpoczyna się od pierwszej instrukcji po nawiasie klamrowym otwierającym `{` .
- Zakończenie wykonywania funkcji następuje po napotkaniu instrukcji `return` lub zamykającego nawiasu klamrowego `}`.
- Instrukcja `return` pozwala zwracać wartość - wynik obliczeń wykonanych w funkcji. Wartość ta jest określonego typu. Nazywamy ją *wartością przekazywaną* (*zwracaną*) *przez funkcję*.
- Jeżeli funkcja nie zwraca żadnej wartości, to mówimy, że wartość przekazywana przez funkcję jest typu `void`.

```
unsigned long silnia (unsigned n)
{
  int i, wynik=1;
  // wykonanie obliczeń
  for (i=2; i<=n; ++i)
    wynik *= i;
  // zwrócenie wartości silnia n!
  return wynik;
}

int min (int a, int b)
{
  // zwrócenie mniejszego z dwóch argumentów
  return ( a<b ? a : b);
}

void drukowanie(int n)
{
  cout << "Wynik =" << n << endl;
}
```

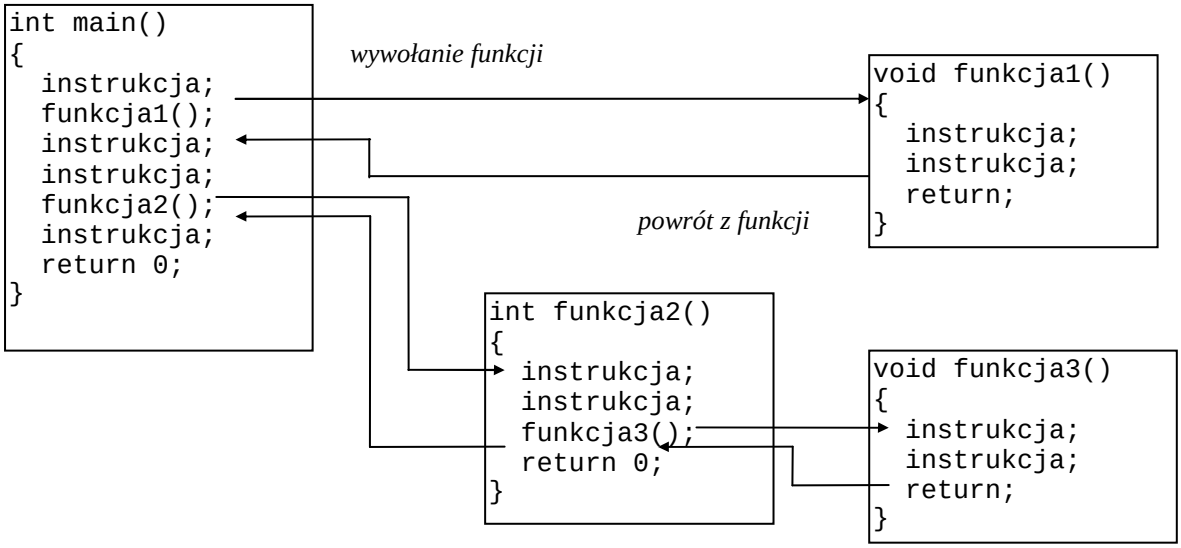
Funkcja `main()`

```
/* Przykład 1: Funkcja zwraca 0 do systemu operacyjnego */
#include <iostream.h>
int main()
{
  cout << "Nazywam się Jan Kowalskin" << endl;
  return 0;  /* wartość zwracana do systemu operacyjnego,
             typ zwracanej wartości odpowiada typowi funkcji */
}
```

- Funkcja `main()` jest funkcją szczególną. Możemy w niej opuścić `return 0`, wtedy zgodnie ze standardem ANSI/ISO C++ funkcja `main()` powinna zwracać liczbę 0. ***Niestety wiele kompilatorów nie respektuje tej zasady i jeżeli jawnie nie umieści się instrukcji `return 0`, to z funkcji zwracana jest wartość przypadkowa.***

Struktura programu

- Każda funkcja stanowi odrębną jednostkę. Nie można definiować funkcji wewnątrz innej funkcji.
- Aby wykonać instrukcje konkretnej funkcji, należy ją *wywołać* (ang. *call*)
- Wywołanie polega na podaniu nazwy funkcji wraz z nawiasami `()`, w których ewentualnie umieszczone są przekazywane do niej wartości (tzw. argumenty aktualne, czyli wartości, na których funkcja ma wykonywać obliczenia). Nawiasy `()` to operator wywołania funkcji. Pominięcie nawiasów spowoduje, że funkcja nie zostanie wywołana.
- Wywołanie funkcji powoduje przekazanie do niej sterowania - wykonywanie bieżącej funkcji jest zawieszane.
- Po zakończeniu ostatniej instrukcji wywołanej funkcji, wznawia się wykonywanie uprzednio zawieszonej funkcji od punktu następującego bezpośrednio po wywołaniu.



// Przykład: Program złożony z kilku funkcji

```
#include <iostream.h>
void wczytywanie() {
  cout << "Czytam dane" << endl;
}
void sortowanie() {
  cout << "Sortuje dane" << endl;
}
void drukowanie() {
  cout << "Drukuje wyniki" << endl;
}
int main () {
  wczytywanie(); // wywołanie funkcji wczytywanie()
  sortowanie();  // wywołanie funkcji sortowanie()
  drukowanie();   // wywołanie funkcji drukowanie()
  cout << "Koniec programu" << endl;
  return 0;
}
```

Definicja funkcji

- Składnia funkcji ma postać:

```
typ_zwracanej_wartości nazwa_funkcji(arg1, arg2, ..., argN)
{
  deklaracja;
  deklaracja;
  ..........
  instrukcja;
  instrukcja;
  ..........
  deklaracja;
  deklaracja;
..........
  instrukcja;
  instrukcja;
..........
  return wyrażenie;
}
```

Uwaga: *w języku C wszystkie deklaracje muszą być zgrupowane razem, na początku funkcji (dokładniej bloku wyznaczanego {}). Każda funkcja ma niepowtarzalną nazwę.*

gdzie:

nazwa_funkcji	Identyfikuje funkcję w programie. Każda funkcja ma nazwę. Nazwa funkcji powinna efektywnie wyrażać zadanie realizowane przez funkcję.
typ_zwracanej_wartości	Określa typ wartości zwracanej do funkcji wywołującej (lub do systemu operacyjnego) jako wynik wykonania funkcji. W języku C++ nie można pomijać *typu_zwracanej_wartości*. Jeżeli funkcja nie zwraca wartości, należy jej przypisać typ `void` . *Uwaga*: W języku C pominięcie typu zwracanej wartości powodowało, że kompilator przypisywał funkcji typ `int`. W C++ zawsze trzeba podać albo typ zwracanej wartości albo `void`.
(*arg1,arg2,...,argN*)	Określa argumenty (parametry) otrzymywane przez funkcję podczas jej wywoływania. Pominięcie *listy_argumentów* (puste nawiasy) oznacza, że do funkcji nie są przekazywane żadne wartości z funkcji wywołującej.
return wyrażenie	Określa wartość zwracaną przez funkcję. Pominięcie wyrażenia (sama instrukcja `return`) oznacza, że funkcja nie zwraca wartości. Można tak postępować tylko z funkcją `void`.

Przykład 1: Funkcja zwraca wartość – jest nią kwadrat przesłanego do funkcji argumentu.

```
int kwadrat(int a)
{
  return a*a;
}
```

Przykład 2: Funkcja nie zwraca wartości – jej zadaniem jest tylko wyświetlenie komunikatu.

```
void komunikat() {
  cout << "Nazywam się Jan Kowalski" << endl;
}
```

Przykład 3: Podczas opracowywania programu przydatna może być funkcja pusta, do późniejszego wypełnienia:

```
void atrapa() { } /* nic nie robi; do wypełnienia później */
```

Kolejność umieszczenia definicji funkcji w programie

- Kolejność umieszczenia definicji funkcji w programie nie jest dowolna.
- Przed pierwszym użyciem funkcji (wywołaniem) kompilator *musi* znać typ wartości zwracanej przez funkcję, oraz liczbę i typy argumentów akceptowanych przez funkcję. Może wtedy przeprowadzić sprawdzenie poprawności wywołania funkcji.
- Rozwiązanie:
  - metoda 1 - właściwa kolejność definicji funkcji w programie (nie jest możliwa, jeśli funkcje się wzajemnie wywołują)
  - metoda 2 - umieszczenie na początku programu *prototypów* (deklaracji, zapowiedzi) funkcji: wiążą one identyfikator funkcji z listą argumentów i typem wartości zwracanej przez funkcję .

Metoda 1

```
int funkcja1(int n)
{
  instrukcja;
  instrukcja;
  return x;
}
```

```
void funkcja3()
{
  instrukcja;
  instrukcja;
}
```

```
void funkcja2(double x, int n)
{
  instrukcja;
  instrukcja;
  funkcja3();
  return y;
}
```

```
int main()
{
  double x;
  int n;
  instrukcja;
  funkcja1(x);
  instrukcja;
  instrukcja;
  funkcja2(x,n);
  instrukcja;
  return 0;
}
```

Metoda 2

```
int funkcja1(int);
double funkcja2(double, int);
void funkcja3();
```

```
int main()
{
  int n;
  double x;
  instrukcja;
  funkcja1(x);
  instrukcja;
  instrukcja;
  funkcja2(x,n);
  instrukcja;
  return 0;
}
```

```
int funkcja1(int n)
{
  instrukcja;
  instrukcja;
  return x;
}
```

```
double funkcja2(double x, int n)
{
  instrukcja;
  instrukcja;
  funkcja3();
  return y;
}
```

```
void funkcja3()
{
  instrukcja;
  instrukcja;
  return;
}
```

Prototyp funkcji

- Składnia:
  typ_zwracanej_wartości nazwa_funkcji(*arg1, arg2, ..., argN*);
- Prototyp określa:
  - nazwę funkcji,
  - typ wartości zwracanej przez funkcję,
  - liczbę argumentów, jakie funkcja spodziewa się otrzymać,
  - typy argumentów,
  - kolejność argumentów.
- Użycie prototypu w programie przed pierwszym wywołaniem funkcji, pozwala umieścić definicje funkcji w dowolnej kolejności.
- Wartości argumentów aktualnych (podanych w wywołaniu funkcji) są przekształcane do typów argumentów formalnych w prototypie funkcji.

Przykład:

```
#include <iostream.h>
int dodaj_liczby(int a, int b);  // prototyp:deklaracja funkcji

int main() {
  int x1=3, x2=5;
  cout << "Suma=" << dodaj_liczby(x1,x2) << endl;
  return 0;
}

int dodaj_liczby(int a, int b)  // definicja funkcji
{
  return(a+b);
}
```

Uwaga

- W prototypie podaje się liczbę argumentów, jakie funkcja spodziewa się otrzymać oraz typy tych argumentów. Nazwa zmiennej jest ignorowana. Równoważne są zatem zapisy:

```
int dodaj_liczby(int a, int b);
int dodaj_liczby(int, int);
```

- Nazwy zmiennych w prototypie podaje się w celu zwiększenia czytelności programu. Czytelniejsza będzie również informacja o ewentualnych błędach i/lub ostrzeżenia z kompilatora.
- ***Jeżeli definicja funkcji znajduje się przed pierwszym jej użyciem, wtedy definicja funkcji działa jako prototyp funkcji.***
- Często spotykane błędy:
  - brak średnika na końcu prototypu funkcji,
  - średnik w deklaracji funkcji,
  - wywołanie funkcji nie pasującej do prototypu.

Działanie prototypu - sprawdzanie zgodności typów

- W języku C++ rygorystycznie przestrzega się zgodności typów. Podczas kompilacji sprawdzane są typy argumentów każdego wywołania funkcji. W wypadku niezgodności typu argumentu użytego w wywołaniu funkcji z typem odpowiedniego argumentu formalnego, kompilator usiłuje wykonać niejawną konwersję, jeżeli tego nie potrafi zrobić, to wyprowadza komunikat o błędzie.

Przykład 1. Niezgodność typów argumentów, program nie skompiluje się

```
#include <iostream.h>

double podziel(int x, double y);
double dodaj(int, double);
int main()
{
  int x=1, y=2;
  double z[2]={2.,5.};
  podziel(x,z);
  dodaj(x,z);
  return 0;
}

double podziel(int x, double y)
{
  cout << y/x;
  return 0;
}
double dodaj(int x, double y)
{
  cout << (y+x);
  return 0;
}
```

Błąd:
Type mismatch in parameter 'y' in call to 'podziel'

Błąd:
Type mismatch in parameter 2 in call to 'dodaj'

- Kompilator sprawdza również, czy liczba argumentów użytych w wywołaniu funkcji odpowiada liczbie argumentów formalnych w definicji funkcji. Jeśli w wywołaniu występuje niewłaściwa liczba argumentów, wyprowadzany jest komunikat o błędzie.

Przykład 2. Niezgodność liczby argumentów. Program nie skompiluje się.

```
#include <iostream.h>

double podziel(int x, double y);
int main()
{
  podziel(1,2.0,3);
  return 0;
}

double podziel(int x, double y)
{
  cout << y/x << endl;
  return 0;
}
```

Błąd:
Extra parameter in call to 'podziel'

Prototypy funkcji bez parametrów w języku C++

Jeżeli funkcja nie ma żadnych argumentów, to może być deklarowana następująco:

```
int fun();     // pusta lista parametrów formalnych - zalecane
int fun(void); // wstawienie zamiast listy parametrów typu void
```

. Przekazywanie argumentów funkcji

- Wywołując funkcję, podajemy jej nazwę oraz w nawiasach listę argumentów aktualnych. Argumenty te odpowiadają argumentom formalnym umieszczonym w definicji funkcji, to znaczy są umieszczone na tej samej pozycji listy argumentów.
- Przykład:

```
#include <iostream.h>
// Funkcja kwadrat: obliczanie kwadratu liczby
int kwadrat(int liczba)
{
  int wynik;
  wynik = liczba*liczba;
  return wynik;
}

int main () {
  int liczba;
  cout << "Podaj liczbę całkowitą: ";
  cin >> liczba;
  cout << "Kwadrat wynosi: " << kwadrat(liczba) << endl;
}
```

argument formalny

Definicja funkcji

wartość zwracana

argument aktualny

wartość funkcji dla aktualnego argumentu

- Domyślnym sposobem inicjowania pamięci argumentów formalnych jest kopiowania wartości argumentów aktualnych do pamięci argumentów formalnych. Technika ta nazywa się *przekazywaniem argumentów przez wartość* (ang. *pass-by-value*).
- Funkcja ma dostęp tylko do lokalnych kopii argumentów do niej przekazanych. Zmiany tych wartości, będą widziane tylko w funkcji i nie będą widoczne w funkcji wywołującej. Po zakończeniu wykonywania funkcji, kopie argumentów są usuwane.
- Argumentem funkcji może być również wyrażenie:
-

```
#include <iostream.h>
#include <math.h>

void drukuj(double wynik)
{
  cout <<"Wynik = " << wynik << endl;
}

int main()
{
  drukuj(2*sin(M_PI/6)); // M_PI stała zdefiniowana w math.h
  return 0;
}
```

Kiedy przekazywanie przez wartość nie wystarcza?

- Zasada przekazywania argumentów przez wartość jest bezpieczna i mało pracochłonna, ponieważ programista nie musi zajmować się zapamiętaniem i odtworzeniem wartości argumentów aktualnych przekazywanych funkcji. Gdyby nie było tego mechanizmu, wówczas każdy argument (którego nie zadeklarowano z modyfikatorem `const`) mógłby się zmieniać z każdym wywołaniem funkcji. Dlatego też zasada ta została wybrana jako domyślny mechanizm przekazywania wartości argumentów.
- Nie zawsze jednak przekazywanie przez wartość jest odpowiednie. Przykłady takich sytuacji:
  - gdy jako argument jest przekazywany obiekt, który zajmuje dużo pamięci - kopiowanie jest kosztowne.
  - gdy wartości przekazane za pomocą argumentów muszą być zmieniane przez funkcję.

```
#include <iostream.h>
void zamien(int, int);

int main() {
  int i=10, j=20;
  cout << "Przed wywolaniem funkcji \ti=" << i << "\tj=" << j <<
endl;
  zamien(i,j);
  cout << "Po wywolaniu funkcji \ti=" << i << "\tj=" << j << endl;
  return 0;
}
// ta funkcja nie zmienia wartości argumentów aktualnych
void zamien(int a, int b) {
  cout << "Przed zamiana \ti=" << i << "\tj=" << j << endl;
  int tymcz=a;
  b=a;
  a=tymcz;
  cout << "Po zamianie \ti=" << i << "\tj=" << j << endl;
}
```

Wynik działania:

```
Przed wywolaniem funkcji  i=10  j=20
Przed zamiana             i=10  j=20
Po zamianie               i=20  j=10
Po wywolaniu              i=10  j=20
```

Jak zmienić wartość argumentów przekazywanych do funkcji?

- Aby funkcja mogła zmienić wartość argumentu, trzeba zastosować specjalną technikę: umożliwić funkcji dostęp do argumentu za pomocą jego adresu w pamięci.
- Technika ta jest omówiona w części „*Wskaźniki i referencje*”

Funkcje o zmiennej liczbie argumentów

- Deklaracja funkcji o zmiennej liczbie parametrów:

  typ_wartości nazwa_funkcji(*parametry_ustalone*, ...)

- Funkcja taka musi pobierać co najmniej jeden stały argument. Informacja o zmiennej liczbie parametrów jest zawarta w symbolu trzech kropek.
- Dostęp do parametrów nie ustalonych uzyskuje się za pomocą następujących makr umieszczonych w pliku nagłówkowym `stdarg.h`:
  - `va_start (va_list wsk_arg, ostatni_arg)` - przekazuje się nazwę ostatniego stałego parametru
  - `va_arg (va_list wsk_arg, typ)` - pobiera wartość kolejnego zmiennego argumentu określonego typu
  - `va_end(va_list wsk_arg)` - zakończenie pobierania (uporządkowanie stosu)

- Przykład: Obliczanie wartości wielomianu określonego stopnia

```
#include <stdio.h>  // wyjście w stylu C
#include <stdarg.h> // dla zmiennej liczby argumentów
#include <math.h>   // dla pow()

double wielomian(double x, int st, ...);
int main()
{
  printf("%lf\n", wielomian(1.0,2,3.,4.,5.));
  printf("%lf\n", wielomian(2.0,3,1.,2.,3.,4.));
  return 0;
}

double wielomian(double x, int st, ...)
{
  double wartosc=0;
  va_list p;
  va_start(p,st);
  for (; st; --st)
    wartosc += va_arg(p,double) * pow(x,st);
  wartosc += va_arg(p,double);
  va_end(p);
  return wartosc;
}
```

Tablice jako argumenty funkcji

- W języku C++ tablica jest traktowana specjalnie. Nigdy nie jest przekazywana do funkcji przez wartość. Zawsze do funkcji przekazywany jest adres pierwszego elementu tablicy. Oznacza to, że zmiany wykonane wewnątrz wywołanej funkcji nie dotyczą lokalnej kopii, ale tablicy będącej argumentem aktualnym.
- Funkcja, do której ma być przesłana tablica nie zna jej rzeczywistego rozmiaru. Informacja ta musi być w jakiś sposób dostarczona funkcji.

- Przykład 1. Drukowanie tablicy jest realizowane w funkcji `drukuj()`. Rozmiar tablicy jest przypisane stałej `MAX`. Jest to stała, która jest zadeklarowana na zewnątrz funkcji, po to aby była rozpoznawana w całym programie.

```
#include <iostream.h>
const int MAX=5;
void drukuj(int tablica[]);
int main() {
  int t[MAX],i;
   for(i=0;i<MAX;++i) t[i]=i;
  drukuj(t);

  return 0;
}
void drukuj(int t[]) {
  int i;
  for(i=0;i<MAX;i++) cout << t[i] <<"\t";
  cout << endl;
}
```

przekazywany jest adres pierwszego elementu tablicy

argumentem formalnym jest tablica, sygnalizowane jest to za pomocą `[]`

Przykład 2. Zerowanie elementów tablicy w funkcji `czysc()`. Rozmiar tablicy podany jest w postaci dodatkowego argumentu.

```
void czysc(int t[], int l_elementow)
{
  for(int i=0;i<l_elementow;++i) t[i]=0;
}

int main() {
  int tab1[10];
  int tab2[20];

  czysc(tab1,10); //wypełni zerami całą tablicę tab1
  czysc(tab2,15); //wypełni zerami pierwsze 15 elementów tab2
  return(0);
}
```

Komentarz:
Można również zapisać tę funkcję z użyciem pętli `while`:

```
void czysc(int t[], int l_elementow) {
  while (l_elementow>0)
    t[--l_elementow]=0;
}
```

Jest to skrócony zapis funkcji:

```
void czysc(int t[], int l_elementow)
{
  while (l_elementow>0) {
    l_elementow--;
    t[l_elementow]=0;
  }
}
```

Przykład 3: Obliczenie wartości wielomianu $w_n(x)= a_0x^n+ a_1x^{n-1}+...+a_{n-1}x+ a_n$ dla danej wartości x.

<u>*Algorytm Hornera*</u>:
Dane: n, $a_0,a_1,...,a_n$ – stopień i współczynniki wielomianu, x - argument
Wynik: wartość wielomianu $w_n(x)= a_0x^n+ a_1x^{n-1}+...+a_{n-1}x+ a_n$
Obliczenia: $y= a_0$
$y = yx+a_i$ *dla* $i=1,2,...,n$

```
// Obliczenie wartości wielomianu (schemat Hornera)
// w[0] zawiera współczynnik przy najwyższej potędze
double W(double x, double a[], int n)
{
  double s=a[0];
  for (int i=1; i<=n; ++i)
    s = s*x +a[i];
  return s;
}
```

- Przykład 4: ***Jeżeli argumentem przekazywanym do funkcji jest element tablicy, jest on traktowany tak jak zwykła zmienna***.

```
#include <iostream.h>
void drukuj(int element);
int main() {
  int t[5],i;
  for(i=0;i<5;++i) t[i]=i;

  for(i=0;i<5;++i) drukuj(t[i]);
  cout << endl;
  return 0;
}
void drukuj(int element)
{
  cout << element;
}
```

przekazywany jest element tablicy typu `int`

argumentem formalnym jest zmienna typu `int`

6.7. Wartość przekazywana przez funkcję - instrukcja `return`

- Do przekazywania wartości obliczonej w funkcji (mówi się również *zwrócenia* wartości) służy instrukcja `return`.
- Instrukcja `return` ma dwie postaci:

```
return wyrażenie;
return; // tylko w funkcji typu void
```

- Pominięcie w funkcji instrukcji `return` powoduje powrót (bez zwracania wartości) po osiągnięciu nawiasu klamrowego zamykającego funkcję. Dotyczy tylko funkcji typu `void`.
- Przykłady:

```
// zwrócenie wartości bezwzględnej argumentu
int Abs(int x)
{

  return ( x<0 ? -x : x );
}

// zwrócenie większej z wartości dwóch argumentów
int Max(int a, int b)
{
  return ( a>b ? a : b )
}

// zwócenie największego wspólnego dzielnika
int Nwd(int a, int b)
{
  int r;
  while (m) // skrót dla while( m != 0)
  {
    r = n % m;
    n = m;
    m = r;
  }
  return n;
}

// Kopiowanie tablicy (Funkcja jest typu void, więc można pominąć return).
  void kopiuj(double skad[], double dokad[], int ile)
  {
  // nie ma nic do kopiowania
    if (ile == 0)
      return;
  // kopiowanie
    for (int i=0; i<ile; ++i)
      dokad[i] = skad[i];
  }
```

Wykorzystanie wartości zwracanej przez funkcję

- Można wyróżnić trzy grupy funkcji:
  - funkcje obliczeniowe - zwracają wynik obliczeń wykonanych na przekazanych do nich argumentach, przykład: `sqrt()` – zwraca pierwiastek kwadratowy, `pow()`- zwraca liczbę podniesioną do określonej potęgi
  - funkcje wykonujące działania na danych i zwracające wartość informującą o tym, czy działanie zakończyło się pomyślnie, przykład: funkcja wejścia w stylu języka C `scanf()`, zwraca ilość wczytanych danych lub -1 w przypadku błędu lub końca pliku
  - funkcje wykonujące działania i nie zwracające żadnej wartości powrotnej; są one deklarowane jako funkcje typu `void`; wynikiem działania funkcji jest na przykład tylko wydrukowanie komunikatu lub zmiana wartości przekazanej jako argument, np. zawartości tablicy.
- Jeżeli funkcja zwraca wartość, można ją umieszczać jako argument w wyrażeniach:
  - umieścić jako wyrażenie proste, np. `cout << "Max=" << Max(a,b) << endl;`
  - przypisać zmiennej, np.: `y=sqrt(x);`
  - wykorzystać w instrukcji warunkowej, np. `if (Max(x,y) > 10) cout << "większy od 10";`

Przykłady:

```
#include <iostream.h>
int dodaj_liczby(int a, int b)
{
  return(a+b);
}
```

```
// krótszy zapis funkcji
int dodaj_liczby(int a, int b)
{
  int wynik;
  wynik=a+b;
  return(wynik);
}
```

jako argument w wyrażeniu

```
// Przykład A.
int main() {
  cout << "3+5=" << dodaj_liczby(3,5)
       << endl;
  return 0;
}
// Przykład B.
int main() {
  int x1, x2;
  x1=3;
  x2=5;
  cout << "3+5=" <<
    dodaj_liczby(x1,x2) << endl;
  return 0;
}
```

jako argument w wyrażeniu

```
// Przykład C.
int main() {
  int x1,x2,suma;
  x1=3;
  x2=5;
  suma=dodaj_liczby(x1,x2);
  cout << "Razem: " << suma << endl;
  return 0;
}
```

przypisanie wartości zwracanej przez funkcję do zmiennej

```
// Przykład D.
int main() {
  int x1,x2;
  x1=3;
  x2=5;
  if (dodaj_liczby(x1,x2)< 10)
    cout << "Za malo." << endl;
  return 0;
}
```

wykorzystanie wartości zwracanej przez funkcję w instrukcji warunkowej