Wstęp do informatyki- wykład 8

Manipulatory

Funkcje

Treści prezentowane w wykładzie zostały oparte o:

- S. Prata, Język C++. Szkoła programowania. Wydanie VI, Helion, 2012
- *www.cplusplus.com*
- Jerzy Grębosz, Opus magnum C++11, Helion, 2017
- B. Stroustrup, Język C++. Kompendium wiedzy. Wydanie IV, Helion, 2014
- S. B. Lippman, J. Lajoie, Podstawy języka C++, WNT, Warszawa 2003.

Pętle for- przykład- trójkąt- choinka

```
/* choinka o wysokości 5
    ^                i=1    4 spacje      1 ^
   ^^^               i=2    3 spacje      3 ^
  ^^^^^              i=3    2 spacje      5 ^
 ^^^^^^^             i=4    1 spacja      7 ^
^^^^^^^^^            i=5    0 spacji      9 ^
i-ty wiersz:  5-i spacji  2*i-1 ^
*/
for (int i = 1; i <= 5; i++)
{
    //dla i-tego wiersza:  5-i spacji
    for (int j = 1; j <= 5 - i; j++) //spacje
        cout<<' ';
    //dla i-tego wiersza:  2*i-1 ^
    for (int j = 1; j <= i * 2 - 1; j++) //^
        cout<<'^';
    //nowa linia
    cout << endl;
}
```

Formatowanie – manipulatory i flagi

- Sposób formatowania danych w operacjach wyjścia można zmienić. Służą do tego **flagi (znaczniki) formatowania** (*format flags*) i **manipulatory**.
- **Manipulatory** są to funkcje zdefiniowane w klasie `ios` i wywoływane poprzez podanie ich nazw jako elementów wstawianych do lub wyjmowanych ze strumienia.

Manipulatory bezargumentowe

Manipulatory bezargumentowe wstawia się do strumienia nie podając nawiasów. Mają one nazwy takie jak flagi.

- `fixed, scientific` — ustawiają formatowanie wyprowadzanych liczb zmiennopozycyjnych.
- W ***notacji naukowej*** (`scientific`) wypisywana jest jedna cyfra przed kropką, maksymalnie tyle cyfr po kropce, ile wynosi aktualna precyzja, następnie litera 'e' i wykładnik potęgi dziesięciu, przez którą należy pomnożyć liczbę znajdującą się przed literą 'e'. Jeśli liczba jest ujemna, to przed nią wypisywany jest znak minus. Na przykład `1.123456e2` oznacza `112.3456`, natomiast `-1.123456e-3` oznacza `-0.001123456`.

Formatowanie – manipulatory bezargumentowe

- ***Format normalny*** (`fixed`) oznacza wypisywanie maksymalnie tylu cyfr po kropce dziesiętnej, ile wynosi aktualna precyzja.
- Jeśli żaden format nie jest ustawiony, to użyty będzie ***format ogólny***, który pozostawia implementacji sposób zapisu (naukowy lub normalny) w zależności od wartości liczby tak, żeby zapis z ustaloną precyzją zajmował jak najmniej miejsca

```
double x = 123.4567891;
double y = 12345678.91;
//format ogólny
cout<<  x << endl;  //  123.457
cout<<  y << endl;  //  1.23457e+07
cout<< fixed << y << endl; //12345678.910000
cout<< scientific << x << endl;  // 1.234568e+02
```

Formatowanie - manipulatory

- **`left, right`** — ustawiają sposób wyrównywania (justowania) wyprowadzanych danych: do lewej, prawej.
- **`endl`** — powoduje wysłanie do strumienia wyjściowego znaku końca linii i opróżnienie bufora związanego z tym strumieniem, czyli natychmiastowe wyprowadzenie znaków z bufora do miejsca przeznaczenia (na ekran, do pliku itd).
- **`boolalpha, noboolalpha`**– wstawianie do strumienia wynikowego wartości logicznych w postaci słów `false` i `true` albo liczb `0` lub `1`

Manipulatory z argumentami

Istnieją też **manipulatory argumentowe**. Stosuje się je analogicznie jak manipulatory bezargumentowe, ale wymagają one argumentów, które, jak dla zwykłych funkcji, podaje się w nawiasach.

Aby używać predefiniowanych manipulatorów argumentowych, należy dołączyć plik nagłówkowy `#include<iomanip>`.

Predefiniowane manipulatory argumentowe zwracają, jak wszystkie manipulatory, referencję do strumienia do którego zostały wstawione

Manipulatory z argumentami

- **`setw(int szer)`** — ustawia szerokość pola dla najbliższej operacji na strumieniu, przy czym określana jest minimalna szerokość pola: jeśli dana zajmuje więcej znaków, to odpowiednie pole zostanie zwiększone. Domyślną wartością szerokości pola jest zero, czyli każda wypisywana dana zajmie tyle znaków, ile jest potrzebne, ale nie więcej

```
int e = 1, f = 10;
cout<< "--------" << endl
    << setw(4) << e
    << setw(4) << f <<endl;
cout<< left << setw(4) << e
    << setw(4) << f << endl <<"--------";
```

```
--------
   1  10
1   10
--------
```

Manipulatory z argumentami

`setprecision(int prec)` — ustawia precyzję wyprowadzanych liczb zmiennopozycyjnych.

- W przypadku formatu *`fixed`* jest to ilość cyfr po kropce dziesiętnej
- W przypadku formatu naukowego *`scientific`* jest to ilość cyfr mantysy po kropce
- Dla formatu ogólnego jest to całkowita liczba cyfr liczby, chyba, że liczba ma mniej cyfr.
- Domyślnie precyzja ustawiona jest na 6.

Manipulatory z argumentami

```
#include <iostream>      // std::cout, std::fixed
#include <iomanip>       // std::setprecision
using namespace std;
int main () {
  double d = 3.14159,  g = 1234567.89;
  cout << g << endl;    //1.23457e+06
  //format ogólny
  cout << setprecision(5) << g << '\n'; //1.2346e+006
  cout << setprecision(5) << d << '\n';  //3.1416
  cout << setprecision(9) << d << '\n';  //3.14159
  cout << fixed; //format z .
  cout << setprecision(5) << d << '\n';  //3.14159
  cout << setprecision(8) << d<<'\n';  //3.14159000
  cout << setprecision(5) << g <<'\n';
//1234567.89000
}
```

Funkcje - wstęp

- Funkcje (podprogramy) - są definicjami instrukcji, które na podstawie danych wejściowych (argumentów) dostarczają w miejscu ich użycia (wywołania) wartości określonego typu.

- Funkcje są więc sposobem na realizację tego samego czy podobnego zadania wielokrotnie, bez potrzeby wielokrotnego powtarzania w naszym kodzie tych samych sekwencji instrukcji.

- Jeśli np. napiszemy funkcję obliczającą pole koła na podstawie zadanego promienia – to tak, jakbyśmy język programowania wyposażyli w nową instrukcję umiejącą właśnie to obliczać. Od tej pory – ile razy w programie potrzebujemy obliczyć pole koła – wywołujemy naszą funkcję.

Funkcje - wstęp

- Funkcję wywołuje się przez podanie jej nazwy i umieszczonych w nawiasie argumentów.
- Wywołanie - użycie funkcji w tekście programu musi być leksykalnie poprzedzone jej definicją lub przynajmniej deklaracją, tj. definicja lub deklaracja funkcji musi wystąpić w tekście programu wcześniej niż jej jakiekolwiek użycie.

Funkcje – deklaracja, definicja i wywołanie

Prosty przykładowy program zawierający funkcję:

```
#include < iostream>
using namespace std;
int suma (int x, int y);  //deklaracja funkcji
int main()
{
      int a = 10, b = 20;
      int c = suma (a, b);   //wywołanie funkcji
      cout << c;
}
int suma (int x, int y)   //definicja funkcji
{
    return x + y; //funkcja zwraca sumę argumentów
}
```

Funkcje – deklaracja, definicja i wywołanie

- Funkcja ma swoją **nazwę** `suma`, która ją identyfikuje. Podobnie jak dla innych nazw występujących w programie – przed pierwszym odwołaniem się do nazwy wymagana jest jej **deklaracja**.
- Deklaracja taka mówi kompilatorowi:
  `suma` jest funkcją wywoływaną z dwoma argumentami typu `int`, a zwracającą jako rezultat wartość typu `int`.
- Przed odwołaniem się do nazwy wymagana jest deklaracja, ale niekoniecznie od razu **definicja**. Sama funkcja może być zdefiniowana później, nawet w zupełnie innym pliku. Definicja zawiera treść funkcji ujętą w `{}` tj. wszystkie instrukcje wykonywane w ramach tej funkcji (tzw. **ciało funkcji**)
- **Wywołanie** funkcji to po prostu napisanie jej nazwy łącznie z nawiasem, gdzie znajdują się argumenty przesyłane do funkcji.

Funkcje – definiowanie funkcji

Funkcje można podzielić na dwie grupy: **niezwracające wartości(bezrezultatowe)** i **zwracające wartość(rezultatowe)**.

Funkcje, które **nie zwracają wartości** są typu pustego **void** *(pusty, próżny)* i zapisujemy je następująco:

```
void nazwaFunkcji(listaParametrów)  //nagłówek f-cji
{
     instrukcje   //treść f-cji
     return; //opcjonalnie
}
```

`listaParametrów` określa liczbę i typy parametrów przekazywanych funkcji

Opcjonalna instrukcja **`return`**; oznacza koniec funkcji, jeśli jej brak funkcja kończy się tam, gdzie zamykający ją nawias klamrowy.

Funkcje – definiowanie funkcji – funkcje typu void

Funkcji typu void używa się m.in. w przypadku, gdy zadaniem funkcji jest wyświetlenie - prezentacja jakichś informacji, np. funkcja pokazująca napis "Czesc! " n razy:

```
void powitanie(int n)//brak zwracanej wartości
{
    for(int i = 0; i < n; i++)
        cout << "Czesc! ";
    cout << endl;
    return; //opcjonalnie
}
```

Funkcja spodziewa się jednej wartości typu int.

Dla parametru n >0 wyświetli n razy napis "Czesc! ", dla n<=0 nie robi nic.

Funkcje – wywołanie funkcji – funkcje typu void

Wywołanie funkcji typu void:

Funkcje typu void wywołujemy w osobnej linii używając jej nazwy i podając parametry, a całość kończąc średnikiem.

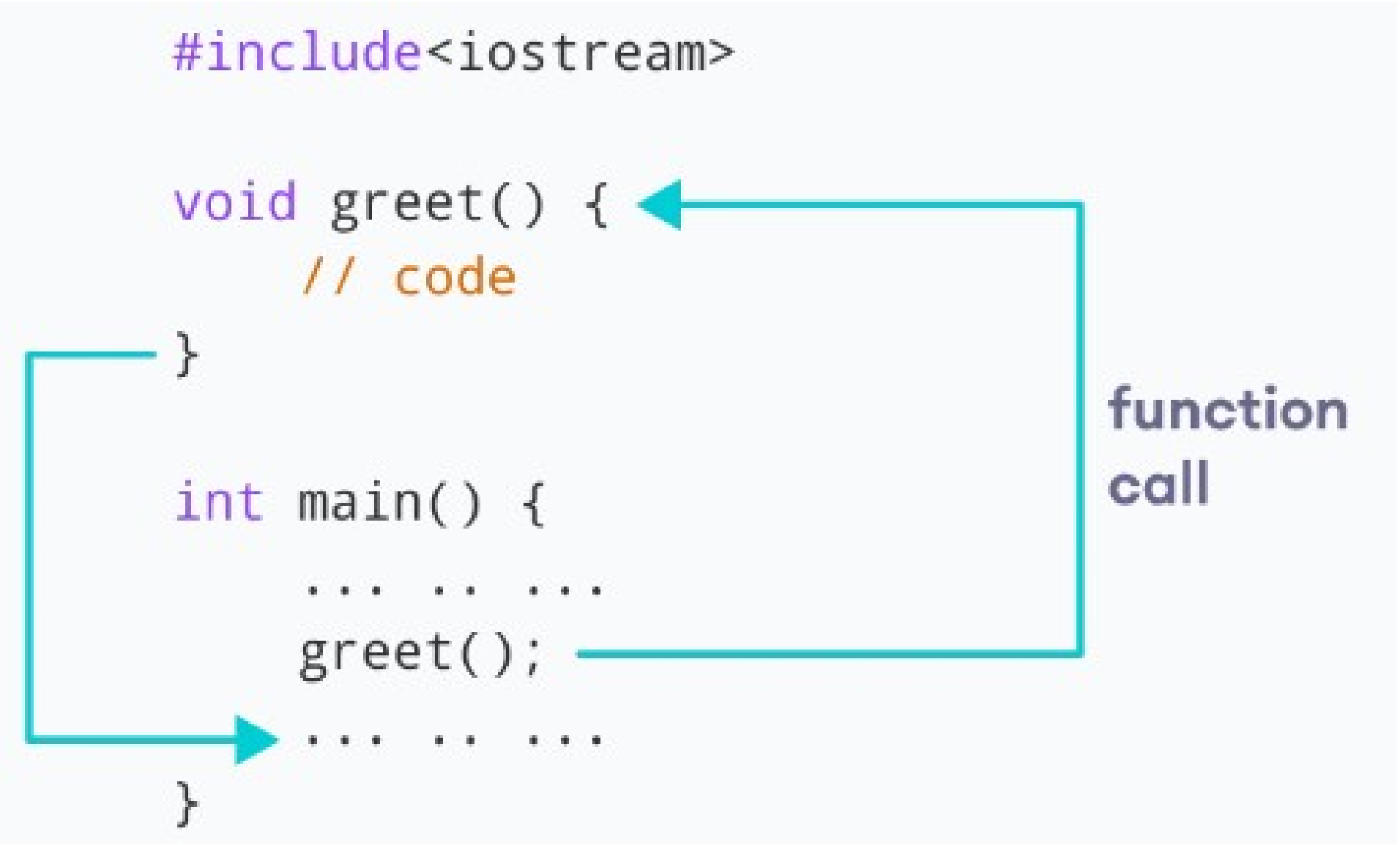
```
#include<iostream>
using namespace std;
void powitanie(int n); //deklaracja funkcji


int main()
{
    powitanie(5); //instrukcja wywołania funkcji
} //na konsoli:Czesc! Czesc! Czesc! Czesc! Czesc!


void powitanie(int n)//definicja funkcji
{
    for(int i = 0; i < n; i++)
        cout << "Czesc! ";
    cout << endl;
}
```

Funkcje – wywołanie funkcji – funkcje typu void

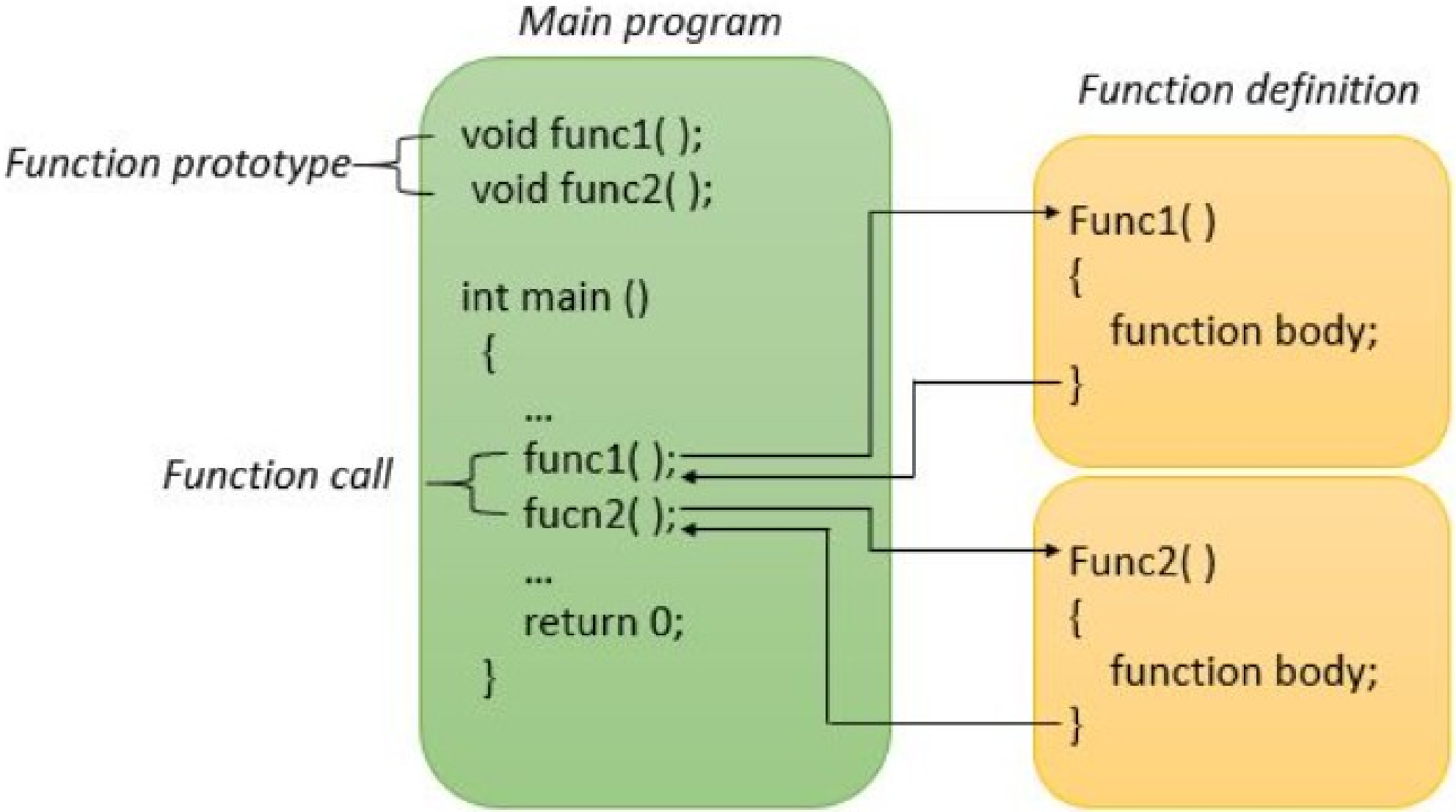
Definicja funkcji przed main: Często zamiast deklaracji funkcji przed **main**, a definicji poniżej **main**, przed funkcją **main** wstawiane są całe definicje funkcji.

```
#include<iostream>
using namespace std;


void powitanie(int n){ //definicja funkcji
    for(int i = 0; i < n; i++)
        cout << "Czesc! ";
    cout << endl;
}


int main()
{
    powitanie(5); //instrukcja wywołania funkcji
} //na konsoli:Czesc! Czesc! Czesc! Czesc! Czesc!
```

Funkcje – wywołanie funkcji – funkcje typu void



Funkcje – wywołanie funkcji – funkcje typu `void`



Struktura programu z funkcjami

```
//Structure of C++ program
#include <iostream>
using namespace std;
return_type function_name(parameter_list); //function prototype

void main()
{
  ........
  function_name();   //function call
  ........
}

return_type function_name(parameter_list)  //function defintion
{
  ........
  function definition
  ........
}
```

Definiowanie funkcji – funkcje zwracające wartość

Jeśli typem zwracanym nie jest `void`, to funkcję nazywamy funkcją **rezultatową**, funkcje takie generują wartość zwracaną do funkcji ją wywołującej, np. funkcja `sqrt(9.0)` zwraca `3,0`.

Ogólna postać definicji takiej funkcji:

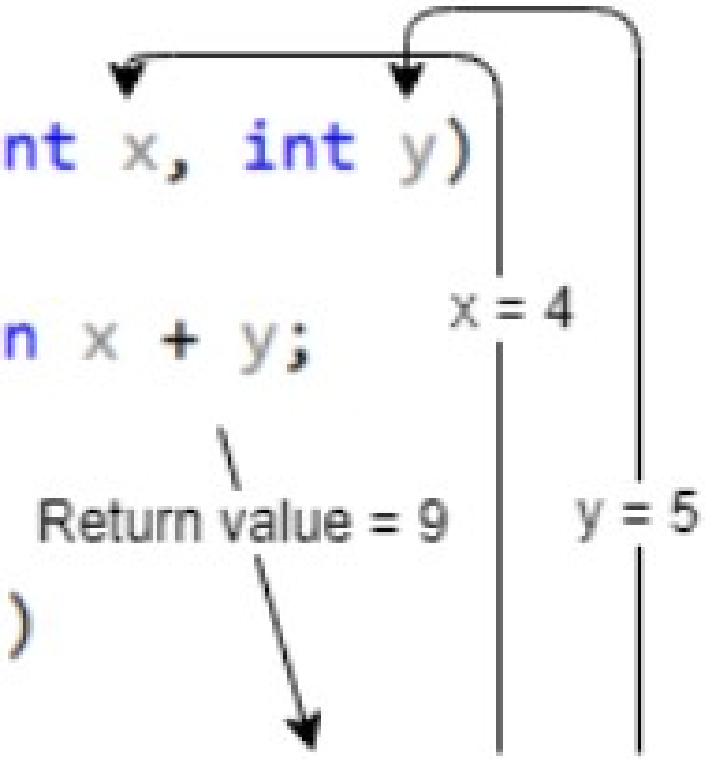
```
nazwaTypu   nazwaFunkcji(listaParametrów)
{
  instrukcje;
  return wartość; //wartość jest rzutowana
                  //na typ nazwaTypu
}
```

Funkcje zwracające wartość muszą mieć instrukcję **`return`** z wartością, która ma być zwrócona do funkcji wywołującej.

Sama zwracana `wartość` może być stałą, zmienną lub wyrażeniem. Typ tej wartości musi dać się rzutować (konwertować) na typ `nazwaTypu`, np. `int` na `double`. Następnie funkcja zwraca tak skonwertowaną wartość.

Funkcje zwracające wartość

```
1   #include <iostream>
2
3   int add(int x, int y)
4   {
5       return x + y;
6   }
7
8   int main()
9   {
10      std::cout << add(4, 5) << std::endl;
11      return 0;
12  }
```



Definiowanie funkcji – funkcje zwracające wartość

Nagłówek funkcji:

```
nazwaTypu   nazwaFunkcji(listaParametrów)
```

Typ wartości zwracanej (`nazwaTypu`):

W C++ funkcja nie może zwracać tablicy. Wszystko inne jest dopuszczalne - liczby całkowite, liczby zmiennoprzecinkowe (czyli typy wbudowane jak `int`, `char`, double ...), wskaźniki (w szczególności wskaźnik może wskazywać na tablicę), a nawet struktury i klasy (typ zdefiniowany przez użytkownika). Typ wartości zwracanej zwany jest zwykle **typem funkcji.**

Nazwa (`nazwaFunkcji`):

Nazwa może być dowolna, byle nie kolidowała z którymś ze słów kluczowych, składała się tylko z liter, cyfr i znaków podkreślenia. Nie może jednak zaczynać się od cyfry.

Definiowanie funkcji – funkcje zwracające wartość

Lista parametrów formalnych(`listaParametrów`).

Lista ta jest ujętą w okrągłe nawiasy listą oddzielonych przecinkami deklaracji pojedynczych parametrów funkcji w postaci (`typ1 nazwa1, typ2 nazwa2`). Nie wolno stosować deklaracji zbiorczych: (`typ nazwa1, nazwa2`). Nazwy wszystkich parametrów muszą być różne.

Lista parametrów może być pusta; obejmujących ją nawiasów pominąć jednak nie można. Jeśli lista parametrów jest pusta, to można zaznaczyć to przez wpisanie wewnątrz nawiasów słowa kluczowego `void`. Nie jest to jednak konieczne.

Definiowanie funkcji – funkcje zwracające wartość

Przykład: funkcja obliczająca sześcian wartości `double`:

```
double cube(double x)
{
   return x*x*x; //zwraca wartość typu double
}
```

W instrukcji **`return`** użyto wyrażenia. Funkcja przy jej wywołaniu oblicza wartość tego wyrażenia i zwraca ją.

Funkcje zwracają wartość w miejsce wywołania, a wartość ta może być potem przypisana zmiennej, wyświetlona, itp.

Zatem wyrażenie będące wywołaniem tej funkcji – samo w sobie – ma wartość.

Definiowanie funkcji – funkcje zwracające wartość

```
#include<iostream>
using namespace std;
double cube(double x);//prototyp- deklaracja f-cji
int main()
{
    double q = cube(1.2); //wywołanie funkcji
    cout << "1.2^3 = " << q <<endl;
    double side;
    cout << " podaj bok :";
    cin >> side;
    cout << "Kostka o boku "<< side
         << " ma pojemnosc "
         << cube(side) <<" cm3" <<endl;
         //2 wywołanie f-cji cube
}
double cube(double x) //definicja funkcji
{
    return x*x*x;
}
```

Definiowanie funkcji – funkcje zwracające wartość

Funkcja kończy swoje działanie po wykonaniu instrukcji **return.**

Jeśli funkcja ma więcej takich instrukcji np. w alternatywnych ścieżkach `if else` to kończy swoje działanie po wykonaniu pierwszej z nich. Np. poniższe `else` jest zbędne, ale ułatwia zrozumienie treści funkcji:

```
int bigger(int a, int b)
{
    if(a>b) return a;
    else return b;
}
/* krótsza wersja:
    if(a>b) return a;
    return b;   */
```

Parametry funkcji i przekazywanie przez wartość

Zajmijmy się teraz sposobem przesyłania argumentów do funkcji.

Najpierw jednak sprawa nazewnictwa.

```
//definicja
double cube(double x)
{
   return x*x*x; //wartość typu double
}
```

nazwy które widzimy w pierwszej linijce definicji funkcji – są to tzw. **argumenty formalne** funkcji. Czasem zwane **parametrami formalnymi**.

Parametry funkcji i przekazywanie przez wartość

```
int main()
{
    double q = cube(3);//wywołanie funkcji w main
    cout << q << endl;
}
```

To natomiast, co pojawia się w nawiasie w momencie wywoływania tej funkcji – czyli w naszym przypadku 3 to tak zwane **argumenty (parametry) aktualne**.

Czyli takie argumenty, z którymi aktualnie funkcja ma wykonać pracę. Często mówi się prościej: **argumenty wywołania funkcji –** bo z tymi argumentami funkcję wywołujemy.

W standardzie C++ używa się słowa argument dla określenia parametrów aktualnych, a słowa parametr dla param. formalnych.

Parametry funkcji i przekazywanie przez wartość