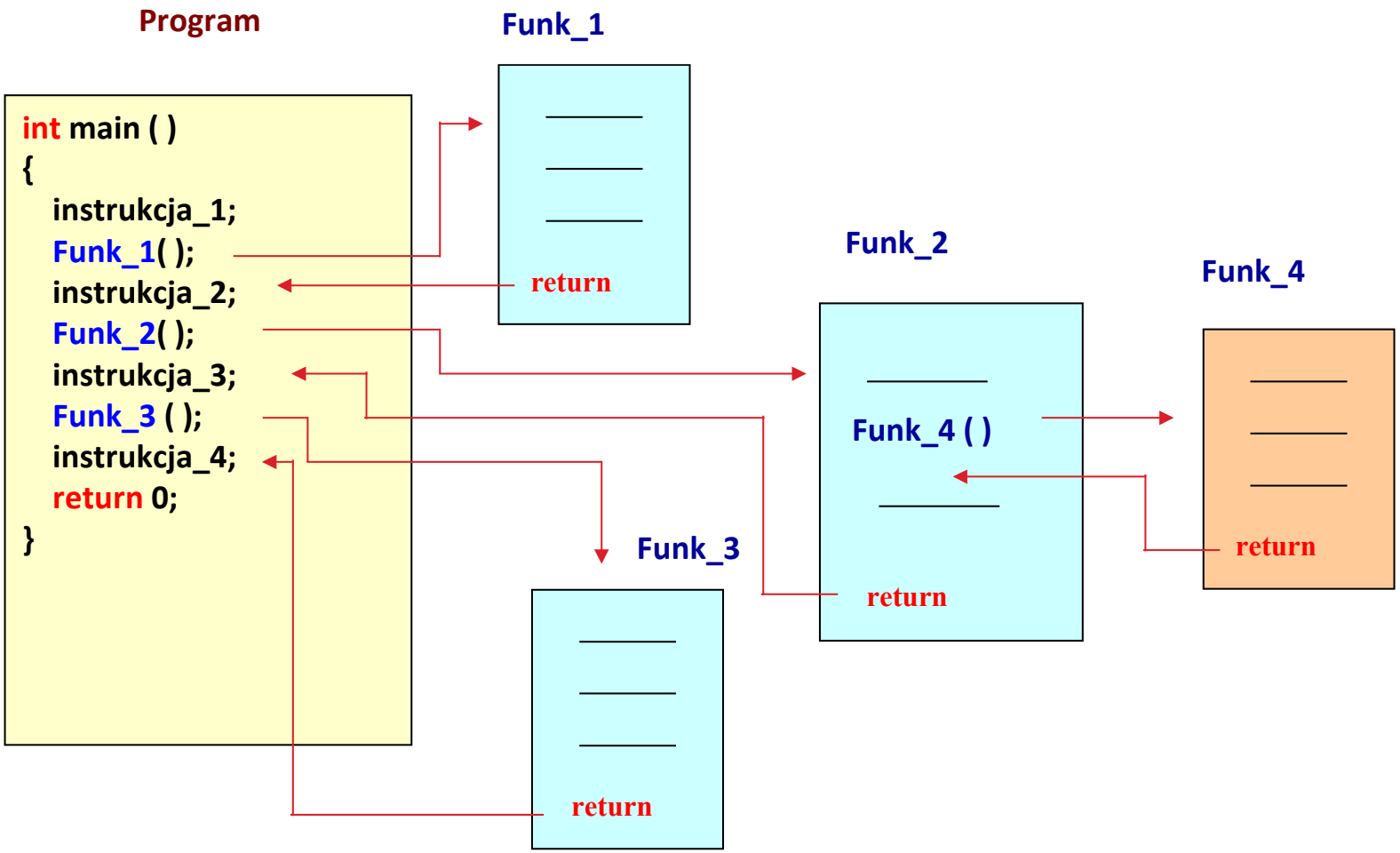
Funkcje

- Podprogram to "mały program" we właściwym programie. Dzięki podprogramom możemy definiować swoje własne "instrukcje".
- Jeśli napiszemy podprogram realizujący operację liczenia pola koła na podstawie zadanego promienia - to tak, jakbyśmy język programowania wyposażyli w nową instrukcję umiejącą właśnie to obliczać. Odtąd ilekroć będziemy potrzebować obliczyć pole koła, wywołamy nasz podprogram (a będzie to tak proste, jak napisanie zwykłej instrukcji).
- W języku C++ wszystkie podprogramy nazywane są *funkcjami*.
- Funkcje w C++ mają dwie odmiany: zwracające wartość i niezwracające wartości.
- Funkcje wywołuje się przez podanie jej nazwy i umieszczonych w nawiasie argumentów.

Funkcje


Program
int main ( )
{
instrukcja_1;
Funk_1( );
instrukcja_2;
Funk_2( );
instrukcja_3;
Funk_3 ( );
instrukcja_4;
return 0;
}
Funk_1
return
Funk_2
Funk_4 ( )
return
Funk_4
return
Funk_3
return

Funkcje

Jeśli funkcja zwraca wartość, wartość tę można przypisać zmiennej.
Na przykład biblioteka <cmath> zawiera funkcję `sqrt()` zwracającą pierwiastek kwadratowy z przekazanej liczby.

Załóżmy, że chcemy obliczyć pierwiastek kwadratowy z 6.25 i przypisać go zmiennej x:

```
x = sqrt(6.25); //zwraca 2.5 i przypisuje tę wartość zmiennej x
```

- Wyrażenie `sqrt(6.25)` wywołuje funkcję `sqrt()`.
  Wyrażenie `sqrt(6.25)` to wywołanie funkcji.
- `sqrt()` jest funkcją wywoływaną.
- Funkcja zawierająca `sqrt()` to funkcja wywołująca.
- Wartość umieszczona w nawiasie (6.25) to dane przekazywane do funkcji nazywana parametrem lub argumentem.
- Funkcja `sqrt()` oblicza wynik 2.5 i odsyła tę wartość do funkcji wywołującej; odsyłana wartość to wartość zwracana.
  O wartości tej można myśleć jako o wartości podstawianej w miejsce wywołania funkcji.

Funkcja suma

```
#include <iostream>
using namespace std;
int suma(int x, int y);                                    //1
/*************************************************/
int main ()
{
    int s;
    cout << "Zaczynamy." << endl;
    s = suma(2, 3);                                        //2
    cout << "Wynik sumowania: " << s;
}
/*************************************************/
int suma(int x, int y)                                     //3
{
    int p;
    cout << "Wewnątrz funkcji suma." << endl;
    p = x + y;
    return p;                                              //4
}
```

Wynik działania programu:

```
Zaczynamy.
Wewnątrz funkcji suma.
Wynik sumowania: 5
```

1. Każda funkcja ma swoją nazwę, która ją identyfikuje. Wiemy, że wszelkie nazwy (np. nazwy zmiennych), przed pierwszym odwołaniem się do nich, muszą zostać zadeklarowane. Wymagana jest więc także deklaracja nazwy funkcji:

```
int suma(int x, int y);                    //1
```

Deklaracja ta (*prototyp* funkcji) mówi kompilatorowi:
`suma` jest funkcją wywoływaną z dwoma argumentami typu `int` i zwracającą jako rezultat wartość typu `int`.

Przed odwołaniem się do nazwy wymagana jest jej deklaracja.
Deklaracja, ale niekoniecznie od razu definicja.
Sama funkcja może być zdefiniowana później, nawet w zupełnie innym pliku. Zdefiniować funkcję, to znaczy napisać jej treść.

Definicja funkcji `suma` znajduje się w **3**.

2. Wywołanie funkcji, to napisanie jej nazwy łącznie z nawiasem, w którym znajdują się argumenty przesyłane do funkcji:

```
s = suma(2, 3);                                    //2
```

Spodziewamy się, że w rezultacie swojej pracy funkcja zwróci jakąś wartość, dlatego widzimy przypisanie tej wartości do zmiennej *s*.

Na dowód tego, że funkcja rzeczywiście zwróciła jakąś wartość, i że nastąpiło przypisanie tej wartości do obiektu *s*, wypisujemy go:

```
cout << "Wynik sumowania: " << s;
```

3. Definicja funkcji zawiera tzw. ciało funkcji (obszar objęty klamrami), czyli wszystkie instrukcje wykonywane w ramach tej funkcji.

```
int suma(int x, int y)                                    //3
{
    int p;
    cout << "Wewnątrz funkcji suma." << endl;
    p = x + y;
    return p;                                             //4
}
```

4. Funkcja kończy swoją pracę i wraca do miejsca, skąd została wywołana (ang. *return* - powrót, zwrot).
Obok słowa `return` widzimy zmienną *p*, której wartość zwracamy jako rezultat wykonania tej funkcji.

Funkcja suma (wersja 2)

```
#include <iostream>
using namespace std;
int suma (int x, int y);
/*************************************/
int main ()
{
    int a, b;
    cout << "Podaj dwie liczby:" << endl;
    cin >> a >> b;
    cout << "Wynik sumowania: " << suma(a, b);
}
/*************************************/
int suma(int x, int y)
{
    return x + y;
}
```

```
Podaj dwie liczby:
2
4
Wynik sumowania: 6
```

Funkcja suma (wersja 3)

```
#include <iostream>
using namespace std;
int suma();
/*************************************/
int main ()
{
    cout << "Wynik sumowania: " << suma();
}
/*************************************/
int suma()
{
    int a, b;
    cout << "Podaj dwie liczby: " << endl;
    cin >> a >> b;
    return a + b;
}
```

Deklaracja `int suma();` lub `int suma(void);` mówi, że funkcja `suma` nie pobiera żadnych parametrów i zwraca wartość typu `int`.

Słowo kluczowe **void** (pusty) odnosi się do pustego typu danych (ang. *void type*).

Funkcja suma (wersja 4)

```
#include <iostream>
using namespace std;
void suma(int x, int y);
/*************************************/
int main ()
{
    int a, b;
    cout << "Podaj dwie liczby: " << endl;
    cin >> a >> b;
    //int s = suma(a, b);    błąd
    suma(a, b);
}
/*************************************/
void suma(int x, int y)
{
    cout << "Wynik sumowania: " << x + y;
    //return x + y;          błąd
}
```

Deklaracja `void suma(int x, int y);` oznacza, że funkcja `suma` przyjmuje dwa parametry typu `int` i nie zwraca żadnej wartości.

Funkcja suma (wersja 5)

```
#include <iostream>
using namespace std;
void suma();
/*************************************/
int main ()
{
    suma();
}
/*************************************/
void suma()
{
    int a, b;
    cout << "Podaj dwie liczby: " << endl;
    cin >> a >> b;
    cout << "Wynik sumowania: " << a + b;
}
```

Deklaracja `void suma();` oznacza, że funkcja `suma` nie przyjmuje żadnych parametrów i nie zwraca żadnej wartości.

Funkcja main

- Mimo że funkcja `main` ma typ wyniku określony jako `int`, nie musi w swojej treści zawierać instrukcji `return`. Takie pominięcie jest możliwe tylko w funkcji `main`. W przypadku braku instrukcji `return 0`, kompilator sam ją dopisze.
- Skoro wraz z zakończeniem funkcji `main`, kończy się program, to kto w wyniku instrukcji `return 0` otrzyma wartość 0?
  Odp.1: System operacyjny, bo to on uruchomił program.
  Odp.2: Jeżeli nasz program został uruchomiony przez inny program, to system operacyjny przekaże rezultat (0) tamtemu drugiemu programowi.
- Zwrócona może być wartość inna niż 0. Najczęściej wartość zwracana oznacza, czy program wykonał się poprawnie, czy z błędami. Tradycyjnie zwrot wartości 0 oznacza poprawne zakończenie całego programu. Wartości inne niż zero mogą oznaczać różnego typu błędy wykonania.
- Funkcja `main` ma specjalne prawa, np. nigdzie jej nie deklarujemy i nie możemy wywołać jej z innej funkcji.

Program oblicza symbol Newtona (wersja 1: bez funkcji)

```
#include <iostream>
using namespace std;
int main ()
{
  int n, k;
  cout << "Podaj kolejno n i k:" << endl;     cin >> n >> k;

  int s_n = 1;
  for (int i = 2; i <= n; i++)
    s_n = s_n * i;

  int s_k = 1;
  for (int i = 2; i <= k; i++)
    s_k = s_k * i;

  int s_nk = 1;
  for (int i = 2; i <= n-k; i++)
    s_nk = s_nk * i;

  int w = s_n / (s_k * s_nk);
  cout << "Symbol Newtona n nad k wynosi " << w;
}
```

Program oblicza symbol Newtona (wersja 2: z funkcją silnia)

```
#include <iostream>
using namespace std;
int silnia(int p);
/*************************************/
int main ()
{
  int n, k;
  cout << "Podaj kolejno n i k:" << endl;     cin >> n >> k;

  int w = silnia(n) / (silnia(k) * silnia(n-k));
  cout << "Symbol Newtona n nad k wynosi " << w;
}
/*************************************/
int silnia(int p)
{
  int s = 1;
  for (int i = 2; i <= p; i++)
    s = s * i;
  return s;
}
```

Kilka instrukcji return

```
#include <iostream>
using namespace std;
string f(int, int);
int main()
{
   int a, b;
   cout << "Podaj kolejno dwie liczby:" << endl;
   cin >> a >> b;
   cout << "Pierwsza liczba jest " << f(a,b) << " (od) drugiej.";
}
string f(int a, int b)
{
   if (a > b) return "większa";
   else if (a < b) return "mniejsza";
        else return "równa";
}
```

- Klasa **string** służy do obsługi napisów (tekstów), jest umieszczona w przestrzeni `std`.
- Funkcja kończy swoje działanie po wykonaniu instrukcji powrotu return.
  Jeśli funkcja ma więcej takich instrukcji, zakończy swoje działanie po wykonaniu pierwszej z nich.

Sposoby przesyłania parametrów do funkcji

1. przez **wartość** 2. przez **referencję** (adres)

1. Domyślnie parametry przesyłane są do funkcji przez wartość.
W poniższym programie

```
#include <iostream>
using namespace std;
void f(int a);
int main()
{
   int a = 2;
   f(a);          //wywołanie
   cout << a;
}
void f(int a)
{
   a = 5;
}
```

w momencie wywołania funkcji **f**, tworzona jest nowa zmienna - kopia parametru **a**. Oznacza to, że funkcja **f** nie otrzymuje oryginału zmiennej **a**, a zatem będzie działać na jej kopii. W ten sposób dane funkcji **main** (czyli tutaj zmienna **a**) są chronione przed działaniami, jakie mają miejsce w funkcji **f**.
Wynikiem powyższego kodu będzie 2, a nie 5.

Sposoby przesyłania parametrów do funkcji

2. Aby zamiast kopii funkcja otrzymała oryginał parametru, należy wysłać go do niej przez referencję (czyli podać funkcji adres parametru w pamięci komputera). W tym celu w nagłówku funkcji f przed nazwą parametru stawiamy operator pobrania adresu &:

```
void f(int &a);
```

Wtedy funkcja f nie tworzy nowej zmiennej - prywatnej kopii parametru a, tylko działa bezpośrednio na nim. Zatem przesyłanie parametrów do funkcji przez adres pozwala tej funkcji na modyfikowanie zmiennych znajdujących się poza nią.

```
#include <iostream>
using namespace std;
void f(int &a);
int main()
{
   int a = 2;
   f(a);
   cout << a;
}
void f(int &a)
{
   a = 5;
}
```

Wynikiem programu będzie 5.

```cpp
#include <iostream>
using namespace std;
void f(int a, int &b);
int main()
{
   int a = 2, b = 3;
   cout << "Przed wywołaniem funkcji:" << endl;
   cout << "a = " << a << ", b = " << b << endl;
   f(a, b);
   cout << "Po wywołaniu funkcji:" << endl;
   cout << "a = " << a << ", b = " << b << endl;
}
void f(int a, int &b)
{
   a = 5;  b = 6;
   cout << "Wewnątrz funkcji:" << endl;
   cout << "a = " << a << ", b = " << b << endl;
}
```

```
Przed wywołaniem funkcji:
a = 2, b = 3
Wewnątrz funkcji:
a = 5, b = 6
Po wywołaniu funkcji:
a = 2, b = 6
```

Przekazywanie tablic do funkcji

- Nazwa tablicy jest jednocześnie adresem jej pierwszego elementu.
- Tablice są zatem przekazywane do funkcji przez adres.
  Przykładowy prototyp funkcji:

```
void f(int tab[]);
```

- Pozwala to zaoszczędzić na czasie i pamięci, które inaczej byłyby potrzebne do kopiowania przekazywanej tablicy. W przypadku dużych tablic narzut związany z ich kopiowaniem może praktycznie uniemożliwiać pracę — nie dość, że potrzebne byłoby dużo pamięci, to jeszcze wiele czasu poświęcano by na kopiowanie danych.
- Z drugiej strony praca z oryginalnymi danymi zwiększa ryzyko przypadkowego uszkodzenia tych danych.
  Aby uniemożliwić funkcji przypadkową modyfikację przekazanej jej tablicy, można użyć słowa kluczowego const:

```
void f(const int tab[]);
```

Przekazywanie tablic do funkcji

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int suma(int n, int tab[]);
int main()
{
   int A[100], n;
   cout << "Podaj rozmiar tablicy (n<=100): ";
   cin >> n;
   srand(time(NULL));
   for (int i = 0; i < n; i++)
      A[i] = rand()%10;
   for (int i = 0; i < n; i++)
      cout << A[i] << " ";
   cout << endl << "Suma elementów tablicy wynosi " << suma(n,A);
}
int suma(int n, int tab[])
{
   int s = 0;
   for (int i = 0; i < n; i++)
      s = s + tab[i];
   return s;
}
```

Rekurencja

- Funkcje w C++ (z wyjątkiem funkcji main) mają tę ciekawą cechę, że mogą wywoływać same siebie.
- Taka możliwość wywołania samej siebie to rekurencja.
- Rekurencja jest ważnym narzędziem w niektórych rodzajach oprogramowania, np. w sztucznej inteligencji.

Właściwie każda funkcja może wywołać samą siebie, problem w tym, że jeśli nie przygotujemy jej na taką pracę, to doprowadzimy do nieskończonej pętli wywołań. Przykład funkcji, która gwarantuje takie kłopoty:

```
void f(int x)
{
  f(x);
}
```

Aby nie spowodować nieskończonej pętli wywołań - funkcja przeznaczona do rekurencyjnego wywołania - powinna posiadać w ciele warunek, w którym decyduje, czy wywołać samą siebie (po raz kolejny), czy nie. Nazywamy go warunkiem zatrzymującym rekurencję.

Rekurencja

```
#include <iostream>
using namespace std;
long silnia(int n);
int main()
{
   int n;
   long w;
   cout << "Podaj n: ";
   cin >> n;
   w = silnia(n);
   cout << n << "! = " << w;
}
long silnia(int n)
{
 if (n == 0)
    return 1;
else
    return n * silnia(n-1);
}
```

```
Podaj n: 5
5! = 120
```

Rekurencja

Przykład nieskończonej liczby wywołań rekurencyjnych w wyniku niepoprawnej definicji funkcji `stad_do_nieskonczonosci(int n)`.

```
#include <iostream>
using namespace std;
int stad_do_nieskonczonosci(int n);
int main ()
{
  int n, k;
  cout << "Podaj liczbę calkowitą: ";
  cin >> n;
  k = stad_do_nieskonczonosci(n);
  cout << "k = " << k << endl;
}
int stad_do_nieskonczonosci(int n)
{
  if (n == 1) return 1;
  else if (n%2 != 0)
          return stad_do_nieskonczonosci(n-1);
       else
          return stad_do_nieskonczonosci(n-2);
}
```