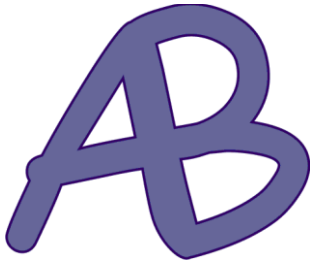


Wykład: 5

Funkcje
Przekazywanie argumentów do funkcji

Funkcje

Funkcje

Deklaracja funkcji (nagłówki funkcji)

Funkcja ma swoją nazwę, która ją identyfikuje. Wszystkie nazwy - przed pierwszym odwołaniem się do nich muszą zostać zadeklarowane.

Wymagana jest więc także deklaracja nazwy funkcji.

Funkcje

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

int nasza_funkcja(int parametr); //deklaracja funkcji

int main()
{
    int a = 10;
    cout << nasza_funkcja(a);
    return 0;
}

int nasza_funkcja(int parametr) //definicja funkcji
{
    cout << ++parametr;
    return 0;
}
```

Nagłówki funkcji

Jeżeli funkcja zdefiniowana jest poniżej funkcji main aby mogła zostać wywołana konieczne jest i jej zadeklarowanie, czyli umieszczenie nagłówka funkcji zakończonego średnikiem przed funkcją main.

```
using namespace std;

void funkcja_01();
void funkcja_02();
void funkcja_03();
void funkcja_04();

int main()
{
    cout << "Dziala funkcja main"<<endl;
    return 0;
}
void funkcja_01()
{
    cout << "dziala funkcja fubkcja_01"<<endl;
}
void funkcja_02()
{
    cout << "dziala funkcja fubkcja_02" << endl;
}
void funkcja_03()
{
    cout << "dziala funkcja fubkcja_03" << endl;
}
void funkcja_04()
{
    cout << "dziala funkcja fubkcja_04" << endl;
}
```

Funkcje z argumentami (parametrami)

Argumenty formalne i aktualne

1. Argumenty formalne
   a) deklarowane są w nagłówku funkcji
   b) są to identyfikatory (nazwy) symbolizujące dane przekazywane do funkcji;
2. Argumenty aktualne
   a) rzeczywiste wartości zmiennych, stałych i wyrażeń podstawiane podczas wywołania w miejsce argumentów formalnych

- ✓ Argumenty formalne opisują jedynie postać informacji przekazywanej do funkcji;
- ✓ Rzeczywistą informację niosą ze sobą dopiero argumenty aktualne.

Funkcje

Przesyłanie argumentów do funkcji przez wartość

Załóżmy, że mamy funkcję:

Argumenty formalne

```
void alarm(int stopień,   int wyjście)
{
cout <<  "Alarm "  << stopień << "stopnia"
<<  "  skierować sie do wyjścia nr " << wyjście;
}
```

W programie wywołujemy tę funkcję tak:

```
int a,  m ; //    . . .
alarm(1,   10) ; alarm (a,  m) ;
```

Argumenty aktualne

Funkcje - Przesyłanie argumentów do funkcji przez wartość

Argumenty przesłane do funkcji – przez wartość - są tylko kopiami. Jakiekolwiek działanie na nich nie dotyczy oryginału.

```
void zwiększ(int formalny)
{
  formalny += 1000; // zwiększenie liczby o 1000
  cout <<   "W funkcji modyfikuje arg formalny\n\t"
       <<  "  i teraz arg formalny =  "<< formalny;
}
```

Wywołując powyższą funkcję w następujący sposób:

```
int aktu = 2 ;
cout << "Przed wywołaniem, aktu = " << aktu << endl;
zwiększ(aktu) ;
cout <<  "Po wywołaniu,   aktu =  "  << aktu << endl;
```

Otrzymamy:

Przed wywołaniem, aktu = 2
W funkcji modyfikuje arg formalny
i teraz arg formalny = 1002 Po wywołaniu,
aktu = 2

Przekazywanie argumentów

Poniższa procedura zamienia ze sobą zawartość zmiennych x i y

```
void zamiana (int x, int y)
{
     int temp;
     temp = x;
     x = y;
     y = temp;
}
```

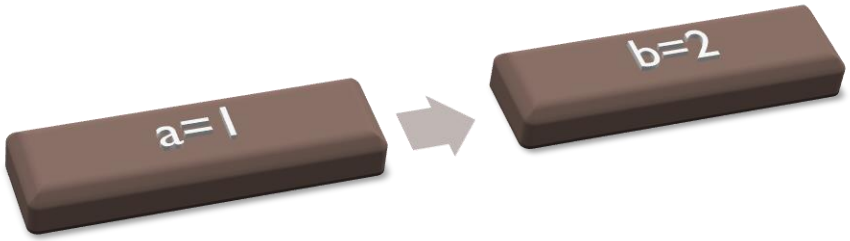
Procedura **po wywołaniu, rezerwuje 3 komórki** pamięci o nazwach:
x, y, temp

Przekazywanie argumentów

Program główny wywołuje procedurę z argumentami a i b

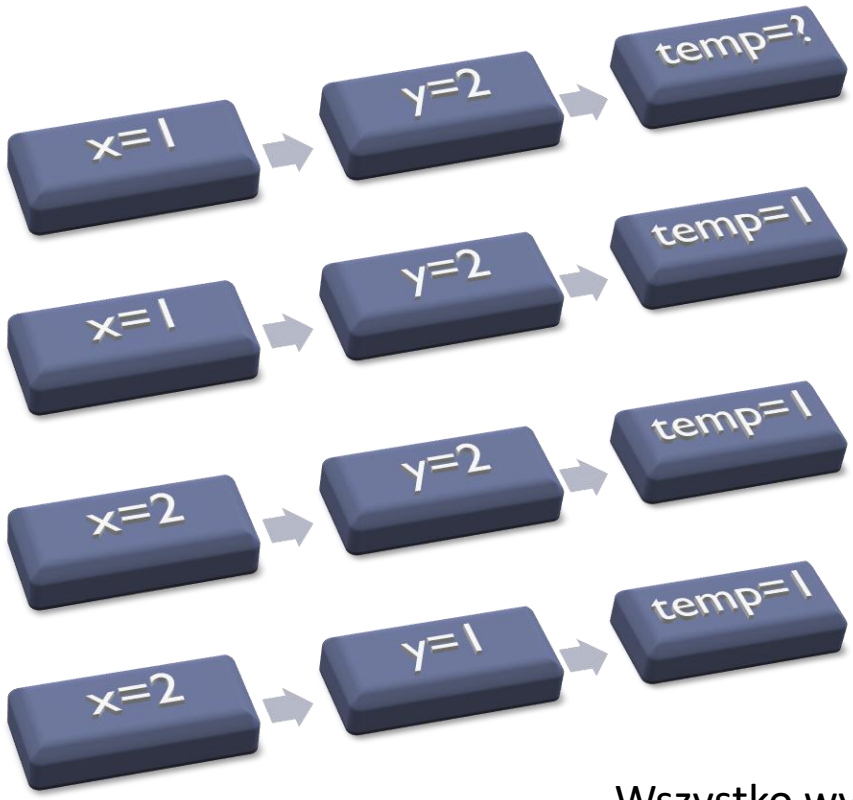
```
int main()
{
    int a = 1;
    int b = 2;
    zamiana(a,b);
    cout << "a= " << a << endl;
    cout << "b= " << b;
    return 0;
}
```

Program rezerwuje 2 komórki pamięci o nazwach: **a, b**



Przekazywanie argumentami

Co będzie się działo w tych trzech komórkach pamięci (x, y, temp) , gdy wywołamy procedurę?



wejście do procedury

temp = x

x = y ;

y = temp;

Wszystko wygląda prawidłowo,
jednak ta procedura tak naprawdę nic nie robi!

Funkcje - Przesyłanie argumentów do funkcji przez referencję

Przesyłanie argumentów przez referencję

```
int nasza_funkcja (int &zmienna);
```

Wywołując funkcję o powyższej deklaracji w następujący sposób:

```
int a1=100;
nasza_funkcja(a1);
```

zamiast wartość zmiennej a1 (liczby 100), do funkcji został wysłany adres zmiennej a1. Na jego podstawie w funkcji stworzona została referencja zmiennej a1 – druga nazwa dla tego samego obiektu w pamięci operacyjnej.

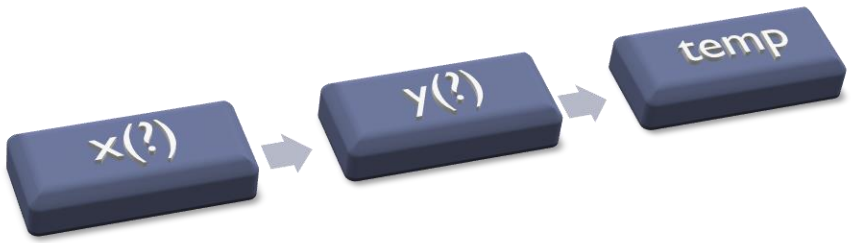
Funkcja może modyfikować wartość zmiennej przekazanej przez referencję.

Przekazywanie argumentów

W tej wersji procedury zastosowano inny sposób przekazywania parametrów:

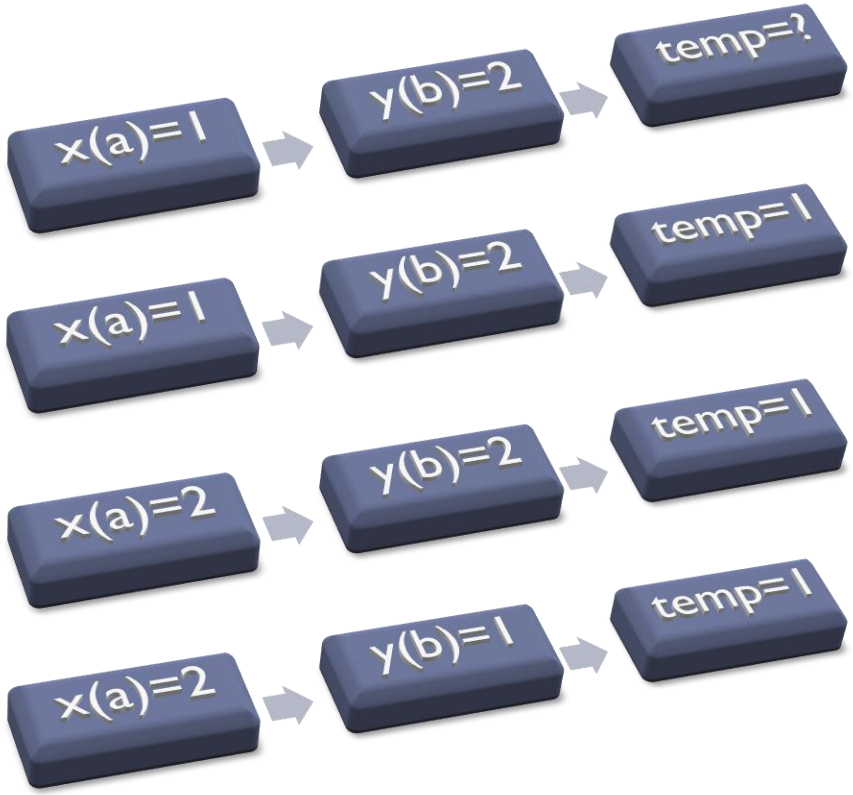
```
void zamiana (int &x, int &y)
{
     int temp;
     temp = x;
     x = y;
     y = temp;
}
```

Tym razem zmienne lokalne x, y zostaną powiązane ze zmiennymi, które procedura otrzyma w trakcie jej wywołania.



Przekazywanie argumentów

Co będzie się działo w tych trzech komórkach pamięci (x, y, temp) , gdy wywołamy procedurę?



wejście do procedury

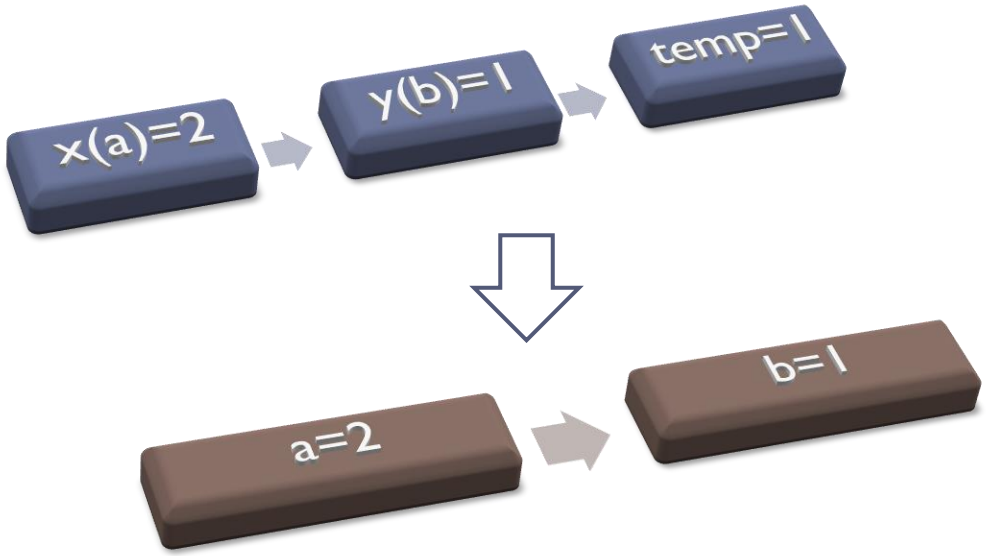
temp = x ;

x = y ; &a = &b;

y = temp; &b= temp;

Przekazywanie argumentów

Po zakończeniu procedury zmienne lokalne x,y,temp znikają, lecz powiązane z nimi zmienne a i b mają już nową wartość.



Po wykonaniu procedury

Metoda ta pozwala na przekazywanie wyniku z wnętrza procedury
do programu ją wywołującego

Funkcje

Argumenty domniemane

Nagłówek funkcji:

```
int nasza_funkcja (int a =100, int argument = 0);
```

Możliwe sposoby wywołania:

```
Nasza_funkcja(10, 10);
Argument = 10;

Nasa_funkcja(10);
Argument = 0;
```

.

Przeciążanie nazw funkcji

Przeciążanie nazw funkcji

Przeciążanie nazw funkcji umożliwia zdefiniowanie kilku funkcji o tej samej nazwie różniących się listą parametrów.

Np.:

```
int funkcja ();
int funkcja (int a);
int funkcja (int a, int b);
int funkcja (int a, int b, inc c);
```

Przeciążanie nazw funkcji

```
#include <iostream>

using namespace std;

void wypisz(int x);
void wypisz(float x);
float srednia(int a, int b);
float srednia(int a, int b, int c);

int main()
{
    int a = 10;
    float b = 3.14;
    wypisz(a);
    wypisz(b);
    cout << srednia(11,20);
    cout <<endl<< srednia(11,20,30);

    return 0;
}
```

Przeciążanie nazw funkcji

```
void wypisz(int x)
{
    cout << "liczba calkowita  x=" << x << endl;
}

void wypisz(float x)
{
    cout << "liczba rzeczywista  x=" << x << endl;
}

float srednia(int a, int b)
{
    float sr;
    sr = ((float)a+b )/2;
    return sr;
}

float srednia(int a, int b, int c)
{
    float sr;
    sr = ((float)a+b+c )/3;
    return sr;
}
```

Zakres ważności nazwy i czasu życia obiektu

Obiekty globalne

Obiekt zdeklarowany na zewnątrz wszystkich funkcji ma zasięg globalny.

- Obiekt jest dostępny wewnątrz wszystkich funkcji znajdujących się w tym pliku.
- Jest znany dopiero od linijki, w której nastąpiła jego deklaracja, w dół, do końca programu.

```
#include <iostream.h>
int liczba;  // zmienna globalna
void jaksa_funkcja();

Int main() {
}

void jaksa_funkcja() {
}
```

Obiekty automatyczne

Zmienne automatyczne - w momencie gdy kończymy blok, w którym je zdefiniowaliśmy automatycznie przestają istnieć. (obiekty automatyczne komputer przechowuje na stosie).

Jeśli po raz drugi wejdziemy do danego bloku (np. przy powtórnym wywołaniu funkcji) to zmienne tam zdefiniowane zostaną powołane do życia po raz drugi.

Wynikają z tego dwa wnioski:

- nie możemy liczyć na to, że przy ponownym wywołaniu funkcji zastaniemy zdefiniowany tam z wartością, którą miał na gdy poprzednio korzystaliśmy z tej funkcji.
- skoro obiekt przestaje istnieć, to nie ma sensu by funkcja zwracała jego adres.

Obiekty automatyczne

- **Zmienne globalne** – są wstępnie inicjalizowane wartością zero.

- **Zmienne automatyczne** - zawierają na stracie losową wartość.

Obiekty lokalne statyczne

Zmienne lokalne statyczne – pozwalają, by zmienna lokalna dla danej funkcji przy ponownym wejściu do tej funkcji miała taką wartość, jak przy ostatnim opuszczaniu tejże funkcji.

W odróżnieniu od zmiennych globalnych są one jednak znane tylko w obrębie funkcji, w której je zdefiniowano.

Obiekty lokalne statyczne

```
#include <iostream.h>
void czerwona(void);

int main()
{
    czerwona();
    czerwona();
    biala();
    czerwona();
    biala();
}

void czerwona(void)
{
    static int ktory_raz;
    ktory_raz++ ;
    cout    <<  "Funkcja czerwona wywo_ana  "<< ktory_raz
            <<  "  raz\n";
}

void biala(void)
{
    static int ktory_raz = 100 ;
    który_raz = który_raz + 1 ;
    cout    <<  "Funkcja bia_a wywo_ana  "<< ktory_raz
            <<  "  raz\n";
}
```

Przesyłanie tablic do funkcji

```
#include <iostream>

using namespace std;

int policz(int t[], int ile);
void zwieksz_o_1(int t[], int ile);

int main()
{
    int tab[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int tab_2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    int n = 9;
    int n_2 = 11;

    cout << "tablica 1: " << policz(tab, n) << endl;
    zwieksz_o_1(tab,n);
    cout << "tablica 1: " << policz(tab, n) << endl;
    //cout << "tablica 2: " << policz(tab_2, n_2) << endl;
    return 0;
}
```

W nagłówku funkcji podajemy tylko informację o typie tablicy – bez jej rozmiaru

W wywołaniu funkcji podajemy tylko nazwę tablicy (bez nawiasów kwadratowych)

Przesyłanie tablic do funkcji

```
int policz(int t[], int ile)
{
    int suma = 0;
    for (int i = 0; i < ile; i++)
    {
        suma += t[i];
    }
    return suma;
}

void zwieksz_o_1(int t[], int ile)
{
    for (int i = 0; i < ile; i++)
        t[i]++;
}
```

W c/c++ tablice przesyłane są przez referencje. Oznacza to, że funkcja może modyfikować ich zawartość.

Literatura:

W prezentacji wykorzystano przykłady i fragmenty:

- Gręboszl J. : ***Symfonia C++, Programowanie w języku C++ orientowane obiektowo***, Wydawnictwo Edition 2000.
- Jakubczyk K.: *Turbo Pascal i Borland C++ Przykłady*, Helion.

Warto zajrzeć także do:

- Sokół R. : ***Microsoft Visual Studio 2012 Programowanie w Ci C++,*** Helion.
- Kerninghan B. W., Ritchie D. M.: ***język ANSI C***, Wydawnictwo Naukowo Techniczne.

Dla bardziej zaawansowanych:

- Gręboszl J. : ***Pasja C++,*** Wydawnictwo Edition 2000.
- Meyers S.: ***język C++ bardziej efektywnie***, Wydawnictwo Naukowo Techniczne