

Podstawy informatyki

Katedra Telekomunikacji, EiT

dr inż. Jarosław Bułat

kwant@agh.edu.pl

Plan prezentacji

- » Funkcje:
 - deklaracja, definicja
 - wywołanie
 - przekazanie argumentów
 - zwrócenie wyniku
- » Zasięg zmiennych w kontekście funkcji
- » Przykłady użycia funkcji

Wiele razy powtarzam ten sam kod

co robić jak żyć?

Funkcje - definicja

```
type name(parameter0, parameter1)
{
    // body (statements)
    return type;
}
```

- » Cel użycia funkcji:
 - uniknięcie powtarzania tego samego kodu wiele razy
 - ukrycie fragmentu kodu (detali implementacji)
- » **type** - typ funkcji == typ zwracanych danych, może być void
- » **name** - nazwa funkcji (zasady jak dla nazw zmiennych)
- » **parameter** - rodzaj i liczba przekazywanych parametrów
- » **body** - instrukcje wewnątrz funkcji

Funkcje - definicja

```
int add(int a, int b) {  
    return a+b;  
}
```

```
void pprint( void ) {  
    cout << "nothing" << endl;  
}
```

```
void pprint() {  
    cout << "nothing" << endl;  
    return;  
}
```

- » “typ” funkcji - dowolny,
 - może być **void** czyli “bez typu”
 - jeżeli zdefiniowany: return ...;
- » Parameter (**argument funkcji**) -
wygląda jak deklaracja zmiennej,
zmienne te zostaną zainicjalizowane podczas wywołania
- » Zasięg zmiennych wewnątrz pprint():
tylko globalne + argumenty
- » Wewnątrz ciała funkcji mogą używać argumentów jak zmiennych **zasięg do końca funkcji !!!**

Funkcje - argumenty

» Podczas wywołania funkcji:

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

```
void pprint(int x) {  
    cout << x << endl;  
}
```

```
int main() {  
    int a = 1;  
  
    pprint(a);  
    pprint(10);  
  
    cout << a << endl;  
}
```

Funkcje - argumenty

» Podczas **wywołania** funkcji:

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

```
void pprint(int x) {
    cout << x << endl;
}
```

```
int main() {
    int a = 1;

    pprint(a);
    pprint(10);

    cout << a << endl;
}
```

Funkcje - argumenty

» Podczas **wywołania** funkcji,

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

```
void pprint(int x) {
    cout << x << endl;
}
```

```
int main() {
    int a = 1;

    pprint(a);
    pprint(10);

    cout << a << endl;
}
```


Funkcje - argumenty

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

```
void pprint(int x) {
    cout << x << endl;
}
```

```
int main() {
    int a = 1;
```

```
    pprint(a);
    pprint(10);
```

```
    cout << a << endl;
}
```

- » Podczas **wywołania** funkcji, Inicjalizowane są argumenty, poprzez przypisanie wartości
- » Wartość argumentu to wyrażenie obliczane podczas wywołania i w miejscu wywołania

Funkcje - argumenty

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

```
void pprint(int x) {
    cout << x << endl;
}
```

```
int main() {
    int a = 1;
```

```
    pprint(a);
    pprint(10);
```

```
    cout << a << endl;
}
```

- » Podczas **wywołania** funkcji, Inicjalizowane są argumenty, poprzez przypisanie wartości
- » Wartość argumentu to wyrażenie obliczane podczas wywołania i w miejscu wywołania
- » Argumenty przekazywane są przez **kopie** (przypisanie)
- » Kopia oznacza:
 - rezerwację pamięci
 - zmiany wewnątrz funkcji nie “wychodzą” na zewnątrz

Funkcje - wywołanie

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

```
int add(int a, int b) {
    return a+b;
}
```

```
int main() {
    int x = 1;
    int y = 2;
    int z;
```

```
z = add(x, y);
```

```
cout << z << endl;
```

```
}
```

» Wywołanie funkcji

» Funkcja `main()` wywoływana jest automatycznie jako pierwsza

» Przekazywanie argumentów:

definicja

```
int add(int a, int b) { return a+b; }
```

wywołanie

```
z = add(x, y);
```

Funkcje - wywołanie

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

```
int add(int a, int b) {
    return a+b;
}
```

```
int main() {
    int x = 1;
    int y = 2;
    int z;
```

```
z = add(x, y);
```

```
cout << z << endl;
```

```
}
```

» Wywołanie funkcji

» Funkcja `main()` wywoływana jest automatycznie jako pierwsza

» Przekazywanie argumentów:

definicja

```
int add(int a, int b) { return a+b; }
```

wywołanie

```
z = add(x, y);
```

Funkcje - wywołanie

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

```
int add(int a, int b) {
    return a+b;
}
```

```
int main() {
    int x = 1;
    int y = 2;
    int z;
```

```
z = add(x, y);
```

```
cout << z << endl;
```

```
}
```

» Wywołanie funkcji

» Funkcja `main()` wywoływana jest automatycznie jako pierwsza

» Przekazywanie argumentów:

definicja

```
int add(int a, int b) { return a+b; }
```

wywołanie

```
z = add(x, y);
```



quiz

PI07_fun1

socrative.com

- login
- student login

Room name:

KWANTAGH

Funkcje - zasięg argumentów

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

void pprint(int x) {
    cout << "1: " << x << endl;
    ++x;
    cout << "2: " << x << endl;
}

int main() {
    int x = 6;

    cout << "0: " << x << endl;
    pprint(x);
    cout << "3: " << x << endl;
}
```

- » Konsekwencje przekazywania argumentu przez kopię:
 - zasięg zmiennej `x` jest ograniczony tylko do funkcji `pprint(...)`
 - zmiana `x` nie zmienia `x`.
- » Rezultat:

0: 6

1: 6

2: 7

3: 6

Funkcje - deklaracja, definicja

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

```
int main() {
    cout << f(71) << endl;
}
```

```
int f(int x){
    return ++x;
}
```

- » Użyłem f(...) przed definicją
- » **error: 'f' was not declared in this scope**

Funkcje - deklaracja, definicja

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

```
int main() {
    cout << f(71) << endl;
}
```

```
int f(int x){
    return ++x;
}
```

- » Użyłem f(...) przed definicją
- » **error: 'f' was not declared in this scope**
- » W miejscu **wywołania funkcji**, nie jest znana **definicja funkcji**.

Funkcje - deklaracja, definicja

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

```
int f(int x);
```

```
int main() {
    cout << f(71) << endl;
}
```

```
int f(int x){
    return ++x;
}
```

- » Użyłem f(...) przed definicją
- » **error: 'f' was not declared in this scope**
- » W miejscu **wywołania funkcji**, nie jest znana **definicja funkcji**.
- » Należy dodać **deklarację funkcji**.
- » Deklaracje funkcji zazwyczaj na górze *.cc lub w *.h
- » Często dzieli się program na:
 - interfejs (*.h) - jak używać
 - implementację (*.cc)

Funkcje - deklaracja, definicja

```
#include<iostream>
```

```
using namespace std;
```

```
int f(int x);
```

← Deklaracja (jak użyć funkcji)

```
int main() {
```

```
    cout << f(71) << endl;
```

```
}
```

← Wywołanie (użycie funkcji)

```
int f(int x){
```

```
    return ++x;
```

```
}
```

← Definicja (implementacja, body)

Funkcje - deklaracja, definicja

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

```
int f(int);
```

← Deklaracja

```
int main() {
    cout << f(71) << endl;
}
```

» Może zawierać typy, bez nazw,
jednakże: nazwy ułatwiają dokumentację

```
int f(int x){
    return ++x;
}
```

```
/**
 * high level decoding FIC - Fast Information Channel
 * @param data pointer to samples_
 * @todo common parts with MSCDecoder()
 */
void FICDecoder(float *data);
```

Funkcje - return

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

```
int f(int x){
    if (x<0){
        return -x;
    }
    return x;
}

int main(){
    cout << f(-10) << endl;
    cout << f(10) << endl;
}
```

- » Funkcja może zwrócić dowolny typ
- » Słowo kluczowe **return**
 - w dowolnym miejscu funkcji
 - bezwarunkowo kończy funkcję
 - jeżeli funkcja ma typ, musi zwrócić ten sam typ
 - funkcja bez typu, return nie zwraca wartości (tylko kończy)
- » Wartość zwracana przez kopiowanie w miejsce wywołania (może być kosztowne!)

Funkcje - return

```
struct Complex{  
    float re;  
    float im;  
};
```

```
Complex f(float re, float im){  
    Complex result = {re, im};  
    return result;  
}
```

```
int main(){  
    Complex r;  
    r = f(3, 4);  
  
    cout << r.re << endl;  
    cout << r.im << endl;  
}
```

- » Jak przekazać więcej niż jedną zmienną przez return
- » Funkcja jest “typu Complex”
- » Zwraca strukturę
- » Struktura jest przypisana do zmiennej **r** po wykonaniu funkcji
- » **Przypisanie przez kopiowanie** więc mało wydajne przy dużej ilości danych

Funkcje - return

```
struct Complex{  
    float re;  
    float im;  
};  
  
Complex f(Complex in){  
    Complex result = {in.im, in.re};  
    return result;  
}
```

```
int main(){  
    Complex in = {3, 4};  
    Complex r = f(in);  
  
    cout << r.re << endl;  
    cout << r.im << endl;  
}
```

- » Podobnie, tylko przez argument przekazywana struktura
- » Przypisanie przez kopiowanie więc mało wydajne przy dużej ilości danych



quiz

PI07_fun2

socrative.com

- login
- student login

Room name:

KWANTAGH

Funkcje - wskaźnik na wynik

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

```
void swap(int *x, int *y){
    int tmp = *y;
    *y = *x;
    *x = tmp;
}
```

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

```
    cout << a << endl;
    cout << b << endl;
}
```

- » Argumentem jest wskaźnik na wynik
- » **Argumenty wywołania** funkcji wskazują gdzie ma zostać umieszczony wynik
- » Argument jest **kopiowany**
 - niemożliwy do zmiany “wstecz”
 - ale **wynik jest zapisany pod adres który wskazuje!**
- » adres wskaźnika nie jest modyfikowany tylko wyluskiwana jest dana

Funkcje - wskaźnik na wynik

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

void swap(int *x, int *y){
    int tmp = *y;
    *y = *x;
    *x = tmp;
}

int main(){
    int tab[] = {10, 20};
    swap(tab, tab+1);
    // swap(&tab[0], &tab[1]);

    cout << tab[0] << endl;
    cout << tab[1] << endl;
}
```

- » Wykorzystanie funkcji do zamiany elementów tablicy
- » Funkcja `swap()` otrzymuje wskaźniki pod którymi są dane, które należy zamienić
- » Problemem w takim zapisie jest zrozumienie czy argumenty są danymi wejściowymi czy wyjściowymi
- » Efektywny sposób przekazywania dużej ilości danych (**zero-copy**)

Funkcje - wskaźnik na wynik

```
#include <iostream>
```

```
using namespace std;
```

```
void setToZero(int *tab, size_t size){  
    for (size_t i = 0; i < size; ++i){  
        tab[i] = 0;  
    }  
}
```

```
int main(){  
    int tab[10];  
    setToZero(tab, 10);  
}
```

- » Zmienna **tab** to jest wskaźnik na pierwszy element tablicy
- » Funkcja **setToZero(...)** modyfikuje dane na które wskazuje wskaźnik a nie samą wartość wskaźnika
- » **Dlatego przekazywanie argumentu przez “kopię” nie ogranicza**

Funkcje - zmienne globalne

```
#include<iostream>
using namespace std;
int tab[10];
```

```
void setToZero(){
    for (size_t i = 0; i < 10; ++i ){
        tab[i] = 0;
    }
}
```

```
int main(){
    // int abc[100]; how to setToZero???
    setToZero();
}
```

- » Dane do przetwarzania jako **zmienna globalna**
- » **Najgorszy pomysł**
 - brak uniwersalności
 - bałagan w kodzie

Funkcje - wyciek pamięci

```
int *createAndSet(size_t size, int value){  
    int *array = new int[size];  
    for (size_t i = 0; i < size; ++i) {  
        array[i] = value;  
    }  
    return array;  
}
```

```
int main(){  
    int *tab;  
  
    tab = createAndSet(10, 666);  
    tab = createAndSet(10, 777);  
  
    delete[] tab;  
}
```

- » Funkcja rezerwuje pamięć: ok
- » Funkcja zwraca przez wartość wskaźnik do niej - ok
- » Kto zwolni pamięć ?!?
- » W przykładzie “memory leak”

```
int *cr  
int *  
for (  
    al  
}  
retur  
}  
  
int ma  
int *  
  
tab :  
tab :  
  
dele  
}
```

In case of fire



1. git commit



2. git push

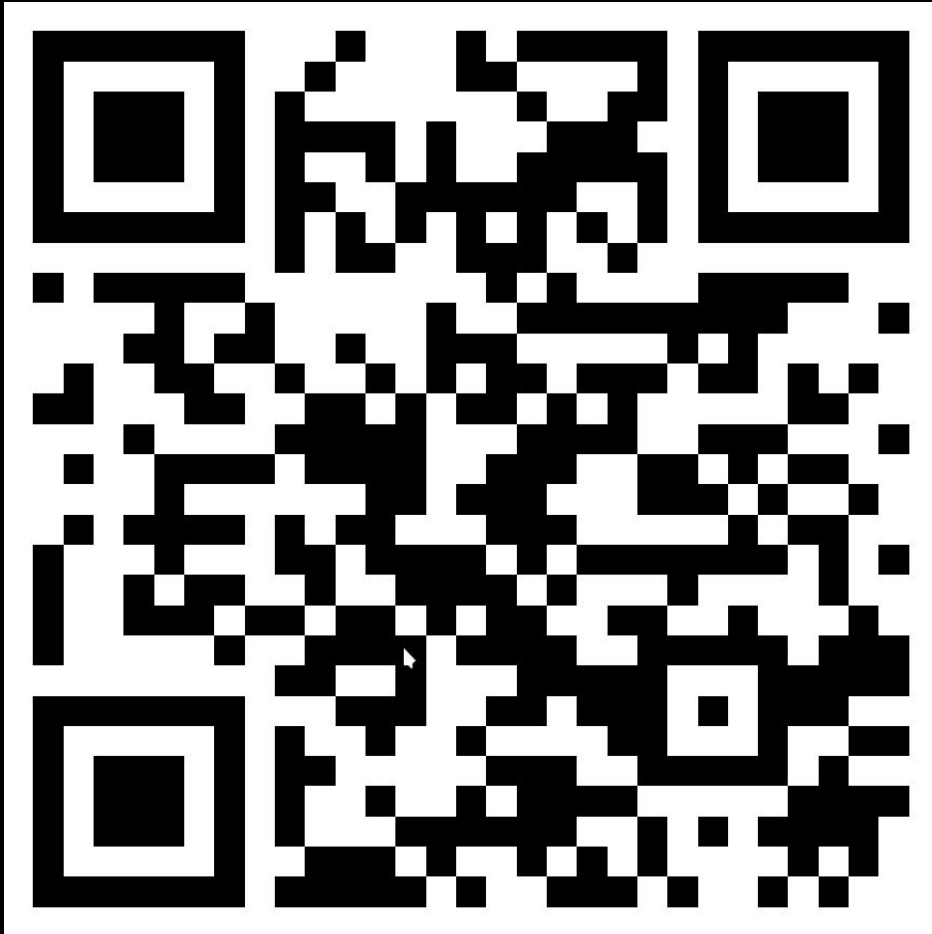


3. leave building

mięć: ok

wartość

ry leak”



quiz

PI07_fun3

socrative.com

- login
- student login

Room name:

KWANTAGH

Pętla w pętli - warunek wyjścia

Problem wyjścia z wewnętrznej pętli (pętli w pętli)

To rozwiązanie jest **najbardziej “eleganckie”**

```
int main(){  
    for (size_t x = 0; x < 10; ++x) {  
        for (size_t y = 0; y < 10; ++y) {  
            if (x > 4 && y > 5) {  
                goto exitLoop;  
            }  
        }  
    }  
    exitLoop:  
    cout << "end" << endl;  
}
```


Pętla w pętli - warunek wyjścia

Problem wyjścia z wewnętrznej pętli (pętli w pętli)

To rozwiązanie jest **najbardziej “eleganckie”**

```
int main(){  
    for (size_t x = 0; x < 10; ++x) {  
        for (size_t y = 0; y < 10; ++y) {  
            if (x > 4 && y > 5) {  
                goto exitLoop;  
            }  
        }  
    }  
exitLoop:  
    cout << "end" << endl;  
}
```

```
void innerLoop(){  
    for (size_t x = 0; x < 10; ++x) {  
        for (size_t y = 0; y < 10; ++y) {  
            if (x > 4 && y > 5) {  
                return;  
            }  
        }  
    }  
}  
  
int main(){  
    innerLoop();  
    cout << "end" << endl;  
}
```

Pętla w pętli - warunek wyjścia

Problem wyjścia z wewnętrznej pętli (pętli w pętli)

To rozwiązanie jest **najbardziej “eleganckie”**

```
int main(){
```

```
    for (size_t x = 0; x < 10; ++x) {
        for (size_t y = 0; y < 10; ++y) {
            if (x > 4 && y > 5) {
                goto exitLoop;
            }
        }
    }
```

```
exitLoop:
```

```
    cout << "end" << endl;
}
```

```
void innerLoop(){
```

```
    for (size_t x = 0; x < 10; ++x) {
        for (size_t y = 0; y < 10; ++y) {
            if (x > 4 && y > 5) {
                return;
            }
        }
    }
```

```
}
```

```
int main(){
    innerLoop();
    cout << "end" << endl;
}
```

Liczby losowy na deterministycznym komputerze

tip: nie da się :-/

generator liczb (pseudo)losowych

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

int main(){

    cout << "0 ... " << RAND_MAX << endl;
    cout << std::rand() << endl;

}
```

- » Funkcja biblioteki standardowej języka C
- » Zwraca wartość całkowitoliczbową w zakresie: 0 RAND_MAX
- » W praktyce jest to typ unsigned int
- » std:: można pominąć

generator liczb (pseudo)losowych

```
#include<iostream>
#include <stdlib.h>
using namespace std;

int main(){

    cout << rand() << endl;

}
```

- » Co jest złego w tym kodzie?
- » Wszystko ok, tylko generuje zawsze te same liczby...
1804289383
1804289383
1804289383
1804289383
1804289383
1804289383

generator liczb (pseudo)losowych

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

int main(){

    cout << rand() << endl;
    cout << rand() << endl;
    cout << rand() << endl;

}
```

- » Rozkład generowanych liczb ma charakter losowy ale jest deterministyczny:
- » Rezultat:
1804289383
846930886
1681692777
- » To nie jest generator liczb losowych tylko pseudolosowych!!!

generator liczb (pseudo)losowych

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

int main(){

    srand(3);
    // srand(0);
    cout << rand() << endl;
    cout << rand() << endl;
    cout << rand() << endl;

}
```

- » Funkcja **srand()** inicjalizuje generator liczb losowych
- .
- .
- .
- wciąż powtarzalne sekwencje tylko inne...

generator liczb (pseudo)losowych

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

int main(){

    srand(time(NULL));
    cout << rand() << endl;

}
```

- » Funkcja `time(NULL)` inicjalizuje generator liczb różnymi wartościami przy każdym* starcie programu
- » Funkcja `time(NULL)` zwraca liczbę sekund, która upłynęła od: 00:00 hours, Jan 1, 1970 UTC current unix timestamp

generator liczb (pseudo)losowych

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

```
int main(){

    srand(time(NULL));
    cout << rand()%10 << endl;
    cout << rand()%100 << endl;
    cout << rand()%1000 << endl;

    for (size_t i = 0; i < 100; ++i) {
        cout << i << ": " << rand()%16 << endl;
    }
}
```

» Przykłady jak uzyskać liczby pseudolosowe z przedziału:

- 0-9
- 0-99
- 0-999
- 0-15

generator liczb (pseudo)losowych

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

int main(){

    srand(time(NULL));

    int size = 100;
    int tab[size];
    for (size_t i = 0; i < size; ++i) {
        tab[i] = rand()%16;
    }
}
```

- » Przykład: zadeklaruj tablicę o rozmiarze **size** i wypełnij ją losowymi wartościami z przedziału 0-15 (4 bity)

Dziękuję