# Methodology and Programming Techniques

**Department of Telecommunications, IEiT**

dr inż. Jarosław Bułat

kwant@agh.edu.pl

# Outline

- Functions:
  - declaration, definition
  - function call
  - passing arguments
  - returned value
- Scope of variable in context of function
- Example of use (many :)

AGH

# Many times I repeat the same code

*“co robić jak żyć?”*

# Functions - definition

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

- Purpose of the function:
  - avoid repeating the same code many times
  - hide the code fragments (implementation details)
- type - type of function == type of return data, may be void
- name - function name (rules for variable names)
- parameter - type and number of parameters passed
- body - instructions inside the function

# Functions - definition

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

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

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

- "type" function - any possible,
  - can be void ie "without type"
  - if defined: return ...;
- Parameters (aguments) - looks like a variable declaration, these variables will be initialized during the function call
- Scope of variables inside pprint(): global only + arguments
- Inside the body of the function I can use arguments as a variable to the end of the function !!!

# Functions - arguments

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

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

int main() {
  int a = 1;

  pprint(a);
  pprint(10);

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

- When calling function,

# Functions - arguments

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

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

int main() {
  int a = 1;

  pprint(a);
  pprint(10);

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

» When calling function,

# Functions - arguments

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

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

int main() {
  int a = 1;

  pprint(a);
  pprint(10);

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

» When calling function,

# Functions - arguments

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

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

int main() {
  int a = 1;

  pprint(a);
  pprint(10);

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

- When calling function, arguments are initialized by assigning values
- The argument is an expression that is evaluated during the call and at the call point

# Functions - arguments

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

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

int main() {
  int a = 1;

  pprint(a);
  pprint(10);

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

- When calling function, arguments are initialized by assigning values
- The argument is an expression that is evaluated during the call and at the call point
- Arguments are passed by copy (assignment)
- Copy means:
  - memory allocation (reservation)
  - changes inside the function are not visible outside

# Functions - arguments

```
#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;
}
```

definition

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

call

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

- Function call
- Function main() is automatically called after program start
- Passing arguments:

# Functions - arguments

- Function call
- Function main() is automatically called after program start
- Passing arguments:

```
#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;
}
```

definition

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

call

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

# Functions - arguments

- Function call
- Function main() is automatically called after program start
- Passing arguments:

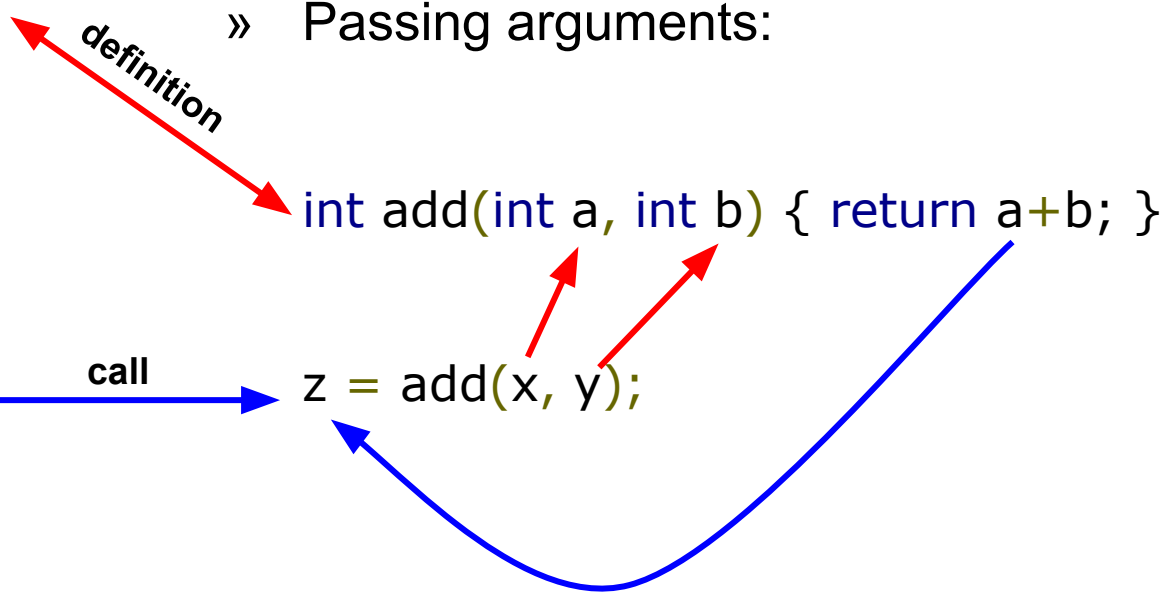
```
#include<iostream>
using namespace std;

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;
}
```



# Functions - scope of arguments

```
#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;
}
```

- The consequences of passing the argument as a copy:
  - range of variable x is limited to pprint(...).
  - change x does not modify x.
- Results:

  0: 6
  1: 6
  2: 7
  3: 6

# Functions - declaration, definition

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



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

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

- I've used f(...) before its definition
- error: 'f' was not declared in this scope

# Functions - declaration, definition

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




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

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

- I've used f(...) before its definition
- error: 'f' was not declared in this scope
- At the function call, there is no known definition of the function

# Functions - declaration, definition

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

int f(int x);

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

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

- I've used f(...) before its definition
- error: 'f' was not declared in this scope
- At the function call, there is no known definition of the function
- To fix it, you need add function declaration.
- Function declarations usually placed at the top of *.cc or in *.h
- Often the program is divided:
  - interface (*.h) - how to use
  - implementation (*.cc)

# Functions - declaration, definition

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

int f(int x);

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

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

← Declaration (how to use the function)

← Call (function call)

← Definition (implementation, body)

# Functions - declaration, definition

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

int f(int);

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

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

← Declaration

» May include types without names, names make documentation clear

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

# Functions - return value

```
#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;
}
```

- The function can return any type
- Keyword return
  - anywhere in the function
  - unconditionally terminates the function
  - if the function has a type, it must return the same type
  - in function without type, return does not return any value (only ends)
- The value returned by copying (can be expensive!)

# Functions - return value

```
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;
}
```

- How to pass more than one variable “by return”
- The function is of the type "Complex"
- Returns the structure
- The structure is assigned to the variable r after the function has been executed
- Assignment by copying, so not very efficient with large amounts of data

# Functions - return value

```
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;
}
```

- Similarly, however, argument is of a “Complex” type
- Assignment by copying, so not very efficient with large amounts of data

# Functions - pointer to the result

```
#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;
}
```

- The argument is a pointer to the result
- Function call arguments indicate where to place the result
- The argument **is copied**
  - impossible to change "back"
  - but the result is saved to the address that indicates!
- The pointer address is not modified, only the data are dereferenced

# Functions - pointer to the result

```
#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;
}
```

- Use a function to replace array elements
- The swap() function receives the pointer to data which should be modified
- The problem with this is to understand whether the arguments are input or output
- **Efficient** way to transfer large amounts of data (**zero-copy**)

# Functions - pointer to the result

```
#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);
}
```

- The tab variable is the pointer to the first element of the array
- The setToZero(...) function modifies the data that is indicated by the pointer and not the pointer value itself
- Therefore, passing the argument by its copy, do not limit C/C++ language

# Functions - global variables

```
#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();
}
```

- Data to be processed as a global variable
  - worst idea (of passing data to function)
  - lack of versatility
  - mess in the code

# Functions - memory leak

```
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;
}
```

- Function memory allocation: ok
- Function returns pointer to the allocated memory - ok
- Who will release memory ?!?
- In this example “memory leak” occurs

# Functions - memory leak

```
int *cr
  int *
  for (
     ar
  }
  retu
}

int ma
  int *

  tab =
  tab =

  delet
}
```


er to the

ory ?!?

ory leak”

# Nested loop - exit condition

Problem with breaking outer loop from internal loop

Presented solution is the **most "elegant"**

```
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;
}
```

# Nested loop - exit condition

Problem with breaking outer loop from internal loop

Presented solution is the **most "elegant"**

```
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;
}
```

# Nested loop - exit condition

Problem with breaking outer loop from internal loop

Presented solution is the **most "elegant"**

```
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;
}
```

# Random numbers on a deterministic computer

tip: imposible :-/

# random number (pseudo)generator

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

int main(){

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

}
```

- Function from standard C library
- Returns integer value in the range: 0 .... RAND_MAX
- it is an unsigned int type
- std:: can be omitted

# random number (pseudo)generator

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

int main(){

  cout << rand() << endl;

}
```

- What's wrong with this code?
- It is ok, only generates the same numbers...
  1804289383
  1804289383
  1804289383
  1804289383
  1804289383
  1804289383
- This is not a random number generator it is  pseudo-random

# random number (pseudo)generator

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

int main(){

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

}
```

- The distribution of the generated numbers is random but is deterministic
- Result:
  1804289383
  846930886
  1681692777

# random number (pseudo)generator

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

int main(){

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

}
```

» The srand() function initializes the random number generator

.

.

.

still repeating sequences only different one...

# random number (pseudo)generator

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

int main(){

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

}
```

- The time(NULL) function initialize a random number generator with different values at each* program start
- The time(NULL) function returns the number of seconds elapsed since:
  `00:00 hours, Jan 1, 1970 UTC`
  `current unix timestamp`

# random number (pseudo)generator

```
#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;
   }
}
```

- Examples of how to get pseudo-random numbers from the following ranges
  0-9
  0-99
  0-999
  0-15

# random number (pseudo)generator

```
#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;
   }
}
```

» Example: declare an array of the size **size** and fill it with random values from 0-15 (4 bits)

# Thank you