

# COMP 2014 Object Oriented Programming

---

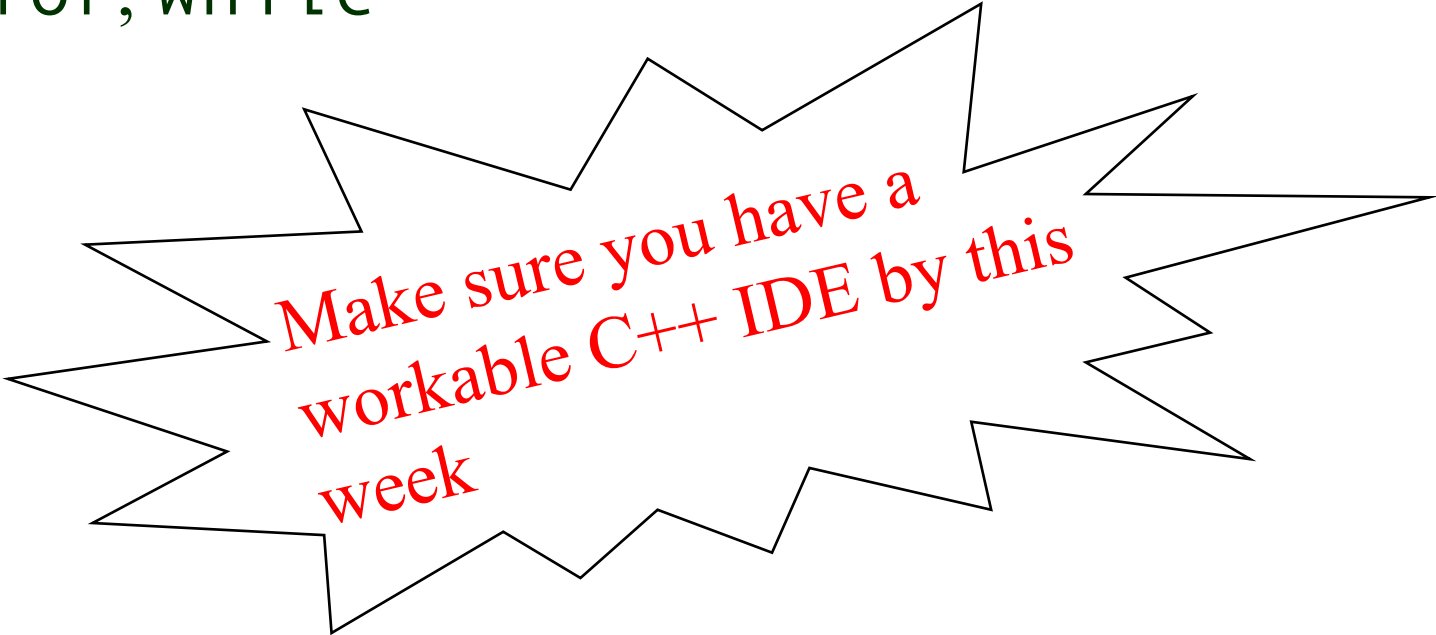
## Lecture 2

### Function calls and parameter passing

# Topics covered in last lecture

---

- ◆ Basic syntax of C++
- ◆ Input and output
- ◆ Flow of control
  - Branching: if-else, switch
  - Loops: for, while



Make sure you have a  
workable C++ IDE by this  
week

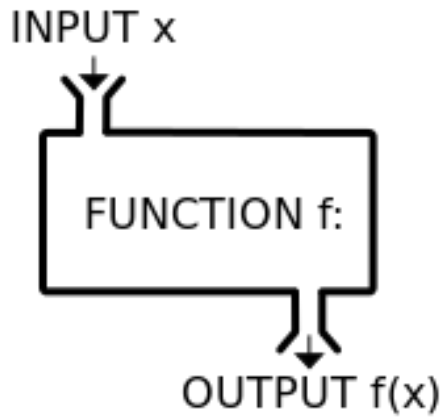
# Topics covered by the lecture

---

- ◆ Concept of function
- ◆ Write a function in C++
- ◆ Parameter passing in a function
  - Call-by-value
  - Call-by-reference
- ◆ Variable scope
- ◆ Function overloading
- ◆ Default arguments

# The concept of function

**Wikipedia Definition:** A function, also known as procedure or subroutine, takes an input and returns an output.



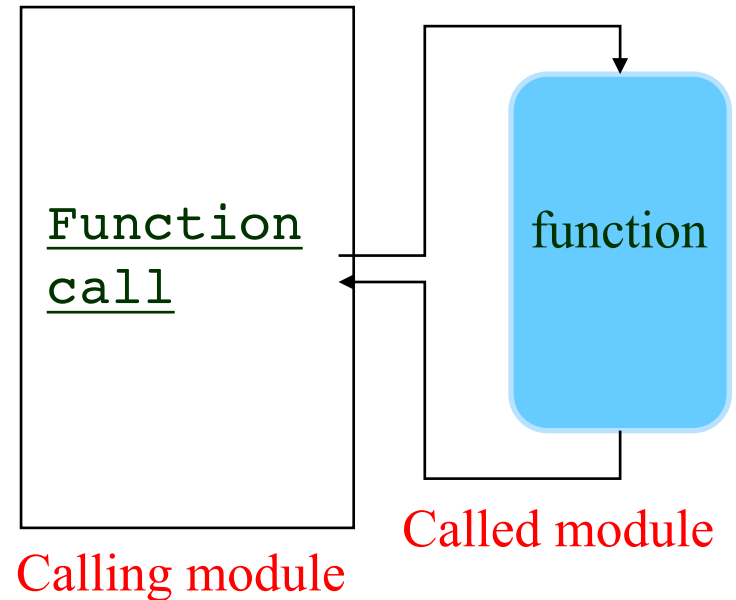
$$f(x) = x^2 + 1$$

$$f(1) = 2$$

$$f(2) = 5$$

$$f(-2) = 5$$

$$f(f(2)) = ?$$



A procedural program is composed of one or more modules, each of which consists of procedures, functions or subroutines.

The miracle of a function is that you can call it several times without extra “cost” of coding.

# Example of a function in C++

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

```
int f(int x) {
    int y = x*x+1;
    return y;
}
```

```
int main() {
    int x;
    cout <<"Input the value of x" <<endl;
    cin >> x;

    cout <<"f(" << x << ")=" << f(x) << endl;
    return 0;
}
```

Function declaration and definition

$$f(x) = x^2 + 1$$

$$f(1) = 2$$

$$f(2) = 5$$

$$f(-2) = 5$$

$$f(f(2)) = ?$$

Function call

# Example of a function in C++

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

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

```
int main() {
    int x;
    cout << "Input the value of x" << endl;
    cin >> x;

    cout << "f(" << x << ")=" << f(x) << endl;
    return 0;
}
```

```
int f(int x) {
    int y = x*x+1;
    return y;
}
```

Function declaration

$$f(x) = x^2 + 1$$

$$f(1) = 2$$

$$f(2) = 5$$

$$f(-2) = 5$$

$$f(f(2)) = ?$$

Function call

Function definition

# Function declaration, definition and call

◆ **Function declaration** (function prototype): *three components in a function declaration,*

- Return type (can be void or any data type)
- Function name (better be meaningful)
- Parameter list (can be none or a few)

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

Can either `int f(int)` or `int f(int x)`. Another example:

```
double monthlyPayment(double, int, double);
```

```
double monthlyPayment(double principal, int years, double rate);
```

Formal arguments

◆ **Function definition**: the code of the function.

- Three parts: *inputs, process & outputs*

◆ **Function call**: use the function.

Format: `FunctionName(Actual arguments)`.

```
int f(int x) {  
    int y = x*x+1;  
    return y;  
}
```

```
f(2);
```

◆ Any function should be **declared/defined** before it is called.

# Return Statements

- ◆ “**return**” statement transfers control back to “*calling*” module
- ◆ Return type can be any valid type, including user defined data type and “**void**”, but needs to match with the function declaration.
- ◆ Any function must have a return statement unless its type is **void**.

```
int main() {  
    ...  
    double result = areaOfCircle(10);  
    cout << result;  
    ....  
}
```

```
double areaOfCircle(int radius) {  
    double area = 3.14*radius*radius;  
    return area;  
}
```

Mind type matching



# Main function

---

- ◆ Any C++ program must have a **main** function and have only one **main** function
- ◆ `main()` normally has **int** type and **returns** 0.
- ◆ `main()` is called by Operating System and **can only** be called by Operating System.
- ◆ If there are more than one free functions, the main function stay at the very bottom unless all other functions are declared on top of it.
- ◆ With OO paradigm, the main function is normally very small, mostly acts as a class driver (as in Java or other OO-based programming languages).

# A not-that-simple example

Calculate monthly payment of a loan, given the input:

- The loan amount: *principal*
- The length of the loan: *years*
- The interest rate (in percentage): *rate*

$$\text{payment} = \text{principal} \times \left( i + \frac{i}{(1+i)^{\text{months}} - 1} \right)$$

Where  $\text{months} = \text{years} * 12$

$i = \text{rate} / (12 * 100)$

# A not-that-simple example of functions

Return type

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

Function name

```
double monthlyPayment(double principal, int years, double rate) {
    int months = years * 12;
    double i = rate/(12.0*100.0);
    double payment = principal * (i + i/(pow(1+i, months)-1));
    return payment;
}
```

formal arguments

Return statement

```
int main() {
    double principal;
    int years;
    double rate;
    cout << "Input the principal, years and rate" << endl;
    cin >> principal >> years >> rate;
    cout << "Monthly payment is " << monthlyPayment(principal, years, rate);
    return 0;
}
```

Variables that can hold values temporally

function call

actual arguments

# Alternative Function Declaration

---

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

```
double monthlyPayment(double, int, double); //function prototype
```

```
int main() {
    double principal;
    int years;
    double rate;
    cout << "Input the principal, years and rate" << endl;
    cin >> principal >> years >> rate;
    cout << "Monthly payment is " << monthlyPayment(principal, years, rate);
    return 0;
}
```

```
double monthlyPayment(double principal, int years, double rate) {
    int months = years * 12;
    double i = rate/(12.0*100.0);
    double payment = principal * (i + i/(pow(1+i, months)-1));
    return payment;
}
```

# Parameters/arguments

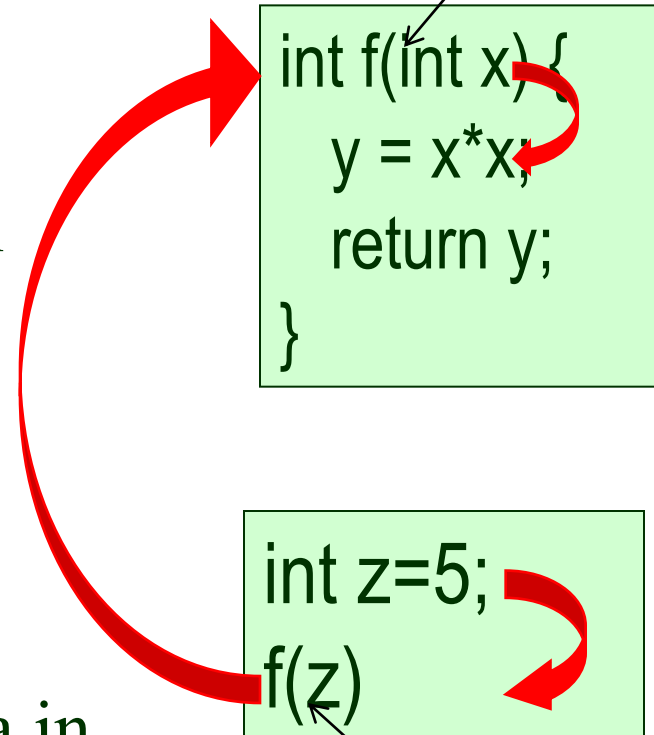
- ◆ Formal arguments:
  - placeholders/variables
  - declared in function declaration
- ◆ Actual arguments
  - data carriers
  - used in the function call
- ◆ Parameter passing
  - actual arguments carry data in calling module and pass them to the called function via formal arguments

formal argument

```
int f(int x) {  
    y = x*x;  
    return y;  
}
```

```
int z=5;  
f(z)
```

actual argument



# Parameter passing

```
#include<iostream>

using namespace std;

void Swap(int a, int b) {
    int temp;

    temp=a;
    a=b;
    b=temp;
}

int main() {
    int a=5;
    int b=8;

    cout << "a=" << a << "; b=" << b << endl;

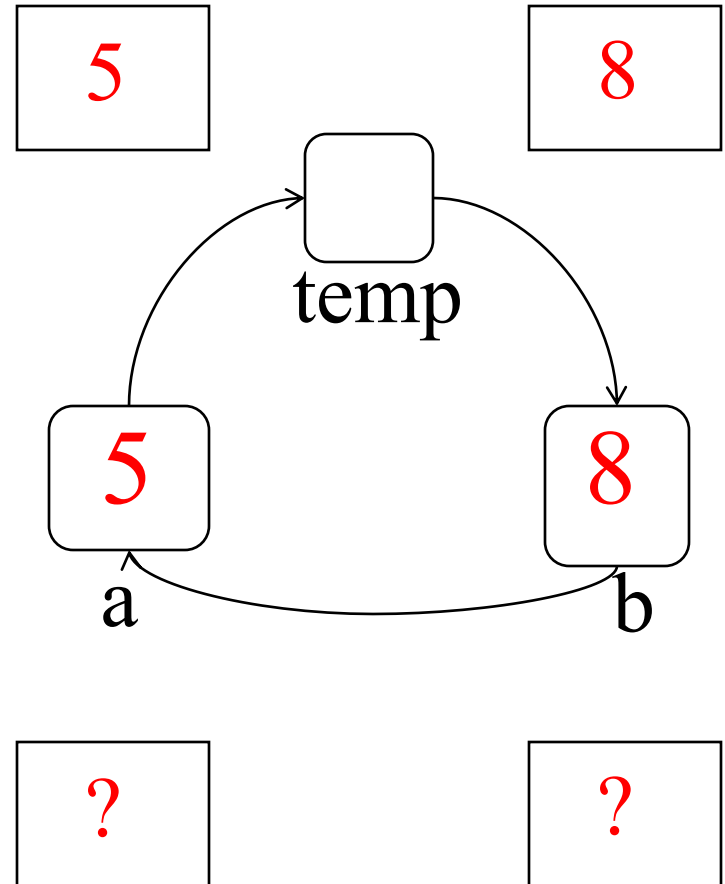
    Swap(a,b);

    cout << "a=" << a << "; b=" << b << endl;
    return 0;
}
```

**formal arguments**

**actual arguments**

Swap two numbers



Call by value

# Call-by-Value Parameters

---

- ◆ The values of the actual arguments are passed (**copied**) to the formal arguments of the called function.
- ◆ The values are **stored** in the formal arguments, which can be considered as "local variables" of the function
- ◆ If the value of the formal arguments are modified, only the “local copy” changes. The function has no access to “actual argument” from the caller.

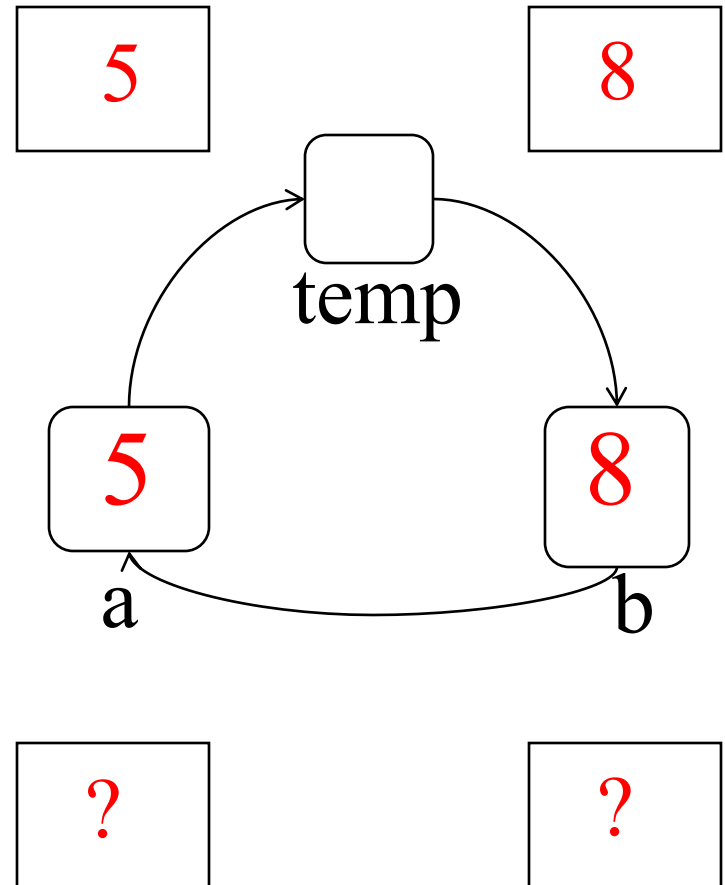
# Another way of parameter passing

Address of operator

```
#include<iostream>
using namespace std;
void Swap(int& a, int& b) {
    int temp;

    temp=a;
    a=b;
    b=temp;
}
void main() {
    int a=5;
    int b=8;

    cout << "a=" << a << "; b=" << b << endl;
    Swap(a,b);
    cout << "a=" << a << "; b=" << b << endl;
}
```



Call by reference



# Call-By-Reference Parameters

---

- ◆ A formal argument that is specified by ampersand “&” is called **reference argument**.
- ◆ The “addresses” of the actual arguments are passed to the reference arguments of the function when the function is called.
- ◆ The reference arguments in the function not only can get the values from those addresses but also can modify the data in the addresses.
- ◆ The calling module can then see the changed values in those addresses.
- ◆ Call-by-reference can be:
  - **Dangerous**: because caller’s data can be changed
  - **Useful**: often is desired, especially when the function has more than one return values. Avoiding **object slicing** is another motivation.
- ◆ There is no such alternation in Java (no call by reference in Java).

# Call-By-Reference vs Call-By-Value

*pass by reference*

cup = 

fillCup(       )

*pass by value*

cup = 

fillCup(       )

# In-class practice

---

Write a function that takes a two-digit integer and return the two digits of the number. For example,

95 => 9 and 5

14 => 1 and 4

See twodigits.cpp

# Scope rules in C++

- ◆ A program can contain number of blocks
  - A block is normally bounded by curly brackets.
  - All **function definitions** are blocks.
  - **while/if-else statements** are blocks

```
int sum = 0;  
for (int i=0; i<10; i++)  
    sum += i;
```

- ◆ A variable declared in a block is valid only in that block

```
int main() {  
    int x = 10;  
    cout << x;  
    return 0;  
}
```

```
int main() {  
    { int x = 10; }  
    cout << x;  
    return 0;  
}
```

- ◆ A variable in an outer block has access to inner blocks not the other way around.

# Local variables to a function

---

- ◆ Local variables
  - Declared inside of a function
  - Available only within that function
- ◆ Can have variables with same names declared in different functions
  - Scope is local: "that function is it's scope"
- ◆ Local variables are preferred: *maintain individual control over data*



Dongmo likes local variables

# Global variables

- ◆ A variable is not declared in any block is called **global variable**
- ◆ A global variable has access to all blocks, especially *accessible by any **functions** and **classes** in the same file.*
- ◆ Global variables may be used for **constants**, say Pi, or for convenience (to avoid too much parameter passing).

```
#include<iostream>
int value = 10;
void change() {
    value *=10;
}

void main() {
    cout << value << endl;
    change();
    cout << value << endl;
}
```

Global variable

Give me your reasons  
whenever you use a  
global variable;  
otherwise, you lose  
marks!

# Function Overloading

C++ is a strongly typed language. Change parameter list of a function will lead to “*another*” function.

```
double average(int n1, int n2) {  
    return ((n1 + n2) / 2.0);  
}  
  
double average(double n1, double n2) {  
    return ((n1 + n2) / 2.0);  
}
```

- ◆ **Function overloading**: the same function name with different parameters (different types or different number of parameters) represents different functions.
- ◆ **Function *signature***: function name & parameter list
  - Must be "unique" for each function definition

# Function call: overloading resolution

- ◆ Given following functions:
  - 1) `void f(int n, double m);`
  - 2) `void f(double n, int m);`
  - 3) `void f(double n, double m);`
- ◆ These calls:
  - `f(5.3, 4);` → Calls 2)
  - `f(4.3, 5.2);` → Calls 3)
  - `f(98, 99);` → Calls ??

See `functioncall.cpp`



# Default Arguments in a function

---

When making a function call, the calling module must provide all values for all formal arguments of the function. However, if part of the formal arguments are given **default** values, the calling module **does not have to** provide values for **those** arguments, in which case, the default values will be used. For example, the following code fragment is illegal:

```
double roundup(double, int); //function prototype
void main() {
    ...
    cout << roundup(3.14159); //illegal, too few parameters
}
```

However, it is ok if we declare the function as follows:

```
double roundup(double, int = 2); //function prototype
```

# Rules for Default Arguments

---

The two rules of using default parameters are:

1. Default values **can only be assigned** in the function prototype (declaration).
2. It is not necessary that all arguments of a function have to be default arguments but **the default arguments must filled from right to left**.

`double roundup(double = 0.0, int = 2);`//function prototype

`double roundup(double, int = 2);`//function prototype

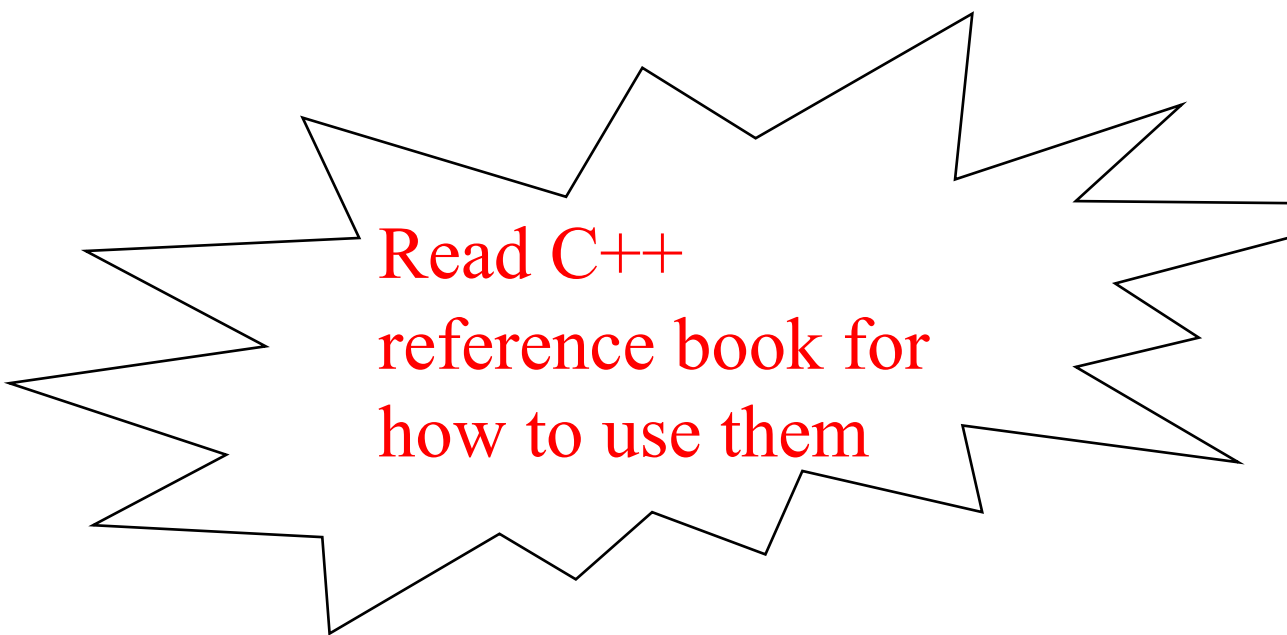
These rules allow C++ not to get '**mixed up**' when matching actual arguments with formal arguments.

`roundup(); roundup(3.14159); roundup(3.14159, 4);`

*Class constructors may take advantage of this capability to initialise data members either explicitly through actual arguments or defaulting to a set of values specified in the function prototype.*

# Pre-defined functions

- ◆ Plentiful pre-defined functions for variety of applications. They are built in library.
- ◆ They are different with different IDEs, some of them inherited from C (in `cstdlib`).
- ◆ Most useful ones are:
  - `<cmath>`
  - `<cstdlib>`
  - `<ctime>`
  - `<algorithm>`
  - `<iomanip>`



Read C++  
reference book for  
how to use them

# Some pre-defined functions

**Display 3.2**    **Some Predefined Functions**

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0, 3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath

# Some pre-defined functions

ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End pro- gram	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

# Summary

---

- ◆ Formal parameter is placeholder, filled in with actual argument in function calls
- ◆ Call-by-value parameters are "local copies" in receiving function body
  - Actual arguments cannot be modified
- ◆ Call-by-reference passes memory addresses of actual arguments
  - Actual arguments can be modified
  - If you don't want the function changes the value of an actual argument, use call-by-value or make the formal argument to be constant
- ◆ Same function name can define multiple: called function overloading

# Summary

---

- ◆ Default arguments: some arguments in a function can be provided with default values in the function declaration.
- ◆ Scope rules:
  - A variable is valid only in its scope
  - Local variables can only be accessed locally.
  - A variable that does not belong to any block is called global variable, which is accessible by any function in the program
- ◆ Pre-defined functions: the functions that have been implemented and placed in C/C++ library.

# Homework

---

- ◆ Read textbook: Chapter 3-4
- ◆ Complete Practical 2 which will be due in week 3
- ◆ Warm up textbook Chapter 5.