

# Programming with C++

## COMP2011: Function I

Cecia Chan

Brian Mak

Dimitris Papadopoulos

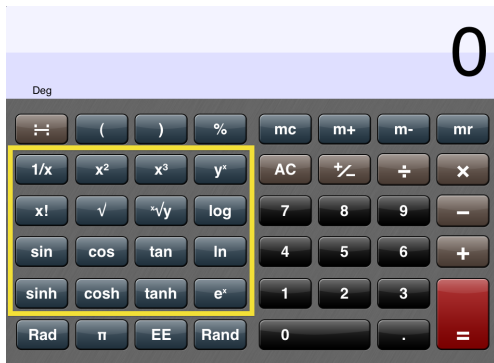
Pedro Sander

Charles Zhang

Department of Computer Science & Engineering  
The Hong Kong University of Science and Technology  
Hong Kong SAR, China



# Scientific Functions

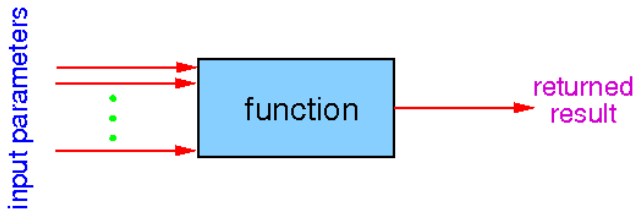


$$z = f(u, v, w, x, y, \dots)$$

- Example:  $\sin(x)$ ,  $\cos(x)$ ,  $\log(x)$ ,  $\text{sqrt}(x)$ , etc.
- We would like to **generalize** the notion of functions in programming languages.

# Part I

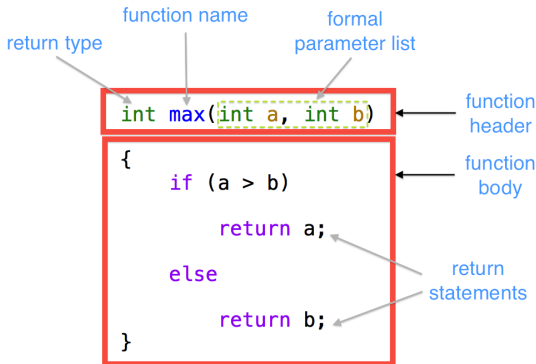
## Function Basics



# Basic Function Syntax

## Syntax: Function Definition

```
<return-type> <function-name> ( <formal-parameter-list> )  
{ <function-body> }
```



# Function Name and Function Call

- Any legal C++ identifier can be used for `<function-name>`.
- Just like naming variables and constants, you should use meaningful names for function names.
  - The name should describe what the function does.
- The function name “`main`” is reserved; you must define it, and define it exactly `once`.
  - Recall that each program can only have one “`main( )`” function.
  - When a program is run, the `shell` — command interpreter of the operating system — looks for the `main( )` function and starts execution from there.

## Syntax: Function Call

`<function-name> ( <actual-parameter-list> )`

# Formal Parameter List & Actual Parameter List

```
/* max function definition */  
int max(int a, int b) { return (a > b) ? a : b; }  
/* max function call */  
cout << max(5, 8) << endl;
```

- **<formal-parameter-list>** appears in the **function definition**: it is a list of **variable declarations** separated by **commas**.

Syntax: **<formal-parameter-list>**

**<type<sub>1</sub> variable<sub>1</sub>>, <type<sub>2</sub> variable<sub>2</sub>>, ..., <type<sub>N</sub> variable<sub>N</sub>>**

- **<actual-parameter-list>** appears in a **function call**: it is a list of **comma-separated objects** passed to the called function.

Syntax: **<actual-parameter-list>**

**<object<sub>1</sub>>, <object<sub>2</sub>>, ..., <object<sub>N</sub>>**

- There is a **one-to-one** correspondence between the **actual parameters** (aka **arguments**) and the **formal parameters**.

# Formal Parameter List & Actual Parameter List ..

- During the function call, the following **initializations** are performed,

$$\begin{aligned} &< type_1 \ variable_1 >= object_1, \\ &< type_2 \ variable_2 >= object_2, \\ &\vdots \\ &< type_N \ variable_N >= object_N \end{aligned}$$

- Since C++ is a **strongly typed** programming language, the data type of an **actual parameter** and its corresponding **formal parameter** must be the same or “matched”.
- A C++ compiler will perform **type checking** to make sure that their types match with each other.
- Exception:** unless an automatic type conversion — **coercion** — can be done, like normal initialization or assignment of an object to a variable of a different type. (More about that later.)

# Function Header & Function Body

- In the function syntax, the first line

`<return-type> <function-name> (<formal-parameter-list>)`

is also called the **function header**, and the rest is the **function body** enclosed in curly braces.

- The **<function-body>** usually consists of the following parts:
  - **constant** declarations
  - **variable** declarations and definitions
  - other C++ statements
  - **return** statement
- It is legal to have an **empty** function body!
- The curly braces must be there, even if there is **zero** or only **one** single statement inside the function body! (That is different from the **if**-statement or **while**-statement, etc. )



# Return Type

- Usually a function returns something — in C++, we call it an object.
- The returned object may be
  - a **signal** to tell the caller about the status of the function: does it run successfully? does it fail?
  - the **result** of some computation. e.g., factorial, sum, etc.
  - a **new object** created by the function. e.g., a new window.
- **<return-type>** specifies the data type of the **single** returned object.
- **<return-type>** can be any of the C++ built-in data types (e.g., char, int, etc.) or user-defined types, **except the array type**. (Array type will be talked later.)

**Question:** Since only a single object is returned by a function, how can you return multiple objects back to the caller?

## Syntax: **return** Statement

**return** < *expression* > ;

- The **return** statement generally returns “2” things to the caller:
  - **program control**: it stops running the called function, and the function caller takes back the control and continue its execution.
  - **an object**: the object (or value) represented by the < *expression* > is returned to the caller.
- The value of < *expression* > in the **return** statement should have the same type as the <**return-type**>. Or, if it can be converted to the <**return-type**> by **coercion**, otherwise it will be a compilation error.
- If a function has a return value, the function body must have at least one **return** statement.

# Example: max

```
#include <iostream>      /* File: max.cpp */
using namespace std;     /* To find the greater value between x and y */

int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
} // Question: can you write with only 1 return statement?

int main()
{
    int x, y;
    cout << "Enter 2 numbers: ";
    cin >> x >> y;

    cout << "The bigger number is " << max(x, y) << endl;
    return 0;
}
```

# Example: Euclidean Distance



# Example: Euclidean Distance ..

```
#include <iostream>      /* File: distance-fcn.cpp */
#include <cmath>          // Math library info
using namespace std;

/* To find the Euclidean distance between 2 points */
double euclidean_distance(double x1, double y1, double x2, double y2)
{
    double x_diff = x1 - x2;
    double y_diff = y1 - y2;
    return sqrt(x_diff*x_diff + y_diff*y_diff);
}

int main()    /* To find the length of the sides of a triangle */
{
    double xA, yA, xB, yB, xC, yC;
    cout << "Enter the co-ordinates of point A: "; cin >> xA >> yA;
    cout << "Enter the co-ordinates of point B: "; cin >> xB >> yB;
    cout << "Enter the co-ordinates of point C: "; cin >> xC >> yC;

    cout << " AB = " << euclidean_distance(xA, yA, xB, yB) << endl;
    cout << " BC = " << euclidean_distance(xB, yB, xC, yC) << endl;
    cout << " CA = " << euclidean_distance(xC, yC, xA, yA) << endl;
    return 0;
}
```

## void: a New Type

- “**void**” means *nothing, emptiness*.
- A function that returns nothing back to the caller has a **return type** of **void**.
- A function that does not take any arguments from the caller may
  - leave the <formal-parameter-list> empty.

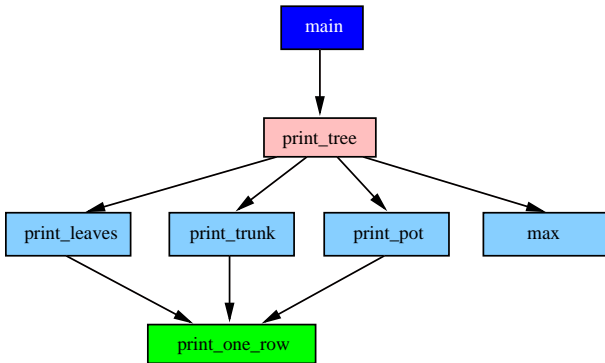
```
int fcn_example( ) { ... }
```

- put the <formal-parameter-list> as **void**.

```
void print_hkust(void) { cout << "hkust" << endl; }
```

## Part II

# Modular Programming




# Why Modular Programming?

- In reality, many application software contains hundreds of thousands, or even million lines of code.
- A common approach to solve complex problems is "divide-and-conquer": divide the problem into smaller parts, and then solve each part in turn.
- In programming, we divide a large program into **modules**, each of which is implemented by a **function**.
- This is called **modular programming**, or **top-down programming**.



## Example: Draw a Tree — Problem Statement

[illegible]

- Task: To draw a tree; the drawing consists of
    - a triangular top (the leaves)
    - a rectangular trunk
    - a trapezium pot
  - Requirements: A user may define
    - the drawing symbols of the 3 parts.
    - the height of the tree.
  - For simplicity, other dimensions (width/height of leaves, trunk, pot) are computed from the tree height using pre-determined formulas.
- 
- A small, stylized green tree with a dense, rounded canopy is planted in a terracotta-colored, ribbed pot. The pot is positioned on a white surface, casting a soft shadow. The tree's trunk is thin and brown, extending from the pot to the base of the foliage.

## Example: Draw a Tree — main( )

```
#include <iostream>      /* File: draw-tree.cpp */
using namespace std;

// Definition of other functions go here ...

int main()
{
    char tree_symbol, trunk_symbol, pot_symbol;
    int tree_height;

    cout << "Enter the character symbols for tree, trunk, and pot: ";
    cin >> tree_symbol >> trunk_symbol >> pot_symbol;

    cout << "Enter height of the tree (an odd integer, please): ";
    cin >> tree_height;

    cout << endl << endl << endl;
    print_tree(tree_height, tree_symbol, trunk_symbol, pot_symbol);

    return 0;
}
```

# Example: Draw a Tree — other Functions

```
int max(int a, int b) { return (a > b) ? a : b; }
```

```
void print_one_row(int num_leading_spaces, int num_symbols, char symbol)
{
    for (int j = 0; j < num_leading_spaces; ++j)
        cout << ' ';

    for (int j = 0; j < num_symbols; ++j)
        cout << symbol;

    cout << endl;
}
```

```
void print_leaves(int tree_height, char tree_symbol)
{
    for (int row = 0, num_leading_spaces = tree_height;
         row < tree_height;
         ++row, --num_leading_spaces)
        print_one_row(num_leading_spaces, 2*row+1, tree_symbol);
}
```

## Example: Draw a Tree — other Functions ..

```
void print_trunk(int tree_width, int trunk_height,
                int trunk_width, char trunk_symbol)
{
    int num_leading_spaces = (tree_width - trunk_width)/2;

    for (int row = 0; row < trunk_height; ++row)
        print_one_row(num_leading_spaces, trunk_width, trunk_symbol);
}
```

```
void print_pot(int tree_width, int pot_height,
               int pot_base_width, char pot_symbol)
{
    int num_leading_spaces = (tree_width - pot_base_width)/2;
    int row = pot_height, width = pot_base_width;

    while (row > 0)
    {
        print_one_row(num_leading_spaces, width, pot_symbol);
        --row;
        ++num_leading_spaces;
        width -= 2;
    }
}
```

## Example: Draw a Tree — other Functions ..

```
/****** The exact formulas are not important! *****/
* The leaves have  $2n+1$  symbols on the  $n$ -th row.
* The trunk is just a rectangle: width = height =  $2/3$  of tree's height.
* pot's height =  $1/3$  of tree's. pot's width =  $2/3$  of tree's.
*/
void print_tree(int tree_height,
               char tree_symbol, char trunk_symbol, char pot_symbol)
{
    int tree_width = 2*tree_height + 1;
    int trunk_width = max(1, tree_width/6);
    int trunk_height = max(1, tree_height/3);
    int pot_width = tree_width * 2/3;
    int pot_base_width = (pot_width % 2) ? pot_width : pot_width + 1;
    int pot_height = max(2, tree_height/3);

    print_leaves(tree_height, tree_symbol);
    print_trunk(tree_width, trunk_height, trunk_width, trunk_symbol);
    print_pot(tree_width, pot_height, pot_base_width, pot_symbol);
}
```

# Remarks: Why Function?

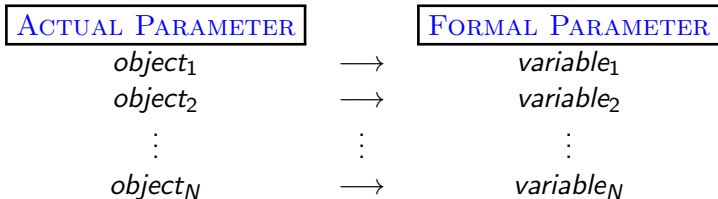
- When you have several segments of codes doing similar things, then they are good candidates for a function.
- A function allows “**write-once-call-many**”: you only write it once they can be called **many times** in the same program with the same or different arguments.
- Functions make programs **easier** to **understand**.
- Functions make programs **easier** to **modify**.
- Functions allow **reusable code**. (e.g., log, sqrt, sin, etc.)
- Functions separate the **concept** (what is done) from the **implementation** (how it is done).
- The last two remarks lead to the creation of **binary libraries** which are a set of compiled functions. These libraries can be **shared**, yet the users do **not** know their implementation. (You'll learn how to do this later.)

## Part III

# Parameter Passing Methods



# How Actual Parameters are Passed to Formal Parameters



- C++ supports 2 ways to pass arguments to a function:
  - 1 pass-by-value (PBV), or call-by-value (CBV)
  - 2 pass-by-reference (PBR), or call-by-reference (CBR)
- Notice that if you call a function with an expression, the expression is first evaluated, and the result is then passed to the function.  
e.g.,  $\max(3 + 5, 2 + 9) \rightarrow \max(8, 11)$  before calling the max function.



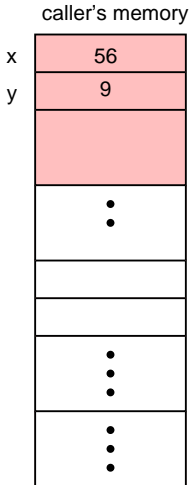
# Pass-by-Value

- In **pass-by-value**, the **value** of an actual parameter is **copied** into the formal parameters of the function.
- If the actual parameter is a **literal constant** (e.g., calling `max(2, 3)`), obviously it won't change.
- If the actual parameter is a **variable** (e.g., calling `max(x, y)`), only its **value** is **copied** to the function, otherwise it has nothing to do with the operation of the function. In particular, its value **cannot be modified** by the function.
- All the function examples presented so far use **pass-by-value** to pass the arguments.

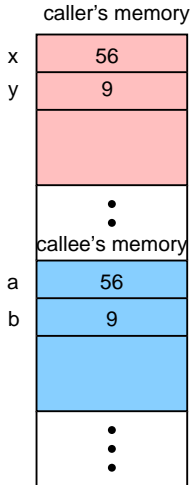
**Question:** What happens if the argument is a big object (e.g. of several MB)?

# Pass-by-Value Illustration

Before calling max(x, y)



After calling max(x, y)



## Syntax: Reference Variable Definition

```
<type> & <variable1> = <variable2>;  
<type> & <variable1> = <variable2>;  
...
```

- A **reference variable** is an **alias** of another variable.
- A **reference variable** must always be **bound** to an object. Therefore, it must be **initialized** when they are **defined**.
- Once a **reference variable** is defined and bound with a variable, you cannot “re-bind” it to another object.

In the example,

- Variables  $a, x, w$  all refer to the **same** integer **object**; similarly, variables  $b, y, z$  also all refer to the **same** integer **object**.
- Variables  $a, x, w$  share the **same memory space**, so that you may modify the value in that memory space through any of them! (Same for  $b, y, z$ .)
- In the line `z = a;`, the **reference variable**  $z$  is not re-bound to  $a$ , but the value of  $a$  is assigned to  $z$ .

# Example: Reference Variables

```
#include <iostream>      /* File: ref-declaration.cpp */
using namespace std;

int main()
{
    int a = 1, b = 2;
    int& x = a;           // now x = a = 1
    int &y = b;           // now y = b = 2
    int &w = a, &z = y; // now w = a = x = 1, z = b = y = 2

    a++;      cout << a << '\t' << x << '\t' << w << endl;
    x += 5;    cout << a << '\t' << x << '\t' << w << endl;
    a = w - x; cout << a << '\t' << x << '\t' << w << endl;

    y *= 10; cout << b << '\t' << y << '\t' << z << endl;
    b--;     cout << b << '\t' << y << '\t' << z << endl;
    z = 999; cout << b << '\t' << y << '\t' << z << endl;

    z = a;           // that is not re-binding z to a
    cout << b << '\t' << y << '\t' << z << endl;
    return 0;
}
```

# Pass-by-Reference Example: swap

```
/* File: pbr-swap.cpp */
#include <iostream>
using namespace std;

void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}

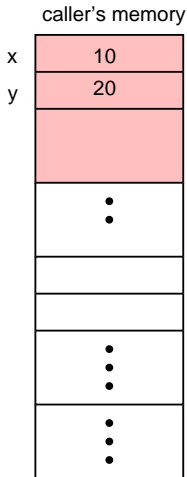
int main()
{
    int x = 10, y = 20;
    swap(x, y);
    cout << "(x , y) = " << '(' << x
         << " , " << y << ')' << endl;
    return 0;
}
```

```
// execution of swap is
// equivalent to running
// the following codes
int& a = x;
int& b = y;
int temp = a;
a = b;
b = temp;

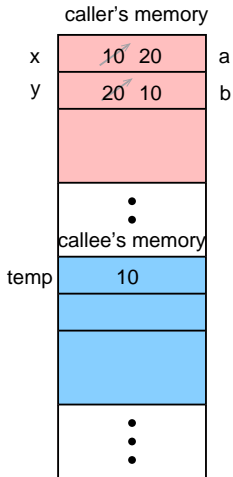
// OR, equivalently
int temp = x;
x = y;
y = temp;
```

# Pass-by-Reference Illustration

Before calling swap(x, y)



After calling swap(x, y)



# Pass-by-Reference

- **Pass-by-reference** does not copy the value of **actual parameters** to the **formal parameters** of the function.
- When an **actual parameter** is **passed by reference**, its corresponding **formal parameter** becomes its **reference variable** (alias).
- In the swap example, on entering the swap function, the following codes are run: 

`int& a = x;    int& b = y;`

That is, the **formal parameters** *a* and *b* are declared as **reference variables** and are initialized or bound to their corresponding **actual parameters** *x* and *y*, respectively.
- You must add the symbol “&” after the type name of the **formal parameter** if you want **pass-by-reference**.
- When an **actual parameter** is **passed by reference** to its **formal parameter**, since they **share the same memory**, any modification made to the **formal parameter** also changes the value of the corresponding **actual parameter**.

# Example: Sort 3 Numbers

```
#include <iostream>          /* File: sort3.cpp */
using namespace std;

void swap(int& x, int& y) /* To swap 2 numbers */
{
    int temp = x;
    x = y;
    y = temp;
}

int main()                  /* To sort 3 numbers in ascending order */
{
    int x, y, z;
    cout << "Enter 3 numbers, x, y, z: ";
    cin >> x >> y >> z;

    if (x > y) swap(x, y);
    if (x > z) swap(x, z);
    if (y > z) swap(y, z);

    cout << "x , y , z = " << x << " , " << y << " , " << z << endl;
    return 0;
}
```



- Function call has **higher precedence** than other operators.  
e.g., the 2 statements in the following example:

```
if (factorial(x) < 200) return -1;  
if (max(a, b) > 10) return -1;
```

may be shortened as

```
if (factorial(x) < 200 || max(a, b) > 10) return -1;
```

The function call `factorial(x)` is executed first, and the returned value is then compared with 200 by the logical `<` operator. The 2nd function call behaves similarly. Finally, the logical `||` operator combines the 2 comparison results.

- Before C++11, you **cannot** define a function inside another function. In other words, all C++ functions, except private class member functions (more about them in *C++ Classes*), are **global** — that is, any C++ function can be called by any other C++ functions if they are properly declared (more about that in *Scope*).
- After C++11, you **can** define local functions inside another function by the **lambda expression**.
- For a function with more than 1 formal parameter, some of them may get their values using **pass-by-value**, while others using **pass-by-reference**. There is no restriction on their number and order.

# Example: Some PBV, Some PBR

```
#include <iostream>          /* File: sum-and-difference.cpp */
using namespace std;

// To find the sum and difference of 2 given numbers
void sum_and_difference(int x, int y, int& sum, int& difference)
{
    sum = x + y;
    difference = x - y;
}

int main()
{
    int x, y, sum, difference;
    cout << "Enter 2 numbers: ";
    cin >> x >> y;

    sum_and_difference(x, y, sum, difference);
    cout << "The sum of " << x << " and " << y << " is " << sum << endl;
    cout << "The difference between " << x << " and " << y << " is "
        << difference << endl;
    return 0;
}
```

- All the **local variables** defined inside a function, including the **formal parameters**, are **destroyed** on **return** of the function call.
  - These **local variables** are created **every time** the function is called.
  - These **local variables** created on the current call are **different** from those created in the previous calls.
  - However, if a formal parameter is a **reference variable**, only itself is destroyed when the function returns, the variable (actual parameter) bound to it still **exists** afterwards.
- **Pass-by-reference** is **more efficient** when a large object has to be passed to a function as **no copying** takes place. However, there is a **risk** that you may accidentally modify the object.

**Question:** Is there a way to pass a large object to a function such that the function cannot modify its value?

# const References as Function Arguments

- You can (and should!) express your intention to keep a reference argument (that is passed to a function) **unchanged** by making it **const**.
- There are 2 advantages:
  1. If you **accidentally** try to **modify** the argument in your function, the compiler will catch the error.

```
// Fine
void call_by_ref(int& x) { x += 10; }
```

```
// Error!
void call_by_const_ref(const int& x) { x += 10; }
```

2. You may pass both **const** and **non-const** arguments to a function that requires a **const reference parameter**.

Conversely, you may pass only **non-const** arguments to a function that requires a **non-const** reference parameter.

Passing Object	const Function Argument	non-const Function Argument
literal constant	✓	X
const Object	✓	X
non-const Object	✓	✓

# Examples: const vs. non-const Reference Arguments

```
1  #include <iostream>      /* File: const-ref-arg.cpp */
2  using namespace std;
3
4  void call_by_ref(int& a) { cout << a << endl; }
5  void call_by_const_ref(const int& a) { cout << a << endl; }
6
7  int main()
8  {
9      int x = 50;
10     const int y = 100;
11     // Which of the following give(s) compilation error?
12     /* passing const literals */
13     call_by_ref(1234);
14     call_by_const_ref(1234);
15
16     /* passing const objects */
17     call_by_ref(y);
18     call_by_const_ref(y);
19
20     /* passing non-const objects */
21     call_by_ref(x);
22     call_by_const_ref(x);
23     return 0;
24 }
```

# Return-by-Value (RBV) and Return-by-Reference (RBR)

## Examples

```
int max(int a, int b) { return (a > b) ? a : b; }

int factorial(int x)
{
    int fx = 1;
    for (int j = 2; j <= x; ++j) fx *= j;
    return fx;
}

/* Function calls */
cout << max(5, 8) << endl;
int result = factorial(10);
```

- Just like you may **PBV** or **PBR**, you may also **return** objects from a function to its caller by **value** or **reference**.
- All the function examples you see so far **return by value**.
- During **RBV**, the returned 'object' is **copied** to the caller.
- If the returned object is big, the return is **slow** due to copying. **RBR** may solve the problem; more about this later.



## Part IV

### Further Examples

# Example: Rock/Paper/Scissors Game



# Example: Rock/Paper/Scissors Game — main()

```
#include <iostream> /* File: rps-game.cpp */
#include <cstdlib>   // Info about random number generator rand()
using namespace std; //rps-game2.cpp: another solution with error messages

// 0/1/2 is used to represent ROCK/PAPER/SCISSORS
const int ROCK = 0, PAPER = 1, SCISSORS = 2;

// Define the game functions here
int print_choice(char player, int choice) { ... }
void print_game_result(int computer_choice, int user_choice) { ... }

int main()
{
    int seed;           // To seed the random number generator
    cout << "Enter an integer: "; cin >> seed;
    srand(seed);        // Initialize random number generator

    int computer_choice = rand()%3; // rand() produces an integer which is
    int user_choice = rand()%3;     // then converted to ROCK/PAPER/SCISSORS

    if (print_choice('C', computer_choice) != 0) return -1; // -1 ⇒ an error
    if (print_choice('U', user_choice) != 0) return -1; // -1 signals an error
    print_game_result(computer_choice, user_choice);
    return 0;
}
```

## Example: Rock/Paper/Scissors Game — other Functions

```
/* To print out the choice picked by the computer or the user */
int print_choice(char player, int choice) // 'C' for computer and 'U' for user
{
    if (player == 'C')
        cout << "Computer";
    else if (player == 'U')
        cout << "User";
    else
        return -1;        // Better also print an error message

    cout << " picks ";

    if (choice == ROCK)
        cout << "rock" << endl;
    else if (choice == PAPER)
        cout << "paper" << endl;
    else if (choice == SCISSORS)
        cout << "scissors" << endl;
    else
        return -1;        // Better also print an error message

    return 0;
}
```

# Example: Rock/Paper/Scissors Game — other Functions ..

```
/* To print game result: "DRAW!", "COMPUTER WINS!", or "PLAYER WINS!" */
```

```
void print_game_result(int computer_choice, int user_choice)
{
    if (computer_choice == user_choice)
        cout << "\tDRAW!" << endl;

    else if (computer_choice == ROCK && user_choice == SCISSORS
        || computer_choice == SCISSORS && user_choice == PAPER
        || computer_choice == PAPER && user_choice == ROCK)
        cout << "\tCOMPUTER WINS!" << endl;

    else
        cout << "\tUSER WINS!" << endl;
}
```