INTRODUCTION TO PROGRAMING

Chapter 4

User-Defined Function

Khoa Công Nghệ Thông Tin
Trường Đại Học Khoa Học Tự Nhiên
ĐHQG-HCM

GV: Thái Hùng Văn

Objectives

In this chapter, you will:

- Learn about predefined functions and how to use them.
- Learn about user-defined functions, discover how to declare, define and call them.
- Examine value-returning functions, including actual and formal parameters
- Explore how to construct and use an user-defined function with call by value method.
- Explore how to construct and use an user-defined function with call by reference method.
- Discover the difference between value and reference parameters
- Learn about the scope of an identifier

Introdution to Functions

What is Function?

- A complex problem is often easier to solve by dividing it into several smaller parts, each of them can be solved by itself. This is called ***structured*** programming.
- These parts are often made into ***functions*** in C/C++. A **function** is a group of statements that together perform a task.
- Every C++ program has at least one function, which is **main()**
- In function body (source code of function), we can also use other functions.
- Functions separate the concept from the implementation; **make programs easier to understand**. Each function solves one of the small problems obtained using ***top-down*** design.
- Functions can be called several times in the same program, allowing **the code to be reused**.

Function Types

- There are two types of function:
  - **Standard library (predefined) functions**
  - **User-defined functions**
- The **StandardLibrary** provides a rich collection of functions for performing **common useful operations** *(input/output, mathematical calculations, string processing, error checking,...)*
- However, the **predefined functions are not enough**. Programmers will often have to build and use user-defined functions.

Predefined Function

- The standard library functions are built-in functions in C/C++ programming language. These are already declared and defined in C/C++ libraries.
- Predefined functions are organized into separate libraries:
  - I/O functions are in **`iostream`** header,
  - Math functions (e.g., **`abs`**, **`ceil`**, **`sqrt`**, etc.) are in **`cmath`** header, …
- Example:

```
#include <iostream>
#include <cmath>
int main()
{
   std::cout << "Square root of 2019 is : " << sqrt (2019);
   return 0;
}
```

Using Predefined Function

- To use a predefined function, you need the name of the appropriate header file; and must use the #**include** directive with the header file name in every **.cpp** file
- You also need to know *Function name, Number of parameters required, Type of each parameter, What the function is going to do*
- *For #include <filename> , the preprocessor searches it in directories pre-designated by the compiler/IDE. This method is normally* ***used to include standard library header files.***
- *For #include "filename" the preprocessor searches first in the same directory as the file containing the directive, and then follows the search path used for the #include <filename> form. This method is normally* ***used to include programmer-defined header files.***

Example

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
  double number, squareRoot;
  cout << "Enter a number: ";     cin >> number;
  // sqrt() is a library function to calculate square root
  squareRoot = sqrt(number);
  cout << "Square root of " << number << " = " << squareRoot;
  return 0;
}
```

User-defined Function

Some Advantages of User-defined Functions

- They allow complicated programs to be divided into manageable pieces.
- A programmer can focus on just a part of the program and construct it, debug it, and perfect it.
- Different people can work on different functions simultaneously
- They can be re-used (even in different programs)
- Enhance program readability
- Can be put together to form a larger program, then a large project can be built.

User-defined Function Types

- There are 4 different types of user-defined functions :
  - Function with no arguments and no return value
  - Function with no arguments and a return value
  - Function with arguments and no return value
  - Function with arguments and a return value
- We can divide into 2 categories :
  - Void functions (no return value)
  - Value-returning functions
- Otherwise, based on arguments we can divide into 4 types:
  - Function with no arguments
  - Function with pass by value method.
  - Function with pass by reference method.
  - Function with pass by pointer method.

Function Definition

```
<type> <function_name> ( <parameter list> )
{
   <local declarations>
   <sequence of statements>
}
```

Some Terminology

Header: Everything before the first brace.

Body: Everything between the braces.

Type: Type of the value returned by the function.

ParameterList: A list of identifiers that provide information for use within the body of the function. Also called **formal parameters**.

"return" statement

- Syntax: **return [<expression>];**
- Usually the **return** statement will appear at the bottom of the function body, sometimes it may appear in other places *(and use in a selection structure, similar to the break statement)*
- The **return** statement terminates the execution of function and returns control to the calling function with the **value of expresion**.
- In **void functions**, the <**expression**> is **Null**
- *Example: Definition of a function that computes the absolute*

```
int absolute(int x) { // the absolute value of an integer
   if (x >= 0)   return x;
   else          return -x;
}
```

"return" statement and "exit" (function)

- Syntax: **exit (<expression>);**
- The **exit()** terminates the execution of function similar to the **return** statement, but it differs from **return** as the **program is also terminated**. (and returns the control back to the operating system with an integer value of <expression>)
- **return expression** returns the value of expression to the calling function, **exit(expression)** always causes the program to terminate and returns an ***exit status*** to the operating system. The value in **expression** is the *exit status* and must be an integer.

Function Prototypes

- A function prototype tells the compiler:
  - number and type of **arguments** to be passed to function
  - type of the **value** that is to be **returned** by the function.
- General Form of a Function Prototype

  <type> <function_name> (<parameter list>) ;
- The parameter list is typically a comma-separated list of types. Identifiers are optional.
  - *exam:* **void f (char c, int i);** *is equivalent to* **void f (char, int);**
- The function definition can be placed anywhere in the program after the function prototypes.
- If a function definition is placed in front of **main()**, there is no need to include its function prototype.

Function Call

- A **function call** has the following syntax:

<function_name> (<argument list>)

Example:

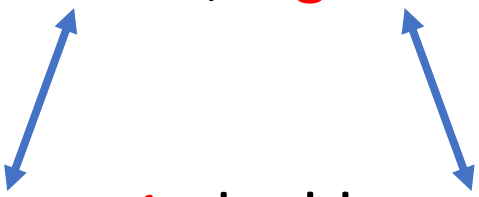
```
int x = absolute(-2019);
int distance = absolute (a-b);
if (distance < x)
    distance = absolute (a-b) + absolute (x-11);
```

- As we have seen, a function is called (or invoked) by writing its name and an appropriate list of arguments within parentheses.
  - The **arguments must match** in **number** and **type** with the parameters in the parameter list of the function definition.

Arguments & Parameters

- one-to-one correspondence between the **arguments** in a function call and the **parameters** in the function definition.
- *For examples:*

```
int argument1;
double argument2 ;
// function call (in another function, such as main)
result = theFunctionName(argument1, argument2);
. . .
// function definition
int thefunctionname(int parameter1, double parameter2){
// Now the function can use the two parameters
// parameter1 = argument 1, parameter2 = argument2
```



Example

```
#include <iostream>
#define PI 3.14
float absolute(float);
float getNumber(){
  float num;
  std::cout << "Enter a number (0 to stop): "; std::cin >> num;
  return num;
}
void main(){
  float a = getNumber(), min = a;
  while (a!=0){
        a = getNumber();
        if (absolute(a-PI) < absolute(min-PI))
           min = a;
  }
  std::cout << "The nearest value to PI is " << min;
}
float absolute(float x) {
    if (x >= 0)       return x;
    else       return -x;
}
```

Void Functions

- A Void Function (VF) does not have a returned type and so functionType in the heading part and the return statement in the body of the VF are meaningless.
- **<type> <function_name> (<parameter list>);** is Value-returning Functions Prototype, and **<type>** is a specific data type. If **<type>** is replaced by **void** then it will be the prototype of VF: **void <function_name> (<parameter list>);**
- In a VF, the return statement has no value, and we can skip if it's at the end of VF body *(else, it is used to exit the VF early).*
- A call to a VF is *(must be)* a **stand-alone statement**. Value-returning Functions may be used in the same way, when the return value is not considered.

Void Function Definition & Call

- Syntax Function Definition:

void <function_name> (<formal parameter list>)

{

<local **declarations**>

<sequence of **statements**>

}

- Syntax Function Call:

<function_name> (< actual argument list>);

Example

```
#include <iostream>
using namespace std;
void Introduce() {
    cout << "* Print Rectangular with size M * N by the * characters *" << endl ;
}
void PrintStars (int NumOfStar) { // print NumOfStar character <*>
    for (int i=0; i< NumOfStar; i++)  cout<< '*';
    cout << endl;
}
int main() {
    int M, N;
    Introduce( );
    cout << "Enter the value of M & N : "; cin >> M >> N;
    for (int i=0; i<M; i++)
            PrintStars (N) ;
    return 0;
}
```

Call by Value & Call by Reference

Call by Value

- Before we discuss function Call by Value (CbV), lets understand 2 terminologies :
  - **Actual parameters:** appear in function calls. (Arguments)
  - **Formal parameters:** appear in function declarations.
- In C/C++, by default, arguments are passed into functions **by value** *(except arrays which is treated as pointers).*
- That is, a **clone copy** of the argument is made and **passed** into the function.
- Changes to the clone copy inside the function has no effect to the original argument in the caller.
- Therefore, arguments can also be a constant, or an expression.

Example of Function call by Value

```
#include <iostream>
void swapnum( int var1, int var2 ) {
   int temp = var1;          // Copying var1 value into temporary variable
   var1 = var2;    // Copying var2 value into var1
   var2 = temp;    //Copying temporary variable value into var2
}
int main( ) {
    int num1 = 3, num2 = 7;
    swapnum (num1, num2);
    std::cout << "After swapping: " <<num1 <<" # " <<num2 <<std::endl;
    swapnum (num1, 2019);        // legal statement
    std::cout << "After swapping: " <<num1 <<" # " <<num2 <<std::endl;
    swapnum (1+2, 3*4);          // legal statement
    std::cout << "After swapping: " <<num1 <<" # " <<num2 <<std::endl;
}
```

What is the result? Why?

Example of Function call by Value - Explain

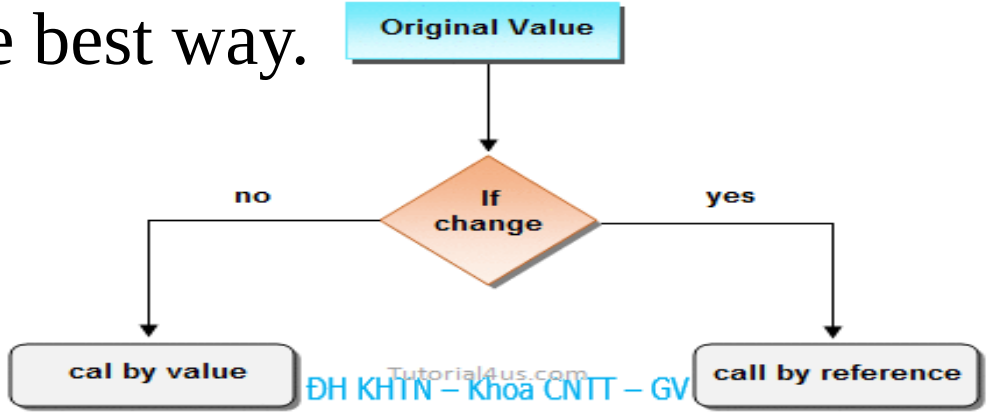
- The result of the above program is:

  After swapping: 3 # 7
  After swapping: 3 # 7
  After swapping: 3 # 7

- The reason: function is called by value for **num1** & **num2**. So actually **var1** & **var2** gets swapped (not num1&num2). As in CbV, **actual parameters are just copied into the formal parameters**.
- In this call ***swapnum (1+2, 3*4);*** the compiler transfer to ***swapnum (3, 12);*** ; pass 3 to **var1** and pass 12 to **var2** ; then swap them and **var1**=12, **var2**=3 ; Finally, the memory location of var1 & var2 is freed, and nothing happens!

Call by Reference

- In many situations, we may wish to modify the original arguments, and can not use the function Call by Value *(e.g , **swapnum** function as above program).*
- In this case, we have to use the function **Call by Reference**, with one of two ways: **Passing by Pointer** or **Passing by Reference**.
- We can pass parameters to a function either by pointers or by reference, and get the same result in both. However, there are also some differences between them, and we have to choose the best way.



Call by Reference - Passing by Pointer

- ***Pass-by-Pointer*** means to pass a pointer argument in the calling function to the corresponding formal parameter of the called function.
- The called function can modify the value of the variable to which the pointer argument points.
- When we use pass-by-pointer, a copy of the pointer is passed to the function, so:
  - If we modify the pointer inside the called function, we only modify the copy of the pointer, but the original pointer remains unmodified and still points to the original variable.
  - If we modify the **content pointed by the pointer** inside the called function, the **content pointed by the original pointer** **will be changed.**

Example of Function Pass_by_Pointer

```
// C++ program to swap 2numbers using method pass by Pointer
#include <iostream>
void swapnum_2 ( int* var1, int* var2 ) {
   int temp = *var1;
   *var1 = *var2;
   *var2 = temp;
}
int main( ) {
    int num1 = 3, num2 = 7;
    swapnum_2 (&num1, &num2);
    std::cout << "After swapping: " <<num1 <<" # " <<num2 ;
}                    // ➔ Output: After swapping: 7 # 3 => Right !
```

Call by Reference - Passing by Reference

- ***Pass-by-Reference*** (PbR) means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function.
- The called function can modify the value of the argument by using its reference passed in.
- PbRs is more efficient than pass-by-value, because it does not copy the arguments. When the called function read or write the formal parameter, it is actually read or write the argument itself.
- The difference between **PbR** and **PbP** is that pointers can be NULL or reassigned whereas references cannot. Use **PbP** if NULL is a valid parameter value or if you want to reassign the pointer

Example of Function Pass_by_Reference

```
// swap two numbers using method pass by Reference
#include <iostream>
void swapnum_3 ( int& var1, int& var2 ) {
   int temp = var1;
   var1 = var2;
   var2 = temp;
}
int main( ) {
    int num1 = 3, num2 = 7;
    swapnum_3 (num1, num2);
    std::cout << "After swapping: " <<num1 <<" # " <<num2 ;
}                    // ➔ Output: After swapping: 7 # 3 => Right !
```

Some other examples

function to return N! = 1*2*...*N	Recursive function to return N! = 1*2*...*N

```
double Factorial (int N)
{
  double F=1;
  for (i=2; i<=N; i++)
    F *= i;
  return F;
}
int main() {
   int a = 19;
   cout <<"Factorial of "<<a
        <<" is  "<< Factorial (a);
   return 0;
}
```

```
double Factorial (int N)
{
  if (N > 1)
        return N*Factorial (N-1);
  else
        return 1;
}
int main() {
   int a = 19;
   cout <<"Factorial of "<<a
        <<" is  "<< Factorial (a);
   return 0;
}
```

Some other examples

// Recursive function to return GreatestCommonDivisor of A and B

// by using use Euclidean algorithm

GCD doesn't change if smaller number is subtracted from a bigger number.	Extended **Euclidean algorithm** - A more efficient solution is to use modulo operator

```
int GCD (int a, int b)
{
        if (a == 0)
                return b;
        if (b == 0)
                return a;
        if (a == b) // base case
                return a;
        if (a > b)
                return GCD (a-b, b);
        return GCD (a, b-a);
}
```

```
int GCD (int a, int b) {
        if (b == 0)
                return a;
        if (a == b) // base case
                return a;
        return GCD (b, a % b);
}
int main() {
   int a = 2019, b = 9102;
   cout <<"GCD of "<<a<<" and "<<b
        <<" is "<< GCD (a, b);
   return 0;
}
```

Some other examples

```
// Return True if N (>1) is a Prime Number and False if not
bool checkPrimeNumber (int N) {
   for (int i = 2; i <= N/2; i++)
      if (N % i == 0) return false;
   return true;
}  // Note: No use input /output functions (cin /cout /...)

int main() {
    cout <<"All of Prime number in [2..2019] : "
    for (int i = 2; i <= 2019; ++i)
        if (checkPrimeNumber ( i ) == true)
            cout << i << "  ";
    return 0;
}
```

Some other examples

```
// Function check <Num> can be Expressed as a Sum of Two Prime Numbers (P1+P2)
bool isSumOfTwoPrime (int Num, int &P1, int &P2) {  // P1 & P2 is 2 Prime Numbers
  for (int i = 2; i <= Num/2; ++i)
     if (checkPrimeNumber ( i ))
          if (checkPrimeNumber ( Num- i )) {
                P1 = i; P2 = Num-i;
                return true;
          }
  return false;
}
int main() {
  int n, prime1, prime2;
  cout << "Enter a positive integer: "; cin >> n;
  if (isSumOfTwoPrime ( n, prime1, prime2 ))
     cout << n << " = " << prime1 << " + " << prime2;
  else cout << n << " can't be expressed as sum of two prime numbers.";
  return 0;
}
```

Some other examples

```
// Solving linear equation aX+b=0.
// Return_value is number of roots (-1 if infinite solution)
int SolveEquation1 (float a, float b, float &x)
{
        if (a == 0)
                if (b == 0) return -1;
                else return 0;
        x = -b/a;
        return 1;
}  // Note: No use input /output functions (cin /cout /...)
```

Some other examples

```
//solving equation ax^2+bx+c=0. Return_value is number of roots (-1 if infinite solution)
// if equation has one root, the root value is x1
int SolveEquation2(float a, float b, float c, float &x1, float &x2) {
        if (a == 0)
                return SolveEquation1(b, c, x1); // reuses SolveEquation1()
        float delta = b*b – 4*a*c;
        if (delta < 0)
                return 0;
        if (delta == 0) {
                x1 = x2 = -b/2/a;
                return 1;
        }
        x1 = (-b – sqrt(delta))/2/a;
        x2 = (-b/a) - x1;
        return 2;
}
```

Quiz

1. What is the difference between
   - call_by_value and call_by_reference?
   - call_by_reference and pass_by_reference?
   - pass_by_reference and pass_by_pointer?
   - pass_by_value and pass_by_pointer?
   - reference variable and pointer variable?
2. Can we create a function with all three forms *(pass_by_value, pass_by_reference and pass_by_pointer)*?
3. What form will help the function run fast? Why?
4. Why should we consider when using input and output statements in a function?

Investigate

1. Function Overloading.
2. Functions with Default Parameters.
3. Static and Global Variables
4. Recursive Functions
5. Project with multiple .CPP files, extern varialbe and functions


End!