

MH1402 Algorithms & Computing II

Lecture 4 Functions (Part1)

Wu Hongjun

Overview

- **Functions**
 - Declaration, definition, function call
- **Variables**
 - Local, global, static variables
- **Passing by Reference**
- Function Overloading
- Separate Compilation and Libraries
- Math functions in C++ standard library
- Recursion

Why use functions in programming?

- The purposes:
 - **Reuse the code**: the code in a function can be easily used throughout the program by simply calling that function
 - **Divide a complex task into smaller, simpler tasks**
- ***The use of function is a breakthrough in programming***
(function in programming was invented in 1950s)
- C++ functions are similar to MATLAB functions
 - Normally we pass input parameters to a function, and get the output values
 - Different in syntax

Functions

- ***Function declaration*** (also called function prototype)
 - Function declaration is optional
 - However, it is better to have function declaration (so as to locate functions at proper places)
- ***Function definition***
 - Define the header and body of a function
- ***Function Call***
 - Using a function
 - A function can be called only after it has been declared or defined

A simple example of function $\text{foo}(x,y) = x^3 + y$;

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

foo, bar, qux are commonly used
in code examples to indicate “anything”

```
double foo(double, double);
```

→ **Declare function foo**

```
double foo(double x, double y)
{
    double z;
    z = x*x*x + y;
    return z;
}
```

Header of function foo

Body of function foo

Define function foo

```
int main() {
    cout << foo(7.0, 2.0) << endl;
    return 0;
}
```

→ **Call function foo**

Function definition

- A function has two parts: header and body
- The syntax of the function header is

return-type function_name(parameter-list)

- The return type of the function
- The name of the function
- And the list of parameters (inputs to the function) , separated by comma

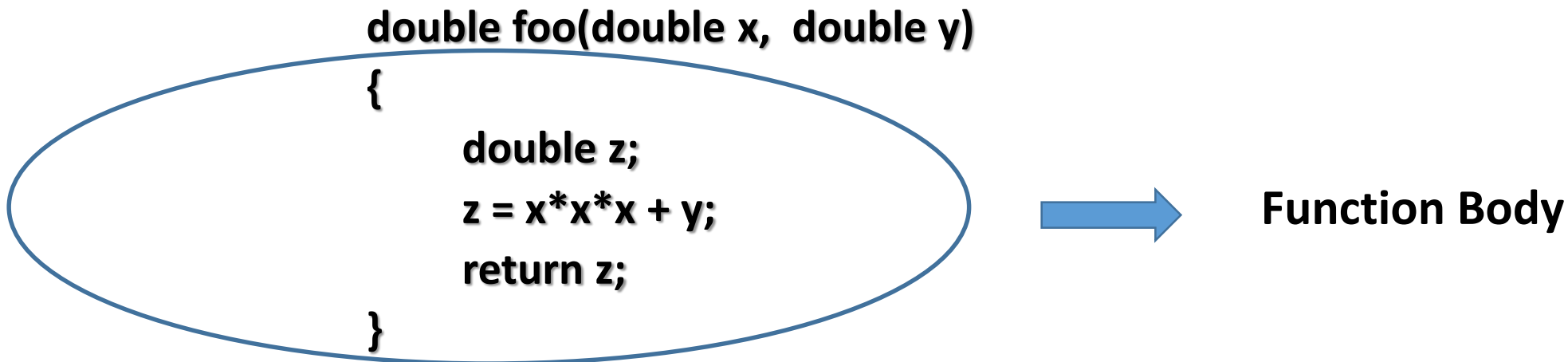
```
double foo(double x, double y)
{
    double z;
    z = x*x*x + y;
    return z;
}
```



Function Header

Function definition

- The body of a function is the block of code following its head
 - It contains the code that performs the function's action { }
 - A function's return statement serves two purposes:
 - 1) terminate the execution of the function
 - 2) return a value to the calling program (**at most one value can be returned**)
To avoid potential programming error, **the type of the value being returned needs to be the same as the return type.**



Function declaration

- **Function declaration is simply the head of the function followed by semicolon**
 - It specifies the function name, the types of its input parameters, and the return type
 - The names of the input parameters can be omitted in function declaration

Example: *double foo(double, double) ;*

- **Alternatively, the function declaration can be**

Example: *double foo(double x, double y) ;*

(the identifiers x and y can be changed to other names)

Function call

- The variables (or values) that are listed in the function's call are called the arguments
- In the example below, the arguments are 7.0 and 2.0

```
int main( )  
{  
    cout << foo(7.0, 2.0) << endl;  
    return 0;  
}
```

- *A function can be called only after it gets declared or defined.*

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

double foo(double x, double y)
{
    double z;
    z = x*x*x + y;
    return z;
}

int main( )
{
    cout << foo(7.0, 2.0) << endl;
    return 0;
}
```

Correct

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

double foo(double, double);

int main( )
{
    cout << foo(7.0, 2.0) << endl;
    return 0;
}

double foo(double x, double y)
{
    double z;
    z = x*x*x + y;
    return z;
}
```

Correct, preferred

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

int main( )
{
    cout << foo(7.0, 2.0) << endl;
    return 0;
}

double foo(double x, double y)
{
    double z;
    z = x*x*x + y;
    return z;
}
```

Error: cannot compile

More on function return

- If a function does not return any value, the return type is **void**
 - The *return* statement in the body of a function is simply **return;**

```
void printNumber(int num)
{
    cout << "number is " << num << endl;
    return;
}
```

```
int main()
{
    int x = 4;
    printNumber(x);
    return 0;
}
```

More on function return

- A function can contain **multiple or zero return statements** in its body

```
int main( )
{
    int n;
    cin >> n ;
    if (n%3 == 0)
    {
        cout << "This number is a multiple of three" << endl;
        return 0;
    }
    cout << "This number is not a multiple of three" << endl;
    return 0;
}
```

More on function return

- **In a function that expects a return value, it is not a good practice to omit return**
- **Return statements don't necessarily need to be at the end;**
- **A function terminates immediately after executing any return statement;**
- **A function terminates after finishing executing the statements in a function even when no return statement was executed**

Function

- **main function**
 - There is **one and only one** main function in a C++ program
 - The computer executes a program by calling its main function.
- **The return type of main function must be *int***
 - 0 being returned in a main function to indicate that the program finished properly
 - If there is no statement *return 0;* at the end of the main function, most of the compilers can still compile the C++ codes (it is good programming style to provide this statement at the end of main function)

Function

- **How to terminate a function (we learned in the previous slides)**
 - Execute a return statement in that function;
 - or finish executing all the statements in that function
- **How to terminate a program? Typically, you can**
 - Method 1: Terminate the main function (**recommended**)
 - Method 2: Call the `exit()` function anywhere in the program.

Example: `exit(0);`

`// to use this function, #include <cstdlib>`

Variables: Local variable

- A local variable is simply a variable declared inside a statement block
 - A local variable is accessible only within that statement block
- The body of a function itself is a statement block, so **variables declared within a function are local to that function (they exist only while the function is executing)**
 - The functions parameters are regarded as being local to the function



Precisely, those values are still in memory, but would get overwritten by other values very soon. To fully understand it, we need to know the details of the implementation of function call using stack (not the focus of this course)

Example to illustrate local variable

```
double foo(double, double);  
  
double foo(double x, double y)  
{  
    double z;  
    z = x*x*x + y;  
    y += 3.5;  
    return z;  
}  
  
int main( )  
{  
    double x = 7.0, y = 2.0;  
    cout << foo(x, y) << endl;  
    cout << y << endl;  
    return 0;  
}
```



```
345  
2
```

Variables: Global Variable

- Global variables are created by placing variable declarations outside any function definition
- Global variables retain their values throughout the execution of the program
 - They are not local variables of any function
- **Global variables can be referenced by any function that follows their declarations**
 - It is convenient to use global variable to pass values into functions,
 - The **risk** is that the value of a global variable may get modified by some function by accident (**poor programming style to declare every variable as global variables!**)

Examples to illustrate that global variables can be referenced by any function that *follows* their declarations

```
int n1;

int function foo( )
{
    n1++;    // correct
    return n1; // correct
}
```

Example 1

```
int function foo( )
{
    n1++;    // incorrect, not in the scope of n1
    return n1; // incorrect,
}

int n1;

int function bar( )
{
    n1++;    // correct
    return n1; // correct
}
```

Example 2

Example to illustrate global variable

```
double y;

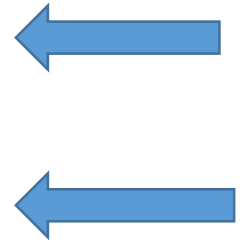
double foo(double);

double foo(double t)
{
    double z;
    z = t*t + y;
    y += 3.5;
    return z;
}

int main( )
{
    double x = 7.0;
    y = 2.0;
    cout << foo(x) << endl;
    cout << y << endl;
    cout << foo(x) << endl;
    cout << y << endl;
    return 0;
}
```



```
51
5.5
54.5
9
```



Local Variable and Global Variable

- In C++, the value of a local variable is **random** if it is not initialized;
- In C++, the value of a global variable is **set to zero** if it is not initialized.

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

int n1;

int main( )
{
    int n2;
    cout << n1 << endl;
    cout << n2 << endl;
    return 0;
}
```

Variables: Static Variable

- In C++, a static variable keeps its value until the end of a program
 - Similar to that of global variable
- Its scope is from the declaration to the end of that statement block
 - Similar to that of local variable
- If a static variable is declared and initialized at the same time, then the initialization of that static variable is executed **only once** during the execution of the program
 - Very different from local variable
- If a static variable is not initialized in a program, its value is set to zero by default in C++
 - Similar to global variable

Examples: static vs local variables

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

int main( )
{
    for (int i = 0; i < 5; i++)
    {
        static int n = 4; //declare static variable
        n++;
        cout << n;
    }
}
```

Output is: 56789

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

int main( )
{
    for (int i = 0; i < 5; i++)
    {
        int n = 4; //local variable
        n++;
        cout << n;
    }
}
```

Output is: 55555

Example: static variable keeps its value even after function call

Output is: 56789

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

int increase( )
{
    static int n = 4;
    n++;
    return n;
}

int main( )
{
    for (int i = 0; i < 5; i++)
    {
        cout << increase();
    }
}
```


Passing by Value vs. Passing by Reference

- **Passing by value**
 - When we pass arguments by value to the parameters of a function (as shown in the previous examples), those parameters of a function are considered as local variables to the function;
 - **changes to those parameters inside a function have no effect outside the function, i.e., the arguments will not be modified**
 - A function can return at most one value, sometimes insufficient

Passing by Value vs. Passing by Reference

- **Passing by reference**
 - Allows us to modify the values of parameters of a function , and **retain the modified values after function call**.
 - To pass an argument to a parameter of a function by reference, simply append an ampersand **&** to the type specified in the function's parameter list.
 - If the value of that parameter gets modified in that function, the value of the corresponding argument is also modified (the same as that of the parameter)
 - **A function can “return” more than one value** when we use passing by reference

Passing by Reference

- Example 1: exchange the values of two objects

```
void interchange(double&, double&);  
  
void interchange(double& x, double& y)  
{  
    double temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

```
int main()  
{  
    double da = 4.1, db = 3.5;  
    cout << da <<" "<< db << endl;  
    interchange(da, db);  
    cout << da <<" "<< db << endl;  
}
```



4.1	3.5
3.5	4.1

(there is a C++ function `swap()` in `#include <algorithm>`)

- The values of the arguments passed by reference can be changed in a function, and the changed values retain outside the function
- **Why the values can get retained ?** (it is a bit difficult to understand)
 - We understand it in this way: in the example given in the previous slide, when we call the function `void interchange(double& x, double& y)` as `interchange(da, db);`
`da` (argument) and `x` (parameter) are treated as the same variable; `db` and `y` are treated as the same variable. Any changes to `x` and `y` also happen to `da` and `db`
 - To fully understand it: in computer memory, `x` and `da` actually use the same memory address to store their values when we call the function, so the values of `x` and `da` are the same. The same happens to `y` and `db`.

Passing by Reference

- Example 2 (almost the same as Example 1):

```
void interchange(double&, double&);  
  
void interchange(double& x, double& y)  
{  
    double temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

```
int main()  
{  
    double da = 4.1, db = 3.5;  
    interchange(da, db);  
    cout << da <<" "<< db << endl;  
    // print: 3.5 4.1  
    double de = 3.2, df = 6.3;  
    interchange(de, df);  
    cout << de <<" "<< df << endl;  
    // print: 6.3 3.2  
    Return 0;  
}
```

Function calls are independent from each other, even with passing by reference

- The argument passed by reference must be variable(s).

```
void interchange(double&, double&);  
  
void interchange(double& x, double& y)  
{  
    double temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

```
int main()  
{  
    double da = 4.1, db = 3.5;  
    cout << da <<" "<< db << endl;  
    interchange(0.5, db);  
    cout << da <<" "<< db << endl;  
}
```


Error: must be a variable here

- **Another Example of passing by reference :
compute the area, circumference of a circle**

```
void compute_circle(double&, double&, double);
```

```
void compute_circle(double& area, double& circumference, double r)
{
    const double PI = 3.141592653589793238463;
    area = PI * r * r;
    circumference = 2 * PI * r;
}
```

```
int main()
{
    double r = 1.0, a, c;
    compute_circle(a, c, r);
    cout << "radius = " << r << endl;
    cout << "area = " << a << endl;
    cout << "circumference = " << c << endl;
}
```



```
radius = 1
area = 3.14159
circumference = 6.28319
```

Difficulties in this lecture

- **The differences between local, global, static variables**
 - What are their scopes?
(Scope means where the variable can be accessed)
 - What are their duration?
(Duration determines when a variable gets created and destroyed)
 - What are their initializations?
- **The difference between passing by value and passing by reference**

Difficulties in this lecture

- The difference between local, global and static variable
 - Scope of local and static variables: from the declaration of the variable
until the end of *that (!!!) statement block*
Scope of global variable: from the declaration of a global variable
until the end of that file
 - Duration of local variable: created when program execution enters its scope;
“destroyed” when program execution leaves its scope;
Duration of global and static variable: created when program starts,
“destroyed” when program ends.
 - Initialization of static and global variables:
set to zero if uninitialized;
the initialization of static variable during declaration is executed only once.