Parameters and Arguments

Chris Wild & Steven Zeil

May 25, 2013

Contents

1 Description

When you write a function, you do not know the name of the variables that will be the function's input and output parameters. Yet in the function, you need some name so that you can write the code.

- A *parameter* is an alias name for the actual argument that will be used when the function is called.
  - More properly, this is called a *formal parameter* of the function.
  - Using parameter aliases is very important for reusability- it allows the same function to be used anywhere in a program, at more than one location of the program or in different programs without worrying about naming conflicts.
  - A parameter is a like a local variable of the function and therefore it has no naming conflicts with anything outside the function (Imagine the chaos if it did).
- The *argument* is the expression that is passed into the function when you write the call.
  - The value of the argument is assigned to the parameter at the time the function is called.
  - For *call by value*, the value of the argument is copied into the parameter.
  - For *call by reference*, the address of the argument is copied into the parameter.

2 Example

```
void myFunction(int anInteger, float aFloat, char aChar)
// anInteger, aFLoat, aChar are the aliases (parameters) for the eventually arguments
{
    //. . .
}
int main( )
{
    int  thisInt = 3; char thisChar = 'a'; float thisFloat = 3.4;
    myFunction(thisInt, thisFloat, thisChar);
     // thisInt, thisFloat, thisChar are the arguments
    myFunction(3, 4.5, 'q'); // can pass in constants as arguments too
}
```

3 Tips

- Even though its not required, naming the parameters in a function prototoype can improve readability
- Chose names for your parameters that indicate the role in the algorihtm implemented by the function

- If the types of the arguments and parameters don't match, the compiler will not be able to use that function

4 Call By Value

4.1 Description

- *Call by value* refers to the passing of arguments into functions whereby a copy is made of the argument and assigned to the parameter.

  In fact, such a parameter is also known as a *copy parameter*.

- Because it is a copy, any changes to the parameter will not effect the value of the original argument.
  - Consequently, these parameters can only be used as *input parameters* to the function.
- That can be either a good or a bad thing. It's good if you want to protect the argument from the function (which you may not have even written). It is bad if you want the function to make a change to the original argument.
- Because a data variable can be of an arbitrary size, making a copy of the argument can be expensive.
- In C++, the default method of passing arguments is by value (except arrays).

4.2 Example

```
void myFunction(int someInt)
{
 someInt++; // change value of someInt
}
int main( )
{
  int myInt = 3;
   myFunction(myInt);
   cout << "After function call: " << myInt << endl;
}
// what will this program print?
// if you didn't answer "after function call: 3",
// then you don't understand call by value
```

4.3 Tips

- Use call by value for int, float and char
- Use call by reference or call by reference const for the rest.

5 Call By Reference

5.1 Description

- *Call by reference* refers to the passing of arguments into functions whereby the address of the argument is assigned to the parameter.
- Any changes to the parameter will also change the value of the original argument.
  - Such parameters can be used as *input parameters* or *output parameters* or combined *input-output* parameters.
  - That can be either a good or a bad thing. It's good if you want the called function to change the argument
  - It is bad if you don't want the function to make a change to the original argument. (There is a way to prevent this however)
- Adding the `const` modifier to a reference parameter tells the compiler not to let the called function change the value. In this way you can have both safety and efficiency.
  - Such parameters can be used only as *input parameters*.
- Since in C++ the default method of passing arguments is by value you need to tell the compiler that you are passing by reference by using "&" after the parameter type.
- A reference parameter is efficient because only the address of the argument has to passed to the function (typically 32 bits on current machines), regardless of the size of the argument itself (which could be thousands of bytes).
  - So, we have two ways of passing input-only parameters to programs: as copy parameters or as const reference parameters. We generally choose between them for efficiency sake.
  - Use call-by-value or copy parameters on small data types (*int*, *float*, *double*, etc.) and on arrays, which aren't small but are actually passed as a single address.
  - Use const reference parameters for some of the larger data types that you will be creating once we being to study structured data.
  - Of course, all that assumes that we are talking about input parameters.
    Output parameters *must* be passed by (non-const) reference.

5.2 Example

```
void myFunction(int &someInt)
{
 someInt++; // change value of someInt
}
int main( )
{
   int myInt = 3;
    myFunction(myInt);
    cout << "After function call: " << myInt << endl;
    return 0;
}
// what will this program print?
// if you didn't answer "After function call: 4",
 // then you don't understand call by reference
 // see also the on parameter passing
```

```
void myFunction(const int &someInt)
{
 someInt++; // change value of someInt
}
int main( )
{
   int myInt = 3;
    myFunction(myInt);
    cout << "After function call: " << myInt << endl;
    return 0;
}
// This program should not compile because you lied to the compiler
 // and went and modified a "const" object.
```

5.3 Tips

- You can read `int &y` as "y is a reference to an integer variable"
- When we start defining and using structured data, we will favor call by reference because of its efficiency (objects can be or arbitrary sixe, and we will use the const modifier to protect objects we don't want to change

6 Experiment

```
//************************
// EXPERIMENT: what is the difference between
// call by value and call by reference?
//
// Programmer: Chris Wild
```

```
// Date: 6 Sep 1999
//************************

#include <iostream.h>

//** swap two integers
void swapByValue(int first, int second)
{
    int temp = first; // need a temporary place to hold swap

    first = second;
    second = temp;
}

void swapByRef(int&, int& );
// this is a function prototype - how do you know - becuase of the ';'
// it means that this function is defined elsewhere (see below)
// this is called a forward reference
// demonstrates two ways of defining low-level funcitons|
// Also notice that we do not have to name the parameter in a function prototype


int main( )
{
   int myFirst = 3;
   int mySecond = 7;

   swapByValue(myFirst, mySecond);
   cout << myFirst << "#" << mySecond << endl;

   swapByRef(myFirst, mySecond);
   cout << myFirst << "#" << mySecond << endl;

}

void swapByRef(int& first, int& second)
{
    int temp = first; // need a temporary place to hold swap

    first = second;
    second = temp;
}
```

Hypothesis:

- What will the first cout print?

- What will the second cout print?
- What would happen if you changed either one of the reference parameters in the "swapByRef" function to a const reference?