Lesson 7 - References
Pass-by-Value Vs Pass-by-Reference

Summary

In this lesson we illustrate the different ways of passing arguments to functions and introduce the concept of Reference types.

New Concepts

Pass-by-value, Pass-by-reference, References, function-overloading.

References

Take a look at the code extract below. On lines 5, 11 we have defined two functions both called `swap`. These two functions are both designed to do the same task, swap two integer variables, yet they behave differently as we shall see.

The `swap` function defined on line 5 accepts two integer values as parameters. When this function is called, we say the parameters are passed by "value". When parameters are passed in this way, the compiler creates copies of the original parameters. If the function modifies these copies, the original values passed to the function will not be affected.

The `swap` function defined on line 11 carries the same name as the previous `swap` function on line 5. This raises a question, is it allowable to declare two functions with the same name? The answer is yes in C++, as long as either the return type or the function parameters are different. This phenomenon is called "function overloading".

Back to the second `swap` function, we see this time `swap` accepts two pointer variables (on line 11). When functions accept pointers as parameters we say they are passed by "reference". We are actually passing the function the memory addresses of the variables. Whatever modifications are made to the variables at those addresses will remain after the function has returned.

```
1  #include <iostream>
2
3  using namespace std;
4
5  void swap(int x, int y) {
6     int temp = x;
7     x = y;
8     y = temp;
9  }
10
11 void swap(int* x, int* y) {
12    int temp = *x;
13    *x = *y;
14    *y = temp;
15 }
```

references.cpp

Now take a look at line 16. We have defined another function called `swap_ref` with a new type of parameter `int&`. These parameters are both "reference" types. References bears similarity to pointers but with some significant differences. To begin with we define a reference in a similar way to a pointer, by stating what type it refers to. This is followed by the reference operator (`&`). So to declare a reference to an integer called `int_ref` we write `int& int_ref`.

Notice the change in syntax in the `swap_ref` function, and compare with the previous `swap` functions. Apart from the parameter declarations, it looks identical to the first `swap` function defined on line 5. It's certainly cleaner than the definition in the second `swap` function which operates on

pointers. This is one of the benefits of references, namely that they allow you to modify the original variables but without the messy syntax of pointers.

```
16 void swap_ref(int& x, int& y) {
17    int temp = x;
18    x = y;
19    y = temp;
20 }
```

references.cpp

Okay, now let's create a `main` function to test the `swap` functions we have created (see line 22). We begin by declaring two integer values we want to swap on line 23, namely `a` and `b`. We get the values for these variables from the user by calling `cin` (lines 26 and 28).

Now let's call the first `swap` function. The compiler knows which function to call because we are passing integer values which match the first `swap` function. After the `swap` function we output the values (line 32) to see the results. If you run the program however you'll notice that the variables `a` and `b` have not been swapped. What happened? Remember that in pass-by-value copies are created, so whatever the `swap` function does it only affects the copies. The result is that the original `a` and `b` were not swapped, and the copies were destroyed when the `swap` function returned.

```
22 int main() {
23    int a, b;
24
25    cout << "Please type in a number" << "\n";
26    cin >> a;
27    cout << "Please type in another number" << "\n";
28    cin >> b;
29
30    cout << "before swap, a = " << a << " and b = " << b << "\n";
31    swap(a, b);
32    cout << "after swap (by value), a = " << a << " and b = " << b << "\n";
```

references.cpp

Now let's try it again but this time using pointers (line 33). First however we have to use the address operator `&` so that the addresses of `a` and `b` are passed to `swap`, rather than their values; the compiler matches then matches this to the second `swap` function when it is called (see line 33). Running the program will show the variables were successfully swapped! This worked because we used pass-by-reference. Instead of creating copies, we gave the `swap` function the original variables.

```
33    swap(&a, &b);
34    cout << "after swap by pointer reference, a = " << a << " and b = " << b << "\n";
35
36    swap_ref(a, b);
37    cout << "after swap using references, a = " << a << " and b = " << b << "\n";
```

references.cpp

Now let's try it again using references by calling the `swap_ref` function. `swap_ref` accepts reference types as parameters, but unlike pointers, we can pass in ordinary integer variables without requiring any new syntax, just as in the first `swap` function (see line 36). As a matter of fact this was the reason we could not use function overloading for `swap_ref`, the compiler would have had no way to differentiate between `swap_ref` and the first `swap` function because the calling syntax would have been identical! Run the program and you'll see the variables `a` and `b` have indeed been swapped again, this time back to their original values.

You may have noticed that references can apparently achieve the same thing as pointers, so why have both and which should you use? Well, references differ from pointers. You have already seen the syntax is somewhat easier to understand, but a reference must always be assigned to some value. For instance, you have seen with a pointer you can initialise it without assigning it a value, or you can set it to `NULL`. You can not do this with a reference and the compiler will complain if you try. On line 38 we show the syntax for initialising a reference type; note that it must immediately be assigned to a variable of the same type.

```
38    int& a_ref = a;
39
40    return 0;
41 }
```

references.cpp

When it comes to implementing your own functions, a good rule of thumb is to always prefer pass-by-reference rather than by pass-by-value. As mentioned, pass-by-value creates copies. Parameters and return types can include types that require a lot of memory, and creating copies every time can be wasteful and bad for performance. Passing by reference eliminates the need to create copies, if the function is called regularly this will provide performance gains in your program.

If you're creating a function whose parameters are passed by reference then you have a choice between using pointers or references. Again, a good rule of thumb is to prefer reference types over pointers. However, if the parameter type points to some memory on the heap, and your function is required to release it, then you must use pointers, because remember that a reference must always be assigned to a value.

Exercises

1. What is wrong with these lines of code...

```
42 int& empty_reference;
43 int& unnamed_reference = 5;
```

references.cpp

2. what is the value of `x_ref` after these lines of code execute...

```
44 int x = 3, y = 4;
45 int& x_ref = x;
46 x_ref = y;
```

references.cpp

3. Why should you never return a reference to a variable from a function, if that variable is "local variable" of that function?

4. Create another function called `swap` which swaps two characters and returns `void`. Ensure that your function swaps the original variables and does not create copies. Call this function from the main function to make sure it works.

5. Create a global variable in the form of an array of 10 integers, i.e:
   `int nums[10] = 7,3,5,2,1,4,6,9,10,8;`
   and create a new function which sorts the numbers of the array into ascending order; use the `swap_ref` function to help you.