Creating Functions in C/C++, Part 2

Call-by-Reference Parameters

What if we want to write a function that actually changes the value of its parameters? For example, suppose we want a function that can be called like this:

\[
\text{swap}(x, y)
\]

that will swap the values of arguments \(x\) and \(y\). In other words if \(x = 10\) and \(y = 20\) before the call, then after the call \(x = 10\) and \(y = 10\). If we write the function like this:

\[
\text{void swap ( int x, int y )}
\{
\quad \text{int temp = x;}
\quad x = y;
\quad y = \text{temp;}
\}
\]

will this do the trick?

Try it and you will see that it does not. Remember that when a function is called, the values of the arguments are copied into the storage cells of the parameters. The function runs and when it terminates, the parameter storage cell contents are not copied back to the arguments. When you think about it, it would make no sense. Suppose we define a function \text{double()} like this:

\[
\text{int double ( int x )}
\{
\quad x = 2\times x;
\quad \text{return x;}
\}
\]

and we call it with the call

\[
\text{cout} \ll \text{double(10)};
\]

If the value of parameter \(x\) were copied back to its argument, it would mean we could replace a constant literal 10 by the value 20, which is impossible.

The kind of parameters we have been using so far are known as call-by-value parameters. This is because the value of the argument is passed to them. So what is the alternative?

The Concept Of A Reference

We mentioned before that variables have several different properties, such as their type, their storage requirements, and their scope. Every variable also has contents and location. In
int x = 5;
int y;

x is the name of a variable of type int with contents of 5. x also has a location. We don't know its actual location exactly but we know that it has some storage location in memory and that location has a specific, numbered address. In the picture below, think of each box as a storage location capable of storing an integer:

```
  x  y
1 | 5 |  . . .
2 |    |    
```

The boxes, which are actually memory words (4-byte units), have addresses but we don't care what they are. The assignment statement

```
y = x;
```

causes the contents of x to be copied into the location of y¹.

```
x  y
1 | 5 | 5 |  . . .
2 |    |    |
```

A reference variable is like another name for a location that already exists. A reference variable really stores an address. If we define two variables x and y as follows:

```
int x = 5;
int& y = x;
```

then y is called a reference to x. The variable y stores the location, or address, of x, but it can be used in place of x. The two statements

```
cout << x;
cout << y;
```

print the same value because y is just a pseudonym, another name, for x. If we increment x and output the value of y, we will see it has changed as well:

```
x++;  
cout << y;
```

and we can increment y and output the value of x and see that it has also changed.

```
y++;  
cout << x;
```

¹Notice the asymmetry of the = operator: the value of its right hand side operand is the contents of the operand, but the value of the left hand side operand is the location of the operand. In other words, putting a variable on the right hand side of “=” causes its value to be extracted whereas putting it on the left hand side causes its location to be found.

The lvalue of a variable is its location. The rvalue is its contents. lvalue and rvalue are just abstractions; they are not really stored anywhere, but they help to make things clearer.
The &-operator used in this way creates a reference variable:

\[ \text{type & identifier = variable;} \]

makes the identifier a reference to the variable. The reference variable type must be the same as type of the variable whose address is being assigned to it.

```c
char c;
char & refTochar = c;
int x;
int & refToInt = x;
```

are two valid reference declarations, but not this:

```c
char c;
int & cref = c;
```

because `cref` is of type `int&` and it should be of type `char&`. It does not matter whether there is space to the left or right of the &-operator. The following three statements are equivalent:

```c
int & y = x;
int & y = x;
int & y = x;
```

We can use reference variables as parameters of functions. They are then known as call-by-reference parameters.

**Examples**

```c
void swap( int & x, int & y )
// replaces x by y and y by x
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a = 10;
    int b = 20;
    swap(a, b);
    cout << a << " " << b << endl;
    return 0;
}
```

This program will print 20 and then 10 because `swap(a, b)` caused the values of `a` and `b` to be swapped. This is because `x` is just another name for `a` in `swap()` and `y` is another name for `b`. Each assignment statement in `swap()` is actually altering the values of `a` and/or `b`. 

void castActors( string & Romeo, 
    string & Juliet )
{
    Romeo = "Leonardo DiCaprio";
    Juliet = "Claire Danes";
}

int main()
{
    string lead_male_role;
    string lead_female_role;

    castActors( lead_male_role, lead_female_role );
    cout << "Romeo will be played by " << lead_male_role << "\n";
    cout << "Juliet will be played by " << lead_female_role << "\n";
    return 0;
}

In this second example, the strings in the main program have no initial value but after the call to castActors(), they are given values. Call-by-reference parameters are the key to writing functions that must give values to multiple variables, such as functions that initialize many variables.

Overloading Functions

Functions can be overloaded in C++, but not in C. Simply put, it means that the same name can be used for two different functions, provided that the compiler can distinguish which function is being called when it tries to compile the code. This can be convenient sometimes, but most of the time there is little need for this feature of C++. Nonetheless, because you may be called upon to read a program that contains overloaded functions, I discuss them very briefly here.

The rule sounds simple at first: The same name can be used for two different functions provided that they have a different number of formal parameters, or they have one or more formal parameters of different types. So these are valid overloads:

    int max( int a, int b );
    int max( int x, int y, int z );

and so are these:

    void sort( int & x, int & y );
    void sort( double & x, double & y );

because the first pair have a different number of parameters and the second pair have different types for their parameters.

These are not valid overloads:

    int max( int a, int b );
    long max( int x, int y );

because they only differ in their return type. (The names of the parameters are irrelevant.) These are not valid either:

    void sort( int & x, int & y );
    void sort( int x, int y );
because the types are the same and kind of parameter passing is not used to distinguish them.

Overloading can get complicated because of type casting and type conversion. If you have two functions such as

```c
void foo( int x, double y);
void foo( double x, int y);
```

and your program makes the call

```c
foo(1,2);
```

which one should the compiler use? Guess what? It can’t really decide either, so it generates an error. Unless you have a good reason to overload function names, it is best to avoid it.

**Default Arguments**

C++, and not C, lets you assign default values to the parameters of a function, in right to left order. In other words, you can declare a function so that the rightmost parameters have default values, as in the following example:

```c
string repeatedstr( string str, int numcopies = 1);
```

The calling program can omit the second argument, in which case numcopies will be assigned 1:

```c
cout << repeatedstr("*");
```

will print a single ‘*’ whereas

```c
cout << repeatedstr("*", 10);
```

will print ten of them.

You can have any number of default arguments but the rule is that a parameter can only have a default argument if the parameter to its right has one. So these are valid:

```c
void foo( int a, int b, int c = 1, int d = 2, int e = 3);
void bar(int a, int b, int c = 1);
```

but not these

```c
void foo( int a = 1, int b);
void bar(int a = 1, int b = 2, int c);
```

Moreover, you can run into problems with overloading and default arguments, as in the following:

```c
void f( int x, int y, int z = 1);
void f( int x, int y);
```
If the program calls \texttt{f(1,2)}, is it the first or the second function that will be invoked? Guess what? The compiler can’t know either, so it generates an error.

The C++ \texttt{iostream} library has two prototypes for \texttt{getline()}:  

\begin{verbatim}
istream& getline ( istream& is, string& str, char delim );
istream& getline ( istream& is, string& str );
\end{verbatim}

Do you think it is overloaded, or do you think that \texttt{getline()} is really defined with the header  

\begin{verbatim}
istream& getline ( istream& is, string& str, char delim = '\n' );
\end{verbatim}

What do you think is the better solution?

The real benefit of being able to assign default values to parameters comes with class constructors, a topic that will be covered when we get to classes. My advice is that you avoid overloading unless you have a really good reason to use it.