C++/Functions and Recursion

This lesson will have to do with using functions and recursion to improve readability and efficiency of your code. This lesson assumes you have a general grasp of C++ syntax. Other than that, I will explain everything as well as I can.

Functions

Simply put, a function is a segment of code that is isolated from the main code segment. A function is called from a section of code. When the function's code has been executed, it returns to the calling code. The general form of a function is:

return_type function_name ([arg1_type arg1_name, ...]) { code }

Here is an example of a function:

```
int addTwoInts(int arg1, int arg2)
{
   int sum = arg1 + arg2;
   return sum;
}
```

- The **int** at the beginning is the return type of the function.
- **addTwoInts** is the function's identifier. It will be what is used to call the function.
- **int arg1** and **int arg2** are called parameters. They are defined in the form `<type> <identifier>`. These will be explained later.
- The code for the function is enclosed in a set of curly brackets **{ }**.
- Every function with a return type other than `void` must have a **return** statement. This is the data that the function will be sending back to the calling code.

Many conventions exist governing the form of functions in C++. Generally, whatever form is most readable to you is the one you should use. For example, some coders will leave the opening `{` on the same line as the function definition while others will give it its own line. Also, some coders would name the above function `add_two_ints`. This is mostly personal preference.

The main Function

The `main` function is a special function. Every C++ program must contain a function named "main". It serves as the entry point for the program. The computer will start running the code from the beginning of the main function. There are two main types of `main` functions; one with and one without parameters. Below are examples of both:

```
// Without Parameters
int main()
{
   ...
}
```

```
// With Parameters
int main(int argc, char * const argv[])
{
   ...
}
```

The reason for having the parameter option for the main function is to allow input from the command line. When you use the `main` function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named `argv`. The array datatype may be beyond you at the moment, but don't fret. In time you will learn of it. Here is an example of a command line statement with a program that uses a `main` function that accepts arguments.

```
C:\> program.exe text 234
```

This command would pass the following information to the `main` function:

- argc = 3
- argv = `{program.exe, text, 234}`
    - Note that the '234' is a string, not an integer value

Since the `main` function has the return type of `int`, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program's execution was. Returning 0 signals that there were no problems.

Function Declarations

Before we discuss calling functions, we must talk about declarations. Programs written in C++ are executed in logical order from the top down. A statement can only use symbols, or named constructs like functions, that are declared in code that has already been executed. Put simply, the compiler must be aware of the name used to describe a function before that function can be called in the code.

For example, to allow `main` to call functions that are defined after the main function itself in the code, we use forward declarations. A forward declaration tells the compiler that a function exists and what its arguments look like, but the definition of function will be elsewhere.

The declaration begins much the same as the definition.

Example:

```
// Definition
int fctn2(int num1, int num2)
{ ... }

// Declaration
int fctn2(int, int);
```

Note that the argument names are optional since the declaration does not need to use the arguments in any way. However, keeping the arguments needed in the declaration may be useful for documentation purposes.

Calling

Calling refers to how a function is used from code.

Example:

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

void fctn1();
int fctn2(int, int);

int main()
{
   int sum;

   fctn1();
   sum = fctn2(2,3);

   return 0;
}

void fctn1()
{
   cout << "This is function 1!" << endl;
}

int fctn2(int num1, int num2)
{
   //the value 2 has been passed as num1
   //the value 3 has been passed as num2

   return num1 + num2;
}
```

First note the use of the two declarations that precede the `main` function. They allow `main` to use `fctn1` and `fctn2` even though they aren't defined until after `main`. The forward declarations ensure that the compiler knows that when it sees the symbols "fctn1" and "fctn2" that those names refer to functions somewhere in the program.

Next, notice that to call `fctn1` all we had to do was enter `fctn1();`. Since it doesn't require any arguments and it has a `void` return value, this is very simple. For `fctn2` however, we see that it not only requires two arguments, but also returns an integer as well. To pass data to a function, you simply list the data in the order it is called for by the function definition. To catch the returned data, we use the assignment operator '=' and assign the returned value to a variable `sum`.

Another thing to note is that functions with a `void` return type, as `fctn1`, do not require a return statement as they do not return anything.

Parameters

Parameters are how data is passed between functions through the call of the function. Above we learned that you list the data you want to pass to a function in the call between the `( )`. The order of the list is determined by the function definition. The first parameter in the list will be assigned to the variable listed first in the function definition. Also, in most cases, you must have the correct number and type of data being passed to the function or you will receive an error when you try to compile your program.

For example:

```
int main
{
   fctn(num1, 12);
}

void fctn(int arg1, int arg2)
{ ... }
```

This would pass the contents of the variable `num1` to `arg1` in the function. Then the integer value '12' would be passed to `arg2`.

Passing Methods

By default, values are passed to a function through a method called `pass by value`. This means that the value of the variable is passed, not the variable itself. This would be like giving a person a copy of your driver's license. They can read all of your information and they can change whatever they want on their copy, but the original is not altered in any way.

If desired, a value can be passed through a method called `pass by reference` where the function is actually given the address of the variable, allowing it direct access to the information. This is done by placing a '&' after the data type in the function definition (this must also be present in any forward declaration). So, if we were to want to pass num1 using pass by reference, we would say:

```
fctn(num1, 23);
```

as before in the `main` function, but in the definition, we would say

```
void fctn(int& arg1, int arg2)
```

Following is a guide as to when one should use pass by reference to alter data, and when one should just use a return statement:

# values altered	Return	Pass by value	Pass by reference
0	N	Y	N
1	Y	Y	N
2+	N	N	Y

You can mix between the use of pass by value and pass by reference from parameter to parameter, but it is not good programming practice to mix the use of pass by reference and returning.

Default Parameters

It's possible to assign default values to parameters that are omitted from the function call. The default values are usually defined in the function declaration like this example:

```
int addTwoInts(int arg1 = 4, int arg2 = 5);
```

In this case, if the parameters aren't provided, they will be assigned 4 and 5 respectively. So, for example, if you had the call

```
temp = addTwoInts();

temp = addTwoInts(33);
```

Neither of these calls would raise an error. In the first one, both values would be set to their defaults. In the second example, the first argument would be passed 33 while the second argument would still be defaulted to 5. You cannot default the Second one without defaulting the first one.

Scope

Scope refers to the level of access an object has. A function can access only global variables and those that are passed to it through arguments. Also, any variables declared inside a function are only available to that function. For example, an integer declared in `main` will not be available to any other function unless it is passed as an argument. Vice versa, an integer declared in a function will not be available to the `main` function.

Recursion

Recursion is a method of function calling in which a function calls itself during execution. Let's start by showing an example and then discussing it.

```
int factorial(int n)
{
   if(n == 1 || n == 0)
   {
      return 1;
   }
   else
   {
      return n*factorial(n-1);
   }
}
```

This is a classic application of recursion. This function calculates the factorial of a number. As you probably know the factorial of n, written n!, is the product of every number from 1 to n. So we can say that 4! = 4×3×2×1. Let's step through what happens in our function when we call `num = factorial(4)`.

- `factorial(4)` is called
  - Since n=4, we take the `else` path. We return 4×`factorial(n-1)`
  - `factorial(3)` is called
    - Since n=3, we take the `else` path. We return 3×`factorial(n-1)`
    - `factorial(2)` is called
      - Since n=2, we take the `else` path. We return 2×`factorial(n-1)`
      - `factorial(1)` is called
        - Since n=1, we take the first path and finally return 1 to the previous function
      - `factorial(1)` returns 1 so `factorial(2)` can return 2×1...2
    - `factorial(2)` returns 2 so `factorial(3)` can return 3×2...6
  - `factorial(3)` returns 6 so `factorial(4)` can return 4×6...24

Many times, a recursive solution to a problem is very easy to program. The drawback of using recursion is that there is a lot of overhead. Every time a function is called, it is placed in memory. Since you don't exit the `factorial` function until n reaches 1, n functions will reside in memory. This isn't a problem for the simple factorial(4), but other functions can lead to serious memory requirements.

Where To Go Next

Topics in C++		
Beginners	**Data Structures**	**Advanced**
Lesson 1: Introduction to C++ Lesson 2: Variables and User Input Lesson 3: Simple Math Lesson 4: Conditional Statements Lesson 5: Loops Lesson 6: Functions and Recursion Lesson 7: More Functions	Lesson 8: Pointers Lesson 9: Classes and Inheritance Lesson 10: Templates 1 Lesson 11: Templates 2	Lesson 12: The STL Lesson 13: STL Algorithms
Part of the School of Computer Science		

1. include<iostream>

int main() {

```
cout<<"hello world\n";
return 0;
```

```
}
```