Functions and Program Structure

Today we will be learning about functions. You should already have an idea of their uses. Cout and cin are both examples of functions. In general, functions perform a number of pre-defined commands to accomplish something productive. Functions that a programmer writes will generally require a prototype. Like a blueprint, the prototype tells the compiler what output value (if any) the function will return, what the function will be called, as well as what arguments (or inputs) the function can be passed (if any).

Functions have many uses. For example, a programmer may have a block of code that he has repeated forty times throughout the program. A function to execute that code would save a great deal of space; and, it would make the program more readable.

Another reason to use functions is to break down a complex program into more manageable pieces. For example, take a menu program that runs complex code when a menu choice is selected. The program would probably best be served by making functions for each menu choice, and then breaking down the complex tasks into smaller, more manageable ones, which could in turn be their own functions. In this way, a program can be designed that makes sense when it is read.

Why use functions?

- Code re-use
- Divide and Conquer, i.e.,
- By decomposing a program into functions, we can divide the work among several programmers
- Clarify the program by keeping details out of the main program
- Each function solves a part of the problem
- Can change one function without changing other parts of the program
- We can test one function separately from the rest

For example,

```
#include < cmath >
int a, base = 4, exp = 2;
a = pow(base, exp);              // sets a = 4^2 = 16
```

"pow" is the function name. The values of base and exp (i.e., the arguments**)** are passed to the pow function.

The function prototype (syntax)

ftype **fname**(**type1** *param1*, **type2** *param2*, …, **typeN** *paramN*)
{
    local variable declarations
    executable statements
    return (*exprn*);
}

- **ftype:** The data type returned. If ftype = **void**, then the function does not return a value.
- **fname:** The function's name.
- **(type1** *param1*, …): In parentheses, a list of **parameters** and their **types** separated by commas. Use **void** here if the function has no parameters.

You can call or invoke a function. To do so, you may need to pass arguments. These arguments are evaluated, and their values are passed to the invoked function. The function parameters "catch" this information (i.e., these argument values) and use them to calculate and return a value.

Example:

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

int mult (int x, int y);        // This prototype specifies that the function mult will accept 2
                                // arguments (both integers), and that it will return an integer.
                                // The trailing semicolon indicates that this is a function declaration
                                // Without the semicolon, the compiler will think you are actually
                                // trying to write the definition of the function here.

int main()
{
int x, y;
cout << "Please input two numbers to be multiplied: "
cin >> x >> y;
cout << "The product of your two numbers is " << mult(x, y);
return 0;
}

int mult (int x, int y)         // When you actually define the function, you begin with the same
                                // prototype from above, MINUS the semicolon. Then there should
                                // always be a brace before and after the code (just like main)
{
return x*y;                     // this function returns the value x*y
}
```

Notice how cout actually outputs what appears to be the mult function. What's really happening is that mult acts as a variable. Because this function returns a value, it's possible for the cout function to output the returned value.

The mult function is defined **below** main. Because its **prototype is above main**, the compiler recognizes it as being defined. Hence, you will not get any errors about mult being undefined, even though its definition is below where it is used.

Example: Find the larger of two numbers.

```
double bigger (double n1, double n2)
{
        double larger;          // local variable declaration
        if (n1 > n2)
                larger = n1;
        else
                larger = n2;
        return (larger);
}
```

Example: Compute the factorial (n! = "n factorial" = n*(n-1)*(n-2)* … * 2 * 1).

```
int factorial (int n)
{
        int i;    // local variable
        product = 1;
        for (i = n; i>1; i--)
                product *= i;
        return (product);
}
```

Function Declarations

A function declaration is terminated by a semicolon.

```
int function1 (type1 param1, type 2 param2, …);     // function DECLARATION
int function1 (type1 param1, type 2 param2, …)      // function DEFINITION
```

- The declaration of a function f1 must occur outside all other functions and before the first function that invokes f1 **OR** inside each function that invokes f1.
- A function declaration inside a function f2 serves as a declaration ONLY for f2.

Example: The following program reads a midterm grade, final grade, and a weight; computes the weighted average, and assigns a course grade.

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

char grade (int exam1, int exam2, float exam1_weight);

main()
{
int ans, mid_term, final;
float weight;
char letter_grade;

do {
        cout << endl << "Compute another grade (1 = Yes, 0 = No)? " << endl;
        cin >> ans;
        if (ans == 1) {
                cout << endl << "Enter midterm, final, weight: " << endl;
                cin >> mid_term>> final >> weight;
                letter_grade = grade (mid_term, final, weight); // calls/invokes function grade
                cout << "Grade is:" << letter_grade << endl;
        }
    } while (ans == 1);
}

char grade (int exam1, int exam2, float exam1_weight)
{
        float average;          // local variable
        average = exam1_weight * exam1 + (1.0 – exam1_weight) * exam2;
        if (average > 7.0)
                return 'P';
        else
                return 'F';
}
```

The main function

This is where the flow begins, i.e., the program starts here. Execution *never* begins in a function; a function must be invoked or called. **main** usually invokes other functions. Some functions can invoke other functions. The program as a whole terminates when **main** does.

The return statement

A function always returns to its invoker. When a function is invoked, the statements in its body begin executing until either a **return** statement *or* the last statement in its body is executed.

If the function does not return a value, the **return** statement is optional. But if used, it can be written:

```
return;                          // all 3 statements are correct if no value is returned
return void;
return (void);
```

Return is the keyword used to force the function to return a value. A function that returns a value must have at least one **return** statement, which is written:

return *exprn*;
return (*exprn)*;

Example:

```
return 25 + num*3;
return (25 + num*3);
```

Arguments and parameters

- Arguments are enclosed in parenthesis and separated by commas if there are more than one.
- Arguments must match the parameters in number, data type, and order. (i.e., when you call a function, you must enter the correct number of arguments of the correct type listed in the order designated by the prototype).
- The value of the arguments represent information passed from the invoking function to the invoked function.
- When a function is invoked, all the functions' arguments are evaluated before the control is passed to the invoked function. However, C++ does NOT guarantee the order of evaluation of the arguments.

Example:

```
int num1 = 5;
void fun (int i, int j);          // declare function "fun"
fun (++num1, ++num1);             // invoke function "fun"
```

The values passed to the function could be
fun (6, 7) or fun (7, 6) ???