LESSON SET

# 6.2 Functions that Return a Value

**PURPOSE**

1. To introduce the concept of scope
2. To understand the difference between static, local and global variables
3. To introduce the concept of functions that return a value
4. To introduce the concept of overloading functions

**PROCEDURE**

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.
3. In the lab, students should complete labs assigned to them by the instructor.

| Contents | Pre-requisites | Approximate completion time | Page number | Check when done |
|---|---|---|---|---|
| Pre-lab Reading Assignment | | 20 min. | 92 | |
| Pre-lab Writing Assignment | Pre-lab reading | 10 min. | 101 | |
| **LESSON 6.2A** | | | | |
| Lab 6.5<br>Scope of Variables | Basic understanding of scope rules and parameter passing | 15 min. | 101 | |
| Lab 6.6<br>Parameters and Local Variables | Basic understanding of formal and actual parameters and local variables | 35 min. | 104 | |
| **LESSON 6.2B** | | | | |
| Lab 6.7<br>Value Returning and Overloading Functions | Understanding of value returning functions and overloaded functions | 30 min. | 106 | |
| Lab 6.8<br>Student Generated Code Assignments | Basic understanding of pass by reference and value. | 30 min. | 110 | |

## PRE-LAB READING ASSIGNMENT

### Scope

As mentioned in Lesson Set 6.1, the scope of an identifier (variable, constant, function, etc.) is an indication of where it can be accessed in a program. There can be certain portions of a program where a variable or other identifier can not be accessed for use. Such areas are considered out of the scope for that particular identifier. The header (the portion of the program before `main`) has often been referred to as the global section. Any identifier defined or declared in this area is said to have **global scope**, meaning it can be accessed at any time during the execution of the program. Any identifier defined outside the bounds of all the functions have global scope. Although most constants and all functions are defined globally, variables should almost **never** be defined in this manner.

**Local scope** refers to identifiers defined within a block. They are active only within the bounds of that particular block. In C++ a **block** begins with a left brace { and ends with a right brace }. Since all functions (including `main`) begin and end with a pair of braces, the body of a function is a block. Variables defined within functions are called **local variables** (as opposed to **global variables** which have global scope). Local variables can normally be accessed anywhere within the function from the point where they are defined. However, blocks can be defined within other blocks, and the scope of an identifier defined in such an inner block would be limited to that inner block. A function's formal parameters (Lesson Set 6.1) have the same scope as local variables defined in the outmost block of the function. This means that the scope of a formal parameter is the entire function. The following sample program illustrates some of these scope rules.

*Sample Program 6.2a:*

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

const PI = 3.14;

void printHeading();


int main()
{
      float circle;
      cout << "circle has local scope that extends the entire main function"
           << endl;

      {
           float square;
           cout << "square has local scope active for only a portion of main."
                << endl;
           cout << "Both square and circle can be accessed here "
                << "as well as the global constant PI." << endl;
      }
      cout << "circle is active here, but square is not." << endl;

      printHeading();

      return 0;
}

void printHeading()
{
      int triangle;

      cout << "The global constant PI is active here "
           << "as well as the local variable triangle." << endl;
}
```

Notice that the nested braces within the outer braces of `main()` indicate another block in which `square` is defined. `square` is active only within the bounds of the inner braces while `circle` is active for the entire `main` function. Neither of these are active when the function `printHeading` is called. `triangle` is a local variable of the function `printHeading` and is active only when that function is active. `PI`, being a global identifier, is active everywhere.

Formal parameters (Lesson Set 6.1) have the same scope as local variables defined in the outmost block of the function. That means that the scope of formal parameters of a function is the entire function. The question may arise about variables with the same name. For example, could a local variable in the function `printHeading` of the above example have the name `circle`? The answer is yes, but it would be a different memory location than the one defined in the `main` function. There are rules of **name precedence** which determine which memory location is active among a group of two or more variables with the same name. The most recently defined variable has precedence over any other variable with the same name. In the above example, if `circle` had been defined in the `printHeading` function, then the memory location assigned with that definition would take precedence over the location defined in `main()` as long as the function `printHeading` was active.

**Lifetime** is similar but not exactly the same as scope. It refers to the time during a program that an identifier has storage assigned to it.

## Scope Rules

1. The scope of a global identifier, any identifier declared or defined outside all functions, is the entire program.
2. Functions are defined globally. That means any function can call any other function at any time.
3. The scope of a local identifier is from the point of its definition to the end of the block in which it is defined. This includes any nested blocks that may be contained within, unless the nested block has a variable defined in it with the same name.
4. The scope of formal parameters is the same as the scope of local variables defined at the beginning of the function.

Why are variables almost never defined globally? Good structured programming assures that all communication between functions will be explicit through the use of parameters. Global variables can be changed by any function. In large projects, where more than one programmer may be working on the same program, global variables are unreliable since their values can be changed by any function or any programmer. The inadvertent changing of global variables in a particular function can cause unwanted side effects.

## Static Local Variables

One of the biggest advantages of a function is the fact that it can be called multiple times to perform a job. This saves programming time and memory space. The values of local variables do not remain between multiple function calls. What this means is that the value assigned to a local variable of a function is lost once the function is finished executing. If the same function is called again that value will not necessarily be present for the local variable. Local variables start "fresh," in terms of their value, each time the function is called. There may be times when a function needs to retain the value of a variable between calls. This can be done by defining the variable to be **static**, which means it is initialized at most once and its memory space is retained even after the function in which it is defined has finished executing. Thus the lifetime of a static variable is different than a normal local variable. Static variables are defined by placing the word **static** before the data type and name of the variable as shown below.

```
static int totalPay = 0;
static float interestRate;
```

## Default Arguments

Actual parameters (parameters used in the call to a function) are often called **arguments**. Normally the number of actual parameters or arguments must equal the number of formal parameters, and it is good programming practice to use this one-to-one correspondence between actual and formal parameters. It is possible, however, to assign default values to all formal parameters so that the calling instruction does not have to pass values for all the arguments. Although these default values can be specified in the function heading, they are usually defined in the prototype. Certain actual parameters can be left out; however, if an actual parameter is left out, then all the following parameters must also be left out. For this reason, pass by reference arguments should be placed first (since by their very nature they must be included in the call).

*Sample Program 6.2b:*

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


void calNetPay(float& net, int hours=40, float rate=6.00);
// function prototype with default arguments specified



int main()
{
      int hoursWorked = 20;
      float payRate = 5.00;
      float pay;               // net pay calculated by the calNetPay function

      cout << setprecision(2) << fixed << showpoint;

      calNetPay(pay);         // call to the function with only 1 parameter
      cout << "The net pay is $" << pay << endl;

      return 0;
}



//
********************************************************************************
//                            calNetPay
//
//   task:      This function takes rate and hours and multiples them to
//              get net pay (no deductions in this pay check!!!).  It has two
//              default parameters.  If the third argument is missing from the
//              call, 6.00 will be passed as the rate to this function.  If the
//              second and third arguments are missing from the call, 40 will be
//              passed as the hours and 6.00 will be passed as the rate.
//
//   data in:   pay rate and time in hours worked
//   data out:  net pay (alters the corresponding actual parameter)
//
//
******************************************************************************



void calNetPay(float& net, int hours, float rate)

{
      net = hours * rate;

}
```

What will happen if `pay` is not listed in the calling instruction? An error will occur stating that the function can not take 0 arguments. The reason for this is that the `net` formal parameter does not have a default value and so the call must have at least one argument. In general there must be as many actual arguments as formal parameters that do not have default values. Of course some or all default values can be overridden.

The following calls are all legal in the example program. Fill in the values that the `calNetpay` function receives for `hours` and `rate` in each case. Also fill in the value that you expect net pay to have for each call.

```
calNetPay(pay);                              The net pay is $__________
   calNetPay receives the value of _______for hours and ________for rate.
```

```
calNetPay(pay,hoursWorked);                 The net pay is $_______
   calNetPay receives the value of _______for hours and ________for rate.

calNetPay(pay, hoursWorked, payRate);       The net pay is $_______
   calNetPay receives the value of _______for hours and ________for rate.
```

The following are not correct. List what you think causes the error in each case.

```
calNetPay(pay, payRate);
calNetPay(hoursWorked, payRate);
calNetPay(payRate);
calNetPay();
```

## Functions that Return a Value

The functions discussed in the previous lesson set are not "true functions" because they do not return a value to the calling function. They are often referred to as procedures in computer science jargon. True functions, or value returning functions, are modules that return exactly one value to the calling routine. In C++ they do this with a `return` statement. This is illustrated by the `cubeIt` function shown in sample program 6.2c.

*Sample Program 6.2c:*

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

int cubeIt(int x);              // prototype for a user defined function
                                // that returns the cube of the value passed
                                // to it.
int main()

{
      int x = 2;
      int cube;

      cube = cubeIt(x);         // This is the call to the cubeIt function.
      cout << "The cube of " << x << " is " << cube << endl;

      return 0;
}

//*******************************************************************
//                              cubeIt
//
//   task:           This function takes a value and returns its cube
//   data in:        some value x
//   data returned:  the cube of x
//
//*******************************************************************


int cubeIt(int x)            // Notice that the function type is int
                             // rather than void

{
        int num;

        num = x * x * x;
        return num;
}
```

The function `cubeIt` receives the value of `x`, which in this case is 2, and finds its cube which is placed in the local variable `num`. The function then returns the value stored in `num` to the function call `cubeIt(x)`. The value 8 replaces the entire function call and is assigned to cube. That is, `cube = cubeIt(x)` is replaced with `cube = 8`. It is not actually necessary to place the value to be returned in a local variable before returning it. The entire `cubeIt` function could be written as follows:

```
int cubeIt(int x)
{
    return x * x * x;
}
```

For value returning functions we replace the word `void` with the data type of the value that is returned. Since these functions return one value, there should be no effect on any parameters that are passed from the call. This means that all parameters of value returning functions should be pass by value, NOT pass by reference. Nothing in C++ prevents the programmer from using pass by reference in value returning functions; however, they should not be used.

The `calNetPay` program (Sample Program 6.2b) has a module that calculates the net pay when given the hours worked and the hourly pay rate. Since it calculates only one value that is needed by the call, it can easily be implemented as a value returning function, instead of by having `pay` passed by reference.

Sample program 6.2d, which follows, modifies Program 6.2b in this manner.

*Sample Program 6.2d:*

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

float calNetPay(int hours, float rate);

int main()

{
      int hoursWorked = 20;
      float payRate = 5.00;
      float netPay;

      cout << setprecision(2) << fixed << showpoint;

      netPay = calNetPay(hoursWorked, payRate);
      cout << " The net pay is $" << netPay << endl;

      return 0;
}
```

*continues*

```
//*****************************************************************************
//                         calNetPay
//
//   task:          This function takes hours worked and pay rate and multiplies
//                  them to get the net pay which is returned to the calling function.
//
//   data in:       hours worked and pay rate
//   data returned: net pay
//
//*****************************************************************************


float calNetPay(int hours, float rate)
{

      return hours * rate;
}
```

Notice how this function is called.

```
paynet = calNetPay (hoursWorked, payRate);
```

This call to the function is not a stand-alone statement, but rather part of an assignment statement. The call is used in an expression. In fact, the function will return a floating value that replaces the entire right-hand side of the assignment statement. This is the first major difference between the two types of functions (`void` functions and value returning functions). A `void` function is called by just listing the name of the function along with its arguments. A value returning function is called within a portion of some fundamental instruction (the right-hand side of an assignment statement, condition of a selection or loop statement, or argument of a `cout` statement). As mentioned earlier, another difference is that in both the prototype and function heading the word `void` is replaced with the data type of the value that is returned. A third difference is the fact that a value returning function MUST have a `return` statement. It is usually the very last instruction of the function. The following is a comparison between the implementation as a procedure (`void` function) and as a value returning function.

| | **Value Returning Function** | **Procedure** |
|---|---|---|
| **PROTOTYPE** | `float calNetPay (int hours, float rate);` | `void calNetPay (float& net, int hours, float rate);` |
| **CALL** | `netpay=calNetPay (hoursWorked, payRate);` | `calNetPay (pay, hoursWorked, payRate);` |
| **HEADING** | `float calNetPay (int hours, float rate)` | `void calNetPay (float& net, int hours, float rate)` |
| **BODY** | `{ return hours * rate; }` | `{ net = hours * rate; }` |

Functions can also return a Boolean data type to test whether a certain condition exists (true) or not (false).

## Overloading Functions

Uniqueness of identifier names is a vital concept in programming languages. The convention in C++ is that every variable, function, constant, etc. name with the same scope needs to be unique. However, there is an exception. Two or more functions may have the same name as long as their parameters differ in quantity or data type. For example, a programmer could have two functions with the same name that do the exact same thing to variables of different data types.

*Example:* Look at the following prototypes of functions. All have the same name, yet all can be included in the same program because each one differs from the others either by the number of parameters or the data types of the parameters.

```
int   add(int a, int b, int c);
int   add(int a, int b);
float add(float a, float b, float c);
float add(float a, float b);
```

When the `add` function is called, the actual parameter list of the call is used to determine which `add` function to call.

## Stubs and Drivers

Many IDEs (Integrated Development Environments) have software debuggers which are used to help locate logic errors; however, programmers often use the concept of stubs and drivers to test and debug programs that use functions and procedures. A **stub** is nothing more than a dummy function that is called instead of the actual function. It usually does little more than write a message to the screen indicating that it was called with certain arguments. In structured design, the programmer often wants to delay the implementation of certain details until the overall design of the program is complete. The use of stubs makes this possible.

*Sample Program 6.2e:*

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

int findSqrRoot(int x);  // prototype for a user defined function that
                         // returns the square root of the number passed to it

int main()
{
      int number;

      cout << "Input the number whose square root you want." << endl;
      cout << "Input a -99 when you would like to quit." << endl;
      cin >> number;

      while (number != -99)
      {
```

*continues*

```
          cout << "The square root of your number is "
               << findSqrRoot(number) << endl;
          cout << "Input the number whose square root you want." << endl;
          cout << "Input a -99 when you would like to quit." << endl;
          cin >> number;
     }
     return 0;
}

int findSqrRoot(int x)
{
     cout << "findSqrRoot function was called with " << x
          << " as its argument\n";
     return 0;
}    // This bold section is the stub.
```

This example shows that the programmer can test the execution of `main` and the call to the function without having yet written the function to find the square root. This allows the programmer to concentrate on one component of the program at a time. Although a stub is not really needed in this simple program, stubs are very useful for larger programs.

A **driver** is a module that tests a function by simply calling it. While one programmer may be working on the `main` function, another programmer may be developing the code for a particular function. In this case the programmer is not so concerned with the calling of the function but rather with the body of the function itself. In such a case a driver (call to the function) can be used just to see if the function performs properly.

*Sample Program 6.2f:*

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

int findSqrRoot(int x); // prototype for a user defined function that
                        // returns the square root of the number passed to it

int main()
{
     int number;

     cout << "Calling findSqrRoot function with a 4" << endl;
     cout << "The result is " << findSqrRoot(4) << endl;

     return 0;
}

int findSqrRoot(int x)
{
     return sqrt(x);
}
```

In this example, the `main` function is used solely as a tool (driver) to call the `findSqrRoot` function to see if it performs properly.

## PRE-LAB WRITING ASSIGNMENT

### Fill-in-the-Blank Questions

1. Variables of a function that retain their value over multiple calls to the function are called _______________variables.
2. In C++ all functions have _______________scope.
3. Default arguments are usually defined in the _______________of the function.
4. A function returning a value should never use pass by _______________ parameters.
5. Every function that begins with a data type in the heading, rather than the word `void`, must have a(n) _______________statement somewhere, usually at the end, in its body of instructions.
6 A(n) _______________is a program that tests a function by simply calling it.
7. In C++ a block boundary is defined with a pair of _______________.
8. A(n) _______________is a dummy function that just indicates that a function was called properly.
9. Default values are generally not given for pass by _______________ parameters.
10. _______________functions are functions that have the same name but a different parameter list.

## LESSON 6.2A

### LAB 6.5 Scope of Variables

Retrieve program `scope.cpp` from the Lab 6.2 folder. The code is as follows:

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


// This program will demonstrate the scope rules.

// PLACE YOUR NAME HERE


const double PI = 3.14;
const double RATE = 0.25;

void findArea(float, float&);
void findCircumference(float, float&);


int main()

{
```

*continues*