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 Scope of Variables	Basic understanding of scope rules and parameter passing	15 min.	101	
Lab 6.6 Parameters and Local Variables	Basic understanding of formal and actual parameters and local variables	35 min.	104	
LESSON 6.2B				
Lab 6.7 Value Returning and Overloading Functions	Understanding of value returning functions and overloaded functions	30 min.	106	
Lab 6.8 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

```
    cout << fixed << showpoint << setprecision(2);
    float radius = 12;

    cout <<" Main function outer block" << endl;
    cout <<" LIST THE IDENTIFIERS THAT are active here" << endl << endl;
    {
        float area;
        cout << "Main function first inner block" << endl;
        cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;

        // Fill in the code to call findArea here

        cout << "The radius = " << radius << endl;
        cout << "The area = " << area << endl << endl;
    }

    {
          float radius = 10;
          float circumference;

          cout << "Main function second inner block" << endl;
          cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;

          // Fill in the code to call findCircumference here

          cout << "The radius = " << radius << endl;
          cout << "The circumference = " << circumference << endl << endl;

    }

    cout << "Main function after all the calls" << endl;
    cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;

    return 0;
}

//  ********************************************************************
//                          findArea
//
//   task:     This function finds the area of a circle given its radius
//   data in:  radius of a circle
//   data out: answer (which alters the corresponding actual parameter)
//
//   ********************************************************************


void findArea(float rad, float& answer)
{

    cout << "AREA FUNCTION" << endl << endl;
    cout << "LIST THE IDENTIFIERS THAT are active here"<< endl << endl;
```

```
    // FILL in the code, given that parameter rad contains the radius, that
    // will find the area to be stored in answer

}

//  ***************************************************************************
//                           findCircumference
//
//   task:     This function finds the circumference of a circle given its radius
//   data in:  radius of a circle
//   data out: distance (which alters the corresponding actual parameter)
//
//   ***************************************************************************


void findCircumference(float length, float& distance)

{
      cout << "CIRCUMFERENCE FUNCTION" << endl << endl;
      cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;

      // FILL in the code, given that parameter length contains the radius,
      // that will find the circumference to be stored in distance

}
```

Exercise 1: Fill in the following chart by listing the identifiers (function names, variables, constants)

GLOBAL	Main	Main (inner 1)	Main (inner 2)	Area	Circum-ference

Exercise 2: For each `cout` instruction that reads:

```
cout << " LIST THE IDENTIFIERS THAT are active here" << endl;
```

Replace the words in all caps by a list of all identifiers active at that location. Change it to have the following form:

```
cout << "area, radius and PI are active here" << endl;
```

Exercise 3: For each comment in bold, place the proper code to do what it says.

NOTE: area = πr^2
circumference = $2\pi r$

Exercise 4: Before compiling and running the program, write out what you expect the output to be.

What value for `radius` will be passed by `main` (first inner block) to the `findArea` function?

What value for `radius` will be passed by `main` function (second inner block) to the `findCircumference` function?

Exercise 5: Compile and run your program. Your instructor may ask to see the program run or obtain a hard copy.

LAB 6.6 Parameters and Local Variables

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

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

// PLACE YOUR NAME HERE

void normalizeMoney(float& dollars, int cents = 150);
// This function takes cents as an integer and converts it to dollars
// and cents.  The default value for cents is 150 which is converted
// to 1.50 and stored in dollars

int main()

{
  int cents;
  float dollars;


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

  cents = 95;
  cout << "\n We will now add 95 cents to our dollar total\n";

  //  Fill in the code to call normalizeMoney to add 95 cents

  cout << "Converting cents to dollars resulted in " << dollars << " dollars\n";

  cout << "\n We will now add 193 cents to our dollar total\n";

  // Fill in the code to call normalizeMoney to add 193 cents

  cout << "Converting cents to dollars resulted in " << dollars << " dollars\n";

  cout << "\n We will now add the default value to our dollar total\n";

  // Fill in the code to call normalizeMoney to add the default value of cents

  cout << "Converting cents to dollars resulted in " << dollars << " dollars\n";
```

```
  return 0;
}

//
*******************************************************************************
//                                normalizeMoney
//
//   task:      This function is given a value in cents.  It will convert cents
//              to dollars and cents which is stored in a local variable called
//              total which is sent back to the calling function through the
//              parameter dollars.  It will keep a running total of all the money
//              processed in a local static variable called sum.
//
//   data in:   cents which is an integer
//   data out:  dollars  (which alters the corresponding actual parameter)
//
//
*******************************************************************************



void normalizeMoney(float& dollars, int cents)

{

     float total=0;

     // Fill in the definition of sum as a static local variable
     _________________sum = 0.0;

     // Fill in the code to convert cents to dollars

     total = total + dollars;
     sum = sum + dollars;

     cout << "We have added another $" << dollars <<"  to our total" << endl;
     cout << "Our total so far is  $" << sum << endl;

     cout << "The value of our local variable total is $" << total << endl;
}
```

Exercise 1: You will notice that the function has to be completed. This function will take `cents` and convert it to `dollars`. It also keeps a running total of all the money it has processed. Assuming that the function is complete, write out what you expect the program will print.

Exercise 2: Complete the function. Fill in the blank space to define `sum` and then write the code to convert `cents` to `dollars`. Example: 789 cents would convert to 7.89. Compile and run the program to get the expected results. Think about how `sum` should be defined.

LESSON 6.2B

LAB 6.7 Value Returning and Overloading Functions

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

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


// This program will input American money and convert it to foreign currency

// PLACE YOUR NAME HERE

// Prototypes of the functions
void  convertMulti(float dollars, float& euros, float& pesos);
void  convertMulti(float dollars, float& euros, float& pesos, float& yen);
float convertToYen(float dollars);
float convertToEuros(float dollars);
float convertToPesos(float dollars);


int main ()


{
      float dollars;
      float euros;
      float pesos;
      float yen;

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

      cout << "Please input the amount of American Dollars you want converted "
           << endl;
      cout << "to euros and pesos" << endl;
      cin >> dollars;

      // Fill in the code to call convertMulti with parameters dollars, euros, and pesos
      // Fill in the code to output the value of those dollars converted to both euros
      // and pesos


      cout << "Please input the amount of American Dollars you want converted\n";
      cout << "to euros, pesos and yen" << endl;
      cin >> dollars;

      // Fill in the code to call convertMulti with parameters dollars, euros, pesos and yen
      // Fill in the code to output the value of those dollars converted to euros,
      // pesos and yen
```

```
      cout << "Please input the amount of American Dollars you want converted\n";
      cout << "to yen" <<endl;
      cin >> dollars;

      // Fill in the code to call convertToYen
      // Fill in the code to output the value of those dollars converted to yen

      cout << "Please input the amount of American Dollars you want converted\n";
      cout << " to euros" << endl;
      cin  >> dollars;

      // Fill in the code to call convert ToEuros
      // Fill in the code to output the value of those dollars converted to euros

      cout << "Please input the amount of American Dollars you want converted\n";
      cout << " to pesos " << endl;
      cin >> dollars;

      // Fill in the code to call convertToPesos
      // Fill in the code to output the value of those dollars converted to pesos

      return 0;
}


// All of the functions are stubs that just serve to test the functions
// Replace with code that will cause the functions to execute properly



//  ************************************************************************
//                           convertMulti
//
//   task:      This function takes a dollar value and converts it to euros
//              and pesos
//   data in:   dollars
//   data out:  euros and pesos
//
//   ************************************************************************

void  convertMulti(float dollars, float& euros, float& pesos)

{
      cout << "The function convertMulti with dollars, euros and pesos "
           << endl <<" was called with " << dollars <<" dollars" << endl << endl;


}
```

continues

```
//  ************************************************************************
//                           convertMulti
//
//   task:     This function takes a dollar value and converts it to euros
//             pesos and yen
//   data in:  dollars
//   data out: euros pesos yen
//
//   ************************************************************************

void  convertMulti(float dollars, float& euros, float& pesos, float& yen)

{
      cout << "The function convertMulti with dollars, euros, pesos and yen"
           << endl << " was called with " << dollars << " dollars" << endl << endl;



}


//  ************************************************************************
//                           convertToYen
//
//   task:          This function takes a dollar value and converts it to yen
//   data in:       dollars
//   data returned: yen
//
//   ************************************************************************

float convertToYen(float dollars)

{
      cout << "The function convertToYen was called with " << dollars <<" dollars"
           << endl << endl;

      return 0;
}

//  ************************************************************************
//                           convertToEuros
//
//   task:          This function takes a dollar value and converts it to euros
//   data in:       dollars
//   data returned: euros
//
//   ************************************************************************
```

```
float convertToEuros(float dollars)
{
      cout << "The function convertToEuros was called with " << dollars
           << " dollars" << endl << endl;

      return 0;
}

//  ************************************************************************
//                           convertToPesos
//
//   task:          This function takes a dollar value and converts it to pesos
//   data in:       dollars
//   data returned: pesos
//
//   ************************************************************************
float convertToPesos(float dollars)

{
      cout << "The function convertToPesos was called with " << dollars
           << " dollars" << endl;

    return 0;
}
```

Exercise 1: Run this program and observe the results. You can input anything that you like for the dollars to be converted. Notice that it has stubs as well as overloaded functions. Study the stubs carefully. Notice that in this case the value returning functions always return 0.

Exercise 2: Complete the program by turning all the stubs into workable functions. Be sure to call true functions differently than procedures. Make sure that functions return the converted dollars into the proper currency. Although the exchange rates vary from day to day, use the following conversion chart for the program. These values should be defined as constants in the global section so that any change in the exchange rate can be made there and nowhere else in the program.

One Dollar = 1.06 euros
9.73 pesos
124.35 yen

Sample Run:

```
Please input the amount of American Dollars you want converted to euros and pesos
9.35

$9.35 is converted to 9.91 euros and 90.98 pesos

Please input the amount of American Dollars you want converted to euros and pesos and yen
10.67
```

```
$10.67 is converted to 11.31 euros, 103.82 pesos, and 1326.81 yen

Please input the amount of American Dollars you want converted to yen
 12.78
$12.78 is converted to 1589.19 yen

Please input the amount of American Dollars you want converted to euros
 2.45
$2.45 is converted to 2.60 euros

Please input the amount of American Dollars you want converted to pesos
 8.75
$8.75 is converted to 85.14 pesos
```

LAB 6.8 Student Generated Code Assignments

Option 1: Write a program that will convert miles to kilometers and kilometers to miles. The user will indicate both a number (representing a distance) and a choice of whether that number is in miles to be converted to kilometers or kilometers to be converted to miles. Each conversion is done with a value returning function. You may use the following conversions.

1 kilometer = .621 miles
1 mile = 1.61 kilometers

Sample Run:

```
Please input
   1 Convert miles to kilometers
   2 Convert kilometers to miles
 3 Quit

1
Please input the miles to be converted
120
   120 miles = 193.2 kilometers

Please input
   1 Convert miles to kilometers
   2 Convert kilometers to miles
 3 Quit
2
Please input the kilometers to be converted
235
   235 kilometers = 145.935 miles

Please input
   1 Convert miles to kilometers
   2 Convert kilometers to miles
 3 Quit
3
```

Option 2: Write a program that will input the number of wins and losses that a baseball team acquired during a complete season. The wins should be input in a parameter-less value returning function that returns the wins to

the `main` function. A similar function should do the same thing for the losses. A third value returning function calculates the percentage of wins. It receives the wins and losses as parameters and returns the percentage (float) to the `main` program which then prints the result. The percentage should be printed as a percent to two decimal places.

Sample Run:

```
Please input the number of wins
80
Please input the number of losses
40
The percentage of wins is 66.67%
```

Option 3: Write a program that outputs a dentist bill. For members of a dental plan, the bill consists of the service charge (for the particular procedure performed) and test fees, input to the program by the user. To non-members the charges consist of the above services plus medicine (also input by the user). The program first asks if the patient is a member of the dental plan. The program uses two overloaded functions to calculate the total bill. Both are value returning functions that return the total charge.

Sample Run 1:

```
Please input a one if you are a member of the dental plan
Input any other number if you are not
1
Please input the service charge
7.89
Please input the test charges
89.56
The total bill is $97.45
```

Sample Run 2:

```
Please input a one if you are a member of the dental plan
Input any other number if you are not
2
Please input the service charge
75.84
Please input the test charges
49.78
Please input the medicine charges
40.22
The total bill is $165.84
```