

6.1**Introduction to Void Functions
(Procedures)**

PURPOSE	<ol style="list-style-type: none"> 1. To introduce the concept of void functions (procedures) 2. To work with void functions (procedures) that have no parameters 3. To introduce and work with void functions (procedures) that have pass by value and pass by reference parameters
----------------	---

PROCEDURE	<ol style="list-style-type: none"> 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 their instructor.
------------------	--

Contents	Pre-requisites	Approximate completion time	Page number	Check when done
Pre-lab Reading Assignment		20 min.	76	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	83	
LESSON 6.1A				
Lab 6.1 Functions with No Parameters	Confident in use of the control structures	15 min.	84	
Lab 6.2 Introduction to Pass by Value	Basic understanding of pass by value.	35 min.	84	
LESSON 6.1B				
Lab 6.3 Introduction to Pass by Reference	Basic understanding of pass by reference.	25 min.	86	
Lab 6.4 Student Generated Code Assignments	Basic understanding of pass by reference and value.	30 min.	89	

PRE-LAB READING ASSIGNMENT

Modules

A key element of structured (well organized and documented) programs is their modularity: the breaking of code into small units. These units, or **modules**, that do not return a value are called **procedures** in most languages and are called **void functions** in C++. Although procedures is the authors' preferred term, this manual uses the word **function** to describe both void functions (discussed in this lesson set) and **value returning functions** (studied in the next lesson set), as this is the terminology used in C++.

The `int main()` section of our program is a function and, up until now, has been the only coded module used in our programs. We also have used pre-defined functions such as `pow` and `sqrt` which are defined in library routines and “imported” to our program with the `#include <cmath>` directive. We now explore the means of breaking our own code into modules. In fact, the `main` function should contain little more than “calls” to other functions. Think of the `main` function as a contractor who hires sub-contractors to perform certain duties: plumbers to do the plumbing, electricians to do the electrical work, etc. The contractor is in charge of the order in which these sub-contract jobs are issued.

The `int main()` function consists mostly of calls to functions just like a contractor issues commands to sub-contractors to come and do their jobs. A computer does many simple tasks (modules) that, when combined, produce a set of complex operations. How one determines what those separate tasks should be is one of the skills learned in software engineering, the science of developing quality software. A good computer program consists of several tasks, or units of code, called modules or functions.

In simple programs most functions are called, or invoked, by the `main` function. Calling a function basically means starting the execution of the instructions contained in that module. Sometimes a function may need information “passed” in order to perform designated tasks.

If a function is to find the square root of a number, then it needs that number passed to it by the calling function. Information is passed to or from a function through **parameters**. Parameters are the components of communication between functions. Some functions do very simple tasks such as printing basic output statements to the screen. These may be instructions to the user or just documentation on what the program will do. Such functions are often called parameter-less functions since they do not require anything passed by the calling procedure.

Sample Program 6.1a:

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

void printDescription(); // Function prototype

int main()
{
    cout << "Welcome to the Payroll Program." << endl;
    printDescription(); // Call to the function
```

```

        cout << "We hoped you enjoyed this program." << endl;

        return 0;
    }

//*****printDescription*****
//          printDescription
//
// Task:      This function prints a program description
// Data in:   none
//
//*****printDescription() // The function heading
{
    cout << "*****"
        << endl << endl;
    cout << "This program takes two numbers (pay rate and hours)"
        << endl;
    cout << "and outputs gross pay. " << endl;
    cout << "*****"
        << endl << endl;

}

```

In this example, three areas have been highlighted. Starting from the bottom we have the function itself which is often called the function definition.

The function **heading** `void printDescription()` consists of the name of the function preceded by the word `void`. The word `void` means that this function will not return a value to the module that called it.¹ The function name is followed by a set of parentheses. Just like the `main` function, all functions begin with a left brace and end with a right brace. In between these braces are the instructions of the function. In this case they consist solely of `cout` statements that tell what the program does.

Notice that this function comes after the `main` function. How is this function activated? It must be called by either the `main` function or another function in the program. This function is called by `main` with the simple instruction `printDescription();`.

A **call** to a function can be classified as the sixth fundamental instruction (see Lesson Set 2). Notice the call consists only of the name of the function (not the word `void` preceding it) followed by the set of parentheses and a semicolon. By invoking its name in this way, the function is called. The program executes the body of instructions found in that function and then returns to the calling function (`main` in this case) where it executes the remaining instructions following the call. Let us examine the order in which the instructions are executed.

¹ In the next lesson set we will see that the word preceding the name of a function can be the data type of the value that the function will return to the calling function.

The main function is invoked which then executes the following instruction:

```
cout << "Welcome to the Pay Roll Program" << endl;
```

Next the call to the function printDescription is encountered which executes the following instructions:

```
cout << "*****" << endl << endl;
cout << "This program takes two numbers (pay rate & hours)" << endl;
cout << "and outputs gross pay" << endl;
cout << "*****" << endl << endl;
```

After all the instructions in printDescription are executed, control returns to main and the next instruction after the call is executed:

```
cout << "We hoped you enjoyed this program" << endl;
```

The first highlighted section of the example is found before main() in what we call the global section of the program. It is called a **prototype** and looks just like the function heading except it has a semicolon at the end. Since our example has the “definition of the function” after the call to the function, the program will give us an error when we try to call it if we do not have some kind of signal to the computer that the definition will be forthcoming. That is the purpose of the prototype. It is a promise (contract if you will) to the compiler that a void function called printDescription will be defined after the main function. If the printDescription function is placed in the file before the main function which calls it, then the prototype is not necessary. However, most C++ programs are written with prototypes so that main() can be the first function.

Pass by Value

The following program, Sample Program 6.1b, is an extension of the code above. This program will take a pay rate and hours worked and produce the gross pay based on those numbers. This can be done in another function called calPaycheck.

Sample Program 6.1b:

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

// Function prototypes
void printDescription();
void calPaycheck(float, int);

int main()
{
    float payRate;
    int hours;

    cout << "Welcome to the Payroll Program." << endl;
```

```
printDescription(); // Call to the printDescription function

cout << endl << "Please input the pay per hour." << endl;
cin >> payRate;

cout << endl << "Please input the number of hours worked." << endl;
cin >> hours;
cout << endl << endl;

calPaycheck(payRate, hours); // Call to the calPaycheck function

cout << "We hope you enjoyed this program." << endl;

return 0;
}

//*****+
//          printDescription
//
// Task:      This function prints a program description
// Data in:   no parameters received from the function call
//
//*****+

void printDescription() // The function heading
{
    cout << "*****" << endl << endl;
    cout << "This program takes two numbers (pay rate and hours) " << endl;
    cout << "and outputs gross pay. " << endl;
    cout << "*****" << endl << endl;
}

//*****+
//          calPaycheck
//
// Task:      This function computes and outputs gross pay
// Data in:   rate and time
//
//*****+

void calPaycheck(float rate, int time)
{
    float gross;

    gross = rate * time;
    cout << "The pay is " << gross << endl;
}
```

The bold sections of this program show the development of another function. This function is a bit different in that it has parameters inside the parentheses of the call, heading and prototype. Recall that parameters are the components of communication to and from a function and the call to that function. The

function `calPaycheck` needs information from the calling routine. In order to find the gross pay it needs the rate per hour and the number of hours worked to be passed to it. The call provides this information by having parameters inside the parentheses of the call `calPaycheck(payRate, hours);`. Both `payRate` and `hours` are called **actual parameters**. They match in a one-to-one correspondence with the parameters in the function heading which are called `rate` and `time`:

```
void calPaycheck(float rate, int time)
```

The parameters in a function heading are called **formal parameters**.

It is important to compare the call with the function heading.

Call	Function heading
<code>calPaycheck(payRate, hours);</code>	<code>void calPaycheck(float rate, int time)</code>

1. The call does not have any word preceding the name whereas the function heading has the word `void` preceding its name.
2. The call must NOT give the data type before its actual parameters whereas the heading MUST give the data type of its formal parameters.
3. Although the formal parameters may have the same name as their corresponding actual parameters, they do not have to be the same. The first actual parameter, `payRate`, is paired with `rate`, the first formal parameter. This means that the value of `payRate` is given to `rate`. The second actual parameter, `hours`, is paired with `time`, the second formal parameter, and gives `time` its value. Corresponding (paired) parameters must have the same data type. Notice that `payRate` is defined as `float` in the main function and thus it can legally match `rate` which is also defined as `float` in the function heading. `hours` is defined as `int` so it can be legally matched (paired) with `time` which is defined as `int` in the function heading.
4. The actual parameters (`payRate` and `hours`) pass their values to their corresponding formal parameters. Whatever value is read into `payRate` in the main function will be given to `rate` in the `calPaycheck` function. This is called **pass by value**. It means that `payRate` and `rate` are two distinct memory locations. Whatever value is in `payRate` at the time of the call will be placed in `rate`'s memory location as its initial value. It should be noted that if the function `calPaycheck` were to alter the value of `rate`, it would not affect the value of `payRate` back in the main function. In essence, pass by value is like making a copy of the value in `payRate` and placing it in `rate`. Whatever is done to that copy in `rate` has no effect on the value in `payRate`. Recall that a formal parameter can have the same name as its corresponding actual parameter; however, they are still two different locations in memory.

How does the computer know which location to go to if there are two variables with the same name? The answer is found in a concept called **scope**. Scope refers to the location in a program where an identifier is accessible. All variables defined in the main function become inactive when another function is called and are reactivated when the control returns to `main`. By the same token, all formal parameters and variables defined inside a function are active only during the time the function is executing. What this means is that an actual parameter and its corresponding formal parameter are never active at the same time. Thus there is no confusion as to which memory location to access even if corresponding

parameters have the same name. More on scope will be presented in the next lesson set.

It is also important to compare the prototype with the heading.

Prototype	Function heading
<code>void calPaycheck(float, int);</code>	<code>void calPaycheck(float rate, int time)</code>

1. The prototype has a semicolon at the end and the heading does not.
2. The prototype lists only the data type of the parameters and not their name. However, the prototype can list both and thus be exactly like the heading except for the semicolon. Some instructors tell students to copy the prototype without the semicolon and paste it to form the function heading.

Let us look at all three parts—prototype, call and heading:

1. The heading MUST have both data type and name for all its **formal parameters**.
2. The prototype must have the data type and can have the name for its **formal parameters**.
3. The call MUST have the name but MUST NOT have the data type for its **actual parameters**.

Pass by Reference

Suppose we want the `calPaycheck` function to only compute the gross pay and then pass this value back to the calling function rather than printing it. We need another parameter, not to get information from the call but to give information back to the call. This particular parameter can not be **passed by value** since any change made in a function to a *pass by value formal parameter* has no effect on its corresponding actual parameter. Instead, this parameter is **passed by reference**, which means that the calling function will give the called function the location of its actual parameter instead of a copy of the value that is stored in that location. This then allows the called function to go in and change the value of the actual parameter.

Example: Assume that I have a set of lockers each containing a sheet of paper with a number on it. Making a copy of a sheet from a particular locker and giving that sheet to you will ensure that you will not change my original copy. This is pass by value. On the other hand, if I give you a spare key to a particular locker, you could go to that locker and change the number on the sheet of paper located there. This is pass by reference.

How does the program know whether a parameter is passed by value or by reference? All parameters are passed by value unless they have the character & listed after the data type, which indicates a pass by reference.

Sample Program 6.1C:

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

// Function prototypes
```

continues

82 LESSON SET 6.1 Introduction to Void Functions (Procedures)

```
void printDescription();           // prototype for a parameter-less function
void calPaycheck(float, int, float&); // prototype for a function with 3
                                         // parameters. The first two are passed
                                         // by value. The third is passed by
                                         // reference

int main()
{
    float payRate;
float grossPay;
float netPay;
    int hours;

    cout << "Welcome to the Payroll Program." << endl;

    printDescription();           // Call to the description function

    cout << endl << "Please input the pay per hour." << endl;
    cin >> payRate;
    cout << endl << "Please input the number of hours worked." << endl;
    cin >> hours;
    cout << endl << endl;

calPaycheck(payRate, hours, grossPay); // Call to the calPaycheck function
netPay = grossPay - (grossPay * .20);

    cout << "The net pay is " << netPay << endl;
    cout << "We hoped you enjoyed this program." << endl;

    return 0;
}

//*****
//          printDescription
//
// Task:      This function prints a program description
// Data in:   none
// Data out:  no actual parameters altered
//*****

void printDescription() // The function heading
{
    cout << "*****" << endl << endl;
    cout << "This program takes two numbers (pay rate and hours) " << endl;
    cout << "and outputs gross pay. " << endl;
    cout << "*****" << endl << endl;
}
```

```

//*****calPaycheck*****
//          calPaycheck
//
// Task:      This function computes gross pay
// Data in:   rate and time
// Data out:  gross (alters the corresponding actual parameter)
//
//*****calPaycheck*****

void calPaycheck(float rate, int time, float& gross)
{
    gross = rate * time;
}

```

Notice that the function `calPaycheck` now has three parameters. The first two, `rate` and `time`, are passed by value while the third has an & after its data type indicating that it is pass by reference. The actual parameter `grossPay` is paired with `gross` since they both are the third parameter in their respective lists. But since this pairing is pass by reference, these two names refer to the SAME memory location. Thus what the function does to its formal parameter `gross` changes the value of `grossPay`. After the `calPaycheck` function finds `gross`, control goes back to the `main` function that has this value in `grossPay`. `main` proceeds to find the net pay, by taking 20% off the gross pay, and printing it. Study this latest revision of the program very carefully. One of the lab exercises asks you to alter it.

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The word _____ precedes the name of every function prototype and heading that does not return a value back to the calling routine.
2. Pass by _____ indicates that a copy of the actual parameter is placed in the memory location of its corresponding formal parameter.
3. _____ parameters are found in the call to a function.
4. A prototype must give the _____ of its formal parameters and may give their _____.
5. A _____ after a data type in the function heading and in the prototype indicates that the parameter will be passed by reference.
6. Functions that do not return a value are often called _____ in other programming languages.
7. Pass by _____ indicates that the location of an actual parameter, rather than just a copy of its value, is passed to the called function.
8. A call must have the _____ of its actual parameters and must NOT have the _____ of those parameters.
9. _____ refers to the region of a program where a variable is “active.”
10. _____ parameters are found in the function heading.

LESSON 6.1A

LAB 6.1 Functions with No Parameters

Retrieve program proverb.cpp from the Lab 6.1 folder. The code is as follows:

```
// This program prints the proverb
// "Now is the time for all good men to come to the aid of their party"
// in a function (procedure) called writeProverb that is called by the main function

//PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

void writeProverb(); //This is the prototype for the writeProverb function

int main()
{
    // Fill in the code to call the writeProverb function

    return 0;
}

// *****
//          writeProverb
//
//  task:      This function prints a proverb
//  data in:   none
//  data out:  no actual parameter altered
// ****

// Fill in the function heading and the body of the function that will print
// to the screen the proverb listed in the comments at the beginning of the
// program
```

Exercise 1: Fill in the code (places in bold) so that the program will print out the proverb listed in the comments at the beginning of the program. The proverb will be printed by the function which is called by the main function.

LAB 6.2 Introduction to Pass by Value

Retrieve program newproverb.cpp from the Lab 6.1 folder. The code is as follows:

```
// This program will allow the user to input from the keyboard
// whether the last word to the following proverb should be party or country:
```

```
// "Now is the time for all good men to come to the aid of their _____"  
// Inputting a 1 will use the word party. Any other number will use the word country.  
  
// PLACE YOUR NAME HERE  
  
#include <iostream>  
#include <string>  
using namespace std;  
  
  
// Fill in the prototype of the function writeProverb.  
  
  
int main ()  
{  
  
    int wordCode;  
  
    cout << "Given the phrase:" << endl;  
    cout << "Now is the time for all good men to come to the aid of their ___"  
        << endl;  
    cout << "Input a 1 if you want the sentence to be finished with party"  
        << endl;  
    cout << "Input any other number for the word country" << endl;  
    cout << "Please input your choice now" << endl;  
    cin >> wordCode; cout  
        << endl;  
    writeProverb(wordCode);  
  
    return 0;  
}  
  
  
// ****  
//          writeProverb  
//  
//      task:      This function prints a proverb. The function takes a number  
//                  from the call. If that number is a 1 it prints "Now is the time  
//                  for all good men to come to the aid of their party."  
//                  Otherwise, it prints "Now is the time for all good men  
//                  to come to the aid of their country."  
//      data in:    code for ending word of proverb (integer)  
//      data out:   no actual parameter altered  
//  
// ****  
  
void writeProverb (int number)
```

```
{
    // Fill in the body of the function to accomplish what is described above
}
```

Exercise 1: Some people know this proverb as “Now is the time for all good men to come to the aid of their country” while others heard it as “Now is the time for all good men to come to the aid of their party.” This program will allow the user to choose which way they want it printed. Fill in the blanks of the program to accomplish what is described in the program comments. What happens if you inadvertently enter a float such as -3.97?

Exercise 2: Change the program so that an input of 1 from the user will print “party” at the end, a 2 will print “country” and any other number will be invalid so that the user will need to enter a new choice.

Sample Run:

Given the phrase:

Now is the time for all good men to come to the aid of their __

Input a 1 if you want the sentence to be finished with party

Input a 2 if you want the sentence to be finished with country

Please input your choice now

4

I'm sorry but that is an incorrect choice; Please input a 1 or 2

2

Now is the time for all good men to come to the aid of their country

Exercise 3: Change the previous program so the user may input the word to end the phrase. The string holding the user’s input word will be passed to the proverb function instead of passing a number to it. Notice that this change requires you to change the proverb function heading and the prototype as well as the call to the function.

Sample Run:

Given the phrase:

Now is the time for all good men to come to the aid of their __

Please input the word you would like to have finish the proverb

family

Now is the time for all good men to come to the aid of their family

LESSON 6.1B

Lab 6.3 Introduction to Pass by Reference

Retrieve program paycheck.cpp from the Lab 6.1 folder. This program is similar to Sample Program 6.1C that was given in the Pre-lab Reading Assignment. The code is as follows:

```
// This program takes two numbers (payRate & hours)
// and multiplies them to get grosspay.
// It then calculates net pay by subtracting 15%
//PLACE YOUR NAME HERE
```

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

//Function prototypes
void printDescription();
void computePaycheck(float, int, float&, float&);

int main()
{
    float payRate;
    float grossPay;
    float netPay;
    int hours;

    cout << setprecision(2) << fixed;

    cout << "Welcome to the Pay Roll Program" << endl;

    printDescription(); //Call to Description function

    cout << "Please input the pay per hour" << endl;
    cin >> payRate;

    cout << endl << "Please input the number of hours worked" << endl;
    cin >> hours;
    cout << endl << endl;

    computePaycheck(payRate,hours,grossPay,netPay);

    // Fill in the code to output grossPay

    cout << "The net pay is $" << netPay << endl;

    cout << "We hope you enjoyed this program" << endl;

    return 0;
}

// *****
//          printDescription
//
//  task:      This function prints a program description
//  data in:   none
//  data out:  no actual parameter altered
// *****
*****
```

continues

```

void printDescription() // The function heading
{
    cout << "*****" << endl << endl;
    cout << "This program takes two numbers (payRate & hours)" << endl;
    cout << "and multiplies them to get gross pay" << endl;
    cout << "it then calculates net pay by subtracting 15%" << endl;
    cout << "*****" << endl << endl;
}

// *****
//          computePaycheck
//
// task:      This function takes rate and time and multiples them to
//             get gross pay and then finds net pay by subtracting 15%.
// data in:   pay rate and time in hours worked
// data out:  the gross and net pay
//
// *****

void computePaycheck(float rate, int time, float& gross, float& net)

{
    // Fill in the code to find gross pay and net pay
}

```

Exercise 1: Fill in the code (places in bold) and note that the function computePaycheck determines the net pay by subtracting 15% from the gross pay. Both gross and net are returned to the main() function where those values are printed.

Exercise 2: Compile and run your program with the following data and make sure you get the output shown.

```

Please input the pay per hour
9.50
Please input the number of hours worked
40

```

```

The gross pay is $380
The net pay is $323
We hoped you enjoyed this program

```

Exercise 3: Are the parameters gross and net, in the modified calPaycheck function you created in Exercise 1 above, pass by value or pass by reference?

Exercise 4: Alter the program so that gross and net are printed in the function compute instead of in main(). The main() function executes the statement

```
cout << "We hoped you enjoyed this program" << endl;
```

after the return from the function calPaycheck.

Exercise 5: Run the program again using the data from Exercise 2. You should get the same results. All parameters should now be passed by value.

LAB 6.4 Student Generated Code Assignments

Option 1: Write a program that will read two floating point numbers (the first read into a variable called first and the second read into a variable called second) and then calls the function swap with the actual parameters first and second. The swap function having formal parameters number1 and number2 should swap the value of the two variables. Note: This is similar to a program you did in Lesson Set 1; however, now you are required to use a function. You may want to look at logicprob.cpp from Lesson Set 1.

Sample Run:

```
Enter the first number
Then hit enter
80
Enter the second number
Then hit enter
70
```

You input the numbers as 80 and 70.
After swapping, the first number has the value of 70 which was the value of the second number
The second number has the value of 80 which was the value of the first number

Exercise 1: Compile the program and correct it if necessary until you get no syntax errors.

Exercise 2: Run the program with the sample data above and see if you get the same results.

Exercise 3: The swap parameters must be passed by _____.
(Assume that main produces the output.) Why?

Option 2: Write a program that will input miles traveled and hours spent in travel. The program will determine miles per hour. This calculation must be done in a function other than main; however, main will print the calculation. The function will thus have 3 parameters: miles, hours, and milesPerHour. Which parameter(s) are pass by value and which are passed by reference? Output is fixed with 2 decimal point precision.

Sample Run:

```
Please input the miles traveled  
475  
Please input the hours traveled  
8  
Your speed is 59.38 miles per hour
```

Option 3: Write a program that will read in grades, the number of which is also input by the user. The program will find the sum of those grades and pass it, along with the number of grades, to a function which has a “pass by reference” parameter that will contain the numeric average of those grades as processed by the function. The main function will then determine the letter grade of that average based on a 10-point scale.

90–100	A
80–89	B
70–79	C
60–69	D
0–59	F

Sample Run:

```
Enter the number of grades  
3  
Enter a numeric grade between 0-100  
90  
Enter a numeric grade between 0-100  
80  
Enter a numeric grade between 0-100  
50  
The grade is C
```

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: $\text{area} = \pi r^2$

$\text{circumference} = 2\pi r$

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
```

