

6.1

Introduction to Void Functions (Procedures)

PURPOSE

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

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
    //
    // Task:      This function prints a program description
    // Data in:   none
    //
    /*******

    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;
    }

```

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

```
void calPaycheck(float, int);
```

Function heading

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

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

```

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

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.