COMPREHENSIVE EDITION

PROGRAMMING AND PROBLEM SOLVING WITH C++

SIXTH EDITION

Nell Dale and Chip Weems

Chapter 8

Functions

Chapter 8 Topics

- **Writing a Program Using Functional Decomposition**
- **Writing a Void Function for a Task**
- **Using Function Arguments and Parameters**

Chapter 8 Topics

- **Differences between Value Parameters and Reference Parameters**
- **Using Local Variables in a Function**
- **Function Preconditions and Postconditions**

Functions

- Every C++ program must have a function called `main`
- Program execution always begins with function `main`
- Any other functions are subprograms that must be explicitly called

Function Calls

One function calls another by using the name of the called function followed by ()s that enclose an argument list, which may be empty

A function call temporarily transfers control from the calling function to the called function

Function Call Syntax

FunctionName(Argument List)

- **The argument list is a way for functions to communicate with each other by passing information**

- **The argument list can contain 0, 1, or more arguments, separated by commas, depending on the function**

Two Parts of Function Definition

```
int  Cube(int  n)
{

    return  n * n * n;

}
```

heading

body

What is in a heading?

type of value returned

name of function

parameter list

```
int Cube (int n)
```

Prototypes

A prototype looks like a heading but must end with a semicolon, and its parameter list needs only to contain the type of each parameter

```
int  Cube(int );  // Prototype
```

Function Calls

When a function is called, temporary memory is allocated for:

- its value parameters;
- any local variables; and
- for the function's name if the return type is not void

Flow of control then passes to the first statement in the function's body

Function Calls

- The called function's statements are executed until a **return statement** (with or without a return value) or the **closing brace** of the function body is encountered
- Then control goes back to where the function was called

```
#include <iostream>
int Cube(int);      // prototype
using namespace std;

void  main()
{
    int       yourNumber;
    int       myNumber;
    yourNumber = 14;
    myNumber     =    9;
    cout << "My Number = " << myNumber;
    cout << "its cube is " << Cube(myNumber) <<
   endl;
    cout << "Your Number = " << yourNumber;
    cout << "its cube is " << Cube(yourNumber)
        << endl;
}
```

arguments

Successful Function Compilation

Before a function is called in your program, the compiler must have previously processed either:

- the function's prototype or
- the function's definition (heading and body)

Return Values

In C++, a value-returning function returns in its identifier one value of the type specified in its heading and prototype (called the return type)

In contrast, a void-function cannot return any value in its identifier

Example

Write a void function called `DisplayMessage()`,

which you can call from `main()`, to describe the

pollution index value it receives as a parameter

Your city describes a pollution index

less than 35 as “Pleasant”,

35 through 60 as “Unpleasant”,

and above 60 as “Health Hazard”

parameter

```
void DisplayMessage(int index)

{
    if(index < 35)
        cout << “Pleasant”;
    else if(index <= 60)
        cout << “Unpleasant”;
    else
        cout << “Health Hazard”;
}
```

The Rest of the Program

```
#include <iostream>
void DisplayMessage(int); // Prototype
using namespace std;

int main()
{
```

The Rest of the Program, cont...

```
cout << "Enter air pollution index";
    cin >> pollutionIndex;
    DisplayMessage(pollutionIndex);  // Call
    return 0;
}
```

argument

Return Statement

```
return;  // No value to return
```

- Is valid only in the body block of a void function
- Causes control to leave the function and immediately return to the calling block, leaving any subsequent statements in the function body unexecuted

Header Files

Header Files contain

- **Named constants like**

  const int INT_MAX = 32767;

- **Function prototypes like**

  float sqrt(float);

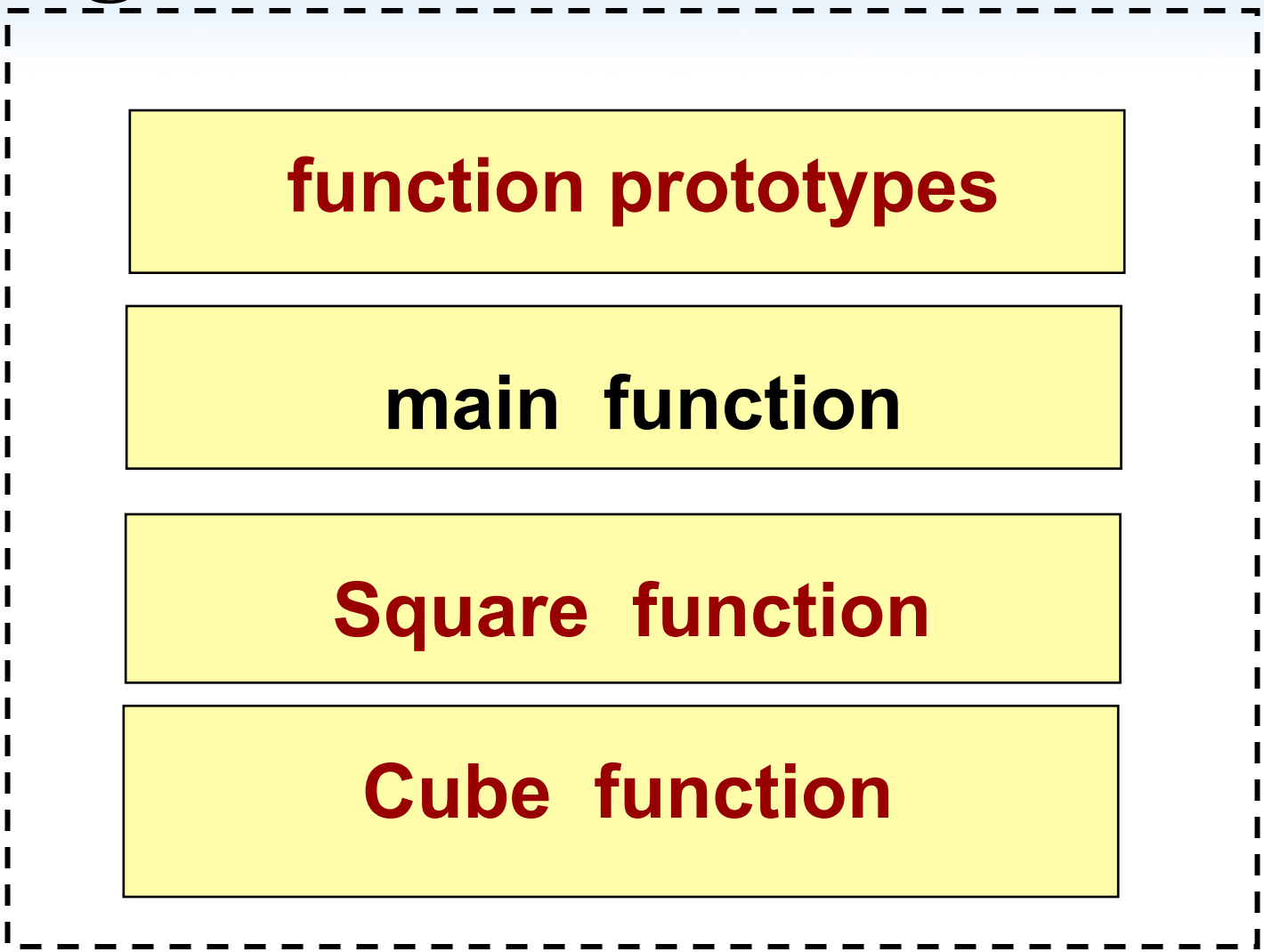
- **Classes like**

  string, ostream, istream

- **Objects like**

  cin, cout

Program with Several Functions



Value-Returning Functions

```
#include <iostream>

int  Square(int);              // Prototypes
int  Cube(int);
using namespace std;

int  main()
{
    cout << "The square of 27 is "
         << Square(27) << endl;
    cout <<  "The cube of 27 is "
         << Cube(27)  <<  endl;
    return 0;
}
```

function calls

Rest of Program

```
int Square(int  n)  // Header and body
{
    return   n * n;
}


int Cube(int  n) // Header and body
{
    return  n * n * n;
}
```

Void Functions Stand Alone

```
#include  <iostream>

void  DisplayMessage(int);// Prototype

using namespace std;

int  main()                          argument
{
    DisplayMessage(15);  // Function call

    cout  <<  "Good Bye"  <<    endl;

    return 0;
}
```

Parameters

parameter

```
void  DisplayMessage(int  n)
{
    cout << "I have liked math for  "
         <<  n  <<  " years"
         << endl;

    return;
}
```

Parameter List

A **parameter list** is the means used for a function to share information with the block containing the call

Classified by Location

Arguments	Parameters
Always appear in a function call within the calling block	Always appear in the function heading, or function prototype

Different Terms

Some C++ books

- Use the term “actual parameters” for arguments

- Those books then refer to parameters as “formal parameters”

Argument in Calling Block

4000

25

age

Value Parameter

The value of the argument (25) is passed to the function when it is called

In this case, the argument can be a variable identifier, constant, or expression

Reference Parameter

The memory address (4000) of the argument is passed to the function when it is called

In this case, the argument must be a variable identifier

Default Parameters

- Simple types, structs, and classes are value parameters by default
- Arrays are always reference parameters
- Other reference parameters are marked as such by having an ampersand (&) beside their type

Use of Reference Parameters

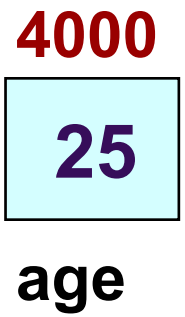
- Reference parameters should be used when the function is to assign a value to, or
- When the function is to change the value of, a variable from the calling block without an assignment statement in the calling block

Using a Reference Parameter

- Is like giving someone the key to your home

- The key can be used by the other person to change the contents of your home!

Main Program Memory



If you pass a **copy** of `age` to a function, it is called **"pass-by-value"** and the function will not be able to change the contents of `age` in the calling block; it is still 25 when you return

Main Program Memory

4000

25

age

BUT, if you pass 4000, the address of age to a function, it is called "pass-by-reference" and the function will be able to change the contents of age in the calling block; it could be 23 or 90 when you return

Additional Terms

- **Pass-by-reference is also called . . .**
  - pass-by-address, or
  - pass-by-location

Can you explain why?

Example of Pass-by-Reference

We want to find 2 real roots for a quadratic equation with coefficients a,b,c.

Write a prototype for a void function named GetRoots() with 5 parameters. The first 3 parameters are type float. The last 2 are reference parameters of type float.

// Prototype

```
void GetRoots(float, float, float,
   float&, float&);
```

Now write the function definition using this information :

This function uses 3 incoming values a, b, c from the calling block. It calculates 2 outgoing values root1 and root2 for the calling block. They are the 2 real roots of the quadratic equation with coefficients a, b, c.

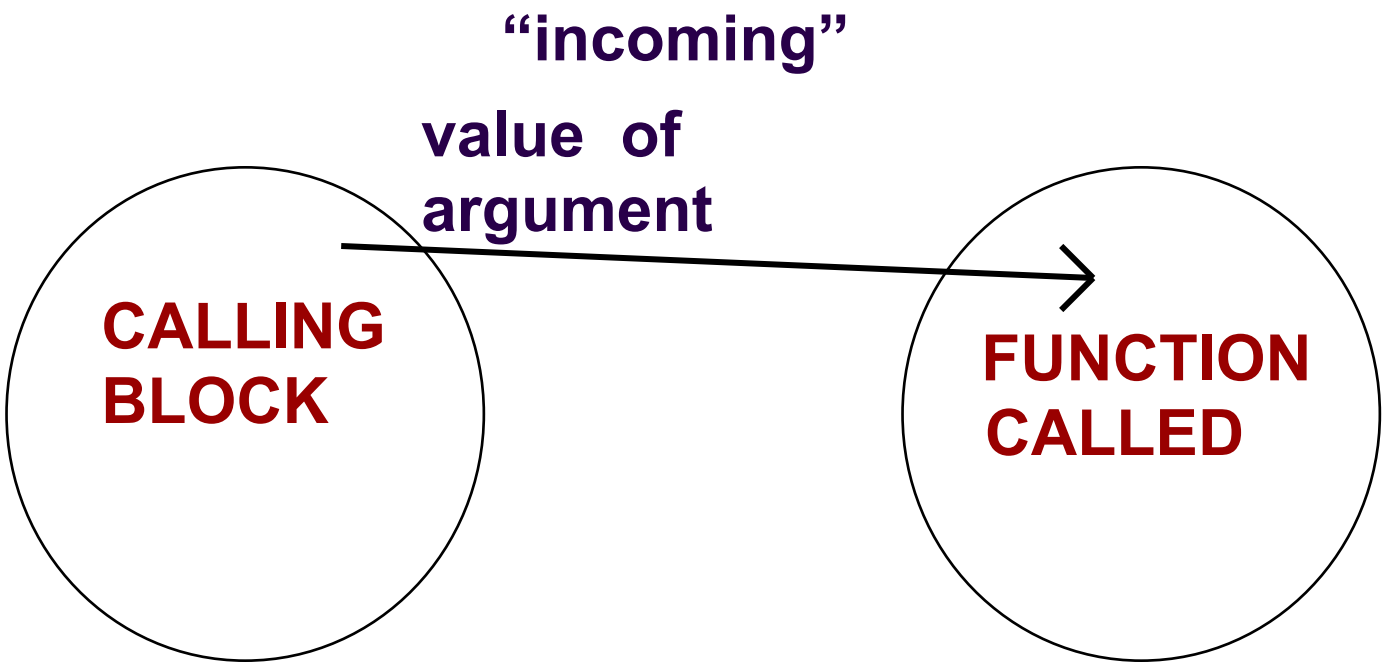
Function Definition

```
void  GetRoots(float a,  float b,  float c,
                    float& root1, float& root2)
{
    float temp; // Local variable

    temp = b * b - 4.0 * a * c;

    root1 =(-b + sqrt(temp)) /(2.0 * a);

    root2 =(-b - sqrt(temp)) /(2.0 * a);

    return;
}
```

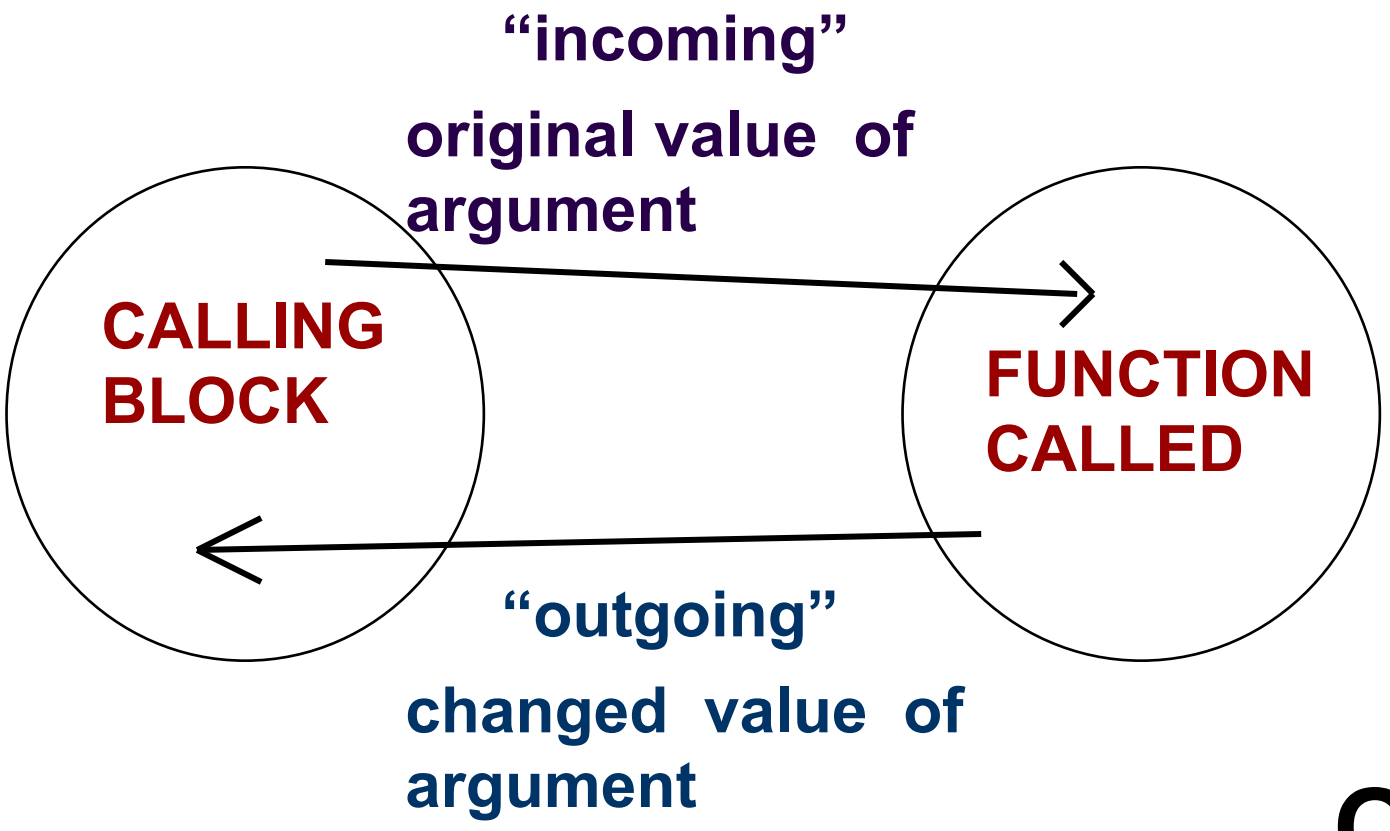
```
#include <iostream>
#include <fstream>
#include <cmath>
// Prototype
void GetRoots(float, float, float, float&, float&);
using namespace std;
void main()
{
```

```
ifstream myInfile;
    ofstream myOutfile;
    float a, b, c, first, second;

    int count = 0;
       ...... // Open files
    while(count <  5)
    {   myInfile >> a >> b >> c;
        // Call
        GetRoots(a, b, c, first, second);
        myOutfile << a << b << c
                  << first << second << endl;
        count++;
    }
       ...... // Close files
}
```

Pass-by-value

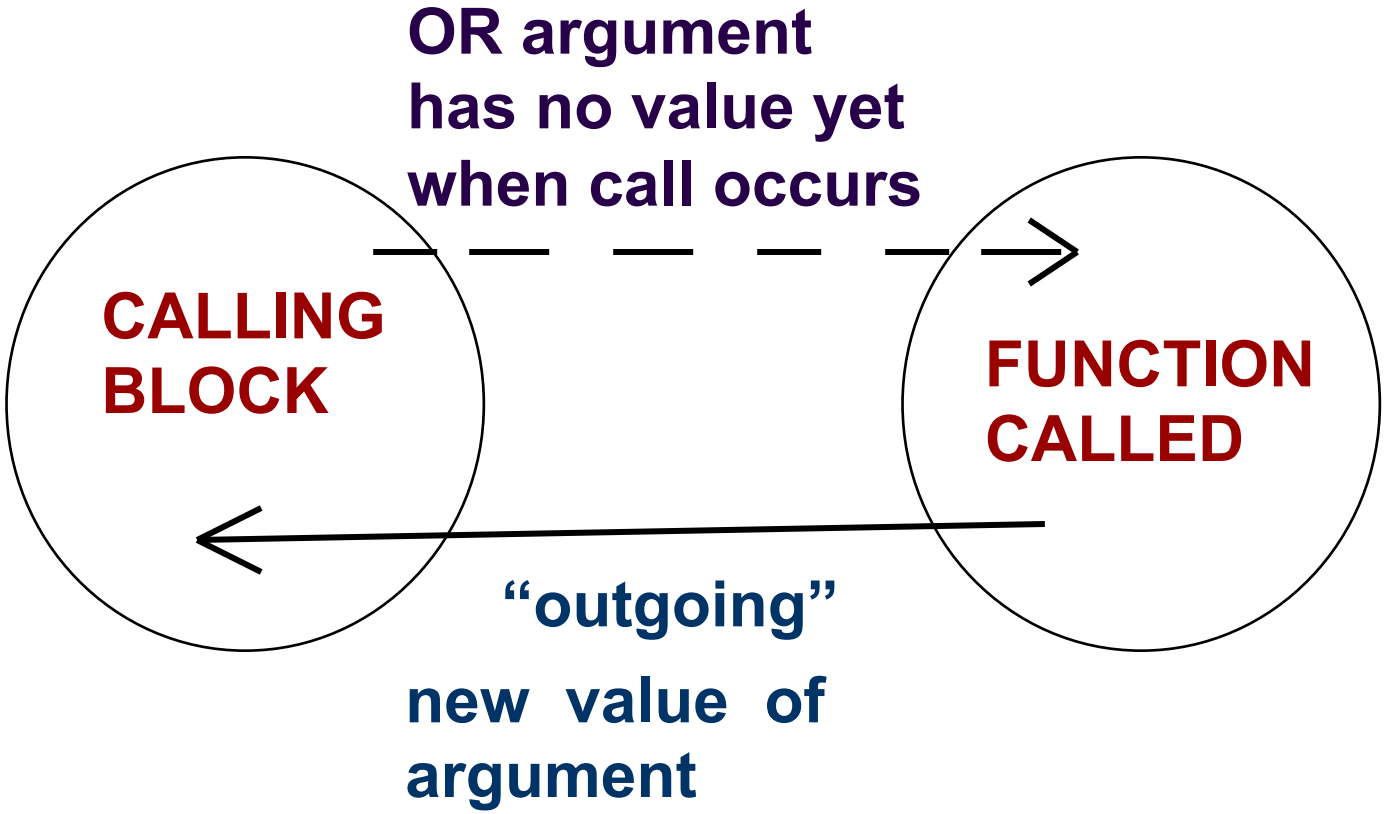


Pass-by-reference



OR,

Pass-by-reference



Data Flow Determines Passing-Mechanism

Parameter Data Flow	Passing-Mechanism
Incoming /* in */	Pass-by-value
Outgoing /* out */	Pass-by-reference
Incoming/outgoing /* inout */	Pass-by-reference

Questions

- ***Why is a function used for a task?***

  To cut down on the amount of detail in your main program (encapsulation)
- ***Can one function call another function?***

  Yes
- ***Can a function even call itself?***

  Yes, that is called recursion; it is very useful and requires special care in writing

More Questions

- ***Does it make any difference what names you use for parameters?***

  No; just use them in function body

- ***Do parameter names and argument names have to be the same?***

  No

Functions are written to specifications

- The specifications state:
- the return type;
- the parameter types;
- whether any parameters are “outgoing” and
- what task the function is to perform with its parameters
- The advantage is that teamwork can occur without knowing what the argument identifiers (names) will be

Write prototype and function definition for

- A void function called GetRating() with one reference parameter of type char

- The function repeatedly prompts the user to enter a character at the keyboard until one of these has been entered: E, G, A, P to represent Excellent, Good, Average, Poor

```
void GetRating(char&);      // Prototype
```

```
void GetRating(char&   letter)
{
    cout << "Enter employee rating." << endl;
    cout << "Use E, G, A, or P : ";
    cin >> letter;
    while((letter  !=  'E')  &&
          (letter  !=  'G')  &&
          (letter  !=  'A')  &&
          (letter  !=  'P'))
    {
        cout << "Rating invalid.  Enter again: ";
        cin   >> letter;
    }
}
```

An Assertion

An assertion is a truth-valued statement--one that is either true or false (not necessarily in C ++ code)

Examples

studentCount > 0

sum is assigned && count > 0

response == ‘y’ or ‘n’

0.0 <= deptSales <= 25000.0

beta == beta @ entry * 2

Preconditions and Postconditions

- A **precondition** is an assertion describing everything that the function requires to be true at the moment the function is invoked
- A **postcondition** describes the state at the moment the function finishes executing, providing the precondition is true
- The *caller* **is responsible for ensuring the precondition**, and the *function code* must ensure the postcondition

For example . . .

Function with Postconditions

```
void GetRating(/* out */  char&   letter)
// Precondition:  None
// Postcondition: User has been
// prompted to enter a letter
// && letter == one of these
// input values: E,G,A, or P
```

Function with Postconditions, continued

```
{
    cout  << "Enter employee rating." << endl;
    cout  << "Use E, G, A, or P : ";
    cin >> letter;
    while((letter != 'E') &&
          (letter != 'G') &&
          (letter != 'A') && (letter != 'P'))
    {
        cout << "Rating invalid. Enter again: ";
        cin   >> letter;
    }
}
```

Function with Preconditions and Postconditions

```
void  GetRoots( /* in */ float a, /* in */ float b,
                /* in */ float c,
                /* out */ float& root1,
                /* out */ float& root2)
//  Precondition: a, b, and c are assigned
//      && a != 0  &&  b*b - 4*a*c != 0
//  Postcondition: root1 and root2 are assigned
//      &&  root1 and root2 are roots of quadratic with
//      coefficients a, b, c
```

Function with Preconditions and Postconditions, continued...

```
{
    float temp;
    temp = b * b - 4.0 * a * c;
    root1 =(-b + sqrt(temp)) /(2.0 * a);
    root2 =(-b - sqrt(temp)) /(2.0 * a);
    return;
}
```

Another Function with Preconditions and Postconditions

```
void  Swap( /* inout */  int&   firstInt,
            /* inout */  int&   secondInt)
//  Precondition:  firstInt and secondInt
//  are assigned
//  Postcondition: firstInt == secondInt@entry
//                 &&  secondInt == firstInt@entry
```

Another Function with Preconditions and Postconditions, cont...

```
{
    int  temporaryInt;

    temporaryInt = firstInt;

    firstInt = secondInt;

    secondInt = temporaryInt;

}
```