CHAPTER 5

FUNCTIONS

Chapter’s Outcome

At the end of this chapter, student should be able to:

a) Learn how to build structured programs that are divided into functions.

b) Understand what is meant by the phrase "scope of variables."

c) Understand how data is passed to functions.

d) Learn how to use the library functions that are included with the compiler.

e) List and explain the four basic types of statements. (This objective is not from our textbook.)

Chapter’s Outline

1) Creating structured programs that are divided into functions
2) Library function and User defined function
3) Scope of variables
4) Parameter passing in functions
5) List and explain the four basic types of statements. (This objective is not from our textbook.)

Creating structured programs that are divided into functions.

- A function is a subpart of a program. **In C++, all programs have at least one function: main().**

- When a program is very small, the entire program can reasonably be done as part of main(). However, **as your programs grow, it will be increasingly more important to divide your program into subtasks, each of which can be performed by an individual function.**

- Every C++ program must have a main function. However, it is wise to develop an algorithm for any given project that divides large tasks into smaller ones. Each of the smaller tasks (let's say, subtasks) can be coded as C++ functions that go together with the main function to make up a structured program.

Reasons to Use Functions

1) The primary reason to use a function is that it provides a way to break a complex task into subtasks which are simpler.

2) Computations that are done more than once can be coded as functions. This eliminates duplicating program instructions.

3) The process of dividing the program into smaller parts helps you to determine the most appropriate algorithm for your problem. (This is called top-down design.)

4) Modularizing your program will make it much easier to read, both for you and for everyone else.

Reasons to Use Functions (cont…)

5) Most importantly, separating individual tasks in functions makes your program more robust and reusable. This is one of the major emphases of object-oriented programming. The more independent and readable your individual functions are, the easier it will be for you or someone else to modify your code later, or to use it as part of a new program. This is the strength of object-oriented design.

6) Using functions makes it easier to code programs.

7) Using functions also makes it easier to debug large programs.

8) Using functions makes it easier to maintain and upgrade a program after the first version has been finished.

Reasons to Use Functions (cont…)

5) Most importantly, separating individual tasks in functions makes your program more robust and reusable. This is one of the major emphases of object-oriented programming. The more independent and readable your individual functions are, the easier it will be for you or someone else to modify your code later, or to use it as part of a new program. This is the strength of object-oriented design.

6) Using functions makes it easier to code programs.

7) Using functions also makes it easier to debug large programs.

8) Using functions makes it easier to maintain and upgrade a program after the first version has been finished.

Guidelines when building programs with multiple functions:

Keep the following guidelines in mind when building programs with multiple functions:

- organization - programs with functions are easier to read and modify
- autonomy - functions should not depend on data or code outside of the function any more than necessary
- encapsulation - functions should keep all of their "details" to themselves
- reusability - since functions are meant to perform single and well-defined tasks, they may be reused in other programs or, even, by other programmers

Types of Functions

There are two types of functions:

1)**Predefined** function

2)**User defined** function

Predefined Functions

Predefined function are functions that have already been defined for you by C++. The C++ language includes libraries of predefined functions that are available for use in your programs. Some predefined function calls for **#include <math.h>** :

- **sqrt(x); returns the square root of x**
  e.g. sqrt(25) will return the value 5

- **pow(x, y); returns the value of x to the y power**
  e.g. pow(2,3) will return the value 2^3, or 8

- **abs(x); returns the absolute value of x**
  e.g. abs(-15) will return the value 15

- **fabs(x); returns the absolute value for double of x**
  e.g. fabs(-3.5) will return 3.5

a) **ceils(x); round up the value of x**
   e.g. ceil (3.4) will return 4.0

b) **floor(x); round down the value of x**
   a) e.g. floor(3.8) will return 3.0

Predefined Functions (cont...)

- In order to use predefined function calls in your program, you need to include the library containing the function you are interested in. For example, to use sqrt and pow, you need to put the following statement near the top of your file:

1) #include <math.h>

   For example, `sqrt(x); pow(x, y); abs(x); fabs(x); ceils(x); floor(x);`

2) #include <stdlib.h>

   For example, `labs(x); rand();`

3) #include <string.h>

   For example, `strcmp(string1, string2); strcpy(var, string); strlen(string);`

3) #include <ctype.h>

   For example, `toupper(); tolower();`

Predefined Functions (cont…)

Some predefined function calls for **#include <stdlib.h>** :

a)**labs(x)**; **absolute value for x;**

e.g. labs(-50000) = 50000

b)**rand()**; **return random number;**

e.g. rand() = return any number

Predefined Functions (cont...)

Some predefined function calls for **#include <string.h>** :

a)**strcmp(string1, string2);** **//string compare**

return true if two strings are different; otherwise returns 0

b)**strcpy(var, string);** **//string copy**

assign the string into variable

- **strlen(string);** **//string length**

  return the length of string

Predefined Functions (cont…)

Some predefined function calls for **#include <ctype.h>** :

a)**toupper();**

change to uppercase

b)**tolower();**

change to lowercase

For more information about predefined function please go to:

http://guinness.cs.stevens-tech.edu/~asatya/courses/cs115/lectures/lect5_1.pdf

User Defined Functions

- Functions are given valid identifier names, just like variables. However, a function name must be followed by parentheses.
- C++ programmers generally add functions to the bottom of a C++ program, that is after the main function. While this is not a requirement of the compiler, it makes your code easier to read and follow by fellow programmers.
- If the functions are listed after the main function, then function prototypes must be included at the top of the program, above the main function.
- The function prototypes "warn" the compiler about the eventual use of the prototyped function and allocate memory.
- An error will definitely result if you call a function from within the main function that has not been prototyped.

User Defined Functions (cont...)

Example:

1)The following program includes a function prototype which "declares" to the program the existence of the function `printMyName` which is detailed below the main function.

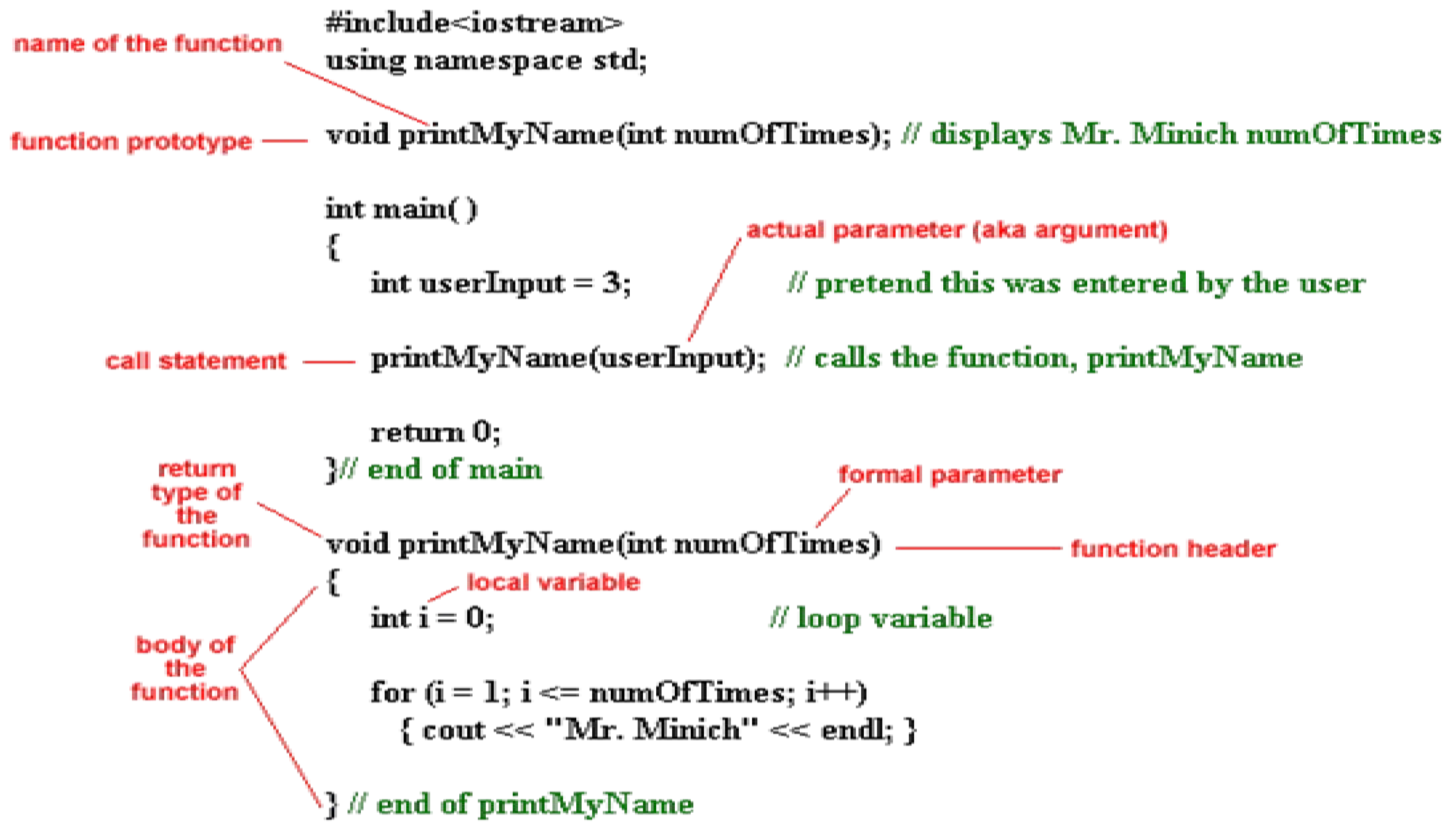
2)The main function "calls" the `printMyName` and sends the argument `userInput` which has a value of 3.

3)Function `printMyName` receives the value and stores it in the parameter `numOfTimes`.

4)The function then prints out the string "Mr. Minich" 3 times using a `for` loop.

User Defined Functions (cont...)

Example of user defined function. User defined function will be used if the function is not provided in the library.



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

void printMyName(int numOfTimes); // displays Mr. Minich numOfTimes

int main( )
{
    int userInput = 3;            // pretend this was entered by the user

    printMyName(userInput);  // calls the function, printMyName

    return 0;
}// end of main

void printMyName(int numOfTimes)
{
    int i = 0;                    // loop variable

    for (i = 1; i <= numOfTimes; i++)
      { cout << "Mr. Minich" << endl; }

} // end of printMyName
```

User Defined Functions Prototypes

- Whenever you have a function call, the function you are calling must be known to the compiler. There are two ways in which you can be sure this occurs:
  1) place the entire function body in the **same file** as the calling function, and **above** it in the file.
     - It is very important that the code for the function body be placed above the code of the calling function, because the compiler reads your file from top to bottom, and must already have "seen" the function before it is called.
  2) place a **prototype**, or definition of your function, in the same file and above the calling function.
     - A prototype tells the compiler that -- even though it doesn't immediately see the body of your function -- it should not be concerned, because your function will be available, either later in the same file or in another file.

User Defined Functions Prototypes (cont...)

- A prototype looks exactly like a function header, with ONE exception ... it is followed by a ";"
- This tells the compiler not to expect the rest of the function (called the "function body"), which would normally follow immediately after the function header.

Example:

```
int my_function() // this is a function header
    {
                // function body
    }


int my_function(); // this is a prototype for function my_function()
```

User Defined Parameters

**** Every function, regardless of which type it is, has a "parameter list."**

- This parameter list is a list of values that are passed to or from the function. Sometimes, however, this parameter list is empty, because no values need to be passed.

- When the parameter list is empty, the parentheses that are used to contain the parameters are empty as shown in the example below.

- Example:

```
void main()
```

OR

```
int main()
```

- Most functions, though, contain at least one parameter in their parameter lists.

User Defined Parameters (cont…)

Why use parameters?

- Parameters allow you to share information between functions.
- Generally, variables that are declared within a function are local to that function. This means that other functions generally do not have access to those variables unless they are "sent" from one function to another by use of parameters.

"pass by value" parameters:

- Parameters that are "passed by value" provide information to the function that is being called.
- When a parameter is passed by value, it does not change anything in the calling function.
- (Parameters that change values in the calling function are called "pass by reference" parameters; these will be discussed in a separate slide.)

User Defined Scope of Variables

- Variables are either **global** or **local** in nature.

Global variables	Local variables
They are declared outside and **above** the main function	They are declared inside of a function, including main
It can be used in **any** function throughout the program	It can be used **only** in that function.
But, it is not wise to use global variables any more than you have to. One small error (for example, miscalculating a value) could lead to multiple, hard-to-find errors in large programs.	They cannot be used or referred to in other functions.

User Defined Local Variables

- A local variable is a variable declared within a block and not accessible outside that block.

- Formal function parameters and variables declared within the body of a function are local to that function:

```
 void PrintLines( int numLines )
        {
        int count;      // Loop control local variable
count = 1;

while (count <= numLines)
        {
                        cout << "**************** " << endl;
                        count++;
                }
        }
```

- `numLines` and `count` are local to the void function `PrintLines`.

User Defined Local Variables (cont...)

- In order to strengthen your program and functions' autonomy and encapsulation, you should avoid the use of global variables and try to use local variables instead.

- The **scope** of a variable is the area in which it can be legally referenced. A global variable's scope is the whole program. A local variable's scope is the function in which it is declared.

- In the example program above (with the function `printMyName`), the variable `userInput` is a local variable which belongs to the main function. The variable `i` is a local variable to the function `printMyName`. (`numOfTimes` isn't really a variable. It is a parameter technically. See objective 3 for more details on parameters.)

Parameter passing in a function

- C++ allows programmers to define their own functions.
- For example the following is a definition of a function which given the co-ordinates of a point (x,y) will return its distance from the origin.

```
// Returns the distance of (x, y) from origin
float distance(float x, float y)
{
    float dist;  //local variable
    dist = sqrt(x * x + y * y);

    return dist;
}
```

Parameter passing in a function (cont...)

- Data is passed to functions as arguments (sometimes referred to as **actual parameters**).
- When a function is "called" by the main function (or another function), one or more arguments are passed to the function.
- On the receiving end, the function accepts these arguments (technically, as **formal parameters**).
- The variable names of the arguments (that is, **actual parameters**) from the "calling" function **do not have to be the same as the names of the formal parameters** in the "called" function.
- But, the datatypes of the arguments and the parameters should match exactly. An error could occur if the data types do not match.

Parameter passing in a function (cont...)

- We type "return 0;" as the last statement in the main functions of all the programs that we write in this C++course.

- This returns a message (of "0") to the operating system indicating that our C++ program ran successfully.

- A returned nonzero value is often used by programmers to indicate that an error occurred during the program's execution.

`Parameter passing` in a function (cont...)

- Often, though, you want your function to return a computed value to the calling function. This value is used in the exact statement that the function was called.

- For example,

```
fahrenheit = celsiusToFahrenheit(celsius);
```

- sends the argument, `celsius`, to the function, `celsiusToFahrenheit`, where it is used to determine the equivalent Fahrenheit temperature.

- Then, the final value is returned from within the function to the main function where it is assigned to the `fahrenheit` variable.

Parameter passing in a function (cont...)

There are technically three ways to pass data as arguments to functions.

You can pass data :

1)by value,

2)by reference,

3)by address.

Parameter passing by value

passing by value is the preferred method. You simply use a variable name, an actual numeric literal, or an expression in the parentheses of the call statement. This method is the safest because it does not actually change the value of the argument in the calling function.

In the example below, the argument named `price` is passed by value to the function `addTax`.

```
#include <iostream.h>

double addTax(double);

int main()
{
     double price = 10.00;
     double finalPrice = 0.0;

     finalPrice = addTax(price); // function call

     cout << "The final price with tax is " << finalPrice << endl;
     return 0;
}// end of main

double addTax(double initialPrice)
{
     return (initialPrice + initialPrice * 0.06);
}// end of addTax
```

Parameter `passing by reference`

- **passing by reference** is to be used when you want the function to actually and permanently change the values of one or more variables. This method is dangerous but beneficial sometimes since it allows you to affect the values of arguments in your calling function.

- You would purposefully pass by reference if you want to return 2 or more values to the calling function. Since it is not possible in C++ to execute two return statements within a function and since it is not possible to return two values in the same return statement (i.e. return `myResult`, `myOtherResult` is illegal), passing by reference can allow you to modify two or more variables in the calling function (i.e. main).

- Also, if you need to pass a really large variable or object, it saves memory to pass by reference. Since a copy of the variable is not being made when you pass by reference, less memory overhead is required.

- For int, double, and char variables, passing by reference to save memory is not really necessary. However, if you were to pass a large "object" such as a piece of clipart or a whole file, you would intentionally pass by reference.

Parameter passing by reference (cont...)

- You must use an ampersand (&) before the formal parameter names in the **function header** (the first line of the **function definition**) to denote passing by reference.In the example below, the argument named price is passed by reference to the function addTax.

```
#include <iostream.h>

   void addTax(double &); // function prototype

   int main()
   {
      double price = 10.00;

      addTax(price); // function call
      cout << "The final price with tax is " << price << endl;
      return 0;
   } // end of main

   void addTax(double & incomingPrice) // function header or definition
   {
      incomingPrice = incomingPrice + incomingPrice * 0.06;
   } // end of addTax
```

Parameter passing by address

- **passing by address** is technically what happens when you pass an array to a function.

- Since arrays are actually pointers to one memory address, you are not passing the actual array but just information that tells the function where the first element of the array is stored in the RAM of the computer.

- Making changes to the array within the function will affect the values within the array permanently, so this method should be used with care.

Library Functions

- There are many functions available to C++ programmers which were written by other programmers. You can use the **#include** compiler directive at the top of your program to take advantage of these functions.

- The source code for those functions does not need to be included within your program. In fact, you do not even have to know exactly how they work.

- You do have to know how many arguments to send to the functions and what datatypes to use for those functions. However, some of the "older" header functions end with a ".h" extension.

- For example, there are many mathematical library functions that can be used in your program if you "include" the **math.h** header file at the top of your program. Similarly, the **ctype.h** header file may be useful as well.

Four Basic Types of Statement in C++

There are four basic types of statement in C++:

- sequence
- selection (`if` and `switch` structures)
- iteration (`for` and `while` loops)
- invocation (`functions`)

Sequence statements

Sequence statements are simply statements that naturally follow one another from the top of the program to the bottom. For example, in the "hello world" program all of the statements are sequence statements.

Selection statements

- Selection statements include if statements and switch statements. Selection statements are sometimes called conditional or decision statements.

- An example of a selection statement is "If I scored a 60 or higher, then I passed the course; otherwise, I failed."

- There are really two possible end results of this statement. Therefore, both paths are not followed. You cannot pass the course and fail the course at the same time. (We will study these kinds of statements in a later chapter.)

- In C++, this if statement might look like this:

```
if (score >= 60)
     {
               cout << "I passed" << endl;
     }
     else
     {
               cout << "I failed" << endl;
     }
```

Iteration/Looping statements

- **Iteration** statements allow the compiler to return to a point higher in the program in order to continuously repeat one or more statements.

- **All loops** including `while`, `do/while`, and `for` loops are examples of iteration statements.

- In C++, this is an example of an iteration statement (`while` loop) that prints "hello world" ten times:

```
while (num < 10)
    {
            cout << "hello world" << endl;
            num = num + 1;
    }
```

Invocation/Function statements

- **Invocation** statements allow you to place a block of statements that you wish to use several times during a program in one section (usually at the bottom of your program).

- You then have the ability to "call" those statements whenever you wish to execute them without having to retype the block of statements over and over again within the program.

- In C++, we use "function calls" as invocation statements. For example, the statement:

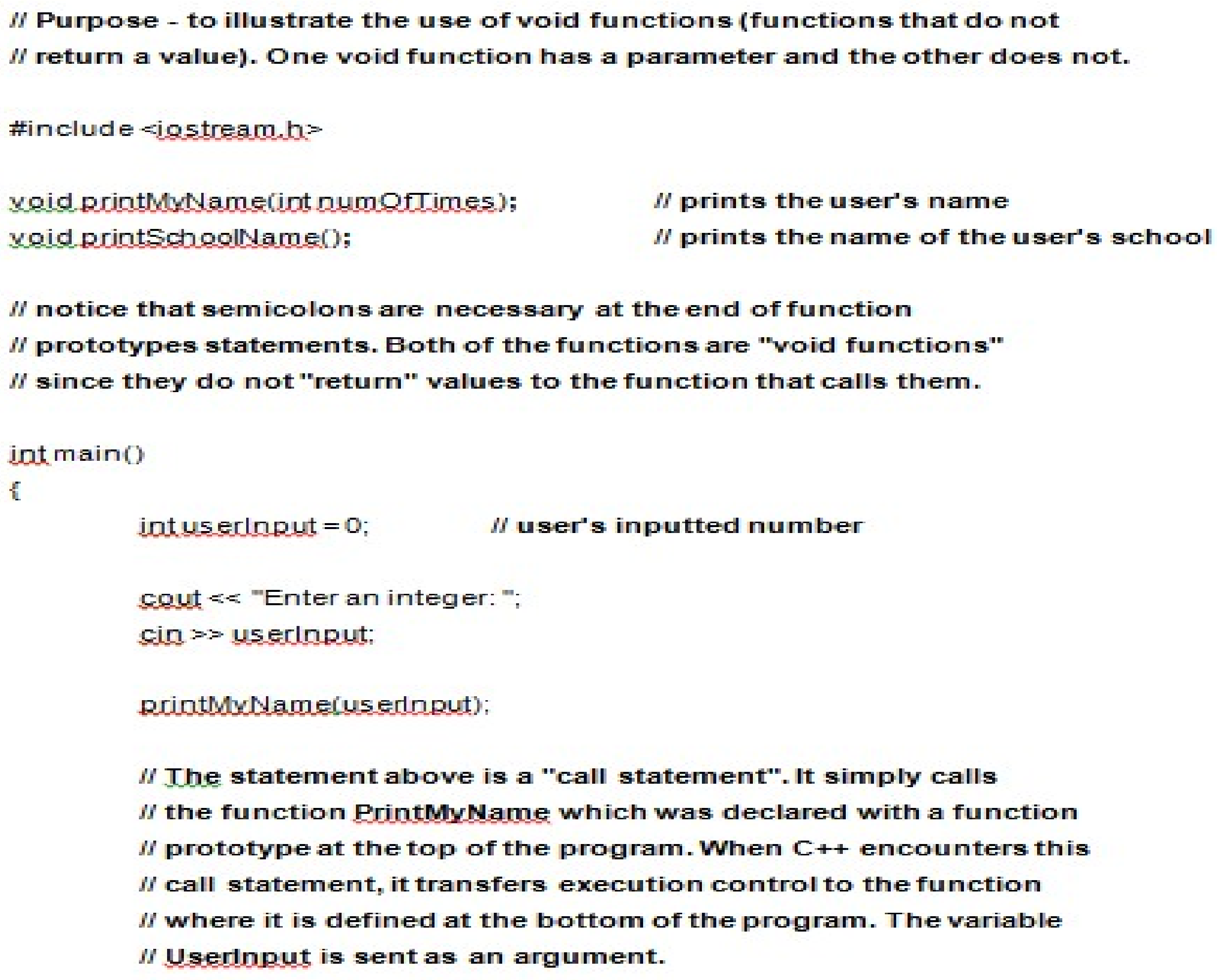
```
displayHelloWorld( );
```

will call (i.e. invoke) the function below:

```
void displayHelloWorld()
    {
            cout << "hello world" << endl;
    }
```

void function (functions that do not return a value)

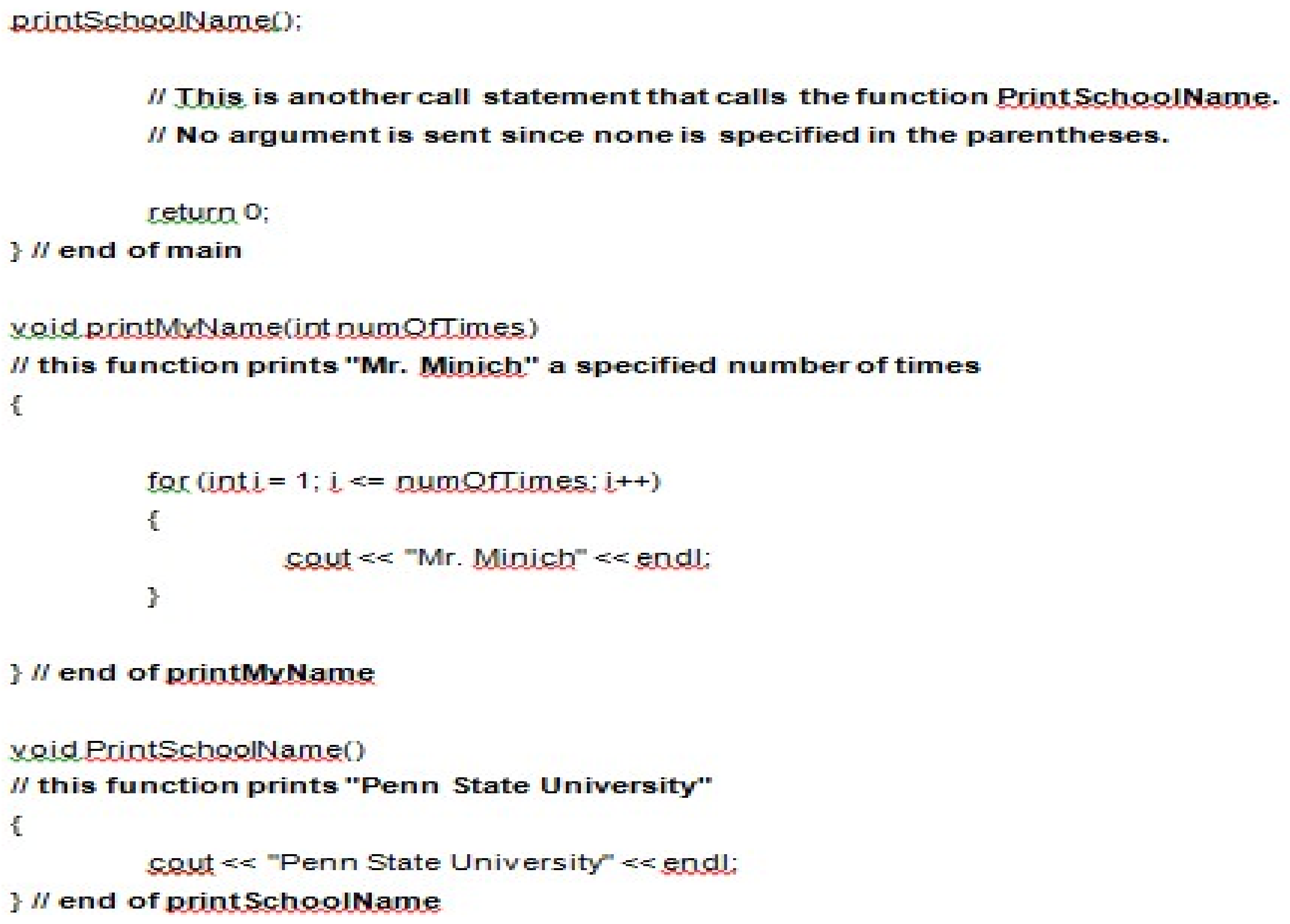
Example 1:



```
// Purpose - to illustrate the use of void functions (functions that do not
// return a value). One void function has a parameter and the other does not.

#include <iostream.h>

void printMyName(int numOfTimes);          // prints the user's name
void printSchoolName();                    // prints the name of the user's school

// notice that semicolons are necessary at the end of function
// prototypes statements. Both of the functions are "void functions"
// since they do not "return" values to the function that calls them.

int main()
{
          int userInput = 0;          // user's inputted number

          cout << "Enter an integer: ";
          cin >> userInput;

          printMyName(userInput);

          // The statement above is a "call statement". It simply calls
          // the function PrintMyName which was declared with a function
          // prototype at the top of the program. When C++ encounters this
          // call statement, it transfers execution control to the function
          // where it is defined at the bottom of the program. The variable
          // UserInput is sent as an argument.
```

void function (functions that do not return a value)

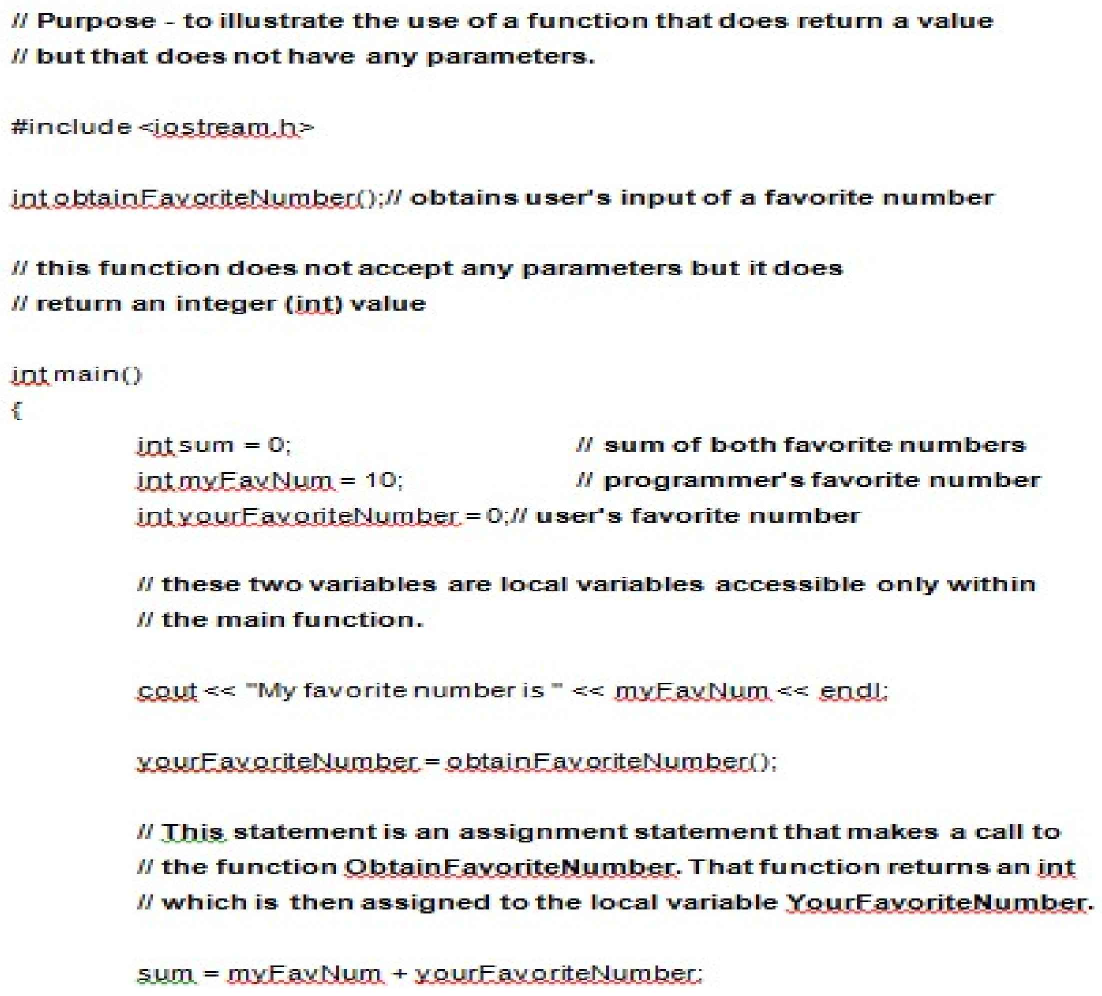
Example 1 (cont...)



```
printSchoolName();

          // This is another call statement that calls the function PrintSchoolName.
          // No argument is sent since none is specified in the parentheses.

          return 0;
} // end of main

void printMyName(int numOfTimes)
// this function prints "Mr. Minich" a specified number of times
{

          for (int i = 1; i <= numOfTimes; i++)
          {
                    cout << "Mr. Minich" << endl;
          }

} // end of printMyName

void PrintSchoolName()
// this function prints "Penn State University"
{
          cout << "Penn State University" << endl;
} // end of printSchoolName
```

Function with return value but without parameter

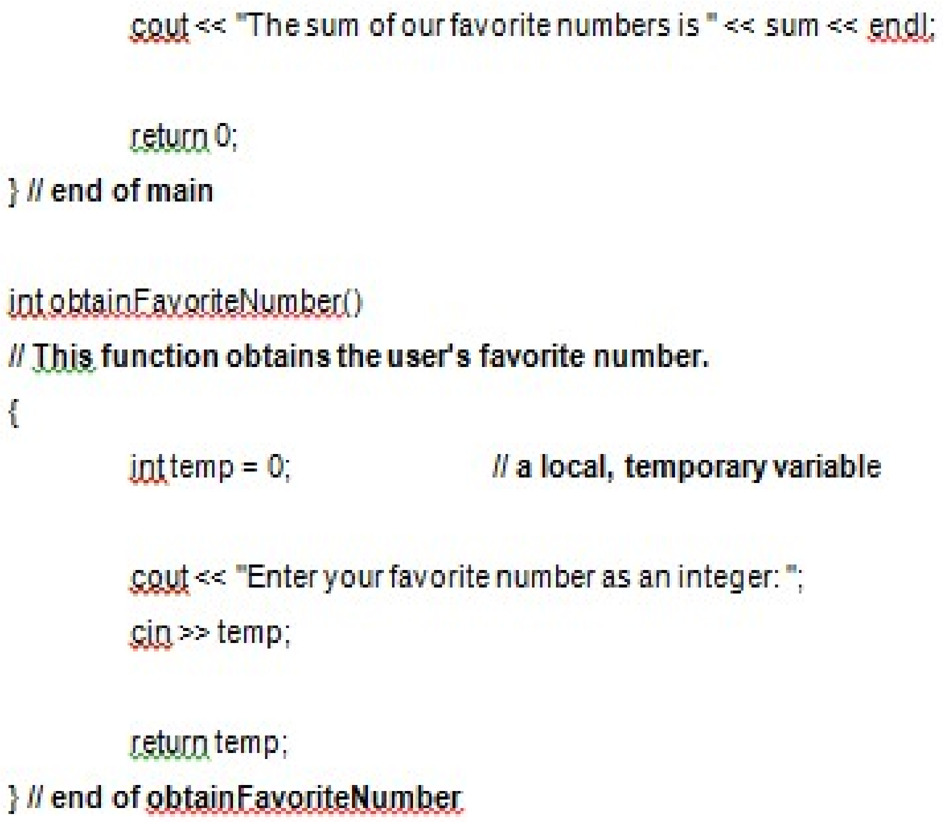
Example 2:



```
// Purpose - to illustrate the use of a function that does return a value
// but that does not have any parameters.

#include <iostream.h>

int obtainFavoriteNumber();// obtains user's input of a favorite number

// this function does not accept any parameters but it does
// return an integer (int) value

int main()
{
          int sum = 0;                          // sum of both favorite numbers
          int myFavNum = 10;                    // programmer's favorite number
          int yourFavoriteNumber = 0;// user's favorite number

          // these two variables are local variables accessible only within
          // the main function.

          cout << "My favorite number is " << myFavNum << endl;

          yourFavoriteNumber = obtainFavoriteNumber();

          // This statement is an assignment statement that makes a call to
          // the function ObtainFavoriteNumber. That function returns an int
          // which is then assigned to the local variable YourFavoriteNumber.

          sum = myFavNum + yourFavoriteNumber;
```

Function **with return value but** without **parameter**

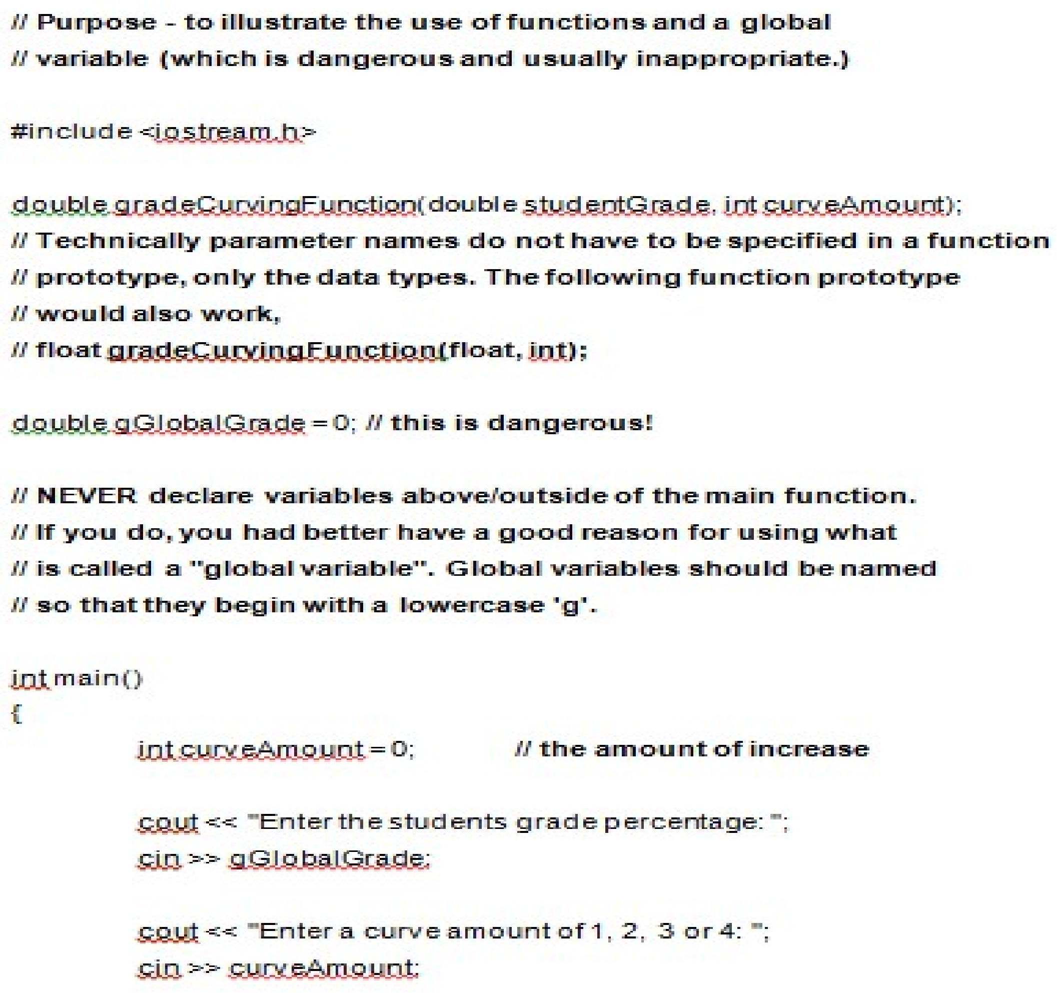
Example 2 (cont…)



```
          cout << "The sum of our favorite numbers is " << sum << endl;

          return 0;
} // end of main

int obtainFavoriteNumber()
// This function obtains the user's favorite number.
{
          int temp = 0;                    // a local, temporary variable

          cout << "Enter your favorite number as an integer: ";
          cin >> temp;

          return temp;
} // end of obtainFavoriteNumber
```

Function with global variables

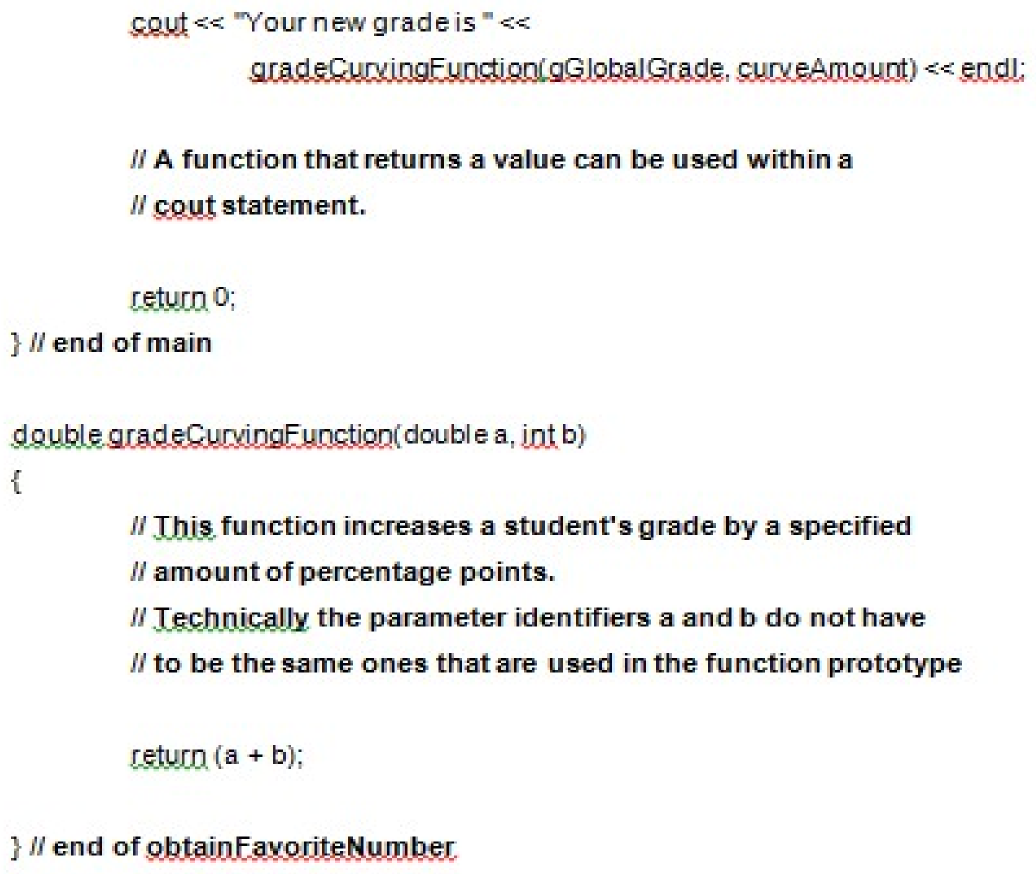
Example 3:



```
// Purpose - to illustrate the use of functions and a global
// variable (which is dangerous and usually inappropriate.)

#include <iostream.h>

double gradeCurvingFunction(double studentGrade, int curveAmount);
// Technically parameter names do not have to be specified in a function
// prototype, only the data types. The following function prototype
// would also work,
// float gradeCurvingFunction(float, int);

double gGlobalGrade = 0; // this is dangerous!

// NEVER declare variables above/outside of the main function.
// If you do, you had better have a good reason for using what
// is called a "global variable". Global variables should be named
// so that they begin with a lowercase 'g'.

int main()
{
        int curveAmount = 0;        // the amount of increase

        cout << "Enter the students grade percentage: ";
        cin >> gGlobalGrade;

        cout << "Enter a curve amount of 1, 2, 3 or 4: ";
        cin >> curveAmount;
```

Function with global variables

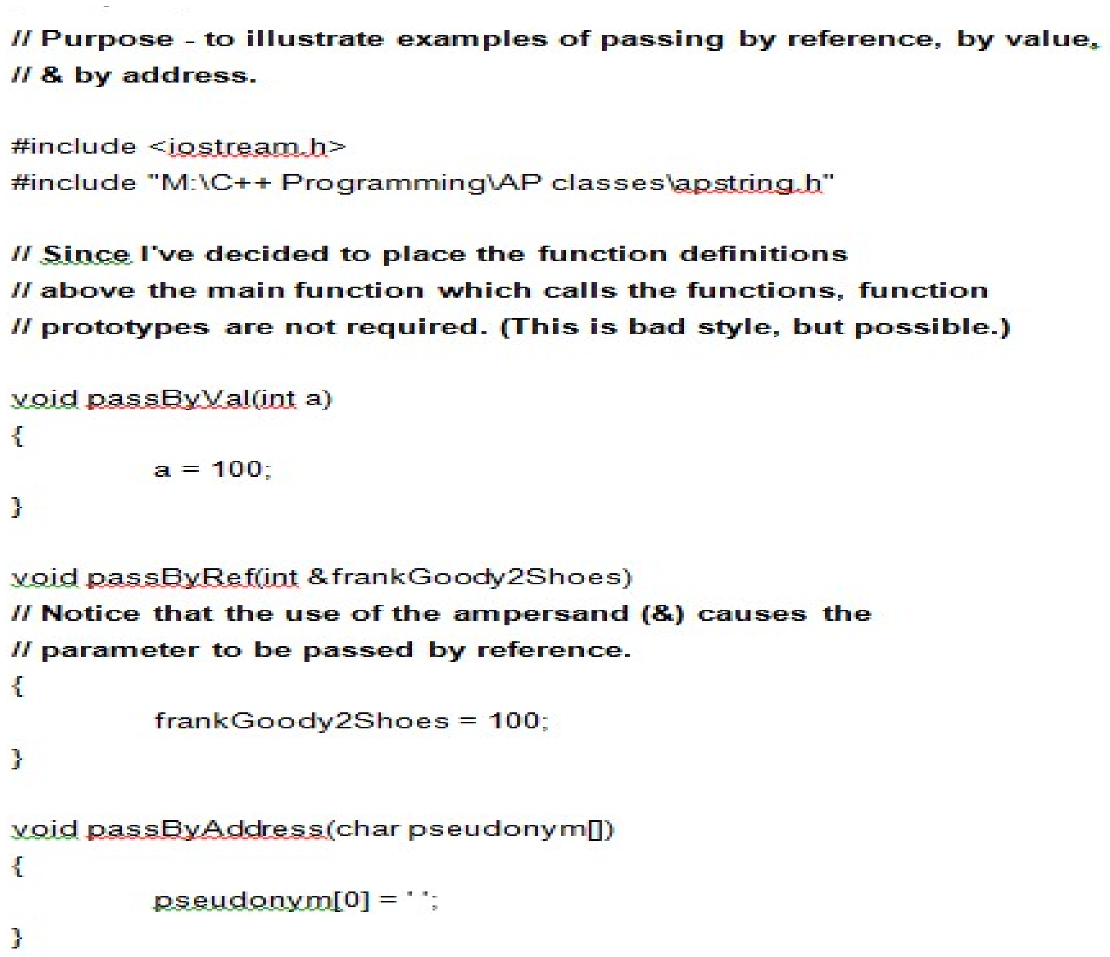
Example 3 (cont…)



```
        cout << "Your new grade is " <<
                gradeCurvingFunction(gGlobalGrade, curveAmount) << endl;

        // A function that returns a value can be used within a
        // cout statement.

        return 0;
} // end of main

double gradeCurvingFunction(double a, int b)
{
        // This function increases a student's grade by a specified
        // amount of percentage points.
        // Technically the parameter identifiers a and b do not have
        // to be the same ones that are used in the function prototype

        return (a + b);

} // end of obtainFavoriteNumber
```

Passing by reference,by value & by address

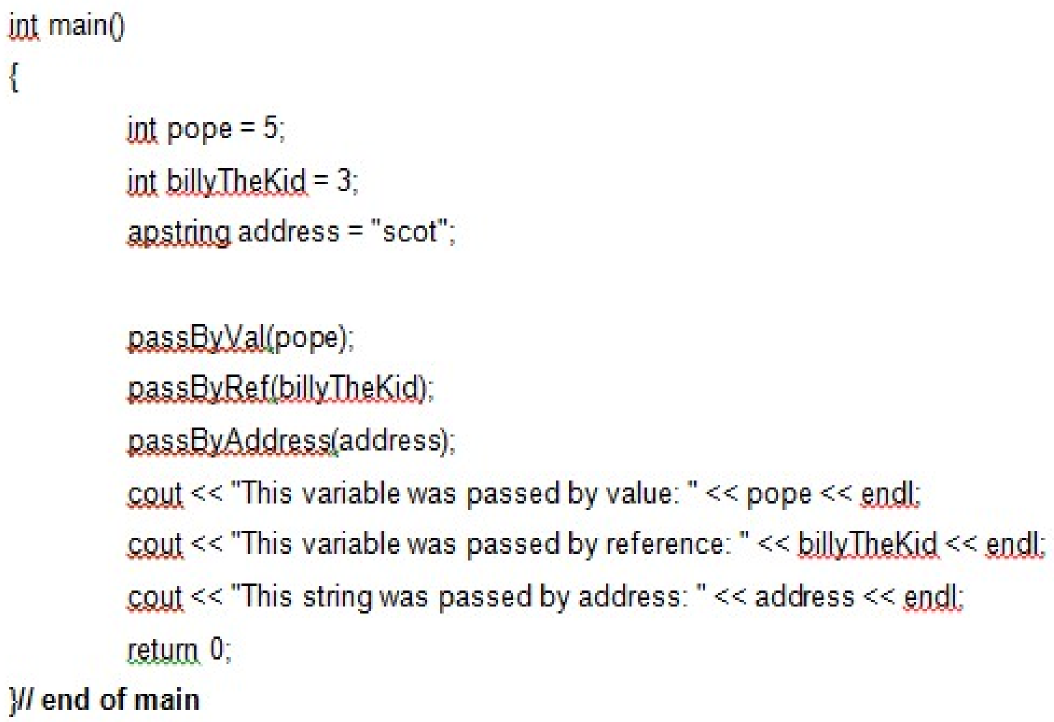
Example 4:



```
// Purpose - to illustrate examples of passing by reference, by value,
// & by address.

#include <iostream.h>
#include "M:\C++ Programming\AP classes\apstring.h"

// Since I've decided to place the function definitions
// above the main function which calls the functions, function
// prototypes are not required. (This is bad style, but possible.)

void passByVal(int a)
{
            a = 100;
}

void passByRef(int &frankGoody2Shoes)
// Notice that the use of the ampersand (&) causes the
// parameter to be passed by reference.
{
            frankGoody2Shoes = 100;
}

void passByAddress(char pseudonym[])
{
            pseudonym[0] = ' ';
}
```

Passing by reference,by value & by address

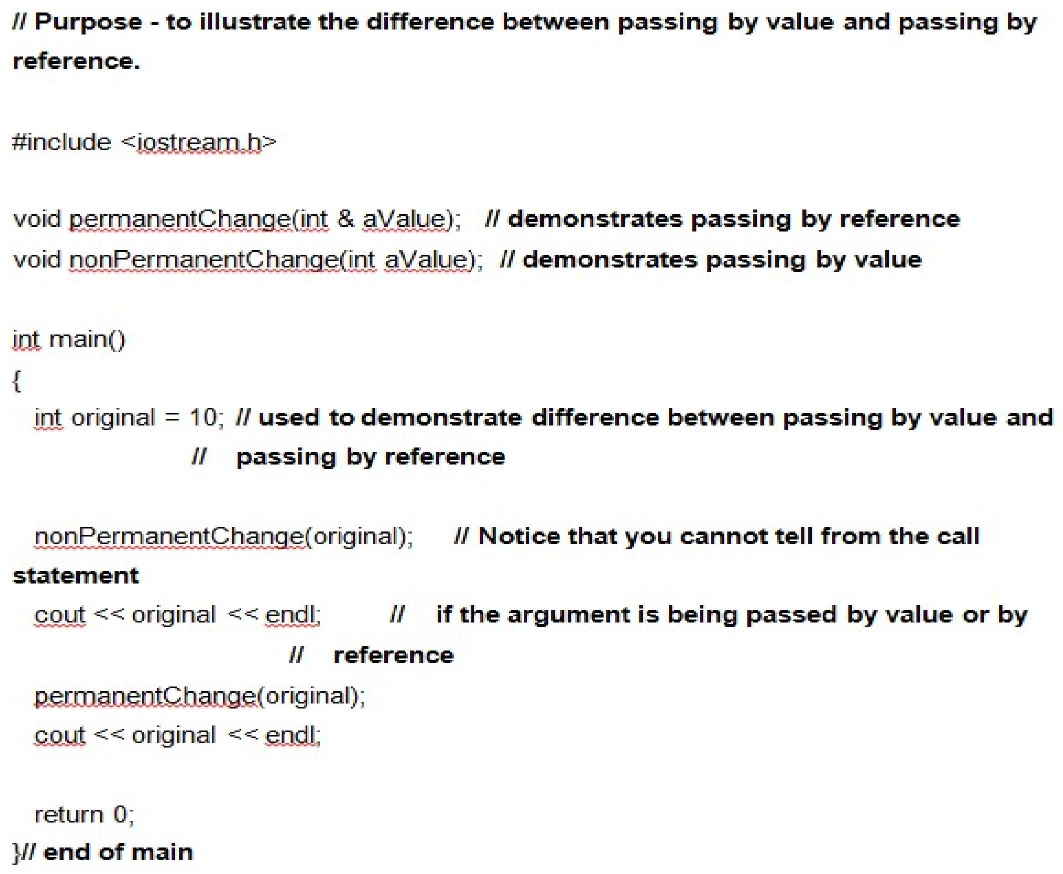
Example 4 (cont…)



```
int main()
{
            int pope = 5;
            int billyTheKid = 3;
            apstring address = "scot";

            passByVal(pope);
            passByRef(billyTheKid);
            passByAddress(address);
            cout << "This variable was passed by value: " << pope << endl;
            cout << "This variable was passed by reference: " << billyTheKid << endl;
            cout << "This string was passed by address: " << address << endl;
            return 0;
}// end of main
```

Difference between passing by value and passing by reference

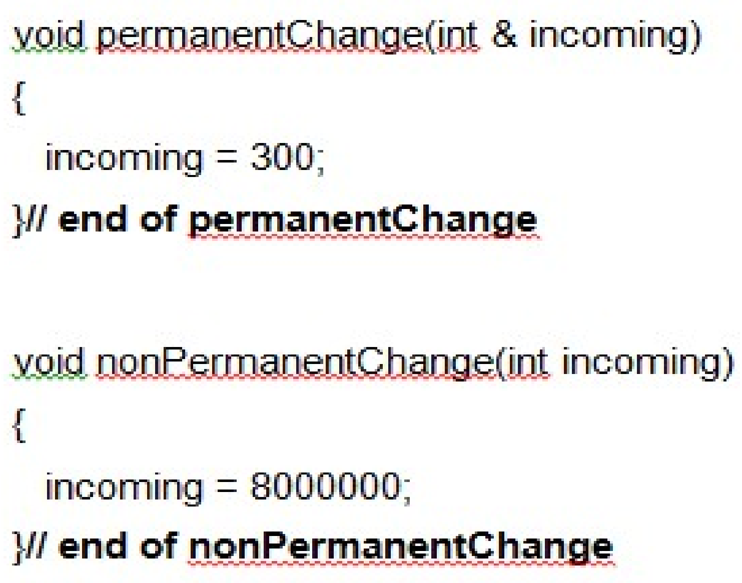
Example 5:



```
// Purpose - to illustrate the difference between passing by value and passing by
reference.

#include <iostream.h>

void permanentChange(int & aValue);   // demonstrates passing by reference
void nonPermanentChange(int aValue);  // demonstrates passing by value

int main()
{
  int original = 10; // used to demonstrate difference between passing by value and
                     //   passing by reference

  nonPermanentChange(original);     // Notice that you cannot tell from the call
statement
  cout << original  << endl;        //   if the argument is being passed by value or by
                                    //   reference
  permanentChange(original);
  cout << original  << endl;

  return 0;
}// end of main
```

Difference between `passing by value` and `passing by reference`

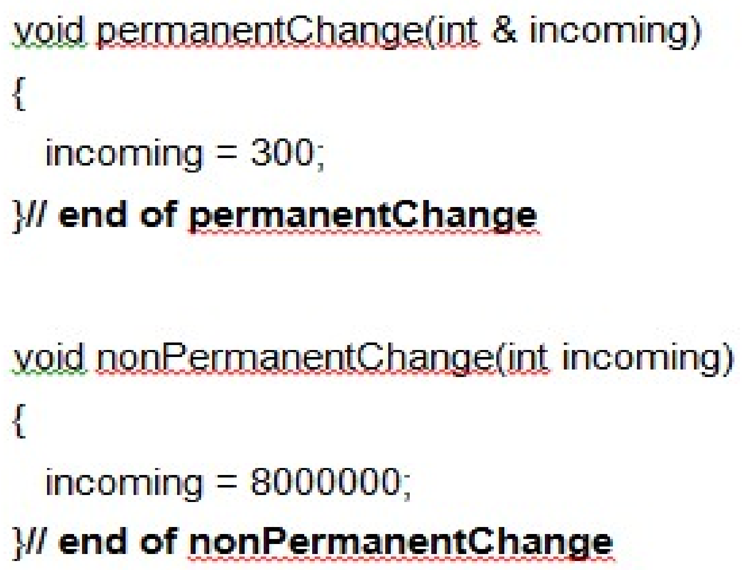
Example 5 (cont…)



```
void permanentChange(int & incoming)
{
   incoming = 300;
}// end of permanentChange

void nonPermanentChange(int incoming)
{
   incoming = 8000000;
}// end of nonPermanentChange
```

Difference between `passing by value` and `passing by reference`

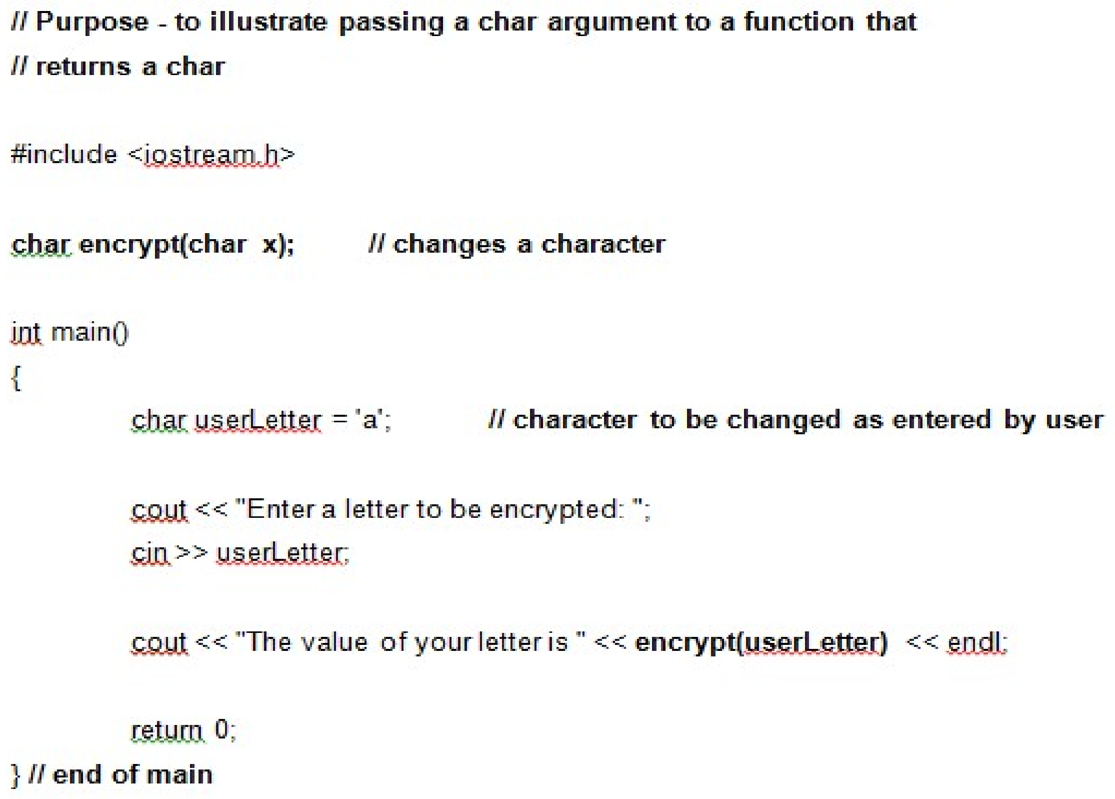
Example 5 (cont…)



```
void permanentChange(int & incoming)
{
   incoming = 300;
}// end of permanentChange

void nonPermanentChange(int incoming)
{
   incoming = 8000000;
}// end of nonPermanentChange
```

Passing a char argument to a function that returns a char

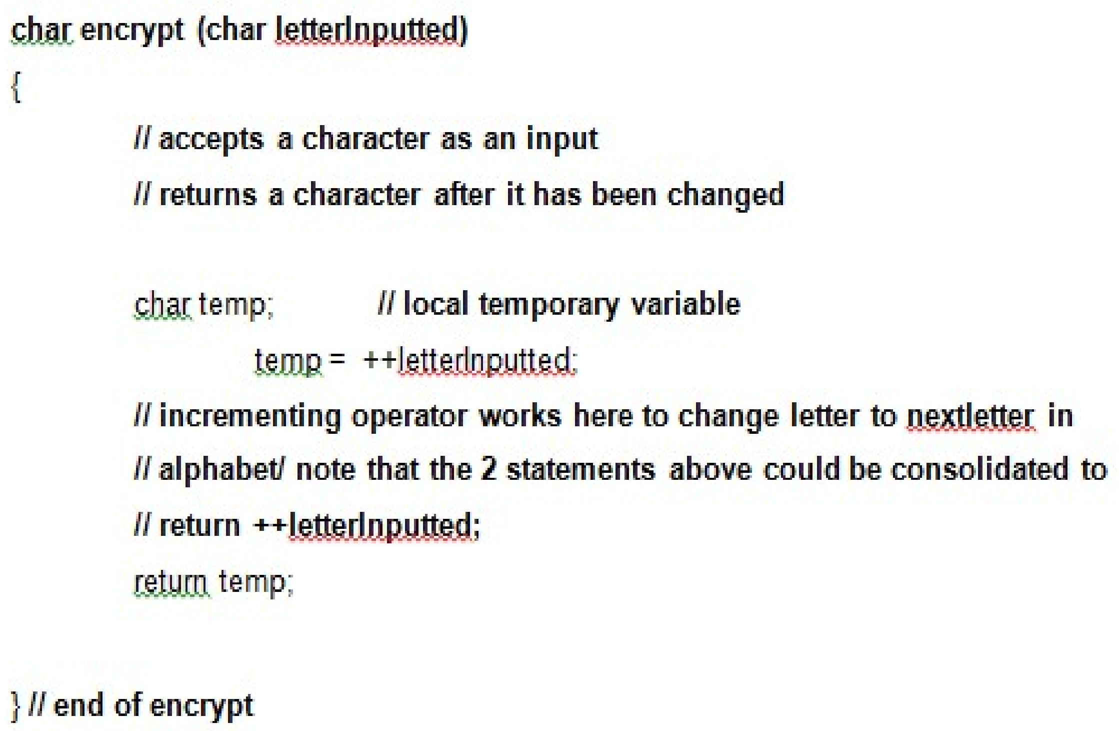
Example 6:



```
// Purpose - to illustrate passing a char argument to a function that
// returns a char

#include <iostream.h>

char encrypt(char  x);        // changes a character

int main()
{
            char userLetter = 'a';          // character  to be changed as entered by user

            cout << "Enter a letter to be encrypted: ";
            cin >> userLetter;

            cout << "The value  of your letter is " << encrypt(userLetter)  << endl;

            return  0;
} // end of main
```

Passing a char argument to a function that returns a char

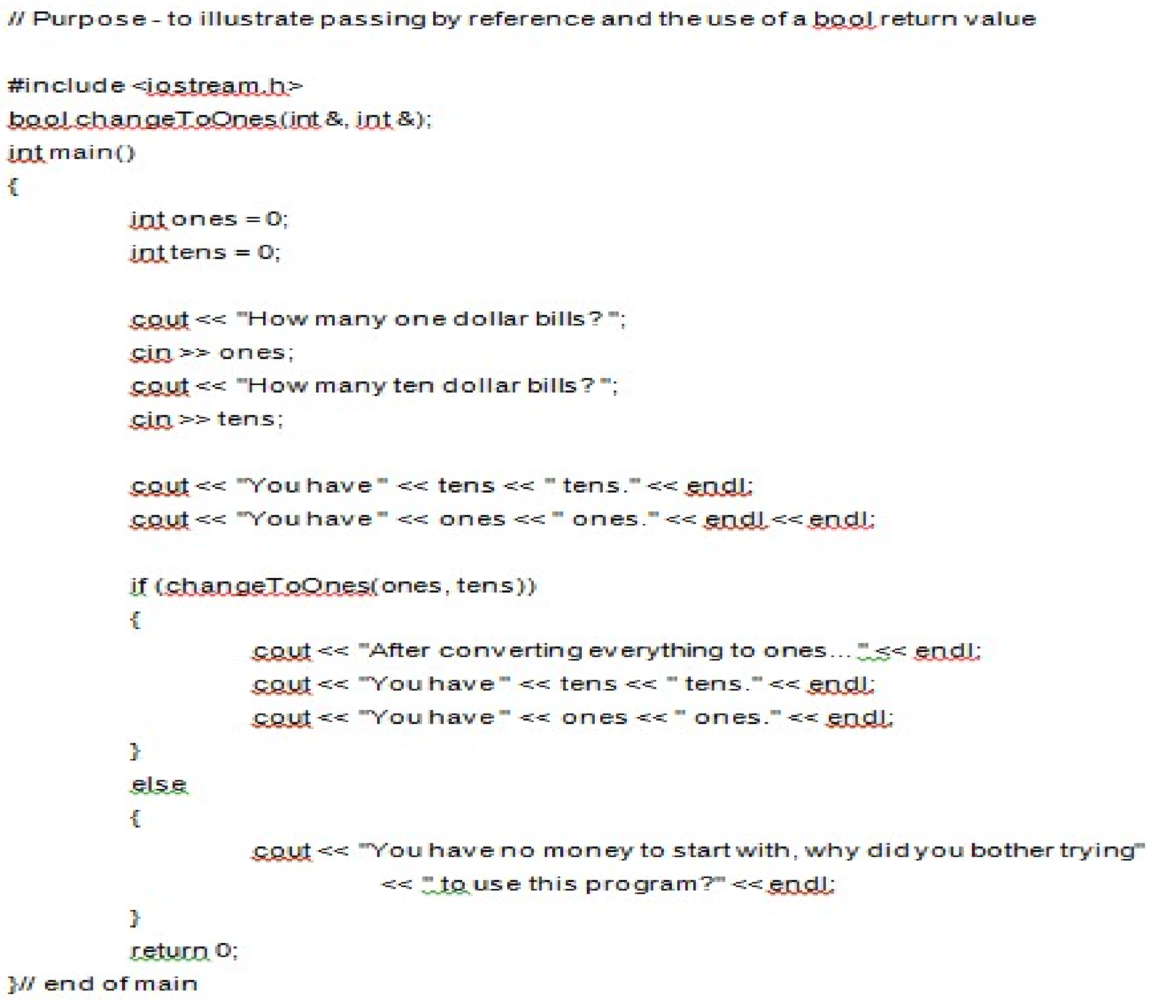
Example 6 (cont…)



```
char encrypt (char letterInputted)
{
        // accepts a character as an input
        // returns a character after it has been changed

        char temp;          // local temporary variable
                temp =  ++letterInputted;
        // incrementing operator works here to change letter to nextletter in
        // alphabet/ note that the 2 statements above could be consolidated to
        // return ++letterInputted;
        return temp;

} // end of encrypt
```

Passing by reference to a function

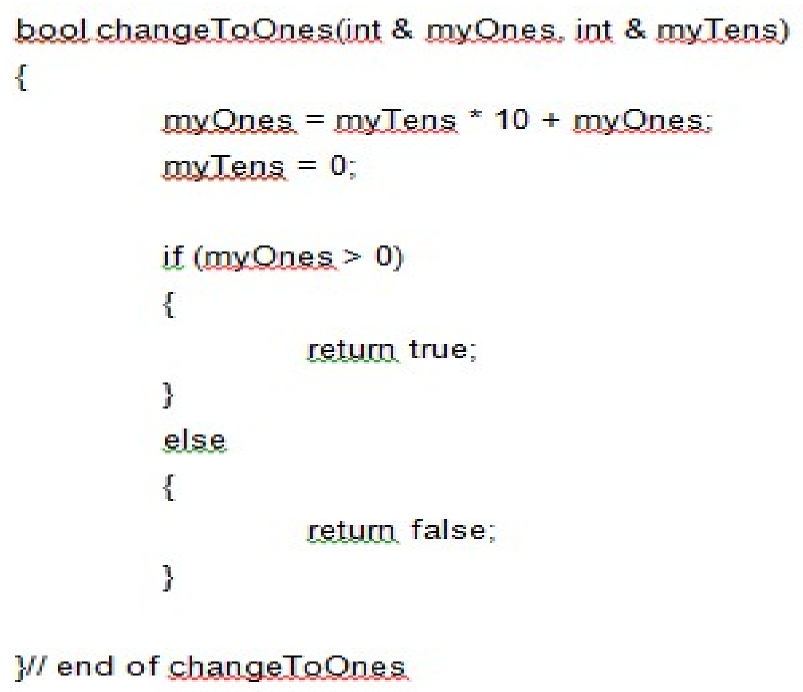
Example 7:



```
// Purpose - to illustrate passing by reference and the use of a bool return value

#include <iostream.h>
bool changeToOnes(int &, int &);
int main()
{
        int ones = 0;
        int tens = 0;

        cout << "How many one dollar bills? ";
        cin >> ones;
        cout << "How many ten dollar bills? ";
        cin >> tens;

        cout << "You have " << tens << " tens." << endl;
        cout << "You have " << ones << " ones." << endl << endl;

        if (changeToOnes(ones, tens))
        {
                cout << "After converting everything to ones... " << endl;
                cout << "You have " << tens << " tens." << endl;
                cout << "You have " << ones << " ones." << endl;
        }
        else
        {
                cout << "You have no money to start with, why did you bother trying"
                        << " to use this program?" << endl;
        }
        return 0;
}// end of main
```

Passing by reference to a function

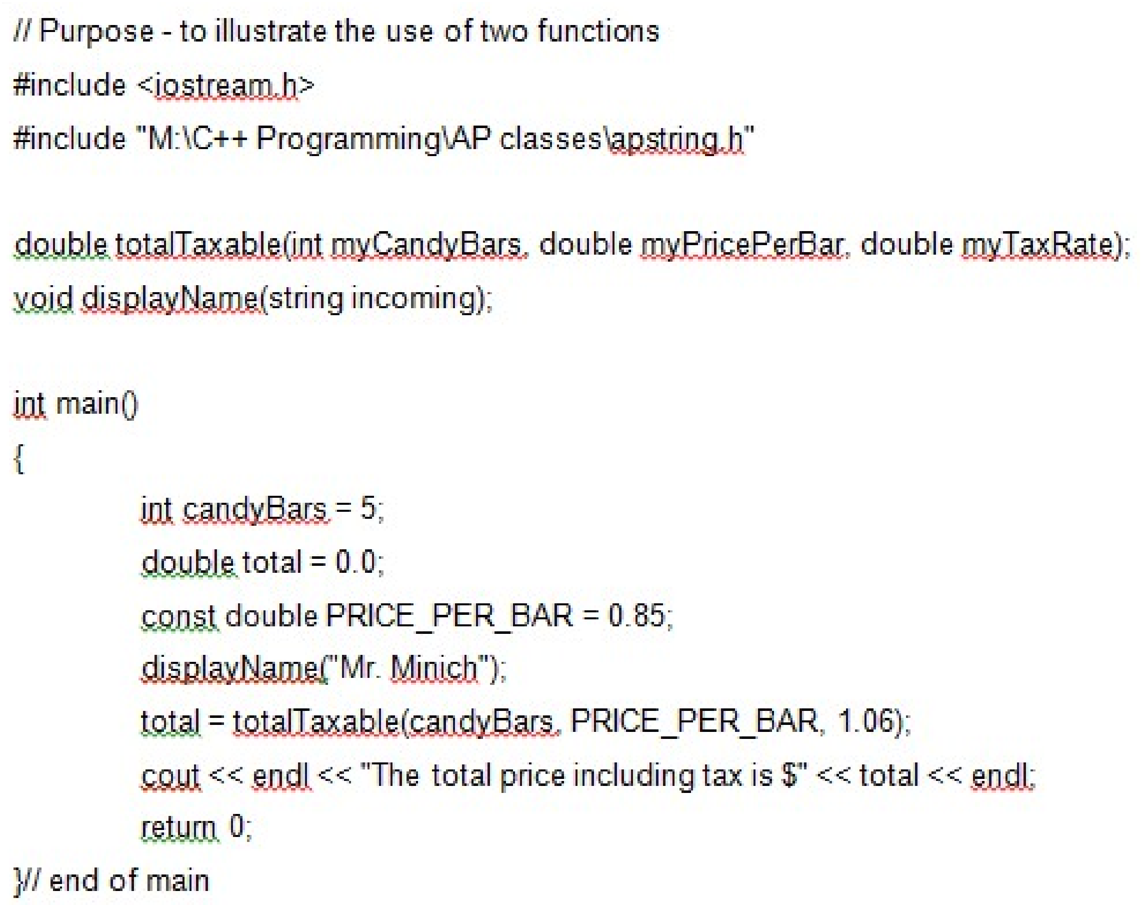
Example 7 (cont…)



```
bool changeToOnes(int & myOnes, int & myTens)
{
        myOnes = myTens * 10 + myOnes;
        myTens = 0;

        if (myOnes > 0)
        {
                return true;
        }
        else
        {
                return false;
        }

}// end of changeToOnes
```

Passing by value to a function

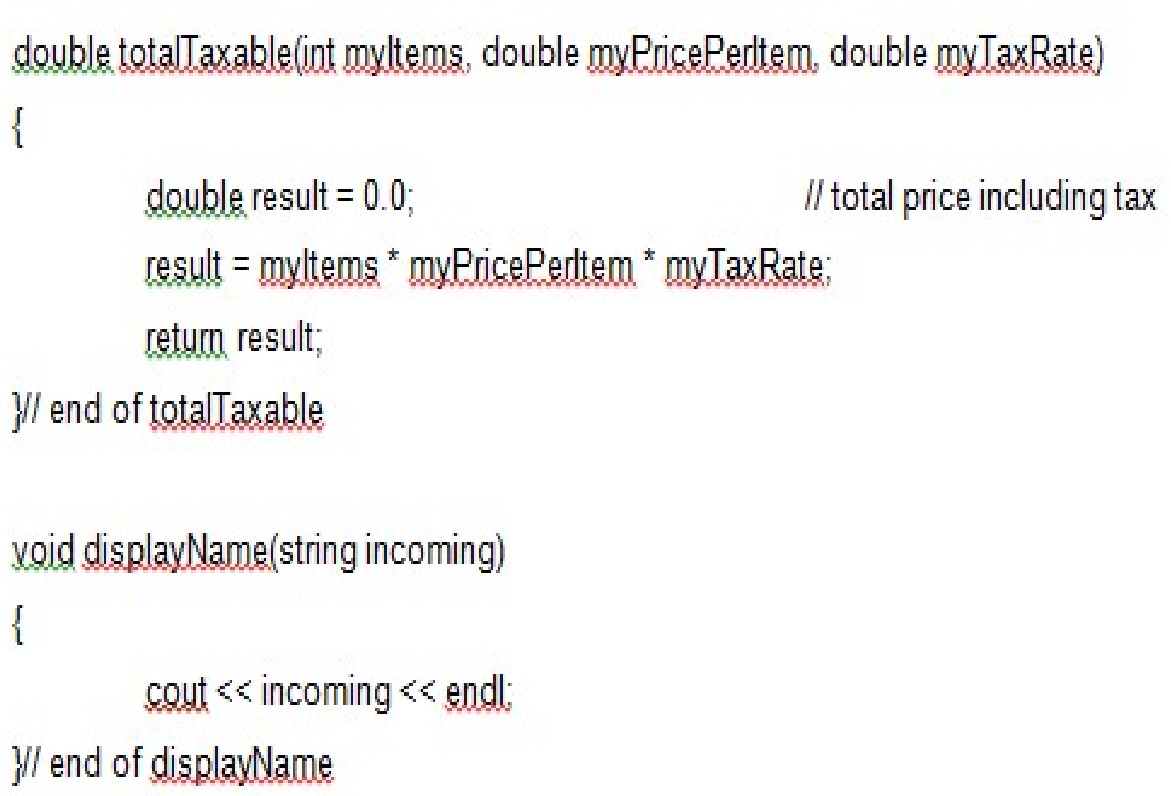
Example 8:



```
// Purpose - to illustrate the use of two functions
#include <iostream.h>
#include "M:\C++ Programming\AP classes\apstring.h"

double totalTaxable(int myCandyBars, double myPricePerBar, double myTaxRate);
void displayName(string incoming);

int main()
{
            int candyBars = 5;
            double total = 0.0;
            const double PRICE_PER_BAR = 0.85;
            displayName("Mr. Minich");
            total = totalTaxable(candyBars, PRICE_PER_BAR, 1.06);
            cout << endl << "The total price including tax is $" << total << endl;
            return 0;
}// end of main
```

Passing by value to a function

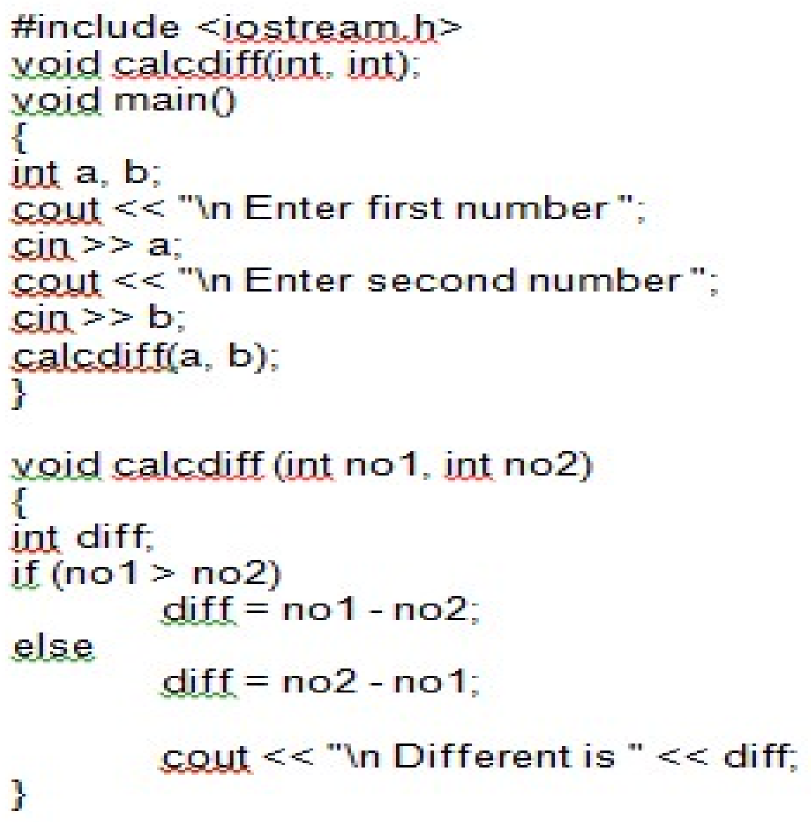
Example 8 (cont…)



```
double totalTaxable(int myItems, double myPricePerItem, double myTaxRate)
{
        double result = 0.0;                          // total price including tax
        result = myItems * myPricePerItem * myTaxRate;
        return result;
}// end of totalTaxable

void displayName(string incoming)
{
        cout << incoming << endl;
}// end of displayName
```

Function with parameter but without return value

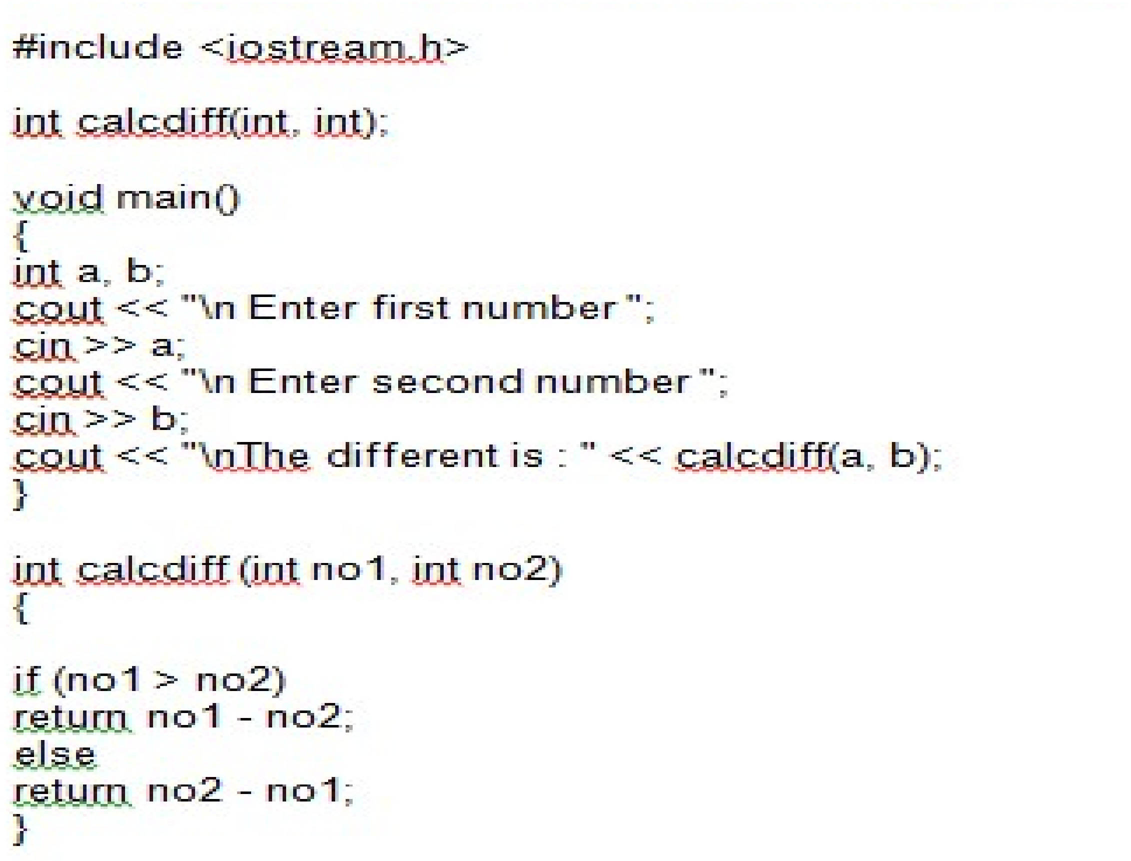
Example 9:



```
#include <iostream.h>
void calcdiff(int, int);
void main()
{
int a, b;
cout << "\n Enter first number ";
cin >> a;
cout << "\n Enter second number ";
cin >> b;
calcdiff(a, b);
}

void calcdiff (int no1, int no2)
{
int diff;
if (no1 > no2)
        diff = no1 - no2;
else
        diff = no2 - no1;

        cout << "\n Different is " << diff;
}
```

Function which return the value and parameter

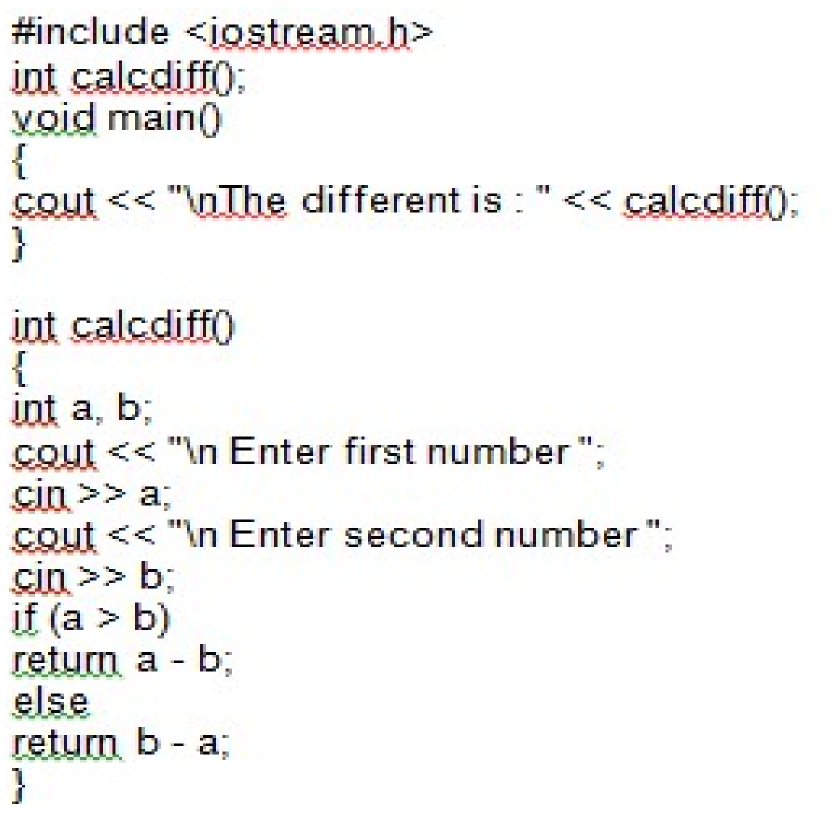
Example 10:



```
#include <iostream.h>

int calcdiff(int, int);

void main()
{
int a, b;
cout << "\n Enter first number ";
cin >> a;
cout << "\n Enter second number ";
cin >> b;
cout << "\nThe different is : " << calcdiff(a, b);
}

int calcdiff (int no1, int no2)
{

if (no1 > no2)
return no1 - no2;
else
return no2 - no1;
}
```

Function which return the value but without parameter

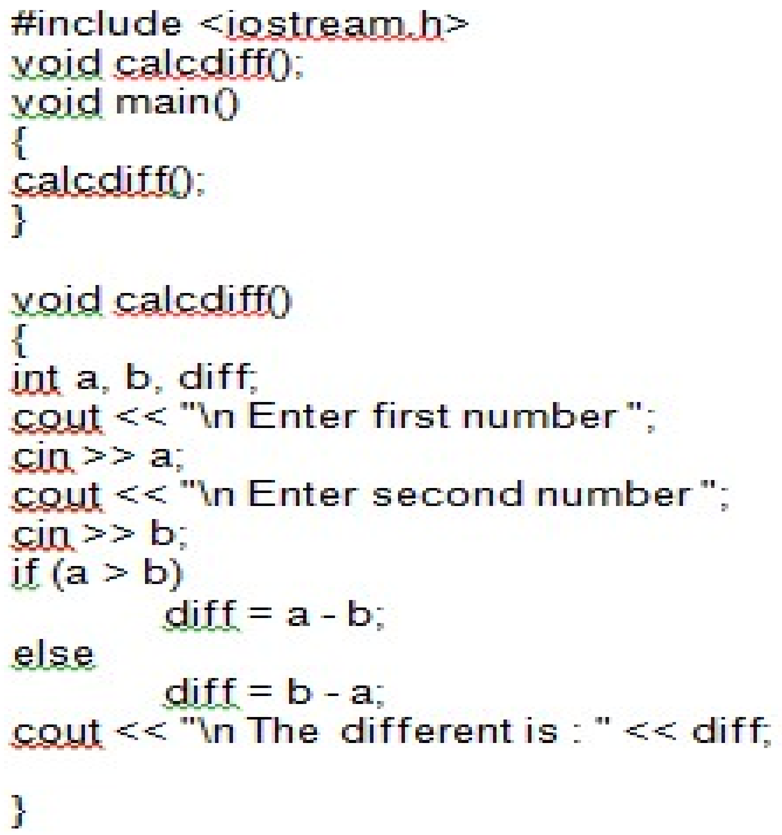
Example 11:



```
#include <iostream.h>
int calcdiff();
void main()
{
cout << "\nThe different is : " << calcdiff();
}

int calcdiff()
{
int a, b;
cout << "\n Enter first number ";
cin >> a;
cout << "\n Enter second number ";
cin >> b;
if (a > b)
return a - b;
else
return b - a;
}
```

Function without return value and parameter (void function)

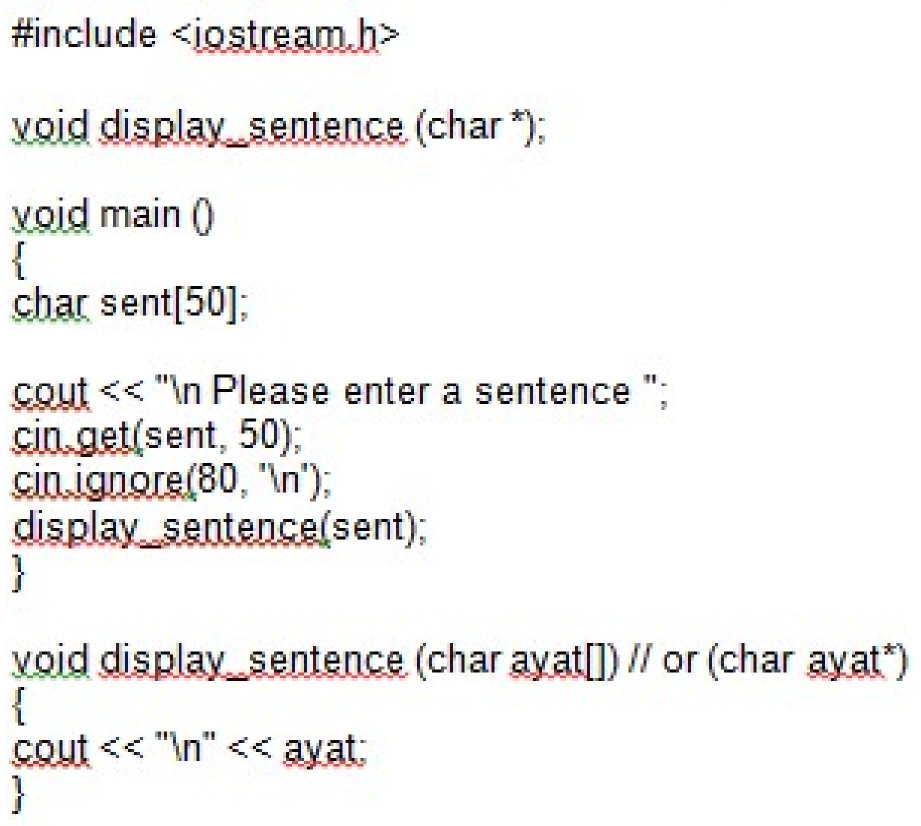
Example 12:



```
#include <iostream.h>
void calcdiff();
void main()
{
calcdiff();
}

void calcdiff()
{
int a, b, diff;
cout << "\n Enter first number ";
cin >> a;
cout << "\n Enter second number ";
cin >> b;
if (a > b)
          diff = a - b;
else
          diff = b - a;
cout << "\n The  different is : " << diff;

}
```

Function which receive a sentence from the main program

Example 13:



```
#include <iostream.h>

void display_sentence (char *);

void main ()
{
char sent[50];

cout << "\n Please enter a sentence ";
cin.get(sent, 50);
cin.ignore(80, '\n');
display_sentence(sent);
}

void display_sentence (char ayat[]) // or (char ayat*)
{
cout << "\n" << ayat;
}
```