

# **Chapter 7**

## **Functions**

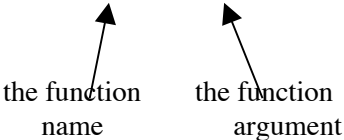
## Functions

Modules in C++ are referred to as functions. The use of functions in a program allows

- a program to be broken into small tasks.
- a program to be written and debugged one small part at a time.
- several programmers to work on the same program.

Functions make a program much easier to read, test and debug. In C++ there are two basic types of functions, ones that do not return a value and those that do return a value.

```
theNum = 9.0;  
theSquareRoot = sqrt(theNum);
```



the function name      the function argument

The sqrt function is a value returning function, specifically the square root of the argument of the function. The main( ) function in a C++ program is typically a non value returning function (a void function). A void function does not return a value to its caller. Functions are subprograms that do not execute until they are *called*.

## Value Returning Functions

Value returning functions are used when only one result is returned and that result is used directly in an expression. Some of the value returning functions in the C++ library are sqrt, fabs, log, pow etc.

Declaration of a value returning function

GENERAL FORM:

```
DataType FunctionName (FormalParameterList)
```

```
{
```

```
    Statement;
```

```
    .
```

```
    .
```

```
}
```

```
// returns the sum of two integers n1 & n2
```

```
int CalcSum(int n1, int n2)
```

```
{
```

```
    return n1 + n2;
```

```
}
```

```
// returns true if code is a 'T' or an 'H' and false otherwise
```

```
bool CheckCode(char code)
```

```
{
```

```
    return (code == 'T' || code == 'H');
```

```
}
```

Many programmers prefer to see the main( ) function as the first function in the program. It is necessary for the compiler to know about a function before it is called in a program. The method used for letting the compiler know about the function without including the actual code for the function is via a *function prototype*. Function declarations may be either function prototypes or function definitions.

### FUNCTION DECLARATIONS

**Function prototype** - a function declaration that does not include the body of the function.

**Function definition** – a function declaration that includes the body of the function.

```
// function prototype
int CalcSum(int n1, int n2);

void main(void)
{
    int firstNum;
    int secondNum;
    int theSum;

    cout << "Enter two integers: ";
    cin >> firstNum >> secondNum;
    // call the function in an assignment expression
    theSum = CalcSum(firstNum, secondNum);
    // call the function in a cout sending constants as arguments
    cout << CalcSum(5,8);
    // call the function in a cout sending expressions as arguments
    cout << CalcSum(2 * 3 + 4, 8 - 12);
}

// function definition
int CalcSum(int n1, int n2)
{
    return n1 + n2;
}
```

## Exercises

1. Write a value returning function that receives three integers and returns the largest of the three. Assume the integers are not equal to one another.
2. Write a value returning function that receives two floating point numbers and returns true if the first formal parameter is greater than the second.
3. Write a value returning function that receives a character and returns true if the character is a vowel and false otherwise. For this example, vowels include the characters 'a', 'e', 'i', 'o', and 'u'.

## Void Functions

A void function is used when

- input/output is to be performed
- more than one value is to be returned

A void function returns values by modifying one or more parameters rather than using a return statement. A void function is called by using the function name and the argument list as a statement in the program.

**Function Call (to a void function)** A statement that transfers control to a void function. In C++ this statement is the name of the function, followed by a list of actual parameters.

GENERAL FORM:

FunctionName ( ArgumentList);

```

#include <iostream>
using namespace std;

void PrintMessage();

void main(void)
{
    char name[10];

    cout << "What is your first name? ";
    cin >> name;
    cout << "\n\nHello " << name << endl << endl;

    // Here is the function call
    PrintMessage();
}

// Following is the function definition - function heading and code
void PrintMessage()
{
    cout << "*****\n";
    cout << "  Welcome to my\n";
    cout << " wonderful program!\n";
    cout << "*****\n";
}

```

## Function prototype (for a void function)

GENERAL FORM:

```
void FunctionName( FormalParameterList );
```

Function definitions must contain two parts – the function heading and the function body.

## Function definition (for a void function)

GENERAL FORM:

```
void FunctionName( FormalParameterList )
{
    statement
    .
    .
}
```

The function PrintMessage from the previous example is what is called a parameterless function. This means that the function requires no input values to execute. Let's look at a function that requires parameters.

```
#include <iostream.h>

// Function prototype
void AddAndPrint(int n1, int n2);

void main(void)
{
    // A call to the function sending the actual parameters
    // In this case the parameters are literal constants of type int
```

```
AddAndPrint(3,5);

cout << "\n\nBack in the main";
}

// Function definition
void AddAndPrint(int n1, int n2)
{
    cout << n1 << " + " << n2 << " is equal to\n\n";
    cout << n1 + n2;
}
```

OUTPUT:

3 + 5 is equal to

8

Back in the main

This program would be more useful if we allowed the user to enter the values to add and print. In the above example, the parameters were literal constants, in the next example, the parameters will be variables.



```

#include <iostream.h>

// Function prototype
void AddAndPrint(int n1, int n2);

void main(void)
{
    int firstNum;    // LOCAL variable for first integer - INPUT
    int secondNum;   // LOCAL variable for second integer - INPUT

    // Get numbers to add from user
    cout << "Enter two integers ";
    cin >> firstNum >> secondNum;

    // Function call sending LOCAL variables
    AddAndPrint(firstNum, secondNum);

    cout << "\n\nBack in the main";
}

// Function definition
void AddAndPrint(int n1, int n2)
{
    cout << endl << n1 << " + " << n2 << " is equal to\n\n";
    cout << n1 + n2;
}

```

OUTPUT:

Enter two integers 21 4

21 + 4 is equal to

25

Back in the main

**Local Variable** – A variable declared within a block and not accessible outside of that block. Local variables are in memory only as long as the function is executing. When the function is called, memory space is created for its local variables. When the function ends (returns) all of its local variables are destroyed.

**Global Variable** – A variable declared outside of all the functions in a program. We will not use global variables in this class.

### **Naming Void Functions**

When you choose a name for a function keep in mind how the function will be called. The name will appear as a statement in the program so the name of the function should sound like a command or an instruction.

```
AddAndPrint(firstNum, secondNum);
```

```
PrintMessage( );
```

```
FindAndPrintSmallest(num1, num2, num3);
```

### **Header Files**

You have been using the `#include` preprocessor directive in your programs from the beginning. These header files specified in the `#include` directive are nothing more than a collection of functions provided by the language for your use.

## Void Functions Exercise

1. Flowchart a void function that receives three integer values and outputs the largest of the three unless they are all equal, in which case, it outputs a message stating they are equal.
2. Write the function prototype for this void function.
3. Write the function definition for this void function.
4. Write a main function that will obtain values for three local variables from the user. Call the function sending these values as its \_\_\_\_\_ , or actual parameters. Output an appropriate message when the program returns to the main function.

## Parameters

There are two types of parameters used in C++ functions, *value parameters* and *reference parameters*. When you send a parameter to a function and its value will NOT BE MODIFIED, use a value parameter. When the function is CHANGING the actual parameter, such as in an input function, use a reference parameter.

**Value Parameter** - A formal parameter that receives a copy of the contents of the corresponding argument (actual parameter).

**Reference Parameter** - A formal parameter that receives the address (location in memory) of the corresponding argument (actual parameter). A reference parameter is noted with an ampersand (&) following the data type of the formal parameter.

### A few notes:

- The ampersand appears ONLY in the function heading, it does not appear in the call.
- Value parameters are coming *in* to the function and reference parameters are going *out* of the function. Some parameters may be coming in with a valid value and going out with a new one. These are said to be *inout* parameters.
- The parameter list in the function definition will show only ONE parameter per line with adequate documentation. Adequate documentation includes a description of what the parameter is used for and whether it is an *in*, *out*, or *inout* parameter.

**Communication:** Communication between the function and the caller is done via the parameter lists. The formal parameters serve as place holders for the actual values (arguments or actual parameters) the caller wishes to use. The formal parameters and the arguments are matched by their relative position in the lists. If the formal parameter is a reference parameter, the parameter receives the address of the argument. If the formal parameter is a value parameter, the parameter receives a copy of the argument.

```
void GetTwoAges(int& age1, int& age2);
```

```
void main(void)
```

```
{
```

```
    int firstAge, secondAge;
```

```
    GetTwoAges(firstAge, secondAge);
```

```
}
```

```
void GetTwoAges(int& age1, int& age2)
```

```
{
```

```
    cin >> age1 >> age2;
```

```
}
```

reference parameters (receive the addresses of firstAge and secondAge)



```

// This program demonstrates the use of value and reference parameters
#include <iostream>
using namespace std;

// This function RECEIVES 2 integers and RETURNS their sum
void CalcSum(int n1, int n2, int& sum);

void main(void)
{
    int firstVal;    // first value to add - INPUT
    int secondVal;   // second value to add - INPUT
    int sum;         // sum of the two values - CALC & OUTPUT

    // Get values from user
    cout << "Enter first value to sum: ";
    cin >> firstVal;
    cout << "Enter second value to sum: ";
    cin >> secondVal;

    // Call function to sum the values
    CalcSum(firstVal, secondVal, sum);

    // Output the sum
    cout << "\n\The sum of the numbers is " << sum << endl;
}

void CalcSum( int n1, int n2, int& sum)
{
    sum = n1 + n2;
}

```

OUTPUT:

```

Enter first value to sum: 23
Enter second value to sum: 21

The sum of the numbers is 44

```

## A FEW POINTS TO REMEMBER

- Value parameters are viewed as constants by the function and should never appear on the left side of an assignment statement or in a *cin* statement.
- Reference parameters are modified by the function and should appear either on the left side of an assignment statement or in a *cin* statement.
- Formal and actual parameters are matched according to their relative positions.
- Arguments (actual parameters) appear in the function call and do not include their data type.
- Formal parameters appear in the function heading and include their data type.
- When the formal parameter is a value parameter, the argument (actual parameter) may be a variable, named or literal constant, or expression. Note that type coercion may take place.
- When the formal parameter is a reference parameter, the argument (actual parameter) **MUST** be a variable of exactly the same data type as the formal parameter.

**THIS PAGE WOULD MAKE A GREAT QUIZ.**

```

// This program calculates a new pay rate for an employee
#include <iostream.h>

// Function prototype for CalcNewRate
void CalcNewRate (float perInc,      // % increase - IN
                  float& rtInc,      // rate increase - OUT
                  float& hrlyRt);    // new hourly rate - INOUT

void main(void)
{
    float percentIncrease;    // % of pay increase - INPUT
    float rateIncrease;       // amount of increase - CALC & OUTPUT
    float hrlyRate;           // current pay rate - INPUT, CALC & OUTPUT

    // Obtain current employee rate info
    cout << "Enter your current hourly pay rate: ";
    cin >> hrlyRate;
    cout << "Enter the percent increase you received: ";
    cin >> percentIncrease;

    // Call function to calculate the new rate
    CalcNewRate (percentIncrease, rateIncrease, hrlyRate);

    // Output the increase & new hourly rate - a "real" program would have formatted output
    cout << "\nYour increase per hour is " << rateIncrease;
    cout << "\nYour new hourly pay rate is " << hrlyRate;
}

void CalcNewRate (float perInc, float& rtInc, float& hrlyRt)
{
    rtInc = hrlyRt * perInc/100.0;
    hrlyRt = hrlyRt + rtInc;
}

```

OUTPUT:

```

Enter your current hourly pay rate: 8.50
Enter the percent increase you received: 3.5

```

```

Your increase per hour is 0.2975
Your new hourly pay rate is 8.7975

```



```

// This program demonstrates the use of value & reference parameters
// and the use of a value returning function
#include <iostream.h>

// Function prototype for CalcNewRate as a value returning function
float CalcNewRate (float perInc,      // % increase - IN
                  float hrlyRt);     // new hourly rate - IN

void main(void)
{
    float percentIncrease;    // % of pay increase - INPUT
    float rateIncrease;       // amount of increase - CALC & OUTPUT
    float hrlyRate;           // current pay rate - INPUT, CALC & OUTPUT

    // Obtain current employee rate info
    cout << "Enter your current hourly pay rate: ";
    cin >> hrlyRate;
    cout << "Enter the percent increase you received: ";
    cin >> percentIncrease;

    // Call function to calculate the new rate - use function name as part of an expression
    rateIncrease = CalcNewRate(percentIncrease, hrlyRate);
    // Calc new hourly rate
    hrlyRate = hrlyRate + rateIncrease;

    // Output the increase & new hourly rate - a "real" program would have formatted output
    cout << "\nYour increase per hour is " << rateIncrease;
    cout << "\nYour new hourly pay rate is " << hrlyRate;
}

// Function definition for a value returning function
float CalcNewRate (float perInc, float hrlyRt)
{
    return hrlyRt * perInc/100.0;
}

```

## Value & Reference Parameter Exercise

1. Flowchart a function that obtains from the user and returns a value for unitCst, units, and taxRt to the calling module. Choose an appropriate name for this function.
2. Flowchart a function that receives the unitCst, units, and taxRt and returns to the calling module the salesTx and totDue. Choose an appropriate name for this function.
3. Flowchart a function that receives the unitCst, units, taxRt, salesTx, and totDue and outputs them in a very readable format. Choose an appropriate name for this function.
4. Write a main function with variables called unitCost, unitsPurch, taxRate, salesTax, and totalDue. Your program should call the functions listed above.
5. Write the well documented code for this program demonstrating all of the necessary documentation and style for CS 1B.

## CS 1B Functions Errors Exercise

Correct the errors in the following program:

```
void GetLoanInfo(float& amt, float& rt, int& term);
float CalcPayment(float amt, float rt, int term, float& pmt);
void OutputLoanInfo(float& amt, float& rt, int& term, float& pmt);

void main(void)
{
    float prinAmt, intRate, monPmt;
    int termOfLoan;

    GetLoanInfo(float amt, float rt, int term);
    CalcPayment(prinAmt, intRate/1200, termOfLoan*12, monPmt);
    OutputLoanInfo(prinAmt, monPmt, intRate, termOfLoan);
}

void GetLoanInfo(float amt, float rt, int term)
{
    cout << "Enter loan amt, interest rate, and term: ";
    cin >> prinAmt >> rt >> termOfLoan;
}

float CalcPayment(float amt, float rt, int term, float& pmt)
{
    pmt = (rt * pow(1 + rt , term)) / (pow(1 + rt, term) - 1) * amt;
}

void OutputLoanInfo(float& amt, float& rt, int& term, float& pmt)
{
    cout << "\n\nThe monthly payment on " << amt << " at " << rt << "% interest is\n"
        << pmt << endl;
}
```

## The Scope of an Identifier

The scope of an identifier deals with when the identifier is active and which parts of the program can see it. The scope of an identifier begins with its most recent definition and continues until the end of the associated block.

**LOCAL SCOPE** - The scope of identifiers declared inside a block (they exist from the time of the declaration until the end of the block).

**GLOBAL SCOPE** - The scope of an identifier that is declared outside of any function (they exist from the time of declaration until the end of the program and may be seen by any function). In general, you will see very few, if any, in well written programs.

C++ function names have global scope (later you will find an exception to this).

A function may not contain another function within it (in other words, you cannot "nest" functions in C++).

**NAME PRECEDENCE or NAME HIDING** - If a function declares a local identifier that is the same name as a global identifier, the local identifier takes precedence inside the function.

The **LIFETIME** of an identifier is the period of time during program execution that the identifier has memory assigned to it.

**STATIC VARIABLE** - A variable that has memory allocated to it for the entire execution of the program.

**AUTOMATIC VARIABLE** - A variable that has memory allocated and deallocated as the block is entered and exited.

**BE PREPARED TO SHOW EXAMPLES OF THE ABOVE**

## Scope Exercises

```
#include <iostream.h>
void BlockOne (int n1, int n2, int n3);
void BlockTwo (int n1, int n2, int n4);
void main(void)
{
    int num1, num2, num3;

    num1=num2=num3=2;
    BlockOne(num1, num2, num3);
    cout << num1 << num2 << num3;
}

void BlockOne( int n1, int n2, int n3)
{
    cout << n1 << n2 << n3 << endl;
    n1 += 5;
    cout << n1 << n2 << n3 << endl;
    BlockTwo(n1, n2, n3);
}

void BlockTwo (int n1, int n2, int n4)
{
    int n3, n5;
    n2 *= 4;
    cout << n1 << n2 << n4 << endl;
}
```

1. n3 and n5 are \_\_\_\_\_ to BlockTwo.
2. The statement BlockOne(num1,num2,num3) is the \_\_\_\_\_.
3. n1, n2 and n3 in BlockOne are the \_\_\_\_\_.
4. In main( ), num1, num2 and num3 are the \_\_\_\_\_ in the statement BlockOne(num1, num2, num3).
5. Show the output from this program.

```

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

int DoSomething(int n1);
void DoSomethingElse(int n1, int& n2);

void main(void)
{
    int num1, num2;
    num1 = DoSomething(4);
    num2 = 1;
    cout << num1 << setw(5) << num2 << endl;
    DoSomethingElse(num1, num2);
    cout << num1 << setw(5) << num2 << endl;
}

int DoSomething(int n1)
{
    return n1 * 2;
}

void DoSomethingElse(int n1, int& n2)
{
    n2 = n2 + n1;
    n1++;
    cout << n1 << setw(5) << n2 << endl;
    n2 = DoSomething(n1);
}

```

```

// Program TryAndConfuseTheStudents
// or program PayAttentionToDetail
#include <iostream.h>

void DoThisCarefully (int& n1, int& n2, int& n3, int& n4, int n5);
void main(void)
{
    const int VALUE = 5;
    int num1, num2, num3, num4;

    num1 = num2 = 5;
    num3 = 4;
    num4 = num1 + num3;

    cout << VALUE << ' ' << num1 << ' ' << num2 << ' ' << num3 << ' ' << num4 << ' '
         << endl;

    DoThisCarefully (num1, num3, num2, num4, VALUE);

    cout << VALUE << ' ' << num1 << ' ' << num2 << ' ' << num3 << ' ' << num4 << ' '
         << endl;
}

void DoThisCarefully ( int& n1, int& n2, int& n3, int& n4, int n5)
{
    int num1;

    num1 = 10;
    n1 += n2;
    n2 = num1 + n3;
    n3 *= n1;
    n4 = n5;
    n5 += 10;
}

```

Show the EXACT output from this program.

## Comparing Void & Value Returning Functions

**The call to a void function appears as a complete statement in the program. Void functions return values by modifying one or more parameters.**

**The call to a value returning function appears as part of an expression. Value returning functions return a single value to the expression and they do not modify parameters.**

Example of a void function:

```
// function definition
void CalcSum (int n1, int n2, int n3, int& sum)
{
    sum = n1 + n2 + n3;
}
```

```
// function call
CalcSum (5, 3, 7, theSum);
```

Example of a value returning function:

```
// function definition
int CalcSum(int n1, int n2, int n3)
{
    return n1 + n2 + n3;
}
```

```
// function call
theSum = CalcSum(5, 3, 7);
```



Convert the following void function to a value returning function.

```
void FindSmaller(int n1, int n2, int& small)
{
    if(n1 < n2)
        small = n1;
    else
        small = n2;
}
```

Convert the following value returning function to a void function.

```
bool CheckEqual(int n1, int n2)
{
    return (n1 == n2);
}
```

## Designing Functions

**Design the interface** - Specify *what* the function does and how it will be called. A collection of functions required for a particular application are typically stored in a separate file called the *specification* file.

- Does it perform any I/O? (Make it a void function.)
- Does it select or calculate a single value to return? (Make it a value returning function.)
- Select a name that indicates what the function does.
- How many pieces of information will be sent to the function?
- Do any of the formal parameters require previously defined values? Will their values be changed by the function? Label each formal parameter as *incoming*, *outgoing*, or *incoming/outgoing*. This is called documenting the **data flow**.
- Write the assertions (preconditions and postconditions). The preconditions and postconditions should be an accurate and concise description of what the function does without specifying how it is done. The user of the function is responsible for meeting the preconditions and your job is to meet the specified postconditions.
- DO NOT WORRY ABOUT THE PARTICULARS OF HOW THE FUNCTION WILL BE IMPLEMENTED. DESIGN ALL OF THE FUNCTION INTERFACES AND THE DRIVER PROGRAM BEFORE YOU START THINKING OF ANY DETAILS. THIS WILL HELP YOU DESIGN A MORE CLEAR AND CONCISE ALGORITHM.

**Design the driver** - Define the interaction between the main function and any other functions used in the program.

- List all variables, named constants, and their data types that will be declared in the main.
- Flowchart the main function and test to see that the flow of control is properly ordered and not missing any components.
- Test the functions and the main program by writing each of the functions as stubs. DO NOT CODE THE FUNCTIONS YET! If the main program is controlled with a loop, test that the entry and exit conditions work before you continue.

Once you are satisfied that you have specified a **clear, concise** set of functions and that their interaction with the main is well defined, you may continue to the implementation phase.

**Implement the Functions** - Write the actual code for the functions **ONE AT A TIME**.

- List any named constants and variables that need to be declared locally.
- Code the function. Is it more than about ten lines? Perhaps it could be broken into two functions.
- Check to make certain that the **ONLY** identifiers appearing in the body of the function are formal parameters or locally declared.
- Check each parameter and its use in the function (data flow).
  - IN**            Should NOT appear on the left side of an assignment statement or in a *cin*.
  - OUT**          Should appear on the left side of an assignment statement or in a *cin*.
  - IN/OUT**      Should appear on the left side of an assignment statement or in a *cin*.
- Check that value returning functions have only one return statement.
- Code the functions one at a time. Test your program leaving the other functions as stubs and code the next function only after the previous one has been successfully incorporated into the program. Adding functions one at a time and testing them will save hours of frustration debugging code. The most common errors are found in the input function - find these errors early! **ALWAYS** implement this function first and confirm that it works by echoing out the input values in the main. The problem is most often the failure to properly use reference parameters.
- These function definitions will typically be placed in a separate file called the *implementation* file. This is referred to as information hiding. The user only needs to know what, not how.

There is a function called *assert* that can be helpful when debugging your program and testing possible problems. The *assert* function is called by sending a valid C++ logical expression as an argument. The program halts and an error message is displayed if the assertion "fails" which is to say the logical expression is false. **This is useful as a debugging tool but is not acceptable in a finished project.**

```

#include <iostream>
#include <assert.h>
using namespace std;

float CalcAvg(int examCt,int scTot);

void main(void)
{
    int score;
    int scoreTot;
    int examCount;
    examCount = 0;
    scoreTot = 0;

    cin >> score;
    while(score != -999)
    {
        scoreTot = scoreTot + score;
        examCount++;
        cin >> score;
    }
    cout << "Average score is " << CalcAvg(examCount,scoreTot);
}

float CalcAvg(int examCt,int scTot)
{
    // call the assert function
    assert(examCt > 0);
    return float(scTot) / examCt;
}

```

Sample runs:

```

-999
Assertion (examCt > 0) failed in "1bconvers.cpp" on line 26

```

```

10
8
4
10
6
-999
Average score is 7.6

```

### Incorporating Error Checking Into Programs Exercise #1

1. Write a basic while loop that sums a stream of integers (input from the keyboard) and continues to do so while the integer entered is not -999. Ask the user to enter values in the range of 1 to 10000 or -999 to quit. Do not do any error checking in this version, assume a perfect user. Save this version as INTSUM.CPP
2. Write a do..while loop below to do the following:
  - read an integer from the keyboard in to a variable called **intVal**
  - output an error message if the number is not in the range 1 to 10000 or the value -999
  - continue looping until a valid value has been entered

Check your algorithm (by hand) with the following input values:

5	loop should end
-25	error message should print and loop should continue
100000	error message should print and loop should continue
-999	loop should end

3. When the algorithm above works, incorporate it in to the program from part 1. Replace the cout/cin combination used to read an integer before the loop and at the bottom of the loop with the thoroughly tested code produced in part 2. Test the program and save it as INTSUM2.CPP

Turn in (IN THIS ORDER)

- this sheet
- listing of INTSUM.CPP (proper style but undocumented)
- listing of INTSUM2.CPP (proper style but undocumented)
- **DO NOT SUBMIT A COPY TO THE MAILBOX**

## **Incorporating Error Checking In To Programs Using Functions Exercise #2**

Modify INTSUM2.CPP as follows and save it as INTSUM3.CPP.

- Place the following function prototype at the top of your program.  
void GetNum(int& num);
- Write the function definition below the main function by taking the code that checks for numbers in the range of 1 to 10000 or -999 and placing it inside the definition. Be sure to change any input variable references from **intVal** to parameter **num**.
- Replace the error checking code found before the while loop in the main and at the bottom of the while loop with calls to GetNum. Be sure the argument name is the name of the input variable (**intVal**) declared in the main. You will use this function to initialize and change LCV **intVal**.

A sample run should look at follows:

**This program sums a stream of integers in the range of 1 to 10000.**

**Enter an integer (1 - 10000 or -999 to exit): 15**

**Enter an integer (1 - 10000 or -999 to exit): -10**

**\*\*\* Invalid Input \*\*\***

**Enter an integer (1 - 10000 or -999 to exit): 500**

**Enter an integer (1 - 10000 or -999 to exit): 100000**

**\*\*\* Invalid Input \*\*\***

**Enter an integer (1 - 10000 or -999 to exit): 45**

**Enter an integer (1 - 10000 or -999 to exit): -999**

**The sum of the integers is: 560**

Turn in (IN THIS ORDER)

- This sheet
- Listing of the code (proper style but undocumented)
- Sample run as above (proper style but undocumented)
- **DO NOT SUBMIT THIS TO THE MAILBOX**

## Functions Flowcharting Exercise

Design a program that accepts numbers in the range of 2 - 2500 or -999. While the number is not -999 call a function that checks to see whether the number is prime or not. This function will return true if the number is prime and false otherwise. A prime number is an integer greater than or equal to 2 whose only divisors are 1 and the number itself. Hint: If  $n$  is not a prime number, it will be exactly divisible by an integer in the range 2 through the square root of  $n$ .

- Step 1: Write function prototypes for the input and check functions.
- Step 2: Make a list of variables and draw the flowchart for the main.
- Step 3: Flowchart the function that checks user input.
- Step 4: Flowchart the function that checks to see if the number is prime.
- Step 5: Document the function prototypes.

## A Few More Notes on Functions

Well documented function prototypes provide several pieces of information to the user of the function:

- the function name
- the number of arguments required
- the data types of the arguments
- the order of the arguments
- information about the state of the arguments before and after the call (pre and post conditions)

### Parameter Descriptions in the Function Heading (Data Flow)

IN	passed by value	value is defined prior to call & not changed
OUT	passed by reference	value is undefined prior to call & is changed
IN/OUT	passed by reference	value is defined prior to call & is changed

These function headings also represent sort of a contract between the function and the user of the function. When you used intrinsic functions such as *pow* and *sqrt* you had certain expectations about the state of the variables you sent to the function. Given two variables *x* and *y*, equal to 3 and 5 respectively, your expectation after a statement such as

```
cout << pow(x,y);
```

is that *x* will contain the value 3 and *y* will contain the value 5. If for some reason *x* came back with a value of 4, this would produce an undesired change to your arguments. This would be an example of an unwanted **side effect**.

**Side Effect** - A side effect is any effect that one function causes to another that is not part of the interface explicitly defined between them.



Consider the following:

```
void SampleFunc(int n1, int n2, int& n3)
{
    n3 = n1 * n2;
    n2++;
    count++;
}
```

Which statements, if any, would produce a side effect?

All side effects are not undesirable and unintentional. Consider the following:

```
value = 3 + 5 * (number = 4);
```

Not only is value assigned 23, the *side effect* of storing 4 to number has also occurred after this assignment statement has been executed. Not a desirable way to program but certainly (or hopefully) intended by the programmer.

Consider the following:

```
#include <iostream>
using namespace std;
void CountEm();
int counter;

// This program processes input lines and prints the number of characters per input line
// as well as the number of lines input.
void main(void)
{
    counter = 0;
    cout << "Enter several lines of input, 'q' to exit\n\n";
    while(cin.get() != 'q')
    {
        counter++;
        CountEm();
    }
    cout << "\n\nYou entered " << counter << " lines of input.\n";
}
void CountEm()
{
    char theCh;
    counter = 0;
    cin.get(theCh);
    while(theCh != '\n')
    {
        counter++;
        cin.get(theCh);
    }
    cout << "There are " << counter << " characters on this line.\n\n";
}
```