**Lab Experience Eleven**

**Objectives:**

1. Understanding the differences between pass by reference and pass by value
2. Creating a program using functions.

**Background**

Scope is a term used to define the area in the program where the variable is known and can be used. The types of scope are block, local, and global. Global scope should be avoided except when passing large amounts of data between functions. I.e. when a two dimensional array is used to represent the palette in a graphics toolbox program.

Function stubbing is used to test each function separately before writing the entire program. A common mistake of most beginning programmers is to write the entire program and then debug it. If a logic error occurred in the program, the programmer debugging the program is literally “looking for the needle in the haystack” because the program could be terminating abnormally in the first function, but the real cause of the error could have been caused in the fourth function not performing the correct task.

Arrays in Computer Science are data structures used to hold large amounts of the same type of data instead of using multiple variables. An array represents consecutive memory locations referenced by the same name with an offset into memory which is represented by the subscript.

**Scope Rules**

If two variables with the same name are used within a program which one does the program use? This is decided by the scoping rules.

Block scope is defined by the block where the variable has been declared. Block scope is also referred to as local scope and local scope takes precedence over global scope.

Global scope is defined from the point of declaration in the program down. Globally defined variables are placed outside any function and can be used by any functions defined below the declaration.

Because of the scoping rules, multiple references to the same name can occur, but each reference could be referring to a different address in memory. Which one is used is dependent on where the declaration of the variable is placed within the program or by the absence of a declaration.

If a function references a variable that has not been defined within the function, either through the parameter list or by a variable declaration, the function looks “outside” its definition to locate the variable. If the variable has been defined globally, the function will use the globally defined variable, if it hasn’t been defined globally, a syntax error will be generated with the message being an undeclared identifier.

It is possible to have two variables with the same name, one defined globally and one defined locally. When this occurs the variable defined locally will be used instead of the globally defined variable.

**Example:**

**#include <iostream>**

**using namespace std;**

The boxes represent the scoping rules. The outside box represents the global scope of the variable x defined above funstuff(). The inside boxes represent the local scope of each function.

Since x is declared in the function main, all references of x in main will use the local declaration.

Since x is not defined in the function funstuff the globally defined variable x will have its value changed. This could be by accident thus the program could produce what is called a side-effect. A side-effect means something has changed and this change could have an adverse effect on the results being generated by the program.

**int x = 5;**

**doublefunstuff();**

**int main(){**

**int x = 3;**

**// more statements**

**return 0;**

**}**

**doublefunstuff(){**

**x = 10;**

**// more statements**

**}**

**Creating Programmer-defined Functions**

When creating a function, the programmer must write the definition. The definition consists of four parts:

1. Return-type: The function can return any valid C++ data type or return nothing. If the function returns nothing the return type will use the keyword **void**.
2. Name: A unique identifier that is descriptive of what the function will accomplish.
3. Parameter line: Used to pass data to the function. If the parameter list is **void** means the function is self-contained and does not need any external information to accomplish its task.
4. Body: The C++ statements necessary to accomplish the task specified for the function.

A function has the following syntax:

return-valuefunctionName(datatype param-1, datatype param-2, …,datatypeparam-n)

Programmer defined identifier following the same rules as naming variables.

Specifies the type of information returned by the function via a return statement.

Parameters are variables defined by the programmer that will be used within the function to solve the specified task. Data will be passed to the function via the parameter list. Each parameter used must be preceded by the data type of the parameter. This is used by the function for syntax checking. The number of parameters used by the function can range from zero to n.

**Examples:**

int main(){ // is a function with a return value of int and zero parameters.

intstrcmp(char \*array1, char \*array2) // is a function with a return value of int and two parameters.

When a function has a return type of void, a return statement is not needed. If a return statement is desired, place **return;** before the right curly brace**}.**

**voiddisplayRetail(double retailPrice){// header**

**// Some C++ statements;**

**}**

**Calling a function**

Before a function is called it must be defined. It was a common practice to define all functions before main, but this is now discouraged. Instead a prototype of the function is placed before the function main with the same characteristics of the function header and the function definition is placed after main. The purpose of the prototype is define the function name and to provide syntax checking between the calling statement and the function definition.

**Example:**

**voiddisplayRetail(double); // prototype note semicolon and only data types**

**// in the parameter list**

**int main(){**

**double price;**

**// some more C++ statements**

**displayRetail(price); // call to the function, note no return type or**

**// data type in front of price.**

**return 0;**

**}// end main**

**voiddisplayRetail(double retailPrice){ // function header and definition.**

**cout<< fixed <<showpoint<<setprecision(2);**

**cout<< “The retail price of the item is $ <<retailPrice<<endl;**

**}**

What happens when the program executes? The value contained in **price** is copied to the variable **retailPrice** and execution is transferred to the function **displayRetail**. Execution is transferred back to the statement following **displayRetail** when the function terminates. The function will terminate when a return statement or a right curly brace is reached.

The variable **price** is commonly referred to as the **actual parameter** (also called argument) to the function and **retailPrice** is called the **formal parameter** since **retailPrice** is used within the body of the function.

The process of copying the contents of the actual parameter to a formal parameter is called **pass by value** or **call by value**. Since **retailPrice** contains a copy of **price**, any changes made to **retailPrice** will not change the value contained in price.

When using function there are essentially three parts that the programmer needs to complete.

1. The heading must have both data type and variable names for all its formal parameters.
2. The prototype must be placed before main and must have the data types. The parameters names can be included within the prototype, but they are ignored.
3. The call must have the name of the function (not preceded by the return type), but must not have the data type for its actual parameters.

**Pass by value**

When a function utilizes **pass by value** the following steps occur when the function is invoked or called:

1. The formal parameter is allocated memory with a unique address. This means a formal parameter and an actual parameter can have the same name, but each will have its own memory address.
2. Type coercion will occur if the data types of the actual parameter and the formal parameter are not the same. I.e. an int converted to double and vice versa.
3. The contents of the actual parameter are copied into the formal parameter thus preventing any accidental changes to the contents of the actual parameter.
4. When the function terminates, the formal parameters memory location is reallocated back to the operating system and is no longer available to the program.

**Pass by reference**

A function can only return a single value with the return statement. What method is used when more than one value needs to be returned to the calling function? pass by reference.

**Example:**

**voiddisplayRetail(double); // prototype note semicolon and only data types**

**// in the parameter list.**

**voidgetData(double &, double &); // prototype Note the &**

**int main(){**

**doublewholesaleCost, pctMarkup;**

**double price;**

**getData(wholesaleCost, pctMarkup); call the function**

**// more C++ statements**

**displayRetail(price); // call to the function, note no return type or**

**// data type in front of price.**

**return 0;**

**}// end main**

**voiddisplayRetail(double retailPrice){ // function header and definition.**

**cout<< fixed <<showpoint<<setprecision(2);**

**cout<< “The retail price of the item is $ <<retailPrice<<endl;**

**}// end displayRetail**

**voidgetData(double &wholesale, double &markup){**

**cout<< “Enter the wholesale cost of the item ==>”;**

**cin>> wholesale;**

**cout<< “\nEnter the percent markup of the item ==>”;**

**cin>> markup;**

**}// end getData**

The & (ampersand) character is used in C++ to differentiate between call by value and call by reference. The formal parameter uses the same memory location as the actual parameter whenever call by reference is used. This means any changes to the formal parameter in the function will change the value of the actual parameter.

When a function utilizes **pass by reference** the following steps occur when the function is invoked or called:

1. The **address** of the actual parameter’s memory location is passed to the formal parameter. This means any changes made within the function to the formal parameter will change the contents of the actual parameter.
2. Type coercion will **NOT** occur if the data types of the actual parameter and the formal parameter are not the same. **The data types of the formal and actual parameters must be identical**.

For example consider the following code snippets from the program on the previous page. The address of **wholesaleCost** is passed to the formal parameter **wholesale** and any changes made to **wholesale** within the function **getData** will change the contents of **wholesaleCost.** The same is true for**pctMarkup**and **markup**.

**getData(wholesaleCost, pctMarkup); call the function**

**voidgetData(double &wholesale, double &markup){**

**cout<< “Enter the wholesale cost of the item ==>”;**

**cin>> wholesale;**

**cout<< “\nEnter the percent markup of the item ==>”;**

**cin>> markup;**

**}// end getData**

**Returning a Value From a Function**

When a function returns a value the function call must be used in an expression, appear on the right hand side of an assignment statement, or in a **cout** statement. If none of these are used the returned value is discarded and execution will continue.

**voiddisplayRetail(double); // prototype note semicolon and only data types**

**// in the parameter list.**

**voidgetData(double &, double &); // prototype Note the &**

**doublecalculateRetailPrice(double, double); //prototype note the return value is double**

**int main(){**

**doublewholesaleCost, pctMarkup;**

**double price;**

**getData(wholesaleCost, pctMarkup); //call the function**

**price = calculateRetailPrice(wholesaleCost, pctMarkup); // place the returned value**

**// in the variable price**

**displayRetail(price); // call to the function, note no return type or**

**// data type in front of price.**

**return 0;**

**}// end main**

**voiddisplayRetail(double retailPrice){ // function header and definition.**

**cout<< fixed <<showpoint<<setprecision(2);**

**cout<< “The retail price of the item is $ <<retailPrice<<endl;**

**}// end displayRetail**

**voidgetData(double &wholesale, double &markup){**

**cout<< “Enter the wholesale cost of the item ==>”;**

**cin>> wholesale;**

**cout<< “\nEnter the percent markup of the item ==>”;**

**cin>> markup;**

**}// end getData**

**doublecalculateRetailPrice(double wholesale, double markup){**

**double retail = wholesale \* (1 + markup/100.0); // the variable retail**

**return retail; // is called a local variable**

**// and is only known within this**

**// function**

**}// end calculateRetailPrice**

The above program is the completed problem posed at the beginning of the functions section. Make sure you understand how information is passed between functions and how a problem can be reduced to a series of sub problems.

**Lab Exercises**

**Directions:**

Start Microsoft word and record the questions and answers to all of the exercises in the lab word document   
Answer the following questions based on material presented in lecture and found in chapters 1-6 of the textbook.

1. Given the function prototype void fun1(int&, double); is the following function call valid or invalid? Why or why not?

double x, y;

fun1(x, y);

**No, type coercion will not occur when the data types are not the same in a pass by reference.**

1. Given the function prototype void fun1(int&, double); is the following function call valid or invalid? Why or why not?

double x, y;

fun1(static\_cast<int>(x), 3);

**No, because the pass by reference is passing a memory location, it cannot be casted.**

1. Given the function prototype void fun1(int&, double); is the following function call valid or invalid? Why or why not?

int x, y;

fun1(x, y);

**Yes, the y can be typed coerced into a double because of pass by value.**

1. Given the function prototype void fun1(int&, double); is the following function call valid or invalid? Why or why not?

int x, y;

double z;

fun1(x + y, z);

**No, the expression is evaluated first, which will become a constant. For example, if x = 2, and y = 3, the function will try to pass the constant 5, but it cannot because it does not have a memory address.**

1. Given the function prototype doublefun2(int , double); is the following function call valid or invalid? Why or why not?

double x, y;

fun2(x, fun2(x, y));

**Yes, since fun2 returns a double, the fun2 that is nested inside will return a double that can be used in the outer fun2.**

1. A prime number is an integer greater than 1 that can only be divided evenly by itself or 1. Examples of prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ... Numbers that are not prime are: Any even number, 9 since it is divisible by 3, multiples of 5, etc. **Note: You must use several functions in the solution of this problem. Please note the number one is not a prime number by definition.**  
   Write a program that will accept from the user a positive integer (make sure you validate the number is positive) and determine if the number entered by the user is a prime number. Notify the user if the number is prime or it is not prime. Your program should also display the number entered by the user.   
     
   The naive or brute-force approach is to check if it is divisible by any number between 1 and itself. Instead use the following algorithm to determine if a number is prime or not.
2. If the number is even and greater than 2, it is not prime.
3. Moreover, you should only test the number against integers less than or equal to the square root of its own value.

To find all kinds of interesting facts about prime numbers and the largest prime number found so far, just [click here](http://www.utm.edu/research/primes/largest.html).

Copy and paste your source code into a word document with several screen shots of all possible case scenarios. I.e. Error checking for a negative number, prime numbers of 2 and 3, and 29, a non-prime number.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

// Programmer: Chris Dang Class: CSCI 1106 Fall 2014

//

// Description: This program will check if a user entered number is a prime

// number or not. The program will then display the number entered as well

// as whether the number is prime or not.

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

#include <string>

#include <cmath> // needed for sqrt

using namespace std;

const string pleaseEnter = "Please enter a positive integer: " ;

const string header = "This program will take in a positive integer and "

"determine if it is prime.\n" ;

bool valPos (int num) ; // validates if num is postive

bool numIsEven(int num) ; // checks for conditions

bool numIsGreaterThanTwo (int num) ;

bool numIsNotDivisibleBySqrtOrLess (int num) ;

bool IsNumIsOne (int num) ;

int main () {

int num ;

cout << header ; // starting info

cout << pleaseEnter;

cin >> num ;

if ((valPos(num)) == true) { // program won't continue if num is zero or less

if (IsNumIsOne(num) == true || numIsEven(num) == true && numIsGreaterThanTwo(num) == true

|| numIsNotDivisibleBySqrtOrLess (num) == false) {

cout << "\nNumber entered: " << num << endl <<

num << " is not a prime number. \n\n" ;

}

else

cout << "\nNumber entered: " << num << endl <<

num << " is a prime number. \n\n" ;

} // end val

return 0 ;

} // End main

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Function: valPos

//

// Description: Validates if input is a positive number.

//

// Pre: The parameter must contain a number

//

// Post: Returns false if it is not a positive number.

// Will return true if it is a positive number.

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool valPos(int num) {

if (num <= 0) { // If num is less than or equal to zero, it is not positive

cout << "\nNumber entered must be positive number.\n" ;

return false;}

else

return true;

} // end valPos

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Function: numIsEven

//

// Description: Checks if a number is even

//

// Pre: The parameter must be a positive integer.

//

// Post: Returns true if number is even, and

// returns false if number is not even

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool numIsEven(int num) {

num = num % 2 ;

if (num == 0) {

return true ;

}

else

return false ;

} // end greaterThanTwoAndEven

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Function: numIsGreaterThanTwo

//

// Description: Checks if a number is greater than the number Two (2)

//

// Pre: The parameter must be a positive integer.

//

// Post: Returns true if number is greater than 2, and

// returns false if number is two or less.

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool numIsGreaterThanTwo (int num) {

if (num > 2)

return true;

else

return false;

} // end numIsGreaterThanTwo

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Function: numIsNotDivisibleBySqrtOrLess

//

// Description: Checks if a number is divisible by the square root of itself or

// integer values less than the root.

//

// Pre: The parameter must be a positive integer.

//

// Post: Returns false if divisible by a number equal to or less than the

// square root of the number. Otherwise it will return true.

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool numIsNotDivisibleBySqrtOrLess (int num) {

bool truth ;

int root = sqrt(num) ;

for ( int i = root; i > 1; i-- ) {

int quotient = num ; //refreshes testVal back to original number

quotient = quotient % i ;

if (quotient == 0)

return false ;

} // end for

return true;

} // end numIsNotDivisibleBySqrtOrLess

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Function: IsNumIsOne

//

// Description: Checks if number is one (1)

//

// Pre: The parameter must be a positive integer.

//

// Post: Returns true if the number is one, and returns false if it is not one

//

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool IsNumIsOne (int num) {

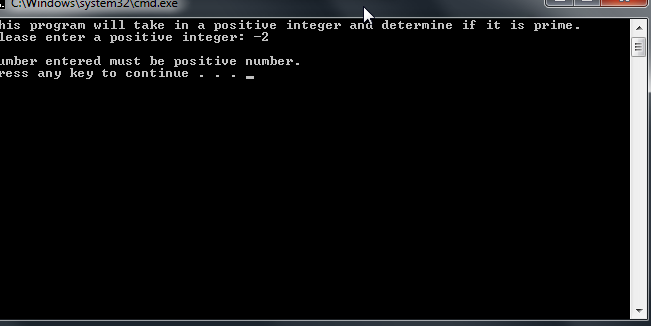
if (num == 1)

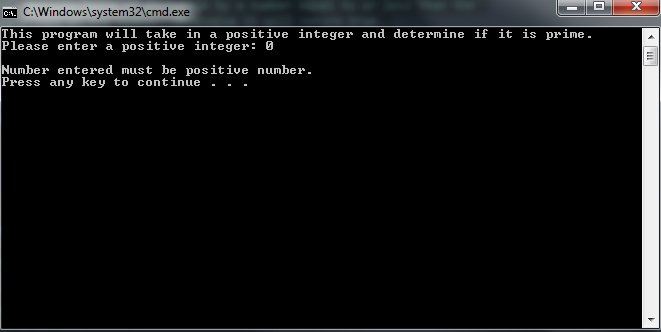
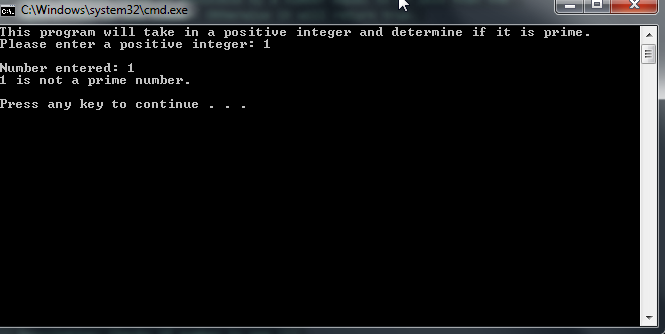
return true ;

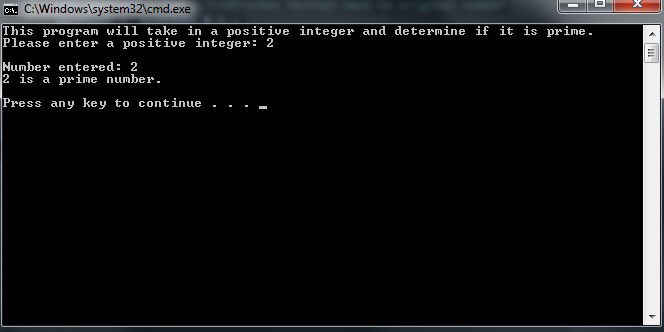
else

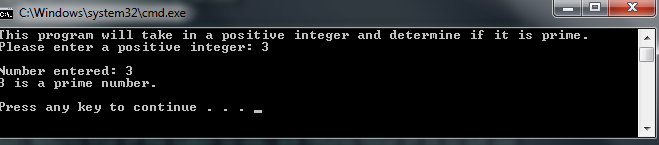
return false ;

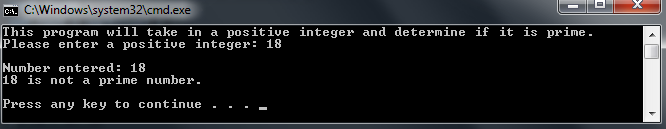
}

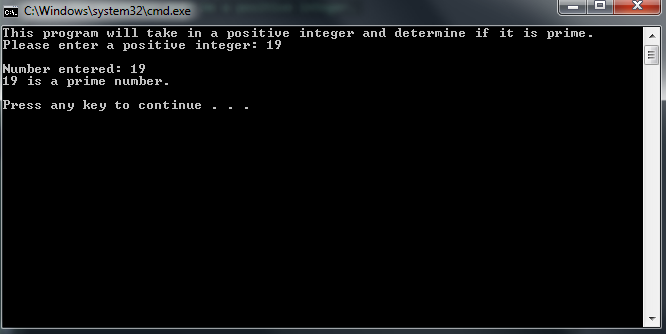


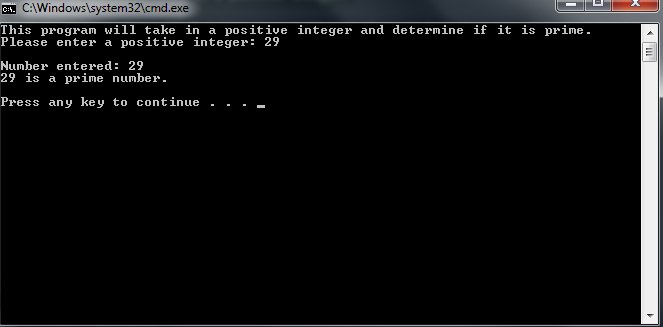
  












**Due Dates:**

As indicated on the Lab Eleven Dropbox Folder.

**What to hand in:**

1. Save the word document using your name and the lab number as the file name. eg. timwrennlab11.docx.
2. Compress the word document and your .cpp file into a single file called yournamelab11.zip.
3. Place the compressed file into the Lab Experience Eleven drop box folder.