Programmer Defined C++ Functions

In this week

- Introduction
- Programmer Defined C++ Functions
- Working with Programmer Defined C++ functions

Introduction
Built-in C++ functions

- Consider the following program that reads a non-negative double data type user input number and then computes and prints the square root of the number

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

int main()
{
    double x;
    cout << "Enter a non-negative number ";
    cin >> x;
    //Check if the user input value is non-negative
    if (x < 0)
        cout << "Please enter a correct input value. Bye" << endl;
    else
    {
        double s = sqrt(x);
        cout << "The square root of " << x << " is " << s << endl;
    }
    system("Pause");
    return 0;
}
```

Introduction
Built-in C++ functions

- How does the program compute the square root of any given non-negative number?
- We say the program is making use of the C++ built-in function named **sqrt**
- Most importantly, we do not even know the actual code that does the square root computation. Instead we are relying on the people who wrote the function and made it part of the mathematical library of C++ language
- We observe therefore that so long as there is a built-in function available that we can use then all we need to know is how to use it. That is
  - What the name of the function is,
  - What information we need to send to the function, and
  - What kind of information the function gives us back.
- In the case of the square root function, we see that
  - The name of the function is **sqrt**
  - We give the function a non-negative double or float data type value, and
  - The function gives us back a double data type value which is the square root of the number we had sent.

Programmer Defined C++ Functions

- Similarly, consider the following program designed to read three double data type user input numbers and then computes and prints the area of a triangle whose sides have lengths equal to the three numbers

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

int main()
{
    double s1, s2, s3;
    cout << "Enter the lengths of the three sides of a triangle: ";
    cin >> s1 >> s2 >> s3;

    //Check to make sure the lengths are correct numbers.
    //If they are not then don't calculate area.
    if (s1 <= 0 || s2 <= 0 || s3 <= 0)
        cout << "Each side of a triangle must have a positive length. Bye." << endl;
    else if (s1 + s2 <= s3 || s1 + s3 <= s2 || s2 + s3 <= s1)
        cout << "These numbers do not satisfy triangle inequality. Bye." << endl;
    else
    {
        double result = triangleArea(s1, s2, s3);
        cout << "The area of the triangle is " << result << endl;
    }
    system("Pause");
    return 0;
}
```

Programmer Defined C++ Functions

- Unfortunately this program does not work. In fact it has a syntax error
- Why? Because this time we are not lucky; there is no C++ built-in function named ***triangleArea*** that can compute and give back the area of a triangle given the lengths of the sides of a triangle
- So what do we do?
- Well we have to write our own function!
- When we write our own function, we call it programmer (or user) defined function
- Once again, in order to write a correct function we specify the following information
  - The name of the function
  - What information the function will receive
  - What information the function will give back

Programmer Defined C++ Functions

- The next question will be where to write the function
- **Answer:-** We write the function below the include directives and using namespaces but above the main program
- The **name a function** must adhere to **C++ identifier naming rules**
- That is a function name must consist of only English alphabets, digits or underscore and it must start with either an English alphabet or an underscore
- The following program shows our main program together with our function

Programmer Defined C++ Functions

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

double triangleArea(double a, double b, double c)
{
    double s = (a + b + c) / 2.0;
    double answer = sqrt(s * (s-a) * (s-b) * (s-c));
    return answer;
}
int main()
{
    double s1, s2, s3;
    cout << "Enter the lengths of the three sides of a triangle: ";
    cin >> s1 >> s2 >> s3;

    //Check to make sure the lengths are correct numbers.
    //If they are not then don't calculate area.
    if (s1 <= 0 || s2 <= 0 || s3 <= 0)
        cout << "Each side of a triangle must have a positive length. Bye." << endl;
    else if (s1 + s2 <= s3 || s1 + s3 <= s2 || s2 + s3 <= s1)
        cout << "These numbers do not satisfy triangle inequality. Bye." << endl;
    else
    {
        double result = triangleArea(s1, s2, s3);
        cout << "The area of the triangle is " << result << endl;
    }
    system("Pause");
    return 0;
}
```

Programmer Defined C++ Functions

- Next let us write a program that reads a positive integer user input that is greater than 1 and then prints either the message **"The number is prime"** or the message **"The number is not prime"**
- Although we could write all the code in the main program, we will rather write a function named ***isPrime*** that will perform the prime check for us and then we will make use of the function in the main program
- Obviously we will send one integer number to the function and we would like to get a **"Yes it is prime"** or a **"No it is not prime"** answer
- It therefore makes sense to design the function such that it gives a Boolean data type result (true to mean yes it is prime and false to mean no it is not prime)
- See below

Programmer Defined C++ Functions

```
#include <iostream>
using namespace std;
bool isPrime(int x)
{
    int count = 0;
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            count++;
    bool check;
    if (count == 0)
        check = true;
    else
        check = false;
    return check;
}
int main()
{
    int n;
    cout << "Please enter a positive integer greater than 1: ";
    cin >> n;
    if (n <= 1)
        cout << "Please enter a correct integer." << endl;
    else
    {
        bool ans = isPrime(n);
        if (ans == true)
            cout << "The number is prime." << endl;
        else
            cout << "The number is not prime." << endl;
    }
    system("Pause");
    return 0;
}
```

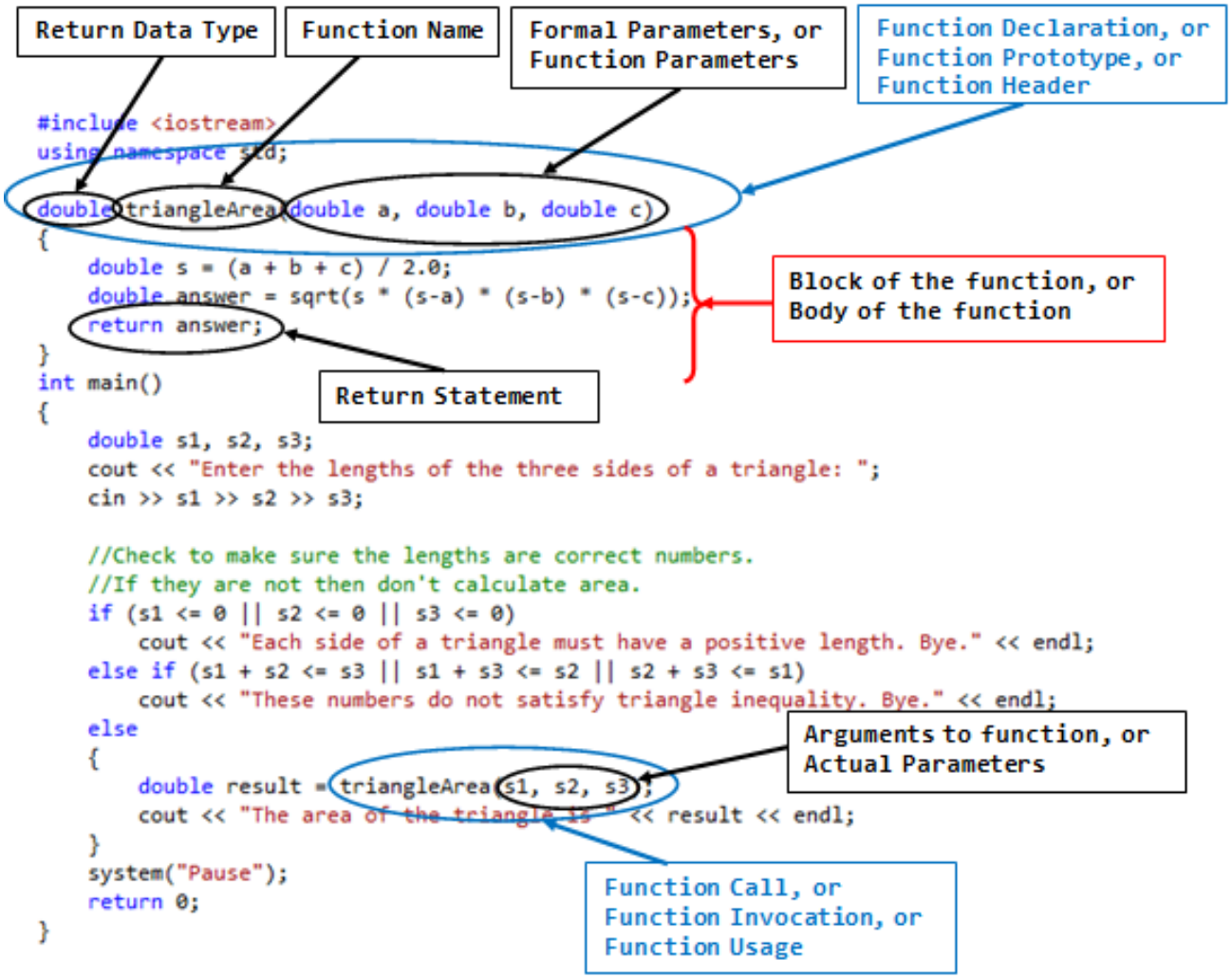
Programmer Defined C++ Functions

- We could also write this function in a more efficient way as follows

```
#include <iostream>
using namespace std;
bool isPrime(int x)
{
    bool check = true;
    for (int k = 2; k < x; k++)
    {
        if (x % k == 0)
        {
            check = false;
            break;
        }
    }
    return check;
}
int main()
{
    int n;
    cout << "Please enter a positive integer greater than 1: ";
    cin >> n;
    if (n <= 1)
        cout << "Please enter a correct integer." << endl;
    else
    {
        bool ans = isPrime(n);
        if (ans == true)
            cout << "The number is prime." << endl;
        else
            cout << "The number is not prime." << endl;
    }
    system("Pause");
    return 0;
}
```

Programmer Defined C++ Functions

Useful Terminologies




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

double triangleArea(double a, double b, double c)
{
    double s = (a + b + c) / 2.0;
    double answer = sqrt(s * (s-a) * (s-b) * (s-c));
    return answer;
}
int main()
{
    double s1, s2, s3;
    cout << "Enter the lengths of the three sides of a triangle: ";
    cin >> s1 >> s2 >> s3;

    //Check to make sure the lengths are correct numbers.
    //If they are not then don't calculate area.
    if (s1 <= 0 || s2 <= 0 || s3 <= 0)
        cout << "Each side of a triangle must have a positive length. Bye." << endl;
    else if (s1 + s2 <= s3 || s1 + s3 <= s2 || s2 + s3 <= s1)
        cout << "These numbers do not satisfy triangle inequality. Bye." << endl;
    else
    {
        double result = triangleArea(s1, s2, s3);
        cout << "The area of the triangle is " << result << endl;
    }
    system("Pause");
    return 0;
}
```

Programmer Defined C++ Functions

The main function

- We note that in C++, the main program is actually a programmer defined function
- The syntax of the language requires it to have an *int* return data type
- Therefore it has a return statement that returns the value
- Returning 0 is an indication to the operating system that the program has finished execution successfully
- Returning an integer different from 0 or a runtime error (which will close the program without returning any value) is an indication to the operating system that the program is corrupted

Programmer Defined C++ Functions

Order of Execution

- Whenever a statement in a C++ program calls a function as in the statement

```
double result = triangleArea(s1, s2, s3);
```

- Then the execution of the program will be **suspended at that statement** and **execution goes (jumps) to the function**
- **Once the function finishes execution**, then execution returns to the statement that called the function and **continues from where it was suspended**

Programmer Defined C++ Functions

Correspondence between arguments and parameters

- The number of arguments to a function must be equal to the number of parameters of the function
- Each argument to a function is sent **(copied)** to a corresponding parameter of the function in the order the arguments/parameters are listed
- The data type of each of the arguments to a function must be the same as the data type of its corresponding parameter (*or the argument must be such that it can be casted automatically to the data type of its corresponding parameter*)
- **A function can have zero or more parameters (arguments)**
- **Example:-** Write a program that prints ten random lower case English alphabets.
- **Answer:-** Although we can write this program without having to write any function, for demonstration purposes let's organize the program such that it calls a function named ***randomLowerCase*** that takes no argument and that returns a random lower case English alphabet. This function will therefore return a char data type as shown below.

Programmer Defined C++ Functions

Correspondence between arguments and parameters

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

char randomLowerCase()
{
    int x = rand() % 26 + 97;
    char c = x;
    return c;
}
int main()
{
    srand(time(0));
    cout << "Here is are ten random lower case English alphabets" << endl;
    for (int i = 0; i < 10; i++)
        cout << randomLowerCase() << endl;
    system("Pause");
    return 0;
}
```

Programmer Defined C++ Functions

The return statement

- The **return statement** of a function is used to **give back (i.e. to return)** a value from the function back to where the function was called from
- The return value from a function can be any C++ data type such as int, float, double, bool, char, etc.
- In this case **the function must have a return statement that must return a value of the same data type as the return data type of the function** or *a value that can be casted to the return data type of the function*
- Most importantly, the return statement can return only one value at a time; that is it can not return two or more values at the same time

Programmer Defined C++ Functions

The return statement

- The return statement also finishes execution of the function
- That is whenever a return statement is executed inside a function, then the function will automatically terminate and execution will return to where the function was called from
- **It is therefore perfectly fine to have more than one return statements inside a function**
- Whenever there are more than one return statements inside a function, then the first return statement that is executed will terminate the execution of the function
- We may therefore re-write the ***isPrime*** function in a more compact way as follows

Programmer Defined C++ Functions

The return statement

```
#include <iostream>
using namespace std;
bool isPrime(int x)
{
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
int main()
{
    int n;
    cout << "Please enter a positive integer greater than 1: ";
    cin >> n;
    if (n <= 1)
        cout << "Please enter a correct integer." << endl;
    else
    {
        bool ans = isPrime(n);
        if (ans == true)
            cout << "The number is prime." << endl;
        else
            cout << "The number is not prime." << endl;
    }
    system("Pause");
    return 0;
}
```

Programmer Defined C++ Functions

void functions

- A **function may also NOT return** any value at all
- However we still need to specify a return data type for the function
- In this case the return data type is specified as **void** to mean nothing
- Such functions are usually called **void functions**
- A void function does not necessarily need to have a return statement
- In this case execution of the function will finish after the last statement in the block of the function is executed

Programmer Defined C++ Functions

void functions

- Whenever a statement in a C++ program calls a void function, then the function call will not receive any result back and as such it can not be used inside any arithmetic or Boolean expressions. Neither can it be used inside a cout statement
- Instead **the function call will be a statement on its own**
- **Example:-** Write a void function named ***printReverse*** that takes a non-negative integer argument and prints the digits of the argument in a reverse
- In this case the function call will be similar to

  printReverse(2573);
- Here the task of the function will be to print the digits of its parameter in reverse and that is it. **It does not give back any result**
- Therefore this function does not necessarily need any return statement and it will finish execution after the last statement in the function body is executed. See below

Programmer Defined C++ Functions

void functions

```
#include <iostream>
using namespace std;
void printReverse(int num)
{
    while(true)
    {
        cout << num%10;
        num = num / 10;
        if (num == 0)
            break;
    }
}

int main()
{
    int n;
    cout << "Please enter a non-negative integer number: ";
    cin >> n;
    if (n < 0)
        cout << "Please enter a correct integer." << endl;
    else
    {
        cout << "The digits of the number you enetered in reverse order are ";
        printReverse(n);
        cout << endl;
    }
    system("Pause");
    return 0;
}
```

It is important to note here that both the *main function* and the *printReverse* function will share the same output window and thus all the *cout* statements irrespective of which function they are in will print to the same output window.

Programmer Defined C++ Functions

void functions and return statement

- We may also optionally have a return statement inside void functions
- In this case the purpose of the return statement will not be to return any value; but rather to terminate the execution of the function and take back execution to where the function was called from
- We may therefore re-write the ***printReverse*** function a little bit more compactly using a return statement as follows

Programmer Defined C++ Functions

void functions and return statement

```
#include <iostream>
using namespace std;
void printReverse(int num)
{
    while(true)
    {
        cout << num%10;
        num = num / 10;
        if (num == 0)
            return;
    }
}

int main()
{
    int n;
    cout << "Please enter a non-negative integer number: ";
    cin >> n;
    if (n < 0)
        cout << "Please enter a correct integer." << endl;
    else
    {
        cout << "The digits of the number you enetered in reverse order are ";
        printReverse(n);
        cout << endl;
    }
    system("Pause");
    return 0;
}
```

Programmer Defined C++ Functions

The main stack and the call stack

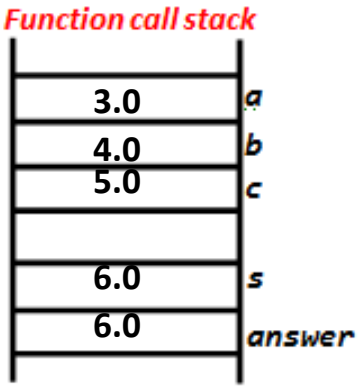
- A C++ function will always have its own memory space named the function **call stack**
- Similarly, the main function has its own memory space which is usually referred to as the **main stack**
- The parameters of a C++ function will therefore reside in the function's call stack
- Moreover any variable declared inside a C++ function will reside in the function's call stack
- Whenever a main function calls a function, the values of the arguments (*which are residing in the main stack*) are sent to the parameters of the function (*which are residing in the call stack*)
- The parameters of the function will therefore always receive **COPIES** of the values of the arguments
- See below

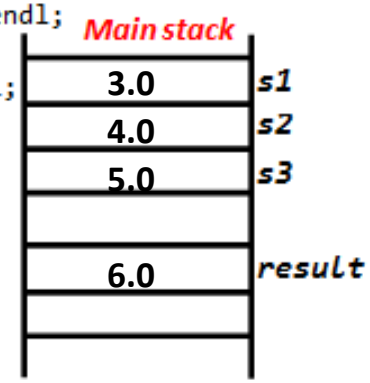
Programmer Defined C++ Functions

The main stack and the call stack

```
double triangleArea(double a, double b, double c)
{
    double s = (a + b + c) / 2.0;
    double answer = sqrt(s * (s-a) * (s-b) * (s-c));
    return answer;
}
int main()
{
    double s1, s2, s3;
    cout << "Enter the lengths of the three sides of a triangle: ";
    cin >> s1 >> s2 >> s3;

    //Check to make sure the lengths are correct numbers.
    //If they are not then don't calculate area.
    if (s1 <= 0 || s2 <= 0 || s3 <= 0)
        cout << "Each side of a triangle must have a positive length. Bye." << endl;
    else if (s1 + s2 <= s3 || s1 + s3 <= s2 || s2 + s3 <= s1)
        cout << "These numbers do not satisfy triangle inequality. Bye." << endl;
    else
    {
        double result = triangleArea(s1, s2, s3);
        cout << "The area of the triangle is " << result << endl;
    }
    system("Pause");
    return 0;
}
```





Programmer Defined C++ Functions

Parameter Passing

- When execution jumps to a function, then C++ can access only the call stack. It can not access the main stack
- This means **the scope of the parameters of a function and any variable declared inside the function is only in the block of the function**
- When a function finishes execution and returns, its call stack will be cleared (deleted or freed) automatically
- Moreover it means a function can use the same variable names as the ones already existing in the main function for its parameters or for any variable declared inside the function block. There will not be any conflict

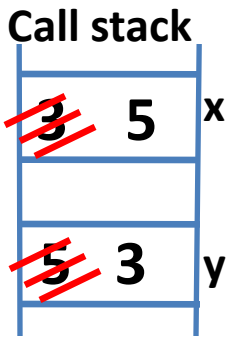
Programmer Defined C++ Functions

Parameter Passing

- An important consequence of the fact that a function will use its own call stack is that **whenever a function modifies any of its parameters then the argument corresponding to the parameter will not be modified**
- This is because once the argument is COPIED to the parameter, then the parameter is residing in the call stack and any modification to the parameter will only modify the call stack memory but it will not affect the main stack memory
- This is called **Parameter Passing by Value**
- Analyze the following program and determine its output.

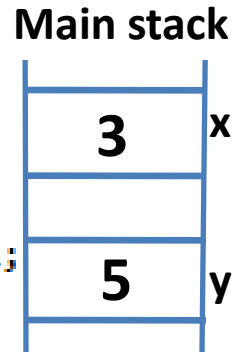
Programmer Defined C++ Functions

Parameter Passing




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

void swap(int x, int y)
{
    cout << "In function, initially x = " << x << " and y = " << y << endl;
    int temp = x;
    x = y;
    y = temp;
    cout << "In function, at the end x = " << x << " and y = " << y << endl;
}
int main()
{
    int x = 3, y = 5;
    cout << "In main, initially x = " << x << " and y = " << y << endl;
    swap(x, y);
    cout << "In main, at the end x = " << x << " and y = " << y << endl;
    system("Pause");
    return 0;
}
```



Programmer Defined C++ Functions

Calling C++ functions

- The arguments to a function can be either variable names in which case they **must** have been initialized with some values; or they can be some literal values such as numbers, characters, Boolean values. See the following examples

```
double s1 = 2.0, s2 = 5.0, s3 = 4.0;
double result1 = triangleArea(s1, s2, s3);
double result2 = triangleArea(2.0, 5.0, 4.0);
double result3 = triangleArea(s1, 5.0, s3);
bool ans1 = isPrime(5);
bool ans2 = isPrime(s2);
```

- Here all the variables result1, result2 and result3 will receive the same returned value from the **triangleArea** function. Similarly both ans1 and ans2 will get the same returned value from the **isPrime** function

Programmer Defined C++ Functions

Calling C++ functions

- A non-void function can be called in any of the following ways
  - Inside an assignment statement as a right hand side operand
    Example: `double x = triangleArea(2.0, 5.0, 4.0);`
  - Inside an arithmetic Expression
    Example: `double x = 2 * triangleArea(2.0, 5.0, 4.0);`
  - Inside a Boolean Expression
    Example: `if (triangleArea(2.0, 5.0, 4.0) > 5.0)`
  - Inside a cout statement
    Example: `cout << triangleArea(2.0, 5.0, 4.0) << endl;`
  - As a statement on its own
    Example: `triangleArea(2.0, 5.0, 4.0);`
- In the first four cases, the returned value of the function will be used in the statement. In the last case, the returned value will be thrown out
- A void function however can be called in only one way which is
  - As a statement on its own
    Example: `printReverse(2060);`

Constant Modifiers

Named Constants, Constant Variables, and Constant Parameters

- In C++,
  - A variables declared as a constant and assigned a literal value is known as a named constant

    Example:- `const float x = 2.5;`
  - A variable declared constant and assigned the value of another variable is known as a constant variable

    Example:-
    ```
    int a;
    cout << "Enter an integer ";
    cin >> a;
    const int y = a;
    ```
  - A parameter of a function declared as a constant is known as a constant parameter

    Example:- `bool isPrime(const int num)`
- None of such variables can modified because they are constants
- Most importantly, the value of a named constant in a C++ program can be determined from the source code of the program (that is without first running the program). However the value of a constant variable or a constant parameter can only be known after running the program
- A named constant simply acts as the name of the literal value assigned to it
- Thus a named constant can be assigned the value of another named constant
- Our ***isPrime*** function with a constant parameter is shown below

Constant Modifiers

Named Constants, Constant Variables, and Constant Parameters

- Whenever we are designing a function that by design does not modify some or all of its parameters, then it is important we designate the parameters that won't be modified as constant parameters which is useful for avoiding any unintended modification

```
bool isPrime(const int x)
{
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
int main()
{
    int n;
    cout << "Please enter a positive integer greater than 1: ";
    cin >> n;
    if (n <= 1)
        cout << "Please enter a correct integer." << endl;
    else
    {
        bool ans = isPrime(n);
        if (ans == true)
            cout << "The number is prime." << endl;
        else
            cout << "The number is not prime." << endl;
    }
    system("Pause");
    return 0;
}
```

Constant Modifiers

Named Constants, Constant Variables, and Constant Parameters

- In order to appreciate the usefulness of constant modifier for parameters, analyze the following program and determine its output.

```
bool isEven(int v)
{
    if (v = 2*(v/2))
        return true;
    else
        return false;
}
int main()
{
    int n;
    cout << "Please enter a non-negative integer: ";
    cin >> n;
    if (n < 0)
        cout << "Please enter a correct integer. Bye" << endl;
    else
    {
        if (isEven(n))
            cout << "The number is even." << endl;
        else
            cout << "The number is odd." << endl;
    }
    system("Pause");
    return 0;
}
```

Constant Modifiers

Named Constants, Constant Variables, and Constant Parameters

- We note that a constant variable in a main program can have different values for every run of the program; but it has a fixed value within a single run of the program. On the other hand a constant parameter or a constant variable defined inside a function can have different values within a single run of a program; but have fixed values within a single function call.

```
void foo(const int a)   //Constant parameter
{
    const int b = a+1;  //Constant variable
    cout << a << "\t" << b << endl;
    return;
}
int main()
{
    const int a = 8;    //Named constant
    int b;
    cout << "Enter an integer ";
    cin >> b;
    const int c = b;    //Constant variable
    const int d = a;    //Named constant

    foo(a);
    foo(b-c);
    foo(b+d);

    system("Pause");
    return 0;
}
```

Pre-condition and Post-condition

- Generally speaking it is assumed that the parameters of a function will always receive correct data values from their corresponding arguments
- This is why in all our examples so far, we had been checking the correctness of the user input values before using these input values as arguments when calling a function
- This is usually emphasized by explicitly writing comments inside the function block that specify what conditions the parameters of the function must satisfy (known as the pre-condition) and what the function is intended to perform (known as the post-condition)
- Whenever a function is given arguments that violate its pre-condition requirements, then the function may get into run time error and crash the program or may give wrong result
- The next example shows our ***isPrime*** function stating explicitly its pre-condition and post-condition for clarity purposes

Pre-condition and Post-condition

```
bool isPrime(const int x)
{
    //Pre-condition:- x is an integer value greater than 1
    //Post-condition:- returns true if x is prime and false otherwise
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
int main()
{
    int n;
    cout << "Please enter a positive integer greater than 1: ";
    cin >> n;
    if (n <= 1)
        cout << "Please enter a correct integer." << endl;
    else
    {
        bool ans = isPrime(n);
        if (ans == true)
            cout << "The number is prime." << endl;
        else
            cout << "The number is not prime." << endl;
    }
    system("Pause");
    return 0;
}
```

Supporting/Helper Functions

- A C++ program can have several functions and the main program can call any of the functions
- Moreover a function may call another function so long as the function being called is written above the function making the call
- The main program or any of the functions may also make use of any C++ built-in functions
- A function that helps the main program or another function in order to perform some computations is known as a supporting or helper function
- Thus whenever we are faced with complicated tasks in our programs, we normally write several supporting or helper functions and make use of these functions in order to make problem solving easier
- Analyze the following program and determine its output. In particular identify the C++ built-in functions and the programmer defined functions in the program

Supporting/Helper Functions

```
bool isPrime(const int x)
{
    //Pre-condition:- x is an integer value greater than 1
    //Post-condition:- returns true if x is prime and false otherwise
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
void printPrimes(const int a, const int b)
{
    //Pre-condition:- a and b are any two integer values
    //Post-condition:- all primes numbers between them are printed
    int smaller = min(a, b);
    int larger = max(a, b);
    int k = smaller >= 2 ? smaller : 2;
    for (; k <=larger; k++)
        if (isPrime(k) == true)
            cout << k << endl;
}
int main()
{
    int a, b;
    cout << "Please enter two integers ";
    cin >> a >> b;
    cout << "All the prime numbers between " << min(a,b) << " and " << max(a,b) << " are" << endl;
    printPrimes(a, b);
    system("Pause");
    return 0;
}
```

Function Declaration and Function Definition

- The part of a function consisting of the return data type, function name and parameter list is known the **function declaration** (or the function prototype or the function header)
  - The function declaration of our isPrime function is therefore

    bool isPrime(const int x)
- The function declaration together with the function block (or function body) is known as the **function definition**
  - The function definition of our isPrime function is therefore

```
bool isPrime(const int x)
{
    //Pre-condition:- x is an integer value greater than 1
    //Post-condition:- returns true if x is prime and false otherwise
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
```

Function Declaration and Function Definition

- As shown in our previous examples, by rule of C++ language we must always write our functions above the main program
- However, we may also write our functions below the main program as long as we write the function declaration above the main program
- When a function declaration is put above the main program, then it is a C++ statement and as such it must be terminated with a semicolon
- We may therefore put our ***isPrime*** function below the main program as follows

Function Declaration and Function Definition

```
#include <iostream>
using namespace std;
bool isPrime(const int x);
int main()
{
    int n;
    cout << "Please enter a positive integer greater than 1: ";
    cin >> n;
    if (n <= 1)
        cout << "Please enter a correct integer." << endl;
    else
    {
        bool ans = isPrime(n);
        if (ans == true)
            cout << "The number is prime." << endl;
        else
            cout << "The number is not prime." << endl;
    }
    system("Pause");
    return 0;
}
bool isPrime(const int x)
{
    //Pre-condition:- x is an integer value greater than 1
    //Post-condition:- returns true if x is prime and false otherwise
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
```

function declaration

function definition

Function Declaration and Function Definition

- It is also allowed not to write the parameter name whenever we write a function declaration statement above the main program

```
#include <iostream>
using namespace std;
bool isPrime(const int);
int main()
{
    int n;
    cout << "Please enter a positive integer greater than 1: ";
    cin >> n;
    if (n <= 1)
        cout << "Please enter a correct integer." << endl;
    else
    {
        bool ans = isPrime(n);
        if (ans == true)
            cout << "The number is prime." << endl;
        else
            cout << "The number is not prime." << endl;
    }
    system("Pause");
    return 0;
}
bool isPrime(const int x)
{
    //Pre-condition:- x is an integer value greater than 1
    //Post-condition:- returns true if x is prime and false otherwise
    for (int k = 2; k < x; k++)
        if (x % k == 0)
            return false;
    return true;
}
```

Function Signature and Function Overloading

- The part of a function declaration consisting of the function name and the parameter list is known as the signature of the function
  - The signature of our ***isPrime*** function is therefore

    ```
    isPrime(const int x)
    ```
- Strictly speaking the constant modifier and the parameter names are not part of the function signature
- Thus in a strict sense the signature of our ***isPrime*** function is therefore

  ```
  isPrime(int)
  ```
- A C++ program can have more than one function with the same name as long as each of the functions have different signatures (that is different number of parameters or different data types for the parameters)
- Having two or more functions with the same name but different signatures in the same C++ program is known as function overloading
- Of course having two functions with the same signature in the same program is not allowed for when we call the function, C++ will not be able to pick one from the other

Function Signature and Function Overloading

- The following program demonstrates function overloading

```
void foo(int x)
{
    cout << "In foo --- int function" << endl;
    return;
}
void foo(float x)
{
    cout << "In foo --- float function" << endl;
    return;
}
void foo(double x)
{
    cout << "In foo --- double function" << endl;
    return;
}
void foo(int x, float y)
{
    cout << "In foo --- int, float function" << endl;
    return;
}
int main()
{
    float a = 1.1;
    foo(1);
    foo(1.1);
    foo(1, 1.1);
    foo(a);
    system("Pause");
    return 0;
}
```

Default Arguments

- The parameters of a function may also be assigned a value on the function declaration which is known as a default argument
- In such a case,
  - If the function is called by sending an argument then the value of the argument replaces the default argument; otherwise the parameter will use the default argument
  - If there are more than one parameters with default arguments and the function is called with fewer arguments than the parameters of the function then the values of the arguments will replace the first parameters they find in the function declaration while the remaining parameters will use their default argument values
- Analyze the following program and determine its output

Default Arguments

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

void foo(const char ch = '?', const int n = 10)
{
    for (int i = 0; i < n; i++)
        cout << ch;
    cout << endl;
}
int main()
{
    foo();
    foo('Y');
    foo('Y', 20);
    foo(48);
    system("Pause");
    return 0;
}
```