

# C++ Basics (continued)

## In this topic

- More on C++ Basics
- Conditional Statements
- C++ Built-in functions
- Loops in C++
- Programmer defined C++ functions

# Unary Arithmetic Operators

- In addition to the binary operators, C++ supports unary operators for integer data type variables
- Unary operators have only one operand
- These operators are ++ and --
- They increment/decrement the value of their operand by 1
- These operators can be placed either on the left hand side or the right hand side of their operands as shown below

```
int a = 4, b = 7, c = 10, d = 1;
a++;          ←increment value of a by 1 (same as a = a + 1;)
++b;          ←increment value of b by 1 (same as b = b + 1;)
c--;          ←decrement value of c by 1 (same as c = c - 1;)
--d;          ←decrement value of d by 1 (same as d = d - 1;)
cout << a << " " << b << " " << c << " " << d << endl;
```

# Unary Arithmetic Operators

- When unary operators are placed inside statements then they will be executed differently depending on which side of their operands they are placed as shown below

```
int a = 4, b = 7, c = 10, d = 1;
cout << a++ << endl; ←print value of a then increment it
cout << ++b << endl; ←increment value of b then print it
cout << c-- << endl; ←print value of c then decrement it
cout << --d << endl; ←decrement value of d then print it
cout << a << " " << b << " " << c << " " << d << endl;
a = ++b - c++ - --d; ←Perform ++b, then --d, then the statement, and finally c++
cout << a << endl; ←prints 1
cout << b << endl; ←prints 9
cout << c << endl; ←prints 10
cout << d << endl; ←prints -1
```

# Pre Increment and Post Increment

- The unary operators are also known as pre increment, pre decrement, post increment or post decrement based on which side (left or right) the unary operator is placed

- **Pre Increment**: ++x

Increments the value of x first and then uses the new value

- **Post Increment**: x++

Uses the current value of x and then increments it

- **Pre Decrement**: --x

Decrements the value of x first and then uses the new value

- **Post Decrement**: x--

Uses the current value of x and then decrements it

# C++ char Data Type

- A C++ char variable is used to store a single character such as an alphabet, a digit, or any other symbol placed in between single quotes in which case the ASCII code of the character is stored in memory using two's complement representation

- **Example**

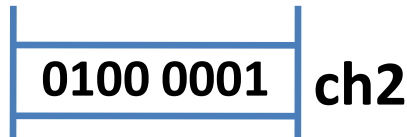
**char ch1 = 'A';**



- Now ch1 is assigned the integer value 65 (ASCII code of 'A') whose two's complement binary representation is shown above
- Moreover, a C++ char variable can be assigned a signed integer number in which case the integer number is stored in memory using two's complement representation

- **Example**

**char ch2 = 65;**




- Now ch2 is assigned the integer value 65 in two's complement representation as shown above

# C++ char Data Type

- A char variable can also be assigned an input value from the keyboard using **cin** command
- In this case the user will have to put one character input from the keyboard (such as an alphabet, a digit or any other symbol) and then the ASCII code of the character input will be stored in memory using two's complement representation
- Single quotes are not needed when typing an input from the keyboard
- **Example**  

```
char ch3;  
cout << "Please enter a character ";  
cin >> ch3; A↵
```


- In this case ch4 will be assigned the integer value 65 as shown above in its two's complement representation

# C++ char Data Type

- Although the actual value stored inside a C++ char variable is a signed integer represented in two's complement; printing a C++ char variable using **cout** command will not print the signed integer stored in the variable
- Instead, the **cout** command will treat the char variable as if it is **unsigned binary representation**, compute its unsigned decimal value **(which will always be between 0 and 255)** and finally print the symbol (**character**) whose **ASCII code** is the unsigned decimal number computed
- Example

```
char ch1 = 321, ch2 = 'A';  
cout << "ch1 is " << ch1 << " and ch2 is " << ch2 << endl;
```

Output  
ch1 is A and ch2 is A

# Type Casting

- Sometimes we may need to convert the data type of the values of certain variables during an arithmetic expression
- For example suppose that we have two **int** variables and we would like to perform division operation between the values of the variables in floating data type
- Then we would convert the data type for the operation
- Example

```
int a = 5, b = 3;
```

```
float c = static_cast<float>(a) / b;
```

- Now, **c** will be assigned  $5.0/3 = 1.67$
- We say the value of the variable **a** is casted from **int** to **float**



# Type Casting

```
int main()
{
    int a = 5, b = 3;
    cout << static_cast<float>(a) / b << endl;
    cout << a / static_cast<float>(b) << endl;
    cout << static_cast<float>(a) / static_cast<float>(b) << endl;
    cout << static_cast<float>(a / b) << endl;
    cout << a << endl;
    cout << b << endl;
    system("Pause");
    return 0;
}
```

- In this case, the first, second and third expressions all give 1.67 that is to say  $5.0/3 = 5/3.0 = 5.0/3.0 = 1.67$
- While the fourth line performs the division  $5/3$  to get the integer 1 and then this integer result is casted to float 1.0
- The variables a and b still remain int data types after the computation and their values unchanged!

# Relational/Comparison Operators

- C++ supports the following relational (comparison) binary operators

Relational Operator	Description
<b>==</b>	<b>is equal to</b>
<b>!=</b>	<b>is not Equal to</b>
<b>&lt;</b>	<b>is less than</b>
<b>&gt;</b>	<b>is greater than</b>
<b>&lt;=</b>	<b>is less than or equal to</b>
<b>&gt;=</b>	<b>is greater than or equal to</b>

- Relational operators give either a **true** or a **false** answer

# Logical Operators

- C++ supports the following logical operators

Logical Operator	Description	Evaluation
!	Logical NOT	false if operand is true; true if operand is false
&&	Logical AND	true if and only if both operands are true; otherwise false
	Logical OR	false if and only if both operands are false; otherwise true

- The operands of logical operators must be true or false values or some expressions that give true or false values
- ! has precedence over && and ||
- Moreover && has precedence over ||
- Same operators are executed left to right order

# Boolean Expressions

- Any C++ expression that gives a Boolean (true or false) result is known as a Boolean expression
- Boolean expressions are constructed using relational and logical operators
- For example, given **int a = 5, b = 3, c = 6;**
  - **a\*c > b** is a Boolean expression and its result **true**
  - **(a <= c) && (b > a)** is a Boolean expression and its result is **false**
  - **(a != c) || !(c > a)** is a Boolean expression and its result is **true**
- Moreover, any numeric value (such as int, float, double, etc) can be used as a Boolean expression where
  - A value **0** is considered as **false**, and
  - A value **1** (or any other number different from 0) is considered as **true**
- Therefore
  - **0** is considered as a Boolean expression and its result **false**
  - **1** is considered as a Boolean expression and its result is **true**
  - **-0.05** is considered as a Boolean expression and its result is **true**

# Boolean Expressions

- Except when used inside a **cout** statement, C++ char type is always processed as a numeric information in two's complement. See the following example

```
int main()
{
    char c1 = -206, c2 = '3';
    int a = c1;           //c1 acts as integer 50
    cout << a << endl;
    bool f1 = 56 > c2;    //c2 acts as integer 51
    cout << f1 << endl;
    float x = a + c2;     //c2 acts as integer 51
    cout << x << endl;
    bool f2 = c1 > 'A';   //c1 acts as integer 50 and 'A' acts as integer 65
    cout << f2 << endl;

    cout << c1 << endl;   //c1 acts as a character '2'
    cout << c2 << endl;   //c2 acts as a character '3'

    char c3 = c1 + c2;    //c1 acts as integer 50 and c2 acts as integer 51
    cout << c3 << endl;   //c3 acts as character 'e'

    cout << c1 + c2 << endl; //c1 acts as integer 50 and c2 acts as integer 51
                           //The result? Integer 101. Why? Because the arithmetic
                           //is performed in int data type

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

# Conditional Statements

## The **if** statement

- C++ supports conditional statements using **if** statements, **if ... else** statements, and **if ... else if ... else** statements
- <if> statement: **Syntax**

```
if (boolean expression) ← The brackets are required!!!  
{  
    Block of if statement  
}
```

- In this case, the **Block of the if** is **executed** only when the **boolean expression** is evaluated to **true**; otherwise it is skipped
- A program can have several if statements each with its own block. In that case each if statement will work independently

# Conditional Statements

## The **if-else** statement

- <if ... else> statement: **Syntax**

```
if (boolean expression)
{
    If Block C++ Statements
}
```

← The brackets ( and ) are required!!!

```
else
{
    Else Block C++ statements
}
```

- Here, the **if block** is **executed** only when the **boolean expression** is evaluated to **true**. In this case the **else block** is automatically skipped!!!
- Similarly, the **else Block** is **executed** only when the **boolean expression** is evaluated to **false**. In this case the **if block** is automatically skipped!!!
- Thus, the if and else blocks are mutually exclusive. **But one of the two blocks is always executed!!!**

# Conditional Statements

## The **if - else if ... else if** statement

- <if ... else if ... else if ...> statement: Syntax

```
if (boolean expression 1)    ← The ( and ) brackets are required!!!
{
    If Block C++ Statements
}
else if (boolean expression 2) ← The ( and ) brackets are required!!!
{
    Else if Block C++ statements
}
else if (boolean expression 3)
{
    Else if Block C++ statements
}
⋮
else if (boolean expression n)
{
    Else if Block C++ Statements
}
```

- Again these <if ... else if ... else if ... > blocks are mutually exclusive. Once a true boolean expression is found, its block is executed and all the remaining else if statements are skipped. That is IF AT ALL, ONLY ONE BLOCK IS EXECUTED
- If none of the boolean expressions is evaluated to true, all the blocks are skipped!



# Conditional Statements

## The **if - else if ... else** statement

- <if ... else if ... else> statement: **Syntax**

```
if (boolean expression 1)
{
    If Block C++ Statements
}
else if (boolean expression 2)
{
    Else if Block C++ statements
}
else if (boolean expression 3)
{
    Else if Block C++ statements
}
:
else
{
    Else Block C++ statements
}
```

- As before all the blocks are mutually exclusive. The else block is executed if none of the boolean expressions is evaluated to true
- **The biggest difference when there is else block is that in this case ONE OF THE BLOCKS WILL ALWAYS NECESSARILY BE EXECUTED!!!**

# Conditional Statements

## Example

```
int main()
{
    int n;
    cout << "Enter an integer ";
    cin >> n;
    if (n >= 100)
    {
        cout << "You entered a big positive number." << endl;
    }
    else if (n <= -100)
    {
        cout << "You entered a big negative number." << endl;
    }
    else if (n < 100 && n > 0)
    {
        cout << "You entered a small positive number." << endl;
    }
    else if (n > -100 && n < 0)
    {
        cout << "You entered a small negative number." << endl;
    }
    else
    {
        cout << "You entered zero." << endl;
    }
    system("Pause");
    return 0;
}
```

Analyze the following program manually and determine the output if the user input value is

- a) -350
- b) 250
- c) 27
- d) 0
- e) -25

# Conditional Statements

## Remarks on Curly Brackets

- If a block has only one statement that belongs to the block, then the curly brackets can be omitted for quick coding purposes

```
int main()
{
    int n;
    cout << "Enter an integer ";
    cin >> n;
    if (n >= 100)
        cout << "You entered a big positive number." << endl;
    else if (n <= -100)
        cout << "You entered a big negative number." << endl;
    else if (n < 100 && n > 0)
        cout << "You entered a small positive number." << endl;
    else if (n > -100 && n < 0)
        cout << "You entered a small negative number." << endl;
    else
        cout << "You entered zero." << endl;
    system("Pause");
    return 0;
}
```

# Conditional Statements

## Block inside Block

- A C++ block can contain other valid blocks inside it. Analyze the following program and determine its for the input values -350, 250, 27, 0, and -25.

```
int main()
{
    int n;
    cout << "Enter an integer ";
    cin >> n;
    if (n >= 100)
        cout << "You entered a big positive number." << endl;
    else if (n <= -100)
        cout << "You entered a big negative number." << endl;
    else
    {
        if (n < 100 && n > 0)
            cout << "You entered a small positive number." << endl;
        else if (n > -100 && n < 0)
            cout << "You entered a small negative number." << endl;
        else
            cout << "You entered zero." << endl;
    }
    system("Pause");
    return 0;
}
```

# Conditional Statements

## Remarks on Curly Brackets

- If the opening and closing curly brackets of a **Block** are omitted, then **only one C++ statement belongs to the block**
- Analyze the following program and determine its output.

```
int main()
{
    int x = -2;
    if (x > 0)
        x = 3 * x;
        cout << x << endl;
    cout << x << endl;

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

# Scope of Variables

- The same variable name can not be declared more than once inside the same block

```
int main()
{
    int x = -5;
    cout << "x has a value " << x << endl;
    //int x;    //Error you can not re-declare
    cout << "Enter an integer ";
    cin >> x;
    cout << "x has value " << x << endl;

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

# Scope of Variables

- Any variable declared inside a block remains accessible ONLY inside the block.
- We say the scope of any C++ variable is only inside the block it is declared in
- Analyze the following program and determine its output

```
int main()
{
    int x = -5;
    if (x < 0)
    {
        cout << "x has value " << x << endl;
        int y = 2;
        cout << "y has value " << y << endl;
    }
    cout << "x has a value " << x << endl;
    //cout << "y has a value " << y << endl;    //This is error

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

# Scope of Variables

- It is allowed to declare a variable inside a block even if the same variable name exists outside the block
- In this case, the inner most block variable shadows an existing same name variable when execution reaches the inner most block

```
int main()
{
    int x = -5;
    if (x < 0)
    {
        cout << "x has value " << x << endl;
        int x = 5;
        cout << "x has value " << x << endl;
        int y = 2;
        cout << "y has value " << y << endl;
    }
    cout << "x has a value " << x << endl;
    //cout << "y has a value " << y << endl; //This is error

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



# C++ Ternary (or Conditional) Operator

- C++ provides a ternary operator (three operands) which provides a convenient conditional operation

- **Syntax**

**bool\_expression ? Statement1 : Statement2;**

- In this case, Statement1 is executed if the boolean expression is true in which case statement2 is skipped; otherwise Statement2 is executed and statement1 is skipped.

# Ternary Conditional Operator Example

```
int main()
{
    int x;
    cout << "Enter an integer ";
    cin >> x;
    x >= 0 ? cout << x << endl : cout << -x << endl;
    system("Pause");
    return 0;
}
```

The ternary conditional operator is the same as the <if ... else> statement

```
int main()
{
    int x;
    cout << "Enter an integer ";
    cin >> x;
    if (x >= 0)
        cout << x << endl;
    else
        cout << -x << endl;
    system("Pause");
    return 0;
}
```

# C++ Built-In Functions

## The cmath library

- C++ provides built-in mathematical functions
- In order to use the mathematical built-in functions, we need to include the math library as follows

**#include <cmath>**

- The math library has many functions such as the absolute value function, the power function, the square root function, rounding functions, trigonometric functions, and much more
- See example code below...

# C++ Built-In Functions

## The cmath library

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int a = -3;
    double b = 6.25, c = -1.097;

    a = abs(a); //absolute value function
    cout << "Absolute value function: = " << a << endl;

    double answer = pow(a, b); //power function
    cout << "Power function " << answer << endl;

    //rounding up function
    cout << "Rounding up function " << ceil(b) << endl;

    //rounding down function
    int x = floor(c);
    cout << "Rounding down function " << x << endl;

    //Rounding up and down functions
    floor(c) > ceil(c) ? cout << "Wow!" << endl : cout << "Expected!" << endl;

    answer = sqrt(b); //Square root function
    cout << "Square root function " << answer << endl;

    //Functions inside an arithmetic
    answer = (0.5 * sqrt(b)) / floor(b);
    cout << answer << endl;

    //Trigonometric functions: sin,cos,tan,asin,acos,atan
    answer = sin(3.14/2); //Trigonometric functions use radian measure NOT degree
    cout << "Sine function " << answer << endl;
    //Please note that C++ trigonometric functions use radian NOT degree

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

# C++ Built-In Functions

## The cmath library

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    double a, b, c;
    cout << "Please enter the coefficients a, b and c: ";
    cin >> a >> b >> c;
    double d = pow(b, 2) - 4*a*c;
    if (d < 0)
        cout << "No real roots." << endl;
    else if (d > 0)
    {
        double x1 = (-b + sqrt(d)) / (2*a);
        double x2 = (-b - sqrt(d)) / (2*a);
        cout << "Two real roots: " << x1 << " and " << x2 << endl;
    }
    else
        cout << "One real root: " << -b/(2*a) << endl;
    system("Pause");
    return 0;
}
```

# C++ Built-In Functions

## Random Number Generator

- In order to work with random numbers, include the `cstdlib` library as follows:

**`#include <cstdlib>`**

- Then use the built-in function **`rand()`** to generate random **integers** in the range **`[0, RAND_MAX)`**
- **`RAND_MAX`** is a constant integer defined in `cstdlib` library whose value is **`32,767`** in MSVC++ 2010 Express Edition
- See the following example...

# C++ Built-In Functions

## Random Number Generator

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    cout << "Generating 5 random integers in the range [0, RAND_MAX)" << endl;
    int a = rand();      //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;
    a = rand();          //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;
    a = rand();          //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;
    a = rand();          //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;
    a = rand();          //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;

    cout << "The value of RAND_MAX is " << RAND_MAX << endl;

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

# C++ Built-In Functions

## Random Number Generator

- Careful observation will reveal that every time we run the previous program, the same five random integers will be printed
- That is, even though the five random numbers are different from each other, repeated execution of the program will print these numbers again and again and again...
- In order to get different output at each execution of the program, we need to seed the random number generator as follows



# C++ Built-In Functions

## Random Number Generator

---

```
#include <iostream>    //Library in order to use cout and cin
#include <cstdlib>      //Library in order to use rand
#include <ctime>        //Library in order to use time
using namespace std;
int main()
{
    srand(time(0)); //Seeding a random number generator

    cout << "Generating 5 random integers in the range [0, RAND_MAX)" << endl;
    int a = rand();    //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;
    a = rand();        //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;
    a = rand();        //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;
    a = rand();        //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;
    a = rand();        //Some random integer in the range [0, RAND_MAX)
    cout << a << endl;

    cout << "The value of RAND_MAX is " << RAND_MAX << endl;

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

# C++ Built-In Random Number Generator Function

- As can be seen in the previous program; the C++ `rand()` function is designed to generate random integers in the range **[0, RAND\_MAX)**
- But what if we want to generate random integers in a restricted interval **[a, b]** where **a** and **b** are some given integers with the condition that **a ≤ b**
- In this case, we need to transform the generated random number to the interval **[a, b]** as follows  
**`int r = rand() % (b-a+1) + a;`**
- Now the value of **r** is in the range **[a, b]** as required

# C++ Built-In Random Number Generator Function

- The following program demonstrates generating random integer numbers in a required range

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    srand(time(0));
    //Let us generate and print a random integer number in the range [-3, 8]
    int a = -3, b = 8;
    int r = rand() % (b-a+1) + a;
    cout << "The generated random integer number in the range [-3, 8] is " << r << endl;

    //Let us generate and print a random integer number in the range [a, b]
    //where a and b are user inputs with the condition that a <= b
    cout << "Enter two integers ";
    cin >> a >> b;
    if (a > b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
    r = rand() % (b-a+1) + a;
    cout << "The generated random integer number in the given range is is " << r << endl;

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

# C++ Built-In Random Number Generator Function

- What if we want to generate a random float or double numbers in a restricted interval **[a, b)** where **a** and **b** are some given float or double values with the condition that  **$a \leq b$**
- In this case, we need to transform the generated random integer number to the float or double value in the interval **[a, b)** as follows

```
float r = static_cast<float>(rand()) / RAND_MAX * (b-a) + a;
```

- Now the value of **r** is float number in the range **[a, b)** as required

# C++ Built-In Random Number Generator Function

- The following program demonstrates generating random float/double numbers in a required range

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    srand(time(0));
    //Let us generate and print a random float number in the range [-3.5, 4.8]
    float a = -3.5, b = 4.8;
    float r = static_cast<float>(rand()) / RAND_MAX * (b-a) + a;
    cout << "The generated random float number in the range [-3.5, 4.8] is " << r << endl;

    //Let us generate and print a random float number in the range [a, b]
    //where a and b are user inputs with the condition that a <= b
    cout << "Enter two float numbers ";
    cin >> a >> b;
    if (a > b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
    r = static_cast<float>(rand()) / RAND_MAX * (b-a) + a;
    cout << "The generated random float number in the given range is " << r << endl;

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

# Loops in C++

- C++ provides three types of loops
- These are

➤ **The *while* loop**

➤ **The *for* loop, and**

➤ **The *do-while* loop**

# The while loop

- **Syntax**

```
while (Boolean_expression)
{
    Block of the while loop
}
```

- The while loop block is executed as long as the Boolean expression is evaluated to true; and execution goes below the block only when the Boolean expression is evaluated to false
- The curly brackets designate the block of the while loop. They can be omitted if the block contains only one statement
- If the curly brackets are omitted then **only the first** statement belongs to the block

# The while loop

## Example

- Analyze the following example and determine its output if the user input values are 12 and 18.

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

int main()
{
    int a, b;
    cout << "Enter two positive integers: ";
    cin >> a >> b;
    if (a <= 0 || b <= 0)
        cout << "You must enter positive integers. Bye." << endl;
    else
    {
        cout << "The common factors of " << a << " and " << b << " are: ";
        int k = 1;
        while (k <= a && k <= b)
        {
            if (a % k == 0 && b % k == 0)
                cout << k << " ";
            k++;
        }
        cout << endl;
        system("Pause");
        return 0;
    }
}
```



# Type Casting and Boolean Expressions

- Sometimes we may use an expression that is not strictly speaking a Boolean expression in the place of the Boolean expressions and rely on automatic type casting
- Analyze the following program and determine its output

```
int main()
{
    int k = 5;
    while (k)
    {
        k *= -1;
        cout << k << endl;
        k > 0 ? k-- : k++;
    }
    system("Pause");
    return 0;
}
```

# The *for* loop

- for loop: **Syntax**

```
for (initialization; boolean_exp; update_action)
{
    Block of the for loop
}
```

- **Initialization** is executed only once before the loop begins
- The block is repeatedly executed as long as the **boolean\_exp** is evaluated to true
- The **update\_action** is executed at the end of each iteration
- The **boolean\_exp** is checked after each **update\_action**

# The *for* loop

## Example

- Consider the following for loop example and determine its output.
- How many iterations does the loop perform?

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Demonstrating a C++ for loop." << endl;
    for (int k = 0; k <= 5; k = k + 1)
    {
        cout << "The value of k is " << k << endl;
    }
    cout << "Good Bye." << endl;
    system("Pause");
    return 0;
}
```

# More on *for* loop

- Each section of the *for* loop (**initialization**, **boolean\_exp**, and **update\_action**) are optional
- If **initialization** is omitted, then no initialization is performed in the **for** loop; but it must be done before the loop
- If the **boolean\_exp** is omitted, then it is always assumed **true** and therefore may create an infinite loop
- If the **update\_action** is omitted, then no update action is performed; can be done inside the block

# More *for* loop examples

Analyze the following two programs and determine their outputs

```
int main()
{
    int i = 0; //Initialization outside the for loop declaration
    for (; i < 10;) //Initialization and update operations omitted
    {
        cout << "i = " << i << endl;
        i++; //update operation inside the for loop block
    }
    system("pause");
    return 0;
}
```

```
int main()
{
    int counter;
    for (counter = 5; counter >= 2;) //update operation omitted
    {
        cout << "counter = " << counter << endl;
        --counter; //update operation inside the for loop block
    }
    system("pause");
    return 0;
}
```

# More *for* loop examples

- It is possible to have multiple variable initializations and multiple variables updates in the for loop; all we need is separate them with commas
- Also it is possible to have a Boolean expression with multiple relational operations connected by logical operators
- Consider the following program and determine its output

```
int main()
{
    for (int i = 0, j = 10, z = 5; i < 10 && j > 0 && z > 0; i++, --j)
    {
        cout << i << "\t" << j << "\t" << z << endl;
        z--;
    }
    system("pause");
    return 0;
}
```

# The *do-while* loop

- **Syntax**

```
do  
{
```

Block of the *do-while* loop

```
} while (Boolean_expression); ← semicolon MUST!!!
```

- First the block is executed, then the Boolean expression is evaluated
- The block is executed as long as the Boolean expression is evaluated to true
- Therefore the block is always executed at least once

# The *do-while* loop

## Example

- Write a program that asks the user to enter a positive integer and then prints the positive integer entered.
- **Remark:-** Note that if the user enters a negative or zero inputs then your program must ask for the input again and again and again until the user enters a positive integer

---

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

int main()
{
    int x;
    do
    {
        cout << "Please enter a positive integer: ";
        cin >> x;
    }while (x <= 0);

    cout << "The positive integer you entered is " << x << endl;

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



# The *break* and *continue* statements

- C++ provides break and continue statements
- When these statements are inside loop constructs, then
  - The break statement forces the loop to terminate
  - The continue statement forces the statements inside the block of the loop which come after the continue statement to be skipped and go to the next iteration. In the case of **for** loop, it goes to the **update section** while in the case of the **while** and **do-while** loops, it goes to the **boolean expression**
- These statements alter the natural flow of execution of loop structures

# The *break* Statement

## Example

- The following program repeatedly reads user input integer numbers and prints them. The program stops when a negative integer number is read.

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

int main()
{
    int x;
    while (true)
    {
        cout << "Please enter any integer. Enter a negative integer to stop. ";
        cin >> x;
        if (x < 0)
            break;
        else
            cout << "You entered " << x << endl;
    }
    cout << "Finished reading and printing." << endl;
    system("Pause");
    return 0;
}
```

# The *continue* statement

## Example

- The following program repeatedly reads user input integer numbers and prints only the positive integers. Moreover the program stops when a negative integer number is read.

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

int main()
{
    int x;
    while (true)
    {
        cout << "Please enter any integer. Enter a negative integer to stop. ";
        cin >> x;
        if (x == 0)
            continue;
        else if (x < 0)
            break;
        else
            cout << "You entered " << x << endl;
    }
    cout << "Finished reading and printing." << endl;
    system("Pause");
    return 0;
}
```

# Nested Loops: Syntax

- Any loop structure can be put inside another loop structure. The general syntax is

*for , while or do-while loop* ← *outer loop*

{

Some C++ statements

*inner loop* → *for , while or do-while loop*

{

Some C++ statements

}

Some C++ statements

}

# Practice Question

- Write a C++ program that reads a positive integer user input **n** and then generates and prints **n** random integers each of which is in the range [10, 100]. In addition your program must print all the factors of each random integer generated as shown in the sample run output given below.

## Sample Run Output

```
How many random numbers do you want to generate? 5 ↵
Generated the random number 45. Its factors are 1 3 5 9 15 45
Generated the random number 20. Its factors are 1 2 4 5 10 20
Generated the random number 17. Its factors are 1 17
Generated the random number 12. Its factors are 1 2 3 4 6 12
Generated the random number 78. Its factors are 1 2 3 6 13 26 39 78
Press any key to continue...
```

# Practice Question

```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
    srand(time(0));
    int n;
    cout << "How many random numbers do you want to generate? ";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        int r = rand() % (100-10+1) + 10;
        cout << "Generated the random number " << r << ". Its factors are ";
        for (int k = 1; k <= r; k++)
        {
            if (r % k == 0)
                cout << k << " ";
        }
        cout << endl;
    }
    system("Pause");
    return 0;
}
```

# Programmer Defined C++ Functions

- Consider the following program designed to read three double data type user input numbers and then compute and print the area of a triangle whose sides have lengths equal to the three user input 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 there is no C++ built-in function named ***triangleArea*** that can compute and return the area of a triangle given the lengths of the sides of a triangle
- So what do we do? Well we have to **define** our own function!
- When we define our own function, we call it programmer (or user) defined function
- In order to define a programmer defined function correctly, we need to specify the following information
  - The return data type of the function,
  - The name of the function,
  - The parameters of the function, and
  - The body (block) of the function



# Programmer Defined C++ Functions

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

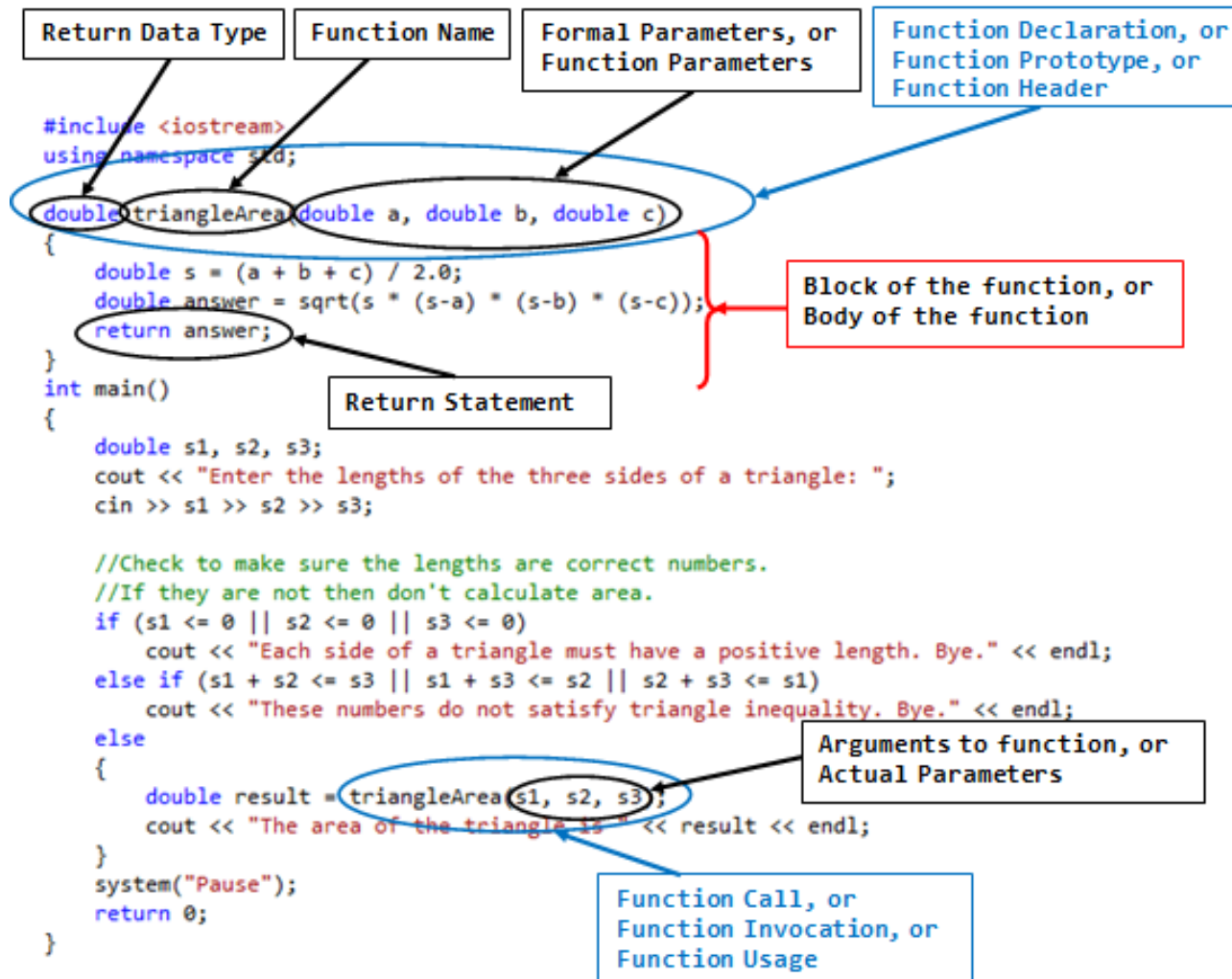
double triangleArea(double a, double b, double c)
{
    double s = (a + b + c) / 2.0;
    double answer = sqrt(s * (s-a) * (s-b) * (s-c));
    return answer;
}

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

## Useful Terminologies



# 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 an integer 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

## 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

```
#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 entered 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;
```

```
}
```

Function call stack

3.0	a
4.0	b
5.0	c
6.0	s
6.0	answer

Main stack

3.0	s1
4.0	s2
5.0	s3
6.0	result

# 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.

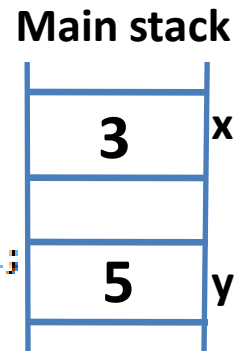
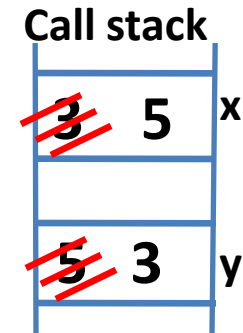
# Programmer Defined C++ Functions

## 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;
}
```



# 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(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;
}
```

# Function Declaration and Function Definition

- As shown in the previous examples, a programmer defined function is placed above the main program
- Alternatively, we may put only the function declaration above the main function and put the function definition below the main program
- When a function declaration is placed above the main program, it is treated as a statement and therefore is terminated by a semicolon

# Function Declaration and Function Definition

```
#include <iostream>
using namespace std;
bool isPrime(int x); ← function declaration
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(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 definition*

# 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(int x)*
- A C++ program can have more than one function with the same function name as long as each of the functions have different signatures
- 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 because 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;
}
```



# The const Qualifier

## Named Constants, Constant Variables, and Constant Parameters

- In C++,
  - A variable 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 but assigned a non literal value is known as a **constant variable**  
**Example:-** `const int z = rand();`  
*OR*  
`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 be 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

# The const Qualifier

- If a variable defined in a program, by design, is not required to be modified; then it is advisable to make the variable a named constant or a constant variable
- Similarly, if a function parameter is not required to be modified; then it is advisable to make the function parameter a constant parameter
- This helps to avoid any unintended modifications of the variables or the parameters
- Most importantly however this helps make code clearer and understandable and minimizes code debugging time

# The const Qualifier

## Named Constants, Constant Variables, and Constant Parameters

- Identify which variables are named constants, constant variables, or constant parameters in the program given below.

```
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;
}
```

# Global Variables

- A variable declared outside of any function block is known as a global variable
- Global variables are usually declared after the include directives but before any function definition
- The scope of global variables is global i.e. they can be accessed from within the main function as well as from within any other function definition
- Global variables are often used to store some constant information needed to be accessible in the entire part of the program
- See a schematic program given below

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

const int SIZE = 25;

RETURN_VALUE foo1(PARAMETERS)
{
    //We can access the variable SIZE here
}

RETURN_VALUE foo2(PARAMETERS)
{
    //We can access the variable SIZE here
}

int main()
{
    //We can access the variable SIZE here
    system("Pause");
    return 0;
}
```

# Static Variables

- C++ static variables are variables defined inside a function and that remain live after the function has finished execution
- This means static variables are defined only once and then the variables and their values are carried forward to the next function calls
- Any subsequent function calls will therefore skip the definition of the static variable
- The scope of static variables is only in the function block where they are defined
- Analyze the program shown below and determine its output

```
#include <iostream>
using namespace std;
void foo()
{
    static int count = 0;
    cout << "count = " << count << endl;
    count++;
}
int main()
{
    for (int i = 0; i < 5; i++)
        foo();
    system("Pause");
    return 0;
}
```