C++ PROGRAMMING

Programming Language

*A **programming language** is an artificial language that can be used to control the behavior of a machine, particularly a computer.*

Programming languages, like human languages, are defined through the use of syntactic and semantic rules, to determine structure and meaning respectively.

The term **syntax** refers to grammatical structure whereas the term **semantics** refers to the meaning of the vocabulary symbols arranged with that structure.

Programming languages are used to facilitate communication about the task of organizing and manipulating information, and to express algorithms precisely.

1.1.1 Levels of Programming Languages

There are three levels of programming languages

i. Machine language
ii. Assembly language
iii. High level language

Machine Language

This is the fundamental language of the computer's processor. It is the lowest level programming language understood by the computer.

While easily understood by computers, machine languages are almost impossible for humans to use because they consist entirely of binary numbers.

Assembly Language

A low level programming language that is similar to machine language. It uses symbolic operation code to represent the machine operation code needed to program a CPU architecture.

A utility program called an **assembler**, is used to translate assembly language statements into the target computer's machine code.

The assembler performs a more or less isomorphic translation (a one-to-one mapping) from mnemonic statements into machine instructions and data.

Example: Assembly language representation is easier to remember (more *mnemonic*)

mov al, 061h

This instruction means:
Move the hexadecimal value 61 (97 decimal) into the processor register named "al".

The mnemonic "mov" is an *operation code* or *opcode*, a comma-separated list of arguments or parameters follows the opcode;

High Level Language
Computer (programming) language that are easier to learn, because it uses English like statement.

High-level languages are used to solve problems and are often described as **problem-oriented languages**
Example: C++, Java, Visual Basic, Cobol, Fortran, PHP etc.

1.1.2 Compilers, Interpreters and Assemblers

A compiler:
Is a language translator that converts high level programs into machine understandable machine codes.

An interpreter:
Is a software that translates a high level language program into machine language.

An assembler
Is a software that converts programs written in assembly language into machine language.

Advantages of Compiler

- Reduced system load
- Protection for source code and programs
- Improved productivity and quality
- Portability of compiled programs

1.1.3 Source code and Object code

Source code:
Is the set of instructions and statements written by a programmer using a computer programming language.

It can be read and easily understood by a human being. This code is later translated into machine language by a compiler. The translated code is referred to as object code.
It contains declarations, instructions, functions, loops and other statements, which act as instructions for the program on how to function.

Object code:
Is a computer program after translation from source code usually into machine language by a compiler.

4.2 Introduction to C++ Program

4.2.1 General Structure of C++ Program

The basic elements of a program are the data declarations, functions, and comments. Let's see how these can be organized into a simple C++ program.

```
/*********************************************************
 * Heading comments                                      *
 *********************************************************/
data declarations
int main(  )
{
    executable statements
    return(0);
}
```

The heading comments tell the programmer all about the program.

The *data declarations* describe the data that the program is going to use.

Our single function is named `main`. The name `main` is special, because it is the first function called.

Following is an example of a basic syntax that will print the word Hello World

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

int main()
{
        cout <<"Hello Word"; // prints hello word
        return 0;
```

```
}
```

Let us look various parts of the above program

1. **#include**
   Is a preprocessor directive for C/C++ language which copies the code from the requested file to the given file just before compilation.

   # is a pound sign which helps to pre-process the program before the compilation & **include** is a simple directive that tells pre-processor to include the library's data (i.e. function declarations).

2. **iostream**
   Is the predefined library function used for input and output also called as header files. iostream is the header file which contains all the functions of program like cout, cin etc. and #include tells the preprocessor to include these header file in the program.

3. **< >**
   This character is used to indicate a standard language header file, in this case is iostream.

4. **using namespace std**
   Using namespace std means that you are going to use classes or functions (if any) from "std" namespace, so you don't have to explicitly call the namespace to access them.

5. **int main()**
   Is the main function where program execution begins. This will return integer value as return. All programs are called main in C++.

6. **{**
   character indicates the start of the program.

7. **cout <<"Hello Word";**
   This cause the word Hello World to be displayed on the screen

8. **<<**
   symbols are classified in C++ as an operator. They are used to separate items in the output stream.

9. **;**
   is the statement separator in C++

10. **}**
    character signifies the end of the program.

11. **return 0**
    This terminate main() function and cause it to return the value 0 to the calling process.

12. // print the hello world

This is the line of comment and always is not print.

Comments in C++

C++ supports both two types of comments

i. Line comments
ii. Block comments.

A line comment begins with two slashes (`//`) and continues to the end of the line on which it is placed. It might take up an entire line, or it might begin after some executable C++ code and take up the rest of the line.

A block comment begins with a single slash and an asterisk (`/*`) and ends with an asterisk and a slash (`*/`); it might be contained on a single line or continued across many lines.

4.2.2 Compile and run C++ program

After you write a C++ program you compile it; that is, you run a program called compiler that checks whether the program follows the C++ syntax

- if it finds errors, it lists them
- If there are no errors, it translates the C++ program into a program in machine language which you can execute

4.3 Data Type

Data Type is a particular kind of data item, as defined by the values it can hold, the programming language used, or the operations that can be performed on it.

4.3.1 Fundamental Data Type in C++

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.

Data types in C++ is mainly divided into three types:

1. Primitive Data Type e.g
   - Characters
   - Integers

- Floating point
- Boolean etc

2. Derived Data Type e.g
   - Function
   - Array
   - Pointer
   - Reference

3. Abstract or User-Defined Data Type e.g
   - Class
   - Structure
   - Union etc

Primitive Data Type

These data types are built-in or predefined data types and can be used directly by the user to declare variables. Example: int, char , float etc.

- **Integer**: Keyword used for integer data types is **int**. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.

- **Character**: Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.

- **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.

4.3.2 Variable Declaration

Declare characters, integers and floats

type variable-name;
Meaning: variable <variable-name> will be a variable of type <type>
Where type can be:
- int //integer
- float //real number
- char //character

Example of declaring variables

```
int a, b, c;
double x;
int sum;
char my-character;
```

4.4 Variable and Constant

Variable and constant are two commonly used mathematical concepts. Simply put, a variable is a value that is changing or that have the ability to change. A constant is a value which remains unchanged.

Variable:
What is a variable?
In C++ a variable is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

Notice that this is temporary storage. When you turn the computer off, these variables are lost. Permanent storage is a different matter. Typically, variables are permanently storted either to a database or to a file on disk.

Variable definition in C++

You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon.
Example
int i;

NOTE:
When declaring variable try to use expressive name such as my Age instead of i. Such name is easier to understand three weeks later when you are scratching your head trying to figure out what you meant when you wrote that line of code.

A variable definition means that the programmer writes some instructions to tell the compiler to create the storage in a memory location. The syntax for defining variables is:

Syntax:

```
data_type variable_name;
```

or

```
data_type variable_name, variable_name, variable_name;
```
Example:
```
int    width, height, age; /* variable definition */
char   letter;
float  area;
double d;
```

Variable Initialization in C++

Variables are declared in the above example, but none of them has been assigned any value. Variables can be initialized, and the initial value can be assigned along with their declaration.

Syntax
```
data_type variable_name = value;
```

Example
```
int    width, height=5, age=32; /* variable definition and initialization */
char   letter='A';
float  area;
double d;

/* actual initialization */width = 10;
area = 26.5;
```

There are some rules that must be in your knowledge to work with C++ variables.

Rules of Declaring variable in C++

- A variable name can consist of Capital letters A-Z, lowercase letters a-z, digits 0-9, and the underscore character.
- The first character must be a letter or underscore.
- Blank spaces cannot be used in variable names.
- Special characters like #, $ are not allowed.
- C++ keywords cannot be used as variable names.
- Variable names are case-sensitive.
- A variable name can be consisting of 31 characters only if we declare a variable more than one characters compiler will ignore after 31 characters.
- Variable type can be bool, char, int, float, double, void or wchar_t.

Here's Program to Show the Usage of Variable in C++

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

int main()
{
    int x = 5;
    int y = 2;
    int Result;
    Result = x * y;
```

```
    cout << Result;
}
```

Another program showing how Global variables are declared and used within a program:

Example

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

// Global Variable declaration:
int x, y;
float f;

int main()
{
    // Local variable
    int tot;
    float f;
    x = 10;
    y = 20;
    tot = x + y;

    cout << tot;
    cout << endl;
    f = 70.0 / 3.0;
    cout << f;
    cout << endl;
```

Constants
A constant, like a variable, is a memory location where a value can be stored. Unlike variables, constants never change in value. You must initialize a constant when it is created.

C++ has two types of constants: literal and symbolic.

A literal constant is a value typed directly into your program wherever it is needed. For example, consider the following statement:

```
long width = 5;
```

This statement assigns the integer variable `width` the value 5. The 5 in the statement is a literal constant. You can't assign a value to 5, and its value can't be changed.

The values `true` and `false`, which are stored in `bool` variables, also are literal constants.

A symbolic constant is a constant represented by a name, just like a variable. The `const` keyword precedes the type, name, and initialization.

Constant Definition in C++

There are two other different ways to define constants in C++. These are:

- By using *const* keyword
- By using *#define* preprocessor

Constant definition by using const keyword

It is similar to variable declaration except that we should add the keyword “const” prior to it. It is important to assign a value to the constant as soon as we declare it.

Syntax:

```
const type constant_name;
```

E.g

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

int main()
{
  const int SIDE = 50;
  int area;
  area = SIDE*SIDE;
  cout<<"The area of the square with side: " << SIDE <<" is: " << area <<
endl;
  system("PAUSE");
  return 0;
}
```

Program out

The area of the square with side: 50 is: 2500
Note:
It is possible to put const either before or after the type.

```
int const SIDE = 50;
```
or
```
const int SIDE = 50;
```

Constant definition by using #define preprocessor

Syntanx

```
#define constant_name;
```

Example

```
#include <iostream>
```

```
using namespace std;

#define VAL1 20;
#define VAL2  6;
#define Newline;

int main()
{
   int tot;
   tot = VAL1 * VAL2;
   cout << tot;
   cout << Newline;
}
```

Importance of Having Constants in Programs

i. **Constants** can make your **program** more readable.
   For example, you can declare: Const PI = 3.141592654. Then, within the body of your **program**, you can make calculations **that have** something to do with a circle.

ii. **Clarity:** Tells the user the significance of the number.
iii. **Maintainability:** Allow the program to be modified easily.
iv. **Safety:** Cannot be altered during program execution.

4.5 C++ Basic Input/output

In every program, there is some data which is taken as input and generate the processed data as reoutput following the input > process > output cycle.

Features of I/O in C++

- C++ IO is type safe
- C++ IO operations are based on streams of bytes and are device independent

The input-output system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as a stream.

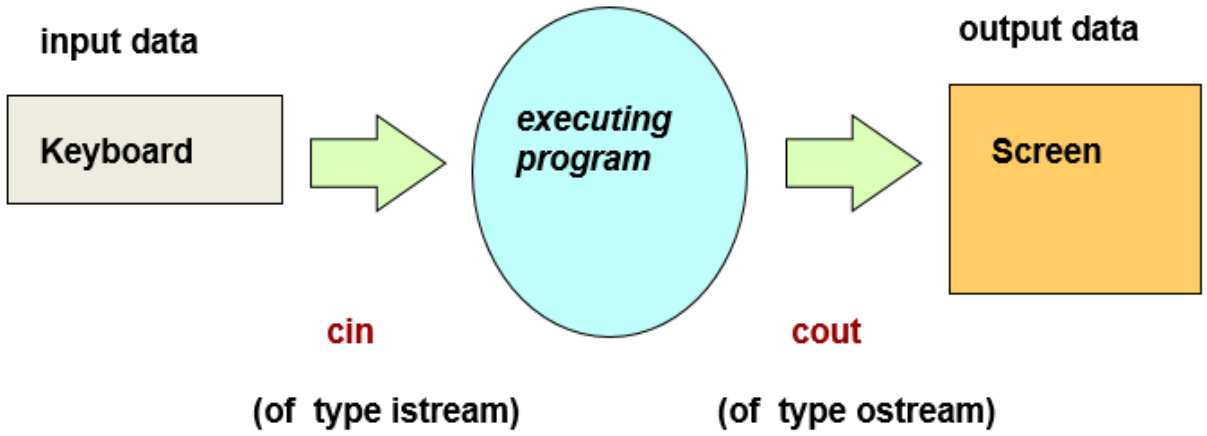
C++ treats input and output as a **stream** of characters.

The functions to allow standard I/O are in **iostream** header file or **iostream.h**.

A stream is a sequence of bytes which acts either as a source from which input data can be obtained or as a destination to which output data can be sent.

The source stream which provides data to the program is called the input stream and the destination stream which receives output from the program is called the output stream.

Keyboard and Screen I/O



Insertion Operator (<<)

Variable **cout** is predefined to denote an output stream that goes to the standard output device (display screen).
The insertion operator << called "put to" takes 2 operands.
The left operand is a stream expression, such as **cout**. The right operand is an **expression** of simple type or a **string constant**.

Syntax
cout << *Expression* << *Expression* . . . ;

cout statements can be linked together using << operator.
These examples yield the same output:
cout << "The answer is " ;
cout << 3 * 4 ;
cout << "The answer is " << 3 * 4 ;

Output Statements (String constant)
String constants (in double quotes) are to be printed as is, without the quotes.
cout<<"Enter the number of candy bars ";

OUTPUT: Enter the number of candy bars
"Enter the number of candy bars **"** is called a **prompt**.

- All user inputs must be preceded by a **prompt** to tell the user what is expected.
- You must insert **spaces** inside the quotes if you want them in the output.
- Do not put a string in quotes on multiple lines.

Output Statements (Expression)
All expressions are computed and then outputted.
E.g
cout << "The answer is " << 3 * 4 ;
OUTPUT: The answer is 12

Escape Sequences
The backslash is called the escape character.
It tells the compiler that the next character is "escaping" it's typical definition and is using its secondary definition.
Examples:

- newline: **\n**
- horizontal tab: **\t**
- backslash: \\
- double quote **\"**

Newline

- cout<<"\n" and cout<<endl both are used to insert a blank line.
- Advances the cursor to the start of the next line rather than to the next space.
- Always end the output of all programs with these statement**.**

Extraction Operator (>>)

- Variable **cin** is predefined to denote an input stream from the standard input device (the keyboard)
- The extraction operator >> called "get from" takes 2 operands. The left operand is a **stream expression**, such as **cin**--the right operand is a variable of simple type.
- Operator >> attempts to extract the next item from the input stream and store its value in the right operand variable.

Input Statements

Syntax

cin >> *Variable* >> *Variable* . . . ;

cin statements can be linked together using >> operator.
These examples yield the same output:

```
cin >> x;
cin >> y;
```

```
cin >> x >> y;
```

Example 1

```
#include <iostream>
using namespace std;
int main()
{
  int x;
  cout << "Type a number: "; // Type a number and press enter
  cin >> x; // Get user input from the keyboard
  cout << "Your number is: " << x;
  return 0;
}
```

Example 2

In this example, the user needs to input two numbers, and then we print the sum:

```
#include <iostream>
using namespace std;
int main()
{
  int x, y;
  int sum;
  cout << "Type a number: ";
  cin >> x;
  cout << "Type another number: ";
  cin >> y;
  sum = x + y;
  cout << "Sum is: " << sum;
  system("PAUSE");
  return 0;
}
```

4.6 Expressions and Assignments

Operators are symbols that tells the compiler to perform specific mathematical or logical manipulation. E.g +, *, /, =, == or <=. They are used to perform operations on variables and values.

Operands are those values on whichwe want to perform operation.

Expression is a combination of operands and operators.

Note:
Operators are characterized by the number of operands that they require.

Types of Operators
Following are types of operators used in C++

i. Arithmetic operators
ii. Assignment operators
iii. Comparison operators
iv. Logical operators
v. Bitwise operators

i. <u>Aritmetic Operators</u>

Arithmetic operators are used to perform common mathematical operations.
Examples of arithmetic operators are:-

- Addition operator (+)

  This adds together two values
  i.e
  x+y
  Consider the given program of addition operator

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << x + y;
  return 0;
}
```

  Its output is: 8

- Subtraction operator (-)

Subtract one value from another
i.e
x-y

Consider the given program of subtraction operator

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << x - y;
  return 0;
}
```

Its output is: 2

- Multiplication operator (*)
  This multiplies two values
  i.e
  x*y

  Consider the given program of multiplication operator

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << x * y;
  return 0;
}
```

  Its output is: 15

- Division operator (/)
  This divides one value from another
  i.e
  x/y

  Consider the given program of division operator

```
#include <iostream>
using namespace std;
int main() {
```

```
  int x = 12;
  int y = 3;
  cout << x / y;
  return 0;
}
```
Its output is: 4

- Modulus operator (**%**)
  This returns the division remainder
  i.e
  x%y

  Consider the given program of modulus operator
```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 2;
  cout << x % y;
  return 0;
}
```
  Its output is: 1

- Increment operator (++)
  This increase the value of a variable by 1
  i.e
  ++x

  Consider the given program of an increment operator
```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  ++x;
  cout << x;
  return 0;
}
```
  Its output is: 6

- Decrement operator (--)

This decrease the value of a variable by 1
i.e
--x

Consider the given program of decrement operator

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  --x;
  cout << x;
  return 0;
}
```

Its output is: 4

ii. <u>Assignment Operators</u>

Assignment operators are used to assign values to variables.
Following are list of all assignment operators

- Operator (=)

  i.e
  x = 5
  Consider the given program of assignment operator (=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  cout << x;
  return 0;
}
```

  Its output is: 5

- Operator (+=)

  i.e
  x += 3 Same As x = x + 3

  Consider the given program of assignment operator (+=)

```
#include <iostream>
using namespace std;
int main() {
```

```
 int x = 5;
 x += 3;
 cout << x;
 return 0;
}
```
Its output is: 8

- Operator (-=)

  i.e

  x -=3 Same As x = x – 3

  Consider the given program of assignment operator (-=)

```
#include <iostream>
using namespace std;
int main() {
 int x = 5;
 x -= 3;
 cout << x;
 return 0;
}
```
  Its output is: 2
- Operator (*=)

  i.e

  x *= 3 Same As x = x * 3

  Consider the given program of assignment operator (*=)

```
#include <iostream>
using namespace std;
int main() {
 int x = 5;
 x *= 3;
 cout << x;
 return 0;
}
```
  Its output is: 15

- Operator (/=)

  i.e

  x /= 3 Same As x = x / 3

Consider the given program of assignment operator (/=)

```
#include <iostream>
using namespace std;
int main() {
  double x = 5;
  x /= 3;
  cout << x;
  return 0;
}
```

Its output is: 1.66667

- Operator (**%=**)

i.e

x %= 3 Same As x = x % 3

Consider the given program of assignment operator (%=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  x %= 3;
  cout << x;
  return 0;
}
```

Its output is: 2

- Operator (**&=**)

i.e

x &= 3 Same As x = x & 3

Consider the given program of assignment operator (&=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  x &= 3;
  cout << x;
  return 0;
}
```

Its output is: 1

- Operator (|=)

  i.e

  x |= 3 Same As x = x | 3

  Consider the given program of assignment operator (|=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  x |= 3;
  cout << x;
  return 0;
}
```

  Its output is: 7

- Operator (^=)

  i.e

  x ^= 3 Same As x = x ^ 3

  Consider the given program of assignment operator (^=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  x ^= 3;
  cout << x;
  return 0;
}
```

  Its output is: 6

- Operator (>>=)

  i.e

  x >>= 3 Same As x = x >> 3

  Consider the given program of assignment operator (>>=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  x >>= 3;
```

```
  cout << x;
  return 0;
}
```
Its output is: 0

- Operator (<<=)

  i.e

  x <<= 3 Same As x = x << 3

  Consider the given program of assignment operator (<<=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  x <<= 3;
  cout << x;
  return 0;
}
```
Its output is: 40

iii. <u>Comparison Operators</u>

Comparison operators are used to compare two values. The return value is either true (1) or false (0).

Following are list of all comparison operators

- Equal to operator (==)

  i.e

  x == y

  Consider the given program of comparison operator (==)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << (x == y); // returns 0 (false) because 5 is not equal to 3
  return 0;
}
```
Its output is: 0

- Not equal (!=)

  i.e

x != y

Consider the given program of comparison operator (!=)

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

int main() {
  int x = 5;
  int y = 3;
  cout << (x != y); // returns 1 (true) because 5 is not equal to 3
  return 0;
}
```

Its output is: 1

- Greater than (>)

  i.e

  x > y

  Consider the given program of comparison operator (>)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << (x > y); // returns 1 (true) because 5 is greater than 3
  return 0;
}
```

  Its output is: 1

- Less than (<)

  i.e

  x < y

  Consider the given program of comparison operator (<)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << (x < y); // returns 0 (false) because 5 is not less than 3
  return 0;
```

```
}
```
Its output is: 0

- Greater than or equal to (>=)

  i.e

  x >= y

  Consider the given program of comparison operator (>=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << (x >= y); // returns 1 (true) because five is greater than, or equal, to 3
  return 0;
}
```

  Its output is: 1

- Less than or equal to (<=)

  i.e

  x <= y

  Consider the given program of comparison operator (<=)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << (x <= y); // returns 0 (false) because 5 is neither less than or equal to 3
  return 0;
}
```

  Its output is: 0

iv. <u>Logical Operators</u>

Logical operators are used to determine the logic between variables or values:

Following are list of all logical operators

- Logical and (**&&**)

  This returns true if both statements are true

  i.e

  x < 5 && x < 10

Consider the given program of logical operator (**&&**)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << (x > 3 && x < 10); // returns true (1) because 5 is greater than 3 AND 5 is less than 10
  return 0;
}
```

Its output is: 1

- Logical or (||)

  This returns true if one of the statements is true

  i.e

  x < 5 || x < 4

  Consider the given program of logical operator (||)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
  int y = 3;
  cout << (x > 3 || x < 4); /* returns true (1) because one of the conditions are true (5 is greater than 3, but 5 is not less than 4) */
  return 0;
}
```

  Its output is: 1

- Logical not (**!**)

  Reverse the result, returns false if the result is true

  i.e

  !(x < 5 && x < 10)

  Consider the given program of logical operator (**!**)

```
#include <iostream>
using namespace std;
int main() {
  int x = 5;
```

```
  int y = 3;
  cout << (!(x > 3 && x < 10)); // returns false (0) because ! (not) is used to reverse the result
  return 0;
}
```

Its output is: 0

Operator precedence

Just like math outside of computer programs, C++ has a defined operator precedence to allow the compiler to determine what to do when you have multiple operators in your expression.

4.7 Decisions

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

C++ handles **decision**-making by supporting the following statements, if statement. switch statement.

Conditions and If statement

C++ has the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

The if Statement

Use the if statement to specify a block of C++ code to be executed if a condition is true.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is true, print some text:

Example1

```
#include <iostream>
using namespace std;
int main() {
  if (20 > 18) {
    cout << "20 is greater than 18";
  }
  return 0;
}
```

Its output is: **20 is greater than 18**

We can also test variables:

Example2

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

int main() {
  int x = 20;
  int y = 18;
  if (x > y) {
    cout << "x is greater than y";
  }
  return 0;
}
```

Its output is: **x is greater than y**

Example explained

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the > operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

Syntax

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

Example

```
#include <iostream>
using namespace std;
int main() {
  int time = 20;
  if (time < 18) {
    cout << "Good day.";
  } else {
    cout << "Good evening.";
  }
  return 0;
}
```

Its output is: **Good evening.**

Example explained

In the example above, time (20) is greater than 18, so the condition is `false`. Because of this, we move on to the `else` condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

The else if Statement

Use the else if statement to specify a new condition if the first condition is false.

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and
condition2 is true
} else {
  // block of code to be executed if the condition1 is false and
```

```
condition2 is false
}
```
Example

```
#include <iostream>
using namespace std;
int main() {
  int time = 22;
  if (time < 10) {
    cout << "Good morning.";
  } else if (time < 20) {
    cout << "Good day.";
  } else {
    cout << "Good evening.";
  }
  return 0;
}
```
Its output is: **Good evening.**

Example explained

In the example above, time (22) is greater than 10, so the **first condition** is `false`. The next condition, in the `else if` statement, is also `false`, so we move on to the `else` condition since **condition1** and **condition2** is both `false` - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

C++ Switch Statements

Use the switch statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- The break and default keywords are optional, and will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
#include <iostream>
using namespace std;
int main() {
int day = 4;
switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;
}
```

Its output is: **Thursday**

The break Keyword

When C++ reaches a `break` keyword, it breaks out of the switch block.
This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.
A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

The `default` keyword specifies some code to run if there is no case match:

Example

```
#include <iostream>
using namespace std;
int main() {
  int day = 4;
  switch (day) {
    case 6:
      cout << "Today is Saturday";
      break;
    case 7:
      cout << "Today is Sunday";
      break;
    default:
      cout << "Looking forward to the Weekend";
  }
  return 0;
}
```

Its output is: Looking forward to the Weekend

Note: The default keyword must be used as the last statement in the switch, and it does not need a break.

4.8 Iteration

The statements that cause a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied are known as **iteration** statements. ...

There are main three types of **iteration** statements used in **C++.**

- while loop
- do while loop
- for loop

Loops can execute a block of code as long as a specified condition is reached.
Loops are handy because they save time, reduce errors, and they make code more readable.

While Loop

The while loop loops through a block of code as long as a specified condition is true:

Syntax

```
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
#include <iostream>
using namespace std;
int main() {
 int i = 0;
 while (i < 5) {
  cout << i << "\n";
  i++;
 }
 return 0;
}
```

Its output is:

```
0
1
2
3
4
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  // code block to be executed
}
while (condition);
```

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
#include <iostream>
using namespace std;
int main() {
  int i = 0;
  do {
    cout << i << "\n";
    i++;
  }
  while (i < 5);
  return 0;
}
```

Its output is:

0
1
2
3
4

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

Syntax

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example1

```
#include <iostream>
using namespace std;
int main() {
  for (int i = 0; i < 5; i++) {
    cout << i << "\n";
  }
  return 0;
}
```

Its output is:

```
0
1
2
3
4
```

Example explained

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Example2

This example will only print even values between 0 and 10:

```
#include <iostream>
using namespace std;
int main() {
  for (int i = 0; i <= 10; i = i + 2) {
    cout << i << "\n";
  }
  return 0;
}
```

Its output is:

```
0
2
4
6
8
10
```

4.9 Functions

***A function** is a block of code which only runs when it is called.*

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Advantages of Breaking Down Programs into Subprograms

Breaking programs into smaller, more manageable pieces has several **advantages**.

- First, breaking a program into smaller modules reduces the compile time. Very large programs can take quite a while.

- Second, it's easier to comprehend — therefore, easier to write, test and debug — a program that consists of a number of well-thought-out but quasi-independent modules, each of which represents a logical grouping of functions.

- Third is the much-vaunted specter of reuse. A module full of reusable functions that can be linked into future programs is easier to document and maintain.

The programmer can break a single program into separate source files generally known as ***modules***. These modules are compiled into machine code by the C++ compiler separately and then combined during the build process to generate a single program.

These modules are also known by compiler geeks as C++ translation units. The process of combining separately compiled modules into a single program is called ***linking.***

Create a Function

C++ provides some pre-defined functions, such as `main()`, which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**

Syntax

```
void myFunction() {
  // code to be executed
}
```

Where

- **myFunction()** is the name of the function
- **void** means that the function does not have a return value.
- inside the function (the body), add code that defines what the function should do

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

To call a function, write the function's name followed by two parentheses `()` and a semicolon `;`
In the following example, `myFunction()` is used to print a text (the action), when it is called:

Example1

```
#include <iostream>
using namespace std;
void myFunction() {
  cout << "I just got executed!";
}
int main() {
  myFunction();// call the function
  return 0;
}
```

Its output is: **I just got executed!**

Note:
A function can be called multiple times

Example2:

```
#include <iostream>
using namespace std;
void myFunction() {
  cout << "I just got executed!\n";
}
int main() {
  myFunction();
  myFunction();
  myFunction();
  return 0;
}
```

Its output is:
I just got executed!
I just got executed!
I just got executed!

Function Declaration and Definition

A C++ function consist of two parts:

- **Declaration:** the function's name, return type, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

Consider the syntax

```
void myFunction() { // declaration
  // the body of the function (definition)
}
```

Note:
If a user-defined function, such as **myFunction()** is declared after the **main()** function, **an error will occur**. It is because C++ works from top to bottom; which means that if the function is not declared above **main(),** the program is unaware of it:

Example

```
#include <iostream>
using namespace std;
int main() {
  myFunction();
  return 0;
}
void myFunction() {
  cout << "I just got executed!";
}
```

Its output is:

In function 'int main()':
'myFunction' was not declared in this scope 4 | myFunction();

However, it is possible to separate the declaration and the definition of the function - for code optimization.

You will often see C++ programs that have function declaration above **main(),** and function definition below **main().** This will make the code better organized and easier to read:

Example

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

// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  cout << "I just got executed!";
}
```

Its output is:

I just got executed!

C++ Functions Parameters

Parameters and Arguments

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

Syntax

```
void functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

The following example has a function that takes a string called **fname** as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example1

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

void myFunction(string fname) {
  cout << fname << " Refsnes\n";
}

int main() {
  myFunction("Liam");
  myFunction("Jenny");
  myFunction("Anja");
  return 0;
}
```

Its output is:

Liam Refsnes
Jenny Refsnes
Anja Refsnes

Note:

When a **parameter** is passed to the function, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `Liam`, `Jenny` and `Anja` are **arguments**.

Default Parameter Value

You can also use a default parameter value, by using the equals sign (`=`). If we call the function without an argument, it uses the default value ("Norway"):

Example

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

void myFunction(string country = "Norway") {
  cout << country << "\n";
}

int main() {
  myFunction("Sweden");
  myFunction("India");
  myFunction();
  myFunction("USA");
  return 0;
}
```

Its output is:

```
Sweden
India
Norway
USA
```

A parameter with a default value, is often known as an "**optional parameter**". From the example above, `country` is an optional parameter and `"Norway"` is the default value.

Multiple Parameters

You can add as many parameters as you want:

Example

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

void myFunction(string fname, int age) {
  cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}
```

Its output is:
Liam Refsnes. 3 years old.
Jenny Refsnes. 14 years old.
Anja Refsnes. 30 years old.

Note
When you are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The `void` keyword, used in the examples above, indicates that the function should not return a value.

If you want the function to return a value, you can use a data type (such as `int`, `string`, etc.) instead of `void`, and use the `return` keyword inside the function:

Example1

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

int myFunction(int x) {
  return 5 + x;
}

int main() {
  cout << myFunction(3);
```

```
  return 0;
}
```

Its output is: **8**

This example returns the sum of a function with **two parameters**:

Example2

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

int myFunction(int x, int y) {
  return x + y;
}

int main() {
  cout << myFunction(5, 3);
  return 0;
}
```

Its output is: **8**

You can also store the result in a variable:

Example3

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

int myFunction(int x, int y) {
  return x + y;
}

int main() {
  int z = myFunction(5, 3);
  cout << z;
  return 0;
}
```

Its output is: **8**

Pass By Reference

In the examples above, we used normal variables when we passed parameters to a function. You can also pass a reference to the function. This can be useful when you need to change the value of the arguments:

A reference variable is a "reference" to an existing variable, and it is created with the `&` operator:

Example1

```
string food = "Pizza";  // food variable
string &meal = food;    // reference to food
```

Now, we can use either the variable name `food` or the reference name `meal` to refer to the `food` variable:

Example2

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

int main() {
  string food = "Pizza";
  string &meal = food;

  cout << food << "\n";
  cout << meal << "\n";
  return 0;
}
```

Its output is
Pizza
Pizza

Example3

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

void swapNums(int &x, int &y) {
  int z = x;
  x = y;
  y = z;
}

int main() {
  int firstNum = 10;
```

```
  int secondNum = 20;

  cout << "Before swap: " << "\n";
  cout << firstNum << secondNum << "\n";

/* Call the function, which will change the values of firstNum and secondNum*/
  swapNums(firstNum, secondNum);

  cout << "After swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  return 0;
}
```

Its output is:

Before swap:
1020
After swap:
2010

Arrays and Strings

Arrays

Definition

An array is a collection of variables that are of similar data types and are alluded by a common name.

The Needs of Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

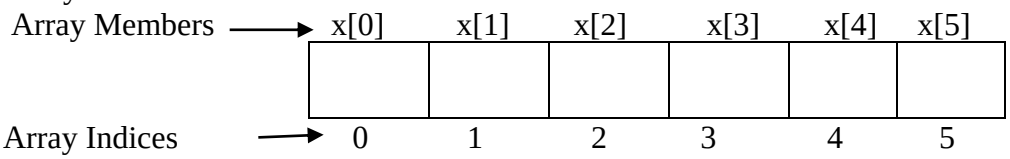
To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store:

Array Declaration (Syntax)
DataType arrayName [*arraySize*];

Access Element in C++ Array
In C++, each element in an array is associated with a number. The number is known as an array index. We can access elements of an array by using those indices.

```
int x[6];
```

Here,
int – type of element to be stored
x – Name of the array
6 – Size of the array (index)
Consider the array x



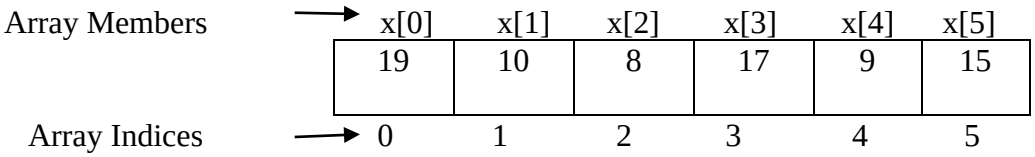
Things to remember:

- The array indices start with 0. Meaning x[0] is the first element stored at index 0.
- If the size of an array is n, the last element is stored at index (n-1). In this example. x[5] is the last element.

C++ Array Initialization

In C++, it is possible to initialize an array during declaration. For example

int x[6] = {19, 10, 8, 17, 9, 15};



Another method to initialize array during declaration

int x[ ] = {19, 10, 8, 17, 9, 15};

Here, we have not mentioned the size of the array. In such cases, the compiler automatically compute the size.

```
string cars[4];
```

We have now declared a variable that holds an array of four strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

C++ Array with empty members

In C++, if an array has a size n, we can store up to n number of elements in the array. However, what will happen if we store less than n number of elements.

For example

int x[6] = {19, 10, 8};

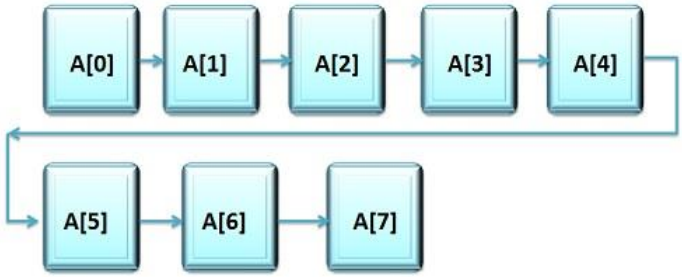
Here the array x has a size of 6. However, we have initialize it with only 3 elements. In such cases, the compiler assigns random values to the remaining places. Oftentimes, this random value is simply 0.

Array Members →	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
	19	10	8	0	0	0
Array Indices →	0	1	2	3	4	5

One and Two Dimensional Array in C++

One Dimension Array (1d)

A one-dimensional array is a list of variables with the same data type.



One-Dimensional Array A[8]

Declaration of 1D Array in C++

Syntax

type variable_name[size];

Here **type** declares the data type of array variable, and **size** defines the number of elements that array will hold.

For example, if we want to declare an array which will contain the balance of each month of the year.

int month_balance[12];

Month _balance is the array variable which will hold the 12 integers, which will represent the balance of each month.

Declaration of arrays in C++ can also be done in the following given manner,

```
int multiple[5] = {11,22,33,44,55};
char ex[10] = {'r', 'y', 't', 'i', 'e', 's', '\0'};
```

Accessing 1D array elements

Now, if we want to access the balance of month 'April', we simply had to mention the variable name followed by a square bracket containing the index value for the month of April.

'month_balance[3]'

But as 'April' is the fourth month of the year, we mentioned '[3]' (3 inside the square bracket) because all arrays have 0 as the index of their first element.

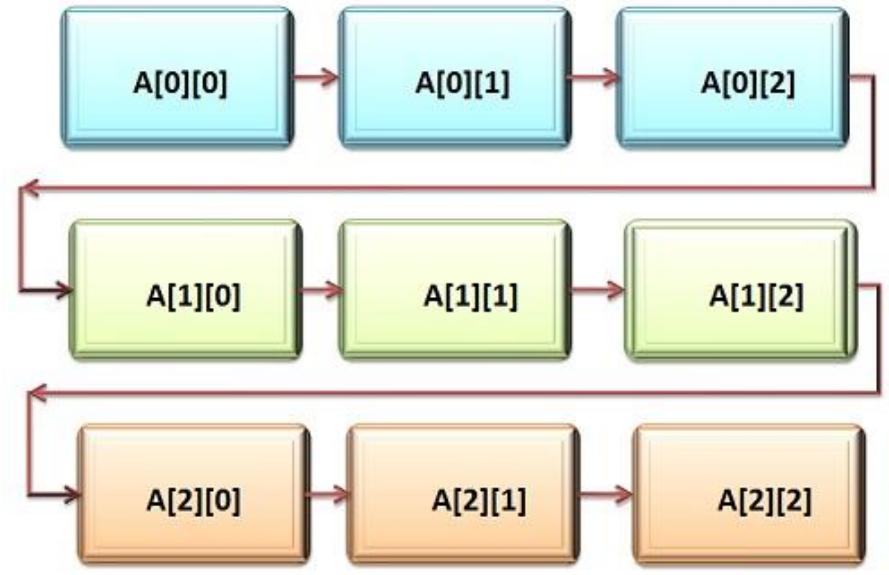
Note:
Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Two-Dimensional Array (2-D array)

A two-Dimensional array can be expressed as 'array of arrays' or 'array of one-dimensional arrays'.

To declare the two-dimensional array variable, we have to specify the array name followed by two square brackets where the second index is the second set of square brackets.

A two-dimensional array is stored in the form of the row-column matrix, where the first index designates the row and second index shows the column. The second or the rightmost index of an array alters very fast as compared to the first or left-most index while accessing the elements of an array.



Two-Dimensional Array A[3][3]

Declaration and Initialization of 2D Array in C++

In C++, the two-dimensional array is declared as;

Syntax

type variable_name[size1][size2];

For example, we want to store the balance of every 30 days in each month of the year, in a 2-D array.

Solution

```
int month_balance[12][30];
```

Initialization of 2D array is quite similar to the 1D array. The below-given example shows the 5×2 matrix of a 2D array.

int a[5][2] = { {0,2}, {1,4}, {2,6}, {3,8}, {4,10}};

Accessing 2D array elements

In order to access the entire 2D array in C++, we have to do looping against the rows and columns as shown below, then use "name_of_the_array[rows][coloumn]" for printing the elements of the array.

Example

```
cout<<"The elements of the Array is: \n";
for(i=0; i<row; i++)
{
for(j=0; j<col; j++)
{
cout<<arr[i][j]<<" ";
}
cout<<"\n";
}
```

Commands for String input

The Use of getline.cin input method

--The C++ **getline()** is a standard library function that is used to read a string or a line from an input stream.

--It is a part of the **<*string*> header file**. The getline() function takes two arguments;

- An input stream object (like cin)
- A string

--The function reads string data from the input stream into the string

The getline() function can be represented in two ways:

1. **Syntax:**
   Isstream& getline(istream& is, string& str, char delim);

   Parameters
   - **is:** It is an object of istream class and tells the function about the stream from where to read the input from.
   - **str:** It is a string object, the input is stored in this object after being read from the stream.
   - **delim:** It is the delimitation character which tells the function to stop reading further input after reaching this character.

2. Syntax:

Istream& getline (istream& is, string& str);

The second declaration is almost the same as that of the first one. The only difference is, the latter does not accept any delimitation character.

Parameters:

- **is:** It is an object of istream class and tells the function about the stream from where to read the input from.

- **str:** It is a string object, the input is stored in this object after being read from the stream.

Below program demonstrates the working of the getline() function

Example

```
// C++ program to demonstrate getline() function

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

int main()
{
    string str;

    cout << "Please enter your name: \n";
    getline(cin, str);
    cout << "Hello, " << str
         << " welcome to GfG !\n";

    return 0;
}
```

Input:

```
Harsh Agarwal
```

Output:

```
Hello, Harsh Agarwal welcome to GfG!
```

We can use getline() function to split a sentence on the basis of a character. Let's look at an example to understand how it can be done.

Example

```
// C++ program to understand the use of getline() function

#include <bits/stdc++.h>
using namespace std;

int main()
{
    string S, T;

    getline(cin, S);

    stringstream X(S);

    while (getline(X, T, ' ')) {
        cout << T << endl;
    }

   6qqqqq K return 0;
}
```

Input:

```
Hello, Faisal Al Mamun. Welcome to GfG!
```

Output:

```
Hello,
Faisal
Al
Mamun.
Welcome
to
GfG!
```

String Operations

String in-built-functions (cstring library)

The “cstring” library defines various string that can be used to perform various operations on strings.

Declaring a character array:

A character array can be declared as

char str[10];

This means that this is a character array i.e. a string with 10 characters.

Note:

This can also be represented as

char *str;

The following example is used for all the functions given below:

```
char str[30];
gets(str);
```

Assume that :C++ string functions" is given as input and str contains this string.

#include<cstring> needs to be written to use any of the following functions mentioned below

No	Functions	Description
1	Strlen (const char *str);	This function returns the length of the string str. The length of the string is the number of characters in the dtring without the terminating character "\0". e.g. cout<<"The length of string:"<<strlen(str); **Output:** The length of the string: 20
2	char*strcpy (char* dest, char*src);	This is a string copy function. The first parameter is the destination string and the second parameter is the source string. The function strcpy will copy the contents of source string to the destination string, including the terminating null-character "\0". e.g. char dest[30]; strcpy(dest, str); cout<<"Destination String:"<<dest; **Output:** Destination String: C++ string functions
3	Char* strcat (char* dest, const char*src);	This is the string concatenation function. The first parameter is the destination string and the second parameter is the source string. The function "**strcat()**" will concatenate/append the contents of source of array to the destination array. The terminating null-character of the destination array is overwritten by the first character of the source array and a null-character is introduced in the destination string at the end of new string. e.g. char dest[50]="Hello"; strcat(dest, str); cout<< "Destination String:"<<dest; Output: Destination String: HelloC++ string functions
4	int strcmp (char* str1, char*str2);	This function compares two strings. The first parameter is the string 1 and the second parameter is string 2 which will be compared by this function. It starts by comparing the first character of both the strings. It compares till it finds a non-matching character or the terminating null-character.

		It returns an integral value. The following is returned by this function.