



**RGPVNOTES.IN**

Program : **B.Tech**

Subject Name: **Basic Computer Engineering**

Subject Code: **BT-205**

Semester: **2nd**



**LIKE & FOLLOW US ON FACEBOOK**

[facebook.com/rgpvnotes.in](https://facebook.com/rgpvnotes.in)

## UNIT II

### Introduction to Algorithms,

An algorithm is a method for solving a computational problem. A formula or set of steps for solving a problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. Algorithms can be expressed in any language, from natural languages like English or French to programming languages like FORTRAN.

We use algorithms every day. For example, a recipe for baking a cake is an algorithm. Most programs, except for some artificial intelligence applications, consist of algorithms. Inventing elegant algorithms that are simple and require the fewest steps possible is one of the principal challenges in programming.

For example, we might be able to say that our algorithm indeed correctly solves the problem in question and runs in time at most  $f(n)$  on any input of size  $n$ .

**Definition.** An algorithm is a finite set of instructions for performing a computation or solving a problem.

**Types of Algorithms Considered.** In this course, we will concentrate on several different types of relatively simple algorithms, namely:

**Selection** -- Finding a value in a list, counting numbers;

**Sorting** -- Arranging numbers in order of increasing or decreasing value; and

**Comparison** -- Matching a test pattern with patterns in a database.

**Properties of Algorithms.** It is useful for an algorithm to have the following properties:

**Input** -- Values that are accepted by the algorithm is called input or arguments.

**Output** -- The result produced by the algorithm is the solution to the problem that the algorithm is designed to address.

**Definiteness** -- The steps in each algorithm must be well defined.

**Correctness** -- The algorithm must perform the task it is designed to perform, for all input combinations.

**Finiteness** -- Output is produced by the algorithm after a finite number of computational steps.

**Effectiveness** -- Each step of the algorithm must be performed exactly, infinite time.

**Generality** -- The procedure inherent in a specific algorithm should be applicable to all algorithms of the same general form, with minor modifications permitted.

### Complexities of Algorithm:

Algorithmic complexity is concerned about how fast or slow algorithm performs.

We define complexity as a numerical function  $T(n)$  - time versus the input size  $n$ . We want to define time taken by an algorithm without depending on the implementation details. But you agree that  $T(n)$  does depend on the implementation. A given algorithm will take different amounts of time on the same inputs depending on such factors as processor speed; instruction set, disk speed, a brand of compiler etc. The way around is to estimate the efficiency of each algorithm asymptotically. We will measure time  $T(n)$  as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two  $n$ -bit integers takes  $n$  steps. Consequently, the total computational time is  $T(n) = c * n$ , where  $c$  is time taken by addition of two bits. On different computers, the addition of two bits might take different time, say  $c_1$  and  $c_2$ , thus the addition of two  $n$ -bit integers takes  $T(n) = c_1 * n$  and  $T(n) = c_2 * n$  respectively. This shows that different machines result in different slopes, but time  $T(n)$  grows linearly as input size increases.

The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science.

We use algorithms to introduce salient concepts and to concentrate on the analysis of algorithm performance, especially computational complexity.

**Concept.** To facilitate the design of efficient algorithms, it is necessary to estimate the bounds on the complexity of candidate algorithms.

**Representation.** Complexity is typically represented via the following measures, which are numbered in the order that they are typically computed by a system designer:

**Work  $W(n)$**  -- How many operations of each given type are required for an algorithm to produce a specified output given  $n$  inputs?

**Space  $S(n)$**  -- How much storage (memory or disk) is required for an algorithm to produce a specified output given  $n$  inputs?

**Time  $T(n)$**  -- How long does it take to compute the algorithm (with  $n$  inputs) on a given architecture?

**Cost  $C(n) = T(n) \cdot S(n)$**  -- Sometimes called the space-time bandwidth product, this measure tells a system designer what expenditure of aggregate computational resources is required to compute a given algorithm with  $n$  inputs.

**Procedure. Analysis of algorithmic complexity generally proceeds as follows:**

Step 1. Decompose the algorithm into steps and operations.

Step 2. For each step or operation, determine the desired complexity measures, typically using Big-Oh notation, or other types of complexity bounds discussed below.

Step 3. Compose the component measures obtained in Step 2 via theorems presented below, to obtain the complexity estimate(s) for the entire algorithm.

Example. Consider the following procedure for finding the maximum of a sequence of numbers. An assessment of the work requirement is given to the right of each statement.

Let $\{a_n\} = (a_1, a_2, \dots, a_n)$	
{ max = $a_1$	# One I/O operation
for $i = 2$ to $n$ do:	# Loop iterates $n-1$ times
{ if $a_i \geq$ max then	# One comparison per iteration
max := $a_i$ }	# Maybe one I/O operation
}	

**Analysis: -**

(1) In the preceding algorithm, we note that there are  $n-1$  comparisons within the loop.

(2) In a randomly ordered sequence, half the values will be less than the mean, and  $a_1$  would be assumed to have the mean value (for purposes of analysis). Hence, there will be an *average* of  $n/2$  I/O operations to replace the value max with  $a_i$ . Thus, there are  $n/2 + 1$  I/O operations.

(3) This means that the preceding algorithm is  $O(n)$  in comparisons and I/O operations. More precisely, we assert that in the average case:

$W(n) = n-1$  comparisons +  $(n/2 - 1)$  I/O operations.

**Tractable** -- An algorithm belongs to class P, such that the algorithm is solvable in polynomial time (hence the use of P for *polynomial*). This means that complexity of the algorithm is  $O(n^d)$ , where  $d$  may be large (which typically implies slow execution).

**Intractable** -- The algorithm in question requires greater than polynomial work, time, space, or cost. Approximate solutions to such algorithms are often more efficient than exact solutions and are preferred in such cases.

**Solvable** -- An algorithm exists that generates a solution to the problem addressed by the algorithm, but the algorithm is not necessarily tractable.

**Unsolvable** -- No algorithm exists for solving the given problem. *Example:* Turing showed that it is impossible to decide whether a program will terminate on a given input.

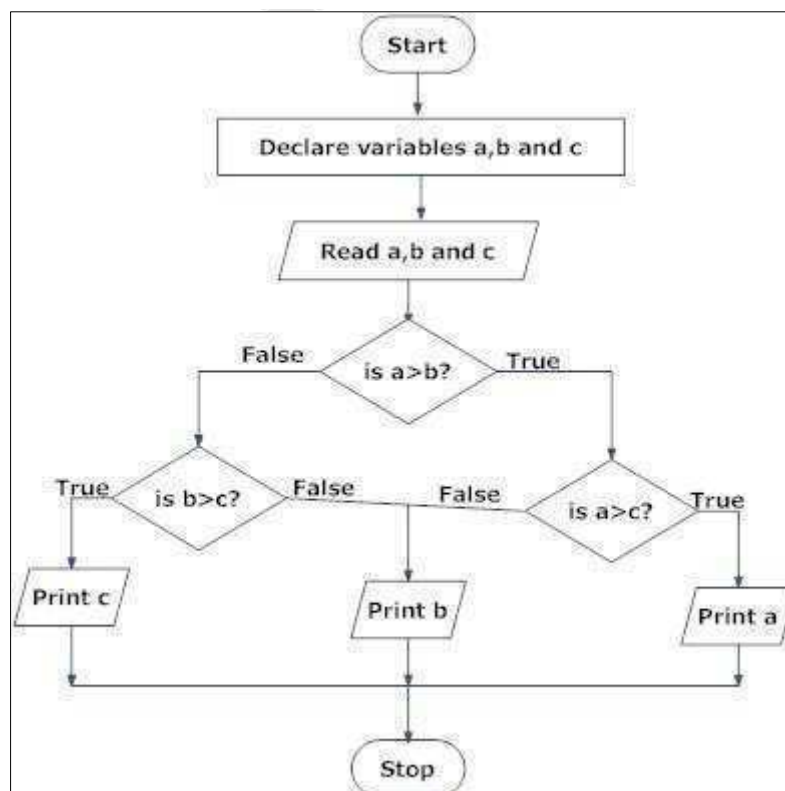
**Class NP** -- If a problem is in Class NP (non-polynomial), then no algorithm with polynomial worst-case complexity can solve this problem.

**Class NP-Complete** -- If any problem is in Class NP-Complete, then any polynomial-time algorithm that could be used to solve this problem could solve all NP-complete problems. Also, it is generally accepted, but not proven, that no NP-complete problem can be solved in polynomial time.

## FLOWCHART

1. The flowchart is a type of diagram (graphical or symbolic) that represents an algorithm or process.
2. Each step in the process is represented by a different symbol and contains a short description of the process step.
3. The flowchart symbols are linked together with arrows showing the process flow direction.
4. A flowchart typically shows the flow of data in a process.
5. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.
6. Flowcharts are generally drawn in the early stages of formulating computer solutions.
7. Flowcharts often facilitate communication between programmers and business people.

Example: - Draw a flowchart to find the largest among three different numbers entered by the user.



## Advantages of Using Flowcharts

1. Communication: a better way of logical communicating.








2. Effective analysis: the problem can be analyzed in more effective way.
3. Proper documentation.
4. Efficient Coding: a guide or blueprint during the systems analysis and program development phase.  
Proper Debugging: helps in debugging process.
5. Efficient Program Maintenance: easy with the help of flowchart.


### Limitations of Using Flowcharts

1. Complex logic: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. Alterations and Modifications: If alterations are required the flowchart may require redrawing completely.
3. Reproduction: As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

### Flowchart Symbols

1. Terminator: "Start" or "End"
2. Process
3. Decision: Yes/No question or True/False
4. Connector: jump in the process flow
5. Data: data input or output (I/O)
6. Delay
7. Arrow: flow of control in a process.

Symbol	Purpose	Description
	Flowline	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of the flowchart.
	Input/output	Used for input and output operation.
	Processing	Used for arithmetic operations and data-manipulations.
	Decision	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flowline
	Off-page Connector	Used to connect flowchart portion on a different page.

Symbol	Purpose	Description
	Predefined Process/Function	Used to represent a group of statements performing one processing task.

## Introduction to Programming

An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner. Without programs, computers are useless.

A program is like a recipe. It contains a list of ingredients (called *variables*) and a list of directions (called *statements*) that tell the computer what to do with the variables. The variables can represent numeric data, text, or graphical images.

There are many programming languages -- C, C++, Pascal, BASIC, FORTRAN, COBOL, and LISP are just a few. These are all high-level languages. One can also write programs in *low-level languages* called assembly languages, although this is more difficult. Low-level languages are closer to the language used by a computer, while high-level languages are closer to human languages.

## Categories of Programming Language

Programming languages fall into two fundamental categories low and high-level languages. Low-level languages are machine-dependent; that is, they are designed to be run on a particular computer. In contrast, high-level languages (for example, COBOL and BASIC) are machine-independent and can be run on a variety of computers.

The hierarchy of programming languages contain various types of programming languages. Through the first four decades of computing, programming languages evolved in generations. The first two generations were low-level and the next two high-level generations of programming languages.

The higher-level languages do not provide us greater programming capabilities, but they do provide a more sophisticated programmer/computer interaction. In short, the higher the level of the language, the easier it is to understand and use. For example, in a fourth-generation language, you need only instruct the computer system what to do, not necessarily how to do it.

## Characteristics of Programming language

The following are the characteristics of a programming language

1. **Readability:** A good high-level language will allow programs to be written in some ways that resemble a quite-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting.
2. **Portability:** High-level languages, being essentially machine independent, should be able to develop portable software.
3. **Generality:** Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become an expert in many diverse languages.

4. **Brevity:** Language should have the ability to implement the algorithm with less amount of code. Programs expressed in high-level languages are often considerably shorter than their low-level equivalents.
5. **Error checking:** Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.
6. **Cost:** The ultimate cost of a programming language is a function of many of its characteristics.
7. **Familiar notation:** A language should have a familiar notation, so it can be understood by most of the programmers.
8. **Quick translation:** It should admit quick translation.
9. **Efficiency:** It should permit the generation of efficient object code.
10. **Modularity:** It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.
11. **Widely available:** Language should be widely available and it should be possible to provide translators for all the major machines and for all the major operating systems.

### Generation of Languages

**1GL** or first-generation language was (and still is) machine language or the level of instructions and data that the processor is given to work on (which in conventional computers is a string of 0s and 1s).

**2GL** or second-generation language is assembler (sometimes called "assembly") language. A typical 2GL instruction looks like this:

```
ADD 12,8
```



An assembler converts the assembler language statements into machine language.

**3GL** or third-generation language is a "high-level" programming language, such as PL/I, C, or Java. Java language statements look like this:

```
public boolean handleEvent (Event evt) {
    switch (evt.id) {
        case Event.ACTION_EVENT: {
            if ("Try me" .equals(evt.arg)) {
```

A compiler converts the statements of a specific high-level programming language into machine language. (In the case of Java, the output is called bytecode, which is converted into appropriate machine language by a Java virtual machine that runs as part of an operating system platform.) A 3GL language requires a considerable amount of programming knowledge.

**4GL** or fourth-generation language is designed to be closer to natural language than a 3GL language. Languages for accessing databases are often described as 4GLs. A 4GL language statement might look like this:

```
EXTRACT ALL CUSTOMERS WHERE "PREVIOUS PURCHASES" TOTAL MORE THAN $1000
```



**5GL** or fifth-generation language is programming that uses a visual or graphical development interface to create source language that is usually compiled with a 3GL or 4GL language compiler. Microsoft, Borland, IBM, and other companies make 5GL visual programming products for developing applications in Java, for example. Visual programming allows you to easily envision object-oriented programming class hierarchies and drags icons to assemble program components.

### **Programming paradigms**

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.

#### **Common programming paradigms include:**

- Imperative which allows side effects,
- Functional which disallows side effects,
- Declarative which does not state the order in which operations execute,
- Object-oriented which groups code together with the state the code modifies,
- Procedural which groups code into functions,
- Logic which has a style of execution model coupled to a style of syntax and grammar, and
- Symbolic programming which has a style of syntax and grammar.

#### **Procedural oriented programming (pop):-**

A program in a procedural language is a list of instruction where each statement tells the computer to do something. It focuses on procedure (function) & algorithm is needed to perform the derived computation.

When the program becomes larger, it is divided into function & each function has clearly defined purpose. Dividing the program into functions & module is one of the cornerstones of structured programming.

E.g.:- c, basic, FORTRAN.

#### **Characteristics of Procedural oriented programming: -**

- It focuses on process rather than data.
- It takes a problem as a sequence of things to be done such as reading, calculating and printing. Hence, many functions are written to solve a problem.
- A program is divided into many functions and each function has clearly defined purpose.
- Most of the functions share global data.
- Data moves openly around the system from function to function.

#### **Drawback of Procedural oriented programming (structured programming):-**



- It's emphasis on doing things. Data is given a second-class status even though data is the reason for the existence of the program.
- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function. Similarly, if new data is to be added, all the function needed to be modified to access the data.
- It is often difficult to design because the components function and data structure do not model the real world.
- For example, in designing graphical user interface, we think what functions, what data structures are needed rather than which menu, menu item and soon.
- It is difficult to create new data types. The ability to create the new data type of its own is called extensibility. Structured programming languages are not extensible.

### Difference between Procedure Oriented Programming (POP) & Object-Oriented Programming (OOP)

Object-Oriented Programming	Procedure Oriented Programming	Points
In OOP, the program is divided into parts called objects.	In POP, the program is divided into small parts called functions.	Divided Into
In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.	In POP, Importance is not given to data but to functions as well as the sequence of actions to be done.	Importance
OOP follows Bottom Up approach.	POP follows Top-Down approach.	Approach
OOP has access specifiers named Public, Private, Protected, etc.	POP does not have any access specifier.	Access Specifiers
In OOP, objects can move and communicate with each other through member functions.	In POP, Data can move freely from function to function in the system.	Data Moving
OOP provides an easy way to add new data and function.	To add new data and function in POP is not so easy.	Expansion
In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.	In POP, the most function uses Global data for sharing that can be accessed freely from function to function in the system.	Data Access
OOP provides Data Hiding so provides more security.	POP does not have any proper way for hiding data so it is less secure.	Data Hiding
In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.	In POP, Overloading is not possible.	Overloading

Example of OOP are: C++, JAVA, VB.NET, C#.NET.	Example of POP are: C, VB, FORTRAN, Pascal.	Examples
--	---	----------

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see the whole world in the form of objects. For example, a car is an object which has certain properties such as color, the number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

**There are a few principle concepts that form the foundation of object-oriented programming:**

### **Object**

This is the basic unit of object-oriented programming. That is both data and functions that operate on data are bundled as a unit called as an object.

### **Class**

When you define a class, you define a blueprint for an object. This doesn't define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

### **Abstraction**

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. For example, a database system hides certain details of how data is stored and created and maintained.

### **Encapsulation**

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you a framework to place the data and the relevant functions together in the same object.

### **Inheritance**

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as a base class, a new class is formed called as the derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

### **Polymorphism**

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

### **Overloading**

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

### **Advantage of OOP**

- Simplicity: software objects model real-world objects, so the complexity is reduced and the program structure is very clear;
- Modularity: each object forms a separate entity whose internal workings are decoupled from other parts of the system;
- Modifiability: it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program since the only public interface that the external world must a class is using methods;
- Extensibility: adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones;
- Maintainability: objects can be maintained separately, making locating and fixing problems easier;

- Re-usability: objects can be reused in different programs.

## Introduction to C++

### History

The C++ programming language is an extension of C that was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides many features that "spruce up" the C language, but more importantly, it provides capabilities for object-oriented programming. A computer cannot understand our language that we use in our day to day conversations, and likewise, we cannot understand the binary language that the computer uses to do its tasks. It is, therefore, necessary for us to write instructions in some specially defined language like C++ which is like natural language and after conversing with the help of compiler the computer can understand it.

### Significant Language Features

Object-oriented programs are easier to understand, correct and modify. Many other object-oriented languages have been developed, including most notably, Smalltalk. The best features of C++ are:

- C++ is a hybrid language-it is possible to program in either a C-like style, an object-oriented style, or both.
- C++ programs consist of pieces called classes and functions. You can program each piece you may need to form a C++ program. The advantage of creating your own functions and classes is that you will know exactly how they work. You will be able to examine the C++ code.

### Areas of Application

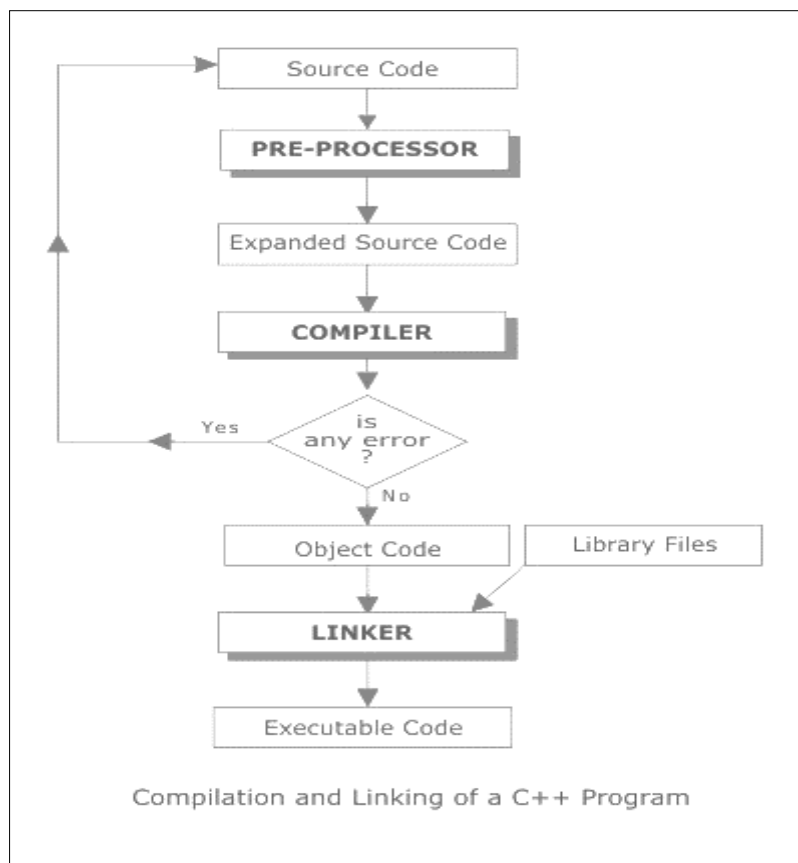
- For Develop Graphical related application like a computer and mobile games.
- To evaluate any kind of mathematical equation use C++ language.
- C++ Language is also used for design OS. Like window xp.
- Google also use C++ for Indexing
- Few parts of Apple OS X are written in C++ programming language.
- Internet browser Firefox is written in C++ programming language
- All major applications of Adobe systems are developed in C++ programming language. Like Photoshop, Image Ready, Illustrator, and Adobe Premier.
- Some of the Google applications are also written in C++, including Google file system and Google Chromium.
- C++ is used for design database like MySQL.

## C++ COMPILER

A C++ compiler is itself a computer program which's the only job is to convert the C++ program from our form to a form the computer can read and execute. The original C++ program is called the "**source code**", and the resulting compiled code produced by the compiler is usually called an "**object file**".

Before compilation, the **preprocessor** performs preliminary operations on C++ source files. A preprocessed form of the source code is sent to the compiler.

After compilation stage object files are combined with predefined libraries by a **linker**, sometimes called a binder, to produce the final complete file that can be executed by the computer. A library is a collection of pre-compiled "object code" that provides operations that are done repeatedly by many computer programs.



### Using Turbo C++ Com

If you are still willing to set up your environment for C++, you need following two software available on your computer.

#### Text Editor:

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows and vim or vi can be used on windows as well as Linux, or UNIX.

The files you create with your editor are called source files and for C++ they typically are named with the extension .cpp, .cp, or .c.

#### C++ Compiler:

This is an actual C++ compiler, which will be used to compile your source code into the final executable program.

Most C++ compilers don't care what extension you give your source code, but if you don't specify otherwise, many will use .cpp by default

Most frequently used and the free available compiler is GNU C/C++ compiler, otherwise, you can have compilers either from HP or Solaris if you have respective Operating Systems.

### C++ BASICS

#### C++ CHARACTER SET

The character set is a set of valid characters that a language can recognize.

<b>Letters</b>	A-Z, a-z
<b>Digits</b>	0-9

<b>Special Characters</b>	Space + - * / ^ \ ( ) [ ] { } = != <> ' " \$ , ; : % ! & ? _ # <= >= @
<b>Formatting characters</b>	backspace, horizontal tab, vertical tab, form feed, and carriage return

## TOKENS

A token is a group of characters that logically belong together. The programmer can write a program by using tokens. C++ uses the following types of tokens.

*Keywords, Identifiers, Literals, Punctuators, Operators.*

### 1. Keywords

These are some reserved words in C++ which have predefined meaning to compiler called keywords. Some commonly used Keyword are given below:

Asm	Auto	Break	case	catch
Char	Class	Const	continue	default
Delete	Do	Double	else	enum
Extern	inline	Int	float	for
friend	Goto	If	long	new
operator	private	Protected	public	register
return	Short	Signed	sizeof	static
struct	switch	Template	this	Try
typedef	union	Unsigned	virtual	void
volatile	While			

### 2. Identifiers

Symbolic names can be used in C++ for various data items used by a programmer in his program. A symbolic name is generally known as an identifier. The identifier is a sequence of characters taken from C++ character set. The rule for the formation of an identifier are:

- An identifier can consist of alphabets, digits and/or underscores.
- It must not start with a digit
- C++ is case sensitive that is upper case and lower-case letters are considered different from each other.
- It should not be a reserved word.

### 3. Literals

Literals (often referred to as constants) are data items that never change their value during the execution of the program. The following types of literals are available in C++.

- Integer-Constants
- Character-constants
- Floating-constants
- Strings-constants

**Integer Constants:** Integer constants are the whole number without any fractional part. C++ allows three types of integer constants.

**Decimal integer constants:** It consists of a sequence of digits and should not begin with 0 (zero). For example, 124, - 179, +108.

**Octal integer constants:** It consists of a sequence of digits starting with 0 (zero). For example. 014, 012.

**Hexadecimal integer constant:** It consists of a sequence of digits preceded by ox or OX.

#### Character constants

A character constant in C++ must contain one or more characters and must be enclosed in single quotation marks. For example, 'A', '9', etc. C++ allows nongraphic characters which cannot be typed directly from the keyboard, e.g., backspace, tab, carriage return etc. These characters can be represented by using an escape sequence. An escape sequence represents a single character. The following table gives a listing of common escape sequences.

Escape Sequence	Nongraphic Character
\a	Bell (beep)
\n	Newline
\r	Carriage Return
\t	Horizontal tab
\0	Null Character

#### Floating constants

They are also called real constants. They are numbers having fractional parts. They may be written in fractional form or exponent form. A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits. For example 3.0, -17.0, -0.627 etc.

#### String Literals

A sequence of characters enclosed in double quotes is called a string literal. String literal is by default (automatically) added a special character '\0' which denotes the end of the string. Therefore, the size of

the string is increased by one character. For example "COMPUTER" will re-represented as "COMPUTER\0" in the memory and its size is 9 characters.

#### 4. Punctuators

The following characters are used as punctuators in C++.

Brackets [ ]	Opening and closing brackets indicate single and multidimensional array subscript.
Parentheses ( )	Opening and closing brackets indicate functions calls, function parameters for grouping expressions etc.
Braces { }	Opening and closing braces indicate the start and end of a compound statement.
Comma,	It is used as a separator in a function argument list.
Semicolon ;	It is used as a statement terminator.
Colon :	It indicates a labeled statement or conditional operator symbol.
Asterisk *	It is used in pointer declaration or as a multiplication operator.
Equal sign =	It is used as an assignment operator.
Pound sign #	It is used as pre-processor directive.

**5. Operators:** Operators are special symbols used for specific purposes. C++ provides six types of operators. Arithmetical operators, Relational operators, Logical operators, Unary operators, Assignment operators, Conditional operators, Comma operator

#### DATA HANDLING

##### BASIC DATA TYPES

C++ supports many data types. The built-in or basic data types supported by C++ are an integer, floating point, and character. These are summarized in table along with description and memory requirement

Type	Byte	Range	Description
Int	2	-32768 to +32767	Small whole number
long int	4	-2147483648 to +2147483647	Large whole number
Float	4	3.4x10 <sup>-38</sup> to 3.4x10 <sup>+38</sup>	Small real number
double	8	1.7x10 <sup>-308</sup> to 1.7x10 <sup>+308</sup>	Large real number



long double	10	3.4x10-4932 to 3.4x10+4932	Very Large real number
char	1	0 to 255	A Single Character

## VARIABLES

It is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different times during the execution of a program.

To understand more clearly, we should study the following statements:

Total = 20.00; In this statement, a value 20.00 has been stored in a memory location Total.

### Declaration of a variable

Before a variable is used in a program, we must declare it. This activity enables the compiler to make available the appropriate type of location in the memory.

```
float Total;
```

You can declare more than one variable of the same type in a single statement

```
int x,y;
```

### Initialization of variable

When we declare a variable its default value is undetermined. We can declare a variable with some initial value.

```
int a = 20;
```

### Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user-defined data types. Following table lists down seven basic C++ data types:

Keyword	Type
Bool	Boolean
Char	Character
Int	Integer
Float	Floating point
Double	Double floating point
Void	Valueless
wchar_t	Wide character

Following is the example, which will produce the correct size of various data types on your computer.

```
#include<iostream>

using namespace std;

int main(){
cout<<"Size of char : "<<sizeof(char)<<endl;
cout<<"Size of int : "<<sizeof(int)<<endl;
cout<<"Size of short int : "<<sizeof(shortint)<<endl;
cout<<"Size of long int : "<<sizeof(longint)<<endl;
cout<<"Size of float : "<<sizeof(float)<<endl;
cout<<"Size of double : "<<sizeof(double)<<endl;
cout<<"Size of wchar_t : "<<sizeof(wchar_t)<<endl;
return0;
}
```

This example uses endl, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using sizeof() operator to get sizeof various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine –

```
Size of char: 1
Size of int: 4
Size of short int: 2
Size of long int: 8
Size of float: 4
Size of double: 8
Size of wchar_t: 4
```



### Type Qualifiers in C++

The type qualifiers provide additional information about the variables they precede.

Meaning	Qualifier
Objects of type const cannot be changed by your program during execution	const
The modifier volatile tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.	volatile
A pointer qualified by restricting is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qu alifier called restrict.	restrict

### STRUCTURE OF C++ PROGRAM

```
#include<header file>
main ()
{
.....
```

```

.....
.....
}

```

A C++ program starts with function called main ( ). The body of the function is enclosed in curly braces. The program statements are written within the braces. Each statement must end with a semicolon;(statement terminator). A C++ program may contain as many functions as required. However, when the program is loaded in the memory, the control is handed over to function main ( ) and it is the first function to be executed.

## Flow of control

### Statements

Statements are the instructions given to the computer to perform any kind of action. Action may be in the form of data movement, decision making etc. Statements form the smallest executable unit within a C++ program. Statements are always terminated by a semicolon.

### Compound Statement

A compound statement is a grouping of statements in which each individual statement ends with a semicolon. The group of statements is called block. Compound statements are enclosed between the pair of braces ({}). The opening brace ( { ) signifies the beginning and closing brace ( } ) signifies the end of the block.

### Null Statement

Writing only a semicolon indicates a null statement. Thus ';' is a null or empty statement. This is quite useful when the syntax of the language needs to specify a statement but the logic of the program does not need any statement. This statement is generally used in for and while looping statements.

## Conditional Statements

Sometimes the program needs to be executed depending upon a condition. C++ provides the following statements for implementing the selection control structure.

- if statement
- if else statement
- nested if statement
- switch statement

### if statement

The syntax of the if statement

```

if (condition)
{
    statement(s);
}

```

From the flowchart, it is clear that if the if the condition is true, the statement is executed; otherwise it is skipped. The statement may either be a single or compound statement.

### if else statement

syntax of the if - else statement

```

if (condition)
    statement1;
else
    statement2;

```

From the above flowchart, the given condition is evaluated first. If the condition is true, statement1 is executed. If the condition is false, statement2 is executed. It should be kept in mind that statement and statement2 can be a single or compound statement.

```

if example      if else example
if (x == 100)
    cout << "x is 100";  if (x == 100)
    cout << "x is 100";
else
    cout << "x is not 100";

```

### Nested if statement

The if block may be nested in another if or else block. This is called nesting of if or else block.

The syntax of the nested if statement

```

if(condition 1)
{
    if(condition 2)
    {
        statement(s);
    }
}
if(condition 1)
    statement 1;
else if (condition 2)
    statement2;
else
    statement3;

```

```

if-else-if example
if(percentage>=60)
    cout<<"Ist division";
else if(percentage>=50)
    cout<<"IIInd division";
else if(percentage>=40)
    cout<<"IIIrd division";
else
    cout<<"Fail" ;

```



### Switch statement

The if and if-else statements permit two-way branching whereas switch statement permits multiple branching. The syntax of the switch statement is:

```

switch (var / expression)
{
    case constant1: statement 1;
    break;
    case constant2: statement2;
    break;
    .
    .
    default: statement3;
    break;
}

```

The execution of switch statement begins with the evaluation of an expression. If the value of expression matches with the constant then the statements following this statement execute sequentially till it

executes break. The break statement transfers control to the end of the switch statement. If the value of expression does not match with any constant, the statement with default is executed.

Some important points about switch statement

- The expression of a switch statement must be of type integer or character type.
- The default case need not be used at last case. It can be placed at any place.
- The case values need not be in specific order.

### Flow of control

#### Looping statement

It is also called a repetitive control structure. Sometimes we require a set of statements to be executed many times by changing the value of one or more variables each time to obtain a different result. This type of program execution is called looping. C++ provides the following construct

- while loop
- do-while loop
- for loop

While loop

Syntax of while loop

```
while(condition)
{
    statement(s);
}
```

The flow diagram indicates that a condition is first evaluated. If the condition is true, the loop body is executed and the condition is re-evaluated. Hence, the loop body is executed repeatedly if the condition remains true. As soon as the condition becomes false, it comes out of the loop and goes to the state next to the 'while' loop.



do-while loop

Syntax of do-while loop

```
do
{statements;
} while (condition);
```

Note: That the loop body is always executed at least once. One important difference between the while loop and the do-while loop the relative ordering of the conditional test and loop body execution. In the while loop, the loop repetition test is performed before each execution the loop body; the loop body is not executed at all if the initial test fails. In the do-while loop, the loop termination test is Performed after each execution of the loop body. Hence, the loop body is always executed least once.

#### for loop

It is a count controlled loop in the sense that the program knows in advance how many times the loop is to be executed.

syntax of for loop for (initialization; decision; increment/decrement)

```
{
    statement(s);
}
```

In the For loop Three operation takes place

- Initialization of loop control variable

- Testing of loop control variable
- Update the loop control variable either by incrementing or decrementing.

Operation (i) is used to initialize the value. On the other hand, operation (ii) is used to test whether the condition is true or false. If the condition is true, the program executes the body of the loop and then the value of loop control variable is updated. Again, it checks the condition and so on. If the condition is false, it gets out of the loop

### Jump Statements

The jump statements unconditionally transfer program control within a function.

- goto statement
- break statement
- continue statement

### The goto statement

goto allows making the jump to another point in the program. goto pqr;

pqr: pqr is known as a label. It is a user-defined identifier. After the execution of goto statement, the control transfers to the line after label pqr.

### The break statements

The break statement, when executed in a switch structure, provides an immediate exit from the switch structure. Similarly, you can use the break statement in any of the loops. When the break statement executes in a loop, it immediately exits from the loop.

## ARRAY

An array is a collection of data elements of same data type. It is described by a single name and each element of an array is referenced by using array name and its subscript no.

### Declaration of Array

Type arrayname[numberofelements];

For example, int Age[5] ;

float cost[30];

### Initialization of One-Dimensional Array

An array can be initialized along with declaration. For array initialization, it is required to place the elements separated by commas enclosed within braces. int A[5] = {11,2,23,4,15}; It is possible to leave the array size open. The compiler will count the array size. int B[] = {6,7,8,9,15,12};

### Referring to Array Elements

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

name[index]

Examples: cout<<age[4]; //print an array element

age[4]=55; // assign value to an array element

cin>>age[4]; //input element 4

Using Loop to input an Array from user

int age [10], i ;

for (i=0 ; i<10; i++)

```
{
    cin>>age[i];
}
```

### Arrays as Parameters

At some moment, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass a complete block of memory by value as a parameter to a function, but we can pass its address.

For example, the following function: `void print(int A[])` accepts a parameter of type "array of int" called A.

In order to pass to this function, an array declared as `int arr[20]`; we need to write a call like this: `print(arr)`;

### Functions in C++

A function is a group of statements that together perform a specific task. Every C++ program has at least one function, which is `main ()`.

The function is used for dividing a large code into a module, due to this we can easily debug and maintain the code. For example, if we write calculator programs at that time we can write every logic in a separate function (For addition `sum ()`, for subtraction `sub()`). Any function can be called many times.

### Advantage of Function

- Code Re-usability
- Develop an application in module format.
- Easily to debug the program.
- Code optimization: No need to write a lot of code.

### Type of Function



There are two types of function in C++ Language. They are;

- Library function or pre-define function.
- User defined function.

### Library function

Library functions are those which are predefined in C++ compiler. The implementation part of pre-defined functions is available in library files that are `.lib/.obj` files. `.lib` or `.obj` files are contained pre-compiled code. `printf()`, `scanf()`, `clrscr()`, `pow()` etc. are pre-defined functions.

### Limitations of Library function

- All predefined function is contained limited task only that is for what purpose function is designed for the same purpose it should be used.
- As a programmer, we do not have any controls on predefined function implementation part is there in machine-readable format.
- In implementation whenever a predefined function is not supporting user requirement then go for user-defined function.

### User-defined function

These functions are created by a programmer according to their requirement, for example, suppose you



want to create a function for add two number then you create a function with name `sum()` this type of function is called a user-defined function.

### Defining a function.

Defining of the function is nothing but give the body of the function that means write logic inside the function body.

#### Syntax

```
return_type function_name(parameter)
{
    function body;
}
```

**Return type:** A function may return a value. The return type is the data type of the value the function returns. Return type parameters and returns statement are optional.

**Function name:** Function name is the name of the function it is decided by programmer or you.

**Parameters:** This is a value which is passed to function at the time of calling of function A parameter is a placeholder. It is optional.


**Function body:** Function body is the collection of statements.

### Function Declarations

A function declaration is a process of tells the compiler about a function name. The actual body of the function can be defined separately.

#### Syntax

```
return_type function_name(parameter);
```

**Note:** At the time of function declaration function must be terminated with ';'. 

### Calling a function.


When we call any function, control goes to function body and execute entire code. For a call, any function just writes the name of the function and if any parameter is required then pass a parameter.

#### Syntax

```
function_name();
```

or

```
variable=function_name(argument);
```

**Note:** At the time of function calling function must be terminated with ';'. 

### Example of Function in C++

```
#include<iostream.h>
```

```
Using namespace std;
```

```
void sum(); // declaring a function
```

```
int a=10,b=20, c;
```

```
void sum() // defining function
```

```
{
c=a+b;
cout<<"Sum: "<<c;
}
void main()
{

sum(); // calling function
}
```

Output

Sum: 30

## Function Arguments in C++

If a function takes any arguments, it must declare variables that accept the values as arguments. These variables are called the formal parameters of the function. There are two ways to pass value or data to function in C++ language which is given below;

- call by value
- call by reference

### Call by value

In call by value, **original value cannot be changed** or modified. In call by value, when you passed a value to the function it is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only but it does not change the value of a variable inside the caller function such as main().

Program Call by value in C++

```
#include<iostream.h>
#include<conio.h>
```

```
void swap(int a, int b)
{
int temp;
temp=a;
a=b;
b=temp;
}
```

```
void main()
{
int a=100, b=200;
clrscr();
swap(a, b); // passing value to function
cout<<"Value of a"<<a;
cout<<"Value of b"<<b;
getch();
}
```

Output

Value of a: 200

Value of b: 100

### Call by reference

In call by reference, the original value is changed or modified because we pass a reference (address). Here, the address of the value is passed to the function, so actual and formal arguments share the same address space. Hence, any value changed inside the function is reflected inside as well as outside the function.

Example Call by Reference in C++

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
void swap(int *a, int *b)
```

```
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
```

```
void main()
```

```
{
    int a=100, b=200;
    clrscr();
    swap(&a, &b); // passing value to function
    cout<<"Value of a"<<a;
    cout<<"Value of b"<<b;
    getch();
}
```

Output

Value of a: 200

Value of b: 100

### Difference Between Call by Value and Call by Reference.

call by Value	call by Reference
This method copies the original value into the function as arguments.	This method copies address of arguments into the function as an argument.
Changes made to the parameter inside the function have no effect on the argument.	Changes made to the parameter affect the argument. Because the address is used to access the actual argument.
Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

### Pointer in C++

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you

must declare a pointer before you can work with it. The general form of a pointer variable declaration is

```
type *var-name;
```

Here, the type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement, the asterisk is being used to designate a variable as a pointer.

Following are the valid pointer declaration –

```
int *ip; // pointer to an integer
double *dp; // pointer to a double
float *fp; // pointer to a float
char *ch // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

```
#include <iostream>
```

```
using namespace std;
```

```
int main () {
    int var = 20; // actual variable declaration.
    int *ip;      // pointer variable

    ip = &var;    // store address of var in pointer variable

    cout << "Value of var variable: ";
    cout << var << endl;

    // print the address stored in ip pointer variable
    cout << "Address stored in ip variable: ";
    cout << ip << endl;

    // access the value at the address available in pointer
    cout << "Value of *ip variable: ";
    cout << *ip << endl;

    return 0;
}
```



**RGPVNOTES.IN**

We hope you find these notes useful.

You can get previous year question papers at  
<https://qp.rgpvnotes.in> .

If you have any queries or you want to submit your  
study notes please write us at  
[rgpvnotes.in@gmail.com](mailto:rgpvnotes.in@gmail.com)



**LIKE & FOLLOW US ON FACEBOOK**

[facebook.com/rgpvnotes.in](https://facebook.com/rgpvnotes.in)