Introduction To Object Oriented Programming

Introduction

The goal of programmers is to develop software that are correct, reliable and maintainable, and satisfy the user requirements. The software development is not a static process, as the software need to be modified or redesigned according to the change in user requirements, business rules and strategies. To cope up with the dynamic nature and complexity of the software different approaches of programming were developed since the invention of the computer. These approaches are known as ***programming paradigms.***

A programming paradigms describes the structure of a program. In other words, it determines how the instructions are placed in a program. Each programming language follows one or the other programming paradigm. There is no specific rules to decide which programming paradigm is to be followed as different paradigms can be used to develop different software. However, whatever paradigm is followed, some of the quality issues of software must be kept in mind. This includes the correctness, usefulness, robustness, maintainability, reusability, portability, reliability and interoperability of the software.

Initially, when computers were invented, the binary language was used to write the programs. However, as programs grew in size, it became difficult to write programs using binary language. Then the assembly language was invented to write large programs. This language, however, was also not user friendly. With the change in the user requirements, the size and the complexity of the program continued to grow, which led to the development of high level language such as BASIC and FORTRAN.

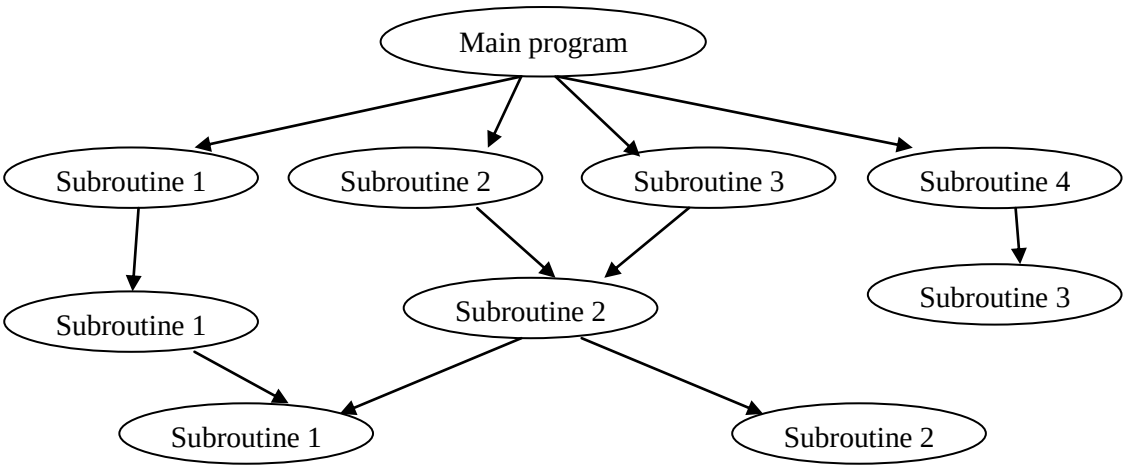
However, these languages provided an unstructured way of writing programs. In ***unstructured programming paradigm***, all the instructions of a program were written one after the other in a single function and hence were suitable for writing only small and simple programs. For large and complex programs, it became difficult to trace and debug errors.

To overcome the limitations of unstructured programming paradigm, other programming paradigm such as procedural, functional, modular, object-oriented, etc. were developed.

Structured Procedural Programming (SPP)

In late 1960s, the high level languages, such as C and Pascal were developed which provided a structured way of writing programs. Structured programming (also known as procedural programming) was a powerful and an easy approach of writing complex programs. In ***procedural programming,*** programs are divided into different procedures (also known as functions, routines or subroutines) and each procedure contains a set of instructions that performs a specific task. This approach follows top-down approach for designing the program. That is, first the entire program is divided into a number of subroutines and these subroutines are again divided into smaller subroutines and so on until each subroutine became an indivisible unit.

Programs in procedural programming consist of controlling procedural known as the ***main***, which controls the execution of other procedural. When a call to a procedure is made, the program controls is passed on to that procedure and all the instructions in that procedure are executed one after the other. After executing all the instructions, the program control returns to the procedural from where the call is made.



Through writing programs using programs using procedural programming paradigm is a simple and easy task, this approach has some limitations also.

1. The emphasis is on the functionality of the software rather than the data used by the function.
2. Procedural programming allows data to move freely from one function to another without any security.
3. In case the needs arises to modify the format of the data, all the functions accessing the data also need to be modified.
4. The procedural programming approach does not represent the real world entities very well as any real-world entity is not only characterized by the functions it performs but also by the properties it possesses.
5. The large programs developed using this approach are difficult to maintain, debug and extend.

Object Oriented Programming (OOP)

To overcome the limitation of procedural programming, object oriented programming paradigm has been developed which has revolutionized the process of software development. It not only includes the best features of structured programming but also includes some new and powerful features. These new OOP features have tremendously helped in the development of well-designed high quality software.

In Object Oriented programming, the programmers not only define the data, but also the operations that can be performed on it together under a single unit and thus, creating different kinds of variables known as objects. An OBJECT is a unit of structural and behavioural modularity that contains a set of properties as well as the associated function.

Features or Characteristics or features of OOP

1. OOP emphasizes on data rather than the functions or the procedures.
2. OOP models the real world very well by binding the data and associated functions together under a single unit.
3. Free movement of data from one function to another.
4. The data of one object can be accessed by the associated functions of that object only. Other functions are not allowed to access that data.
5. The objects of the entire system can interact with each other by sending messages to each other.
6. The programs written in OOP are easy to maintain and extend as new objects can be easily added to the existing system whenever required without modifying the other objects.
7. OOP follows the bottom-up approach for designing the programs. That is, first objects are designed and then these objects are combined to form the entire programs.

Basic Concepts of Object Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

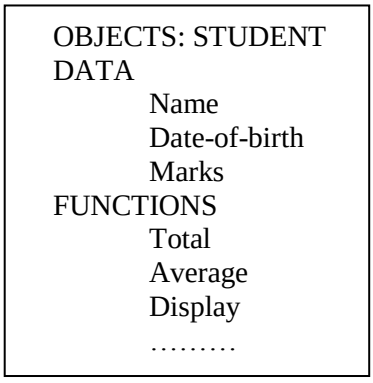
- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer" and "account" are to object in a program, then the customer object may send a message to the count object requesting for the bank balance. Each object contain data, and code to

manipulate data. Objects can interact without having to know details of each other's data or code. It is a sufficient to know the type of message accepted, and the type of response returned by the objects.



Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types.

For examples, Mango, Apple and orange members of class fruit. Classes are user-defined that types and behave like the built-in types of a programming language. The syntax used to create an object is not different then the syntax used to create an integer object in C. If fruit has been defines as a class, then the statement

Fruit Mango;

Will create an object **mango** belonging to the class **fruit.**

Data Abstraction and Encapsulation

The wrapping up of data and function into a single unit (called class) is known as ***encapsulation***. Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

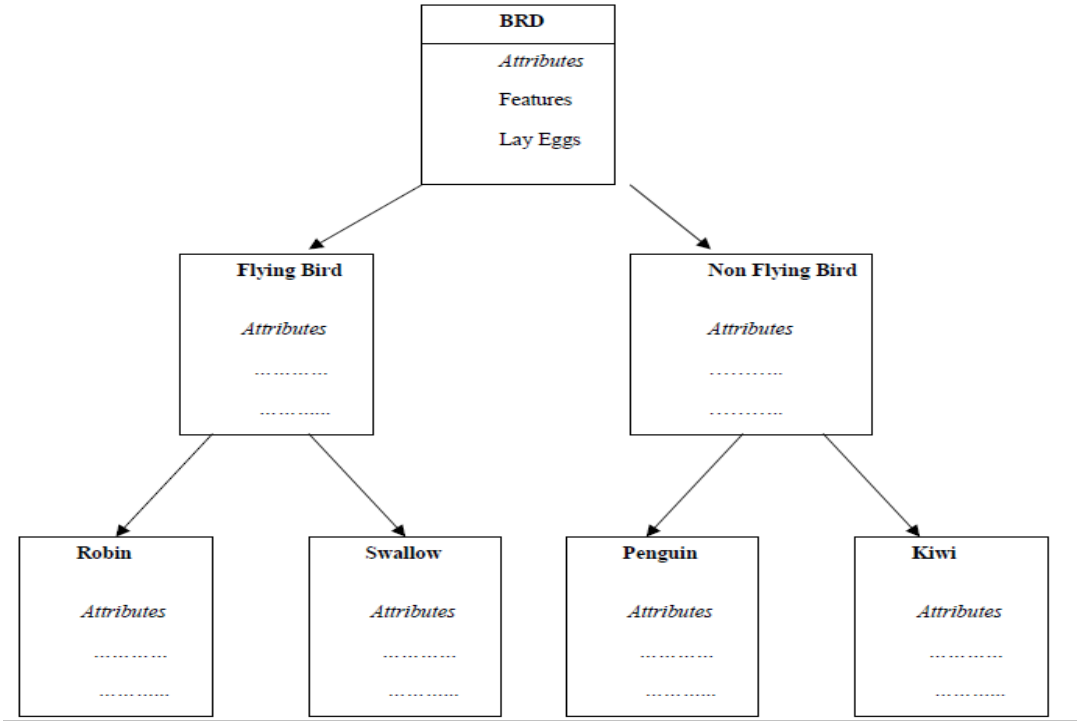
Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created.

The attributes are some time called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member function*.

Inheritance

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of *hierarchical classification.* For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in the following figure,

In OOP, the concept of inheritance provides the idea of *reusability.* This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes. The real appeal and power of the inheritance mechanism is that it Allows the programmer to reuse a class i.e almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduced any undesirable side-effects into the rest of classes.

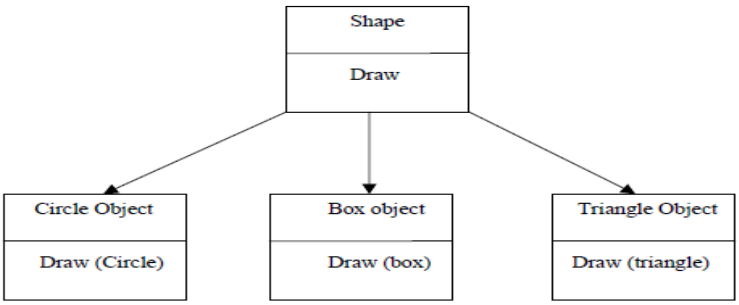


Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation.

For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as *operator overloading*.

Following figure illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as *function overloading.*



Polymorphism

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in above figure by inheritance, every object will have this procedure.

Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

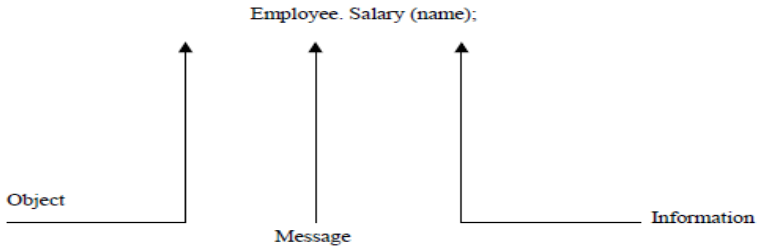
1. Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

1. Creating classes that define object and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. *Message passing* involves specifying the name of object, the name of the function (message) and the information to be sent. Example



Object has a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Benefits of OOP

OOP offers several benefits to both the program designer and the user. Object-Orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.

The principal advantages are:

- Through inheritance, we can eliminate redundant code extend the use of existing
- Classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure program that cannot be invaded by code in other parts of a programs.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map object in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more detail of a model can implemental form.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current product may be

superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

Disadvantages of object-oriented programming

1. Steep learning curve:

The thought process involved in object-oriented programming may not be natural for some people, and it can take time to get used to it. It is complex to create programs based on interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially.

2. Larger program size:

Object-oriented programs typically involve more lines of code than procedural programs.

3. Slower programs:

Object-oriented programs are typically slower than procedure based programs, as they typically require more instructions to be executed.

4. Not suitable for all types of problems:

There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

Difference between Structured Programming and Object Oriented Programming

Structured Programming	Object Oriented Programming
1. Structured Programming is designed which focuses on process/ logical structure and then data required for that process.	1. Object Oriented Programming is designed which focuses on data.
2. Structured programming follows top-down approach.	2. Object oriented programming follows bottom-up approach.
3. Structured Programming is also known as Modular Programming and a subset of procedural programming language.	3. Object Oriented Programming supports inheritance, encapsulation, abstraction, polymorphism, etc.
4. In Structured Programming, Programs are divided into small self contained functions.	4. In Object Oriented Programming, Programs are divided into small entities called objects.
5. Structured Programming is less secure as there is no way of data hiding.	5. Object Oriented Programming is more secure as having data hiding feature.
6. Structured Programming can solve moderately complex programs.	6. Object Oriented Programming can solve any complex programs.
7. Structured Programming provides less reusability, more function dependency.	7. Object Oriented Programming provides more reusability, less function dependency.
8. Less abstraction and less flexibility.	8. More abstraction and more flexibility.

Application of OOP

Object oriented paradigm has touched many major application areas of software development. Some of the application areas where OOP has been used to develop software are listed here.

1) **Simulation and modeling:** Simulation is the technique of representing the real world entities with the help of a computer program. Simula-67 and Smalltalk are two object oriented languages which are designed for making simulation.
2) **User-interface design**: Another popular application of OOP has been in the area of designing graphical user interface such as windows.

3) **Developing computer games:** OOP is also used for developing computer games such as Diablo, Startcraft and warcraft III. These games offer virtual reality environments in which a number of objects interact with each other in complex ways to give the desired result.
4) **Scripting:** in recent years, OOP has also been used for developing HTML, XHTML and XML document for the internet.
5) **Object database:** these days OOP concepts have also been introduced in database systems to develop a new DBMS named object databases.

Some other areas of application include office automation systems, real-time system, decision support system, artificial intelligence (AI) and expert system, neural networks and parallel programming, and computer and computer aided design (CAD) system

Object Oriented Languages

The high level languages that implements the concepts of object-oriented programming is known as an object oriented language also called an OO language. In general an object oriented language must support all or some of these OO concept.

1. Encapsulation and data hiding.
2. Inheritance
3. Polymorphism and dynamic binding
4. All built in and user defined data types are objects
5. All operations are performed using the message passing techniques.

Depending on the extent to which they support OO concepts, the OO language are classified into several categories which are listed below:

1) **Pure languages :** Languages that not only support but also enforce all object oriented concepts are called pure OO languages.
2) **Hybrid languages :** Languages that support some of the OO concepts are called hybrid languages. Java, Python, and C# are examples of hybrid language.
3) **Multi-Paradigm language** : Languages that support many programming paradigm, one of which is object oriented paradigm are called multi-paradigm languages. C++ is the example of multi paradigm language.
4) **Object based languages :** Languages that support the concept of abstract data types and also other OO concepts like encapsulation, data hiding and operator overloading are known as object based language. However, these languages do not support the concept of inheritance and dynamic binding. Ada and Module-2 are example of object based languages.

Introduction to C++

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented version of C.
C+ + is a superset of C. Almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ complier.

The most important facilities that C++ adds on to C care classes, inheritance, function overloading and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

Comparison between C++ and C

C	C++
1) scanf and printf are used for performing input/output operation and are specified in the header file <stdio.h>	1) cin and cout are used for performing input/output operation and are specified in the header file <iostream.h>. In addition, by using cin.form and cin.scan, the functionality of printf and scanf of C are retained.
2) Only multiline comments (/*……..*/) are supported	2) Both single line (//……..) and multiline comments (/*…..*/) are supported.
3) No extra space is included for terminating character '\0' while initializing the size of character array	3) An extra space is included for the terminating character '\0' while initializing the size of character array.
4) Structure and union contain only data	4) Structures and unions can include functions along with data.
5) Only int values can be assigned to the enum variable.	5) C++ does not permit to assign int values to enum variables. Only a valid enumerator can be assigned to an enum variable.
6) enum declared within a structure has a global scope.	6) enum declared within a structure or a class has a local scope.
7) Only traditional style cast is supported.	7) Four casting operators, namely const_cast, dynamic_cast, static_cast, and reinterpret_cast in addition to traditional style cast are supported.
8) While declaring a function with no arguments, the use of keyword 'void' in the parameter list is mandatory.	8) While declaring a function with no arguments, the use of keyword 'void' in the parameter list is optional.
9) malloc () and free() functions are used for managing memory.	9) New and delete operators are used for managing memory.
10) It follows top-down approach for programming.	10) It follows bottom-up approach for programming.

Steps in Developing (OOP) C++ programs

Generally, C++ programs are created by writing instructions with the help of a text editor. Once the instruction are written and saved in a file known as the source file, the file is compiled and then executed to get the required output. A compiler translate the program written in high level language to the machine language.

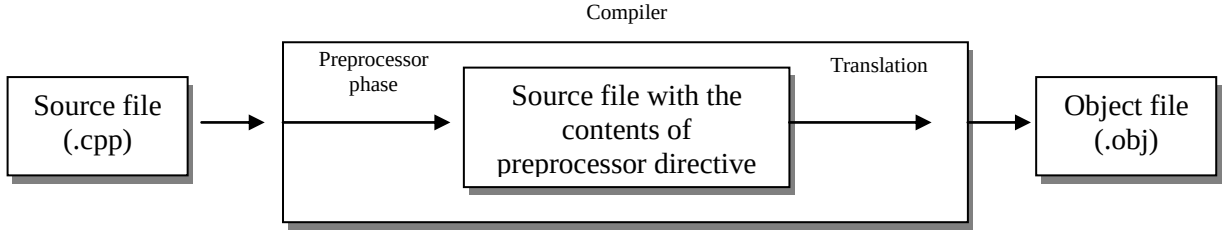
All the three steps, namely **writing the source code**, **compiling the programs** and **linking and executing the programs** are discussed below.

1) Writing the Source Code:

Source code can be defined as the instructions that are written in a source file using the text editor. Text editor are the software that allows creating and editing source files. Now a days, there are several text editor available for writing programs. For example, C++ compilers do not require a separate text editor, rather than they provide interface known as Integrated Development Environment (IDE) to create, edit, compile, link, execute and debug C++ programs.

2) Compiling the programs:

During compilation, a source code written in high level language is translated into its equivalent object code in machine language. However, before translation, the preprocessor phase is executed. The preprocessor phase is not an autonomous phase, it is a part of compilation and occurs each time a source file is compiled.



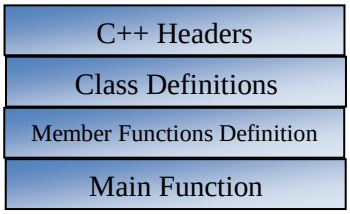
Preprocessor directives are the instructions used to include headers, define symbolic constants, provide conditional compilation, etc. A preprocessor directive is preceded by a '#' and must appear on a single independent line, generally at the beginning of the source file. After the preprocessor phase, the source code is translate into its machine language code. This machine language code, also known as the object code, is stored in an object file that has the **.obj** extension.

3) Linking and executing the program:

Though the source file is translated into its equivalent object file after compilation, it cannot be executed. This is because the definitions of certain pre-defined programming elements like library functions are not present in the headers. The definitions of such programming elements are stored in the form of their object code in the standard library. Thus, to create an executable program, the file containing the object code of these definitions must be linked with the object file. The executable program is stored in a file that has **.exe** extension. This executable file is loaded into the primary memory before execution and the process is known as **loading.**

Structure of Object Oriented Programs (Structure of C++ Programs)

Programs are a sequence of instruction or statements. These statements form the structure of a C++ program. C++ program structure is divided into various sections, namely *header*, *class definition, member functions definitions* and *main function.*



Structure of a C++ Programs

A simple C++ Program

```
//A simple C++ Program          ⟶ This is a comment and is ignored by the compiler
#include <iostream.h>           ⟶ Header
Using namespace std;            ⟶ This tells the compiler to use std namespace
int main( )                     ⟶ main function
{

        cout<< "First C++ Programs"; ⟶ This is the body of main().
        return 0;                    It contains the executable code.
}
```

Comments

Comment are a vital element of a program that is used to increase the readability of a program and to describe its functioning. Comments are not executable statements and, here, do not increase the size of a file.

C++ supports two comment styles: single line comment and multiline comment. Single line comments are used to define line-by-line descriptions. Double slash (//) is used to represent single line comments.

To understand the concept of single line comment,

//An example to demonstrate
// single line comment.

Multiline comments are used to define multiple lines descriptions and are represented as /* …… */

For example, consider this statement

/ An example to demonstrate*
*Multiline comment */*

Headers

A program includes various programming elements lie built in functions, classes, keywords, constants, operators, etc. that are already defined in the standard C++ library. In order to use such pre-defined elements in a program, an appropriate header must be included in the program. The standard headers contain information like prototype, definition and return type of library functions, data type of constants etc. as a result, programmers do not need to explicitly declare the pre-defined programming elements. Standard headers are specified in a program through the preprocessor directive '#include'. In the iostream header is used. When the compiler processes the instruction '#include<iostream>, it includes the contents of iostream in the program. This enables the programmer to use standard input, output and error facilities that are provided only through th standard stream.

The standard streams defined in <iostream> are listed below

cin (pronounced "see in") :

It is the standard input stream that is associated with the standard input device (keyboard) and is used to take the input from users.

cout (pronounced "see out"):

It is the standard output stream that is associated with the standard output device (monitor) and is used to display the output to users.

cerr (pronounced "see err"):

It is the standard error stream that is associated with the standard error device (monitor) and is used to report errors to the users. The cerr object does not have a buffer (temporary storage area) and hence, immediately reports errors to users.

clog (pronounced "see log"):

It is the buffered error stream that is associated with the standard error device (monitor) and is used to report errors to users. Unlike cerr, clog reports errors to users only when the buffer is full.

Namespace

Since its creation, C++ has gone through many changes by the C++ standards committee. One of the new features added to the language is namespace. A namespace permits grouping of various entities like classes, objects, functions and various C++ tokens, etc.

The main function

The main() is a startup function that starts the execution of a C++ program. All C++ statements that need to be executed are written within main (). The compiler executes all the instructions written within the opening and closing curly braces '{ }' that enclose the body of main (). Once all the instructions in main () are executed, the control passes out of main () terminating the entire program and returning a value to the operating system.

By default, main() in C++ returns an int value to the operating system. Therefore, main () should end with the return 0 statement. A return value zero indicates success and non-zero value indicates failure or error.

The Input / Output Operator

In C++, input and output (I/O) operator are used to take input and display output. The operator used for taking the input is known as **the extraction or get from operator (>>),** while the operator used for displaying the output is known as **the insertion or put to operator (<<)**

Input Operator

The input operator, commonly known as the **extraction operator (>>),** is used with the standard input stream, **cin** which treats data as a stream of characters. These character flow from cin to the program through the input operator. The input operator works on two operands, namely the cin stream on its left and a variable on its right. Thus the input operator takes (extracts) the value through cin and stores it in the variable.

For example;

```
cin >> a;
```

Output Operator

The output operator, commonly known as the **insertion operator (<<),** is used with the standard output stream **cout**. Like cin, cout also treats data as stream of characters. These characters flow from the program to cout through the output operator. The output operator works on two operands, namely the cout stream on its left and the expression to be displayed on its right. The output operator directs (insert) the value to cout.

For example;

```
cout << a;
```

Cascading of Input / Output Operator

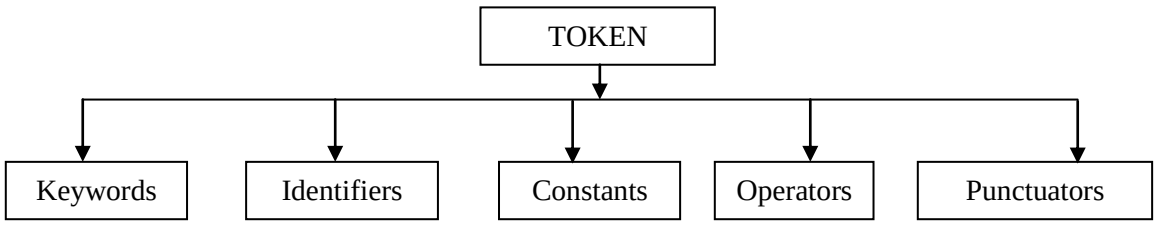
The cascading of the input and output operator refers to the consecutive occurrence of input or output operators in a single statement.

For example;

```
cin >> a >> b;
cout << "The value of a is " << a;
cout << "The value of b is " << b;
```

TOKENS

A Token is defined as the smallest unit of program, when a program is compiled the compiler scans the source code and parses it into tokens to find syntax errors, C++ token are broadly classified into keywords, identifiers, constants, operators and punctuators.



1) Keywords

Keywords are the predefined words that have special significance in any language. Every keyword is reserved for a specific purpose and, hence, cannot be used as user defined names (identifiers). All the keywords of C++ are listed below.

keywords						
asm	const_cast	export	inline	public	static_cast	typename
auto	continue	explicit	int	register	switch	using
bool	default	extern	long	return	this	union
break	delete	float	mutable	reinter_pret_cast	throw	unsigned
case	do	for	new	short	true	virtual
catch	double	friend	namespace	signed	typedef	void
char	dynamic_cast	false	operator	sizeof	template	volatile
class	else	goto	private	struct	try	while
const	enum	if	protected	static	typeid	wchar_t

2) Identifiers

Identifier are the names given to uniquely identify various programming elements like variables, arrays, functions, classes, structure, namespace and so on while defining identifiers in C++, programmers must follow the rules listed below.

a) An identifier must be unique in program.
b) An identifier must contain only upper case and lower case letters, underscore character (_) or digits 0 to 9.
c) An identifier must start with a letter or an underscore.
d) An identifier in upper case is different from that in lower case.
e) An identifier must be different from a keyword.
f) An identifier must not contain other characters such as '*', ';' and whitespace characters (tabs, space and newline)

Some valid and invalid identifiers in C++ are listed:

```
Po178_ddm          // valid
_789vt4            // valid
902gtl             // invalid as it starts with a digit
Tyy;d90            // invalid as it contains the ';' character
For                // invalid as it is a C++ keyword
Fg029 neo          // invalid as it contains space.
```

3) Constants

Constants also known as literals, are the values that a program cannot change during its execution. C++ constants are broadly classified into three categories namely, numeric constants, character constants and string constants.

a) Numeric Constants

Numeric constants refer to the number consisting of a sequence of digits (with or without decimal point) that can be either positive or negative. By default numeric constants are positive. Numeric constants are further classified as integer constants and floating point constants

Type	Description	Example
Integer Constants	Integer constants refer to integer valued numbers. They can be represented by three different number system namely decimal, octal, and hexadecimal.	65, -658, 0621, 38A
Floating point constants	Floating point constants refer to the real numbers, that is, the numbers with a decimal point. Floating point constants are also written in the floating point notation in which the constants is divided into a mantissa and an exponent.	65.75, -74.53, 37E.5

b) Character Constants

Character constants refer to a single character enclosed in single quotes(' '). The examples of character constant are 'a', 'N', '8', '&' etc. all character constants are internally stored as integer values.

c) String Constants

String constants refer to a sequence of any number of characters enclosed in the double quotes (" "). The examples of string constants are "hello", "name", "date" etc. note that string constants are always terminated by the Null ('\0') character.

Operators

Operator s are the symbols which represent various computations (such as addition, subtraction, etc.) performed on various data items. These data items on which operators act are known as **operands**.

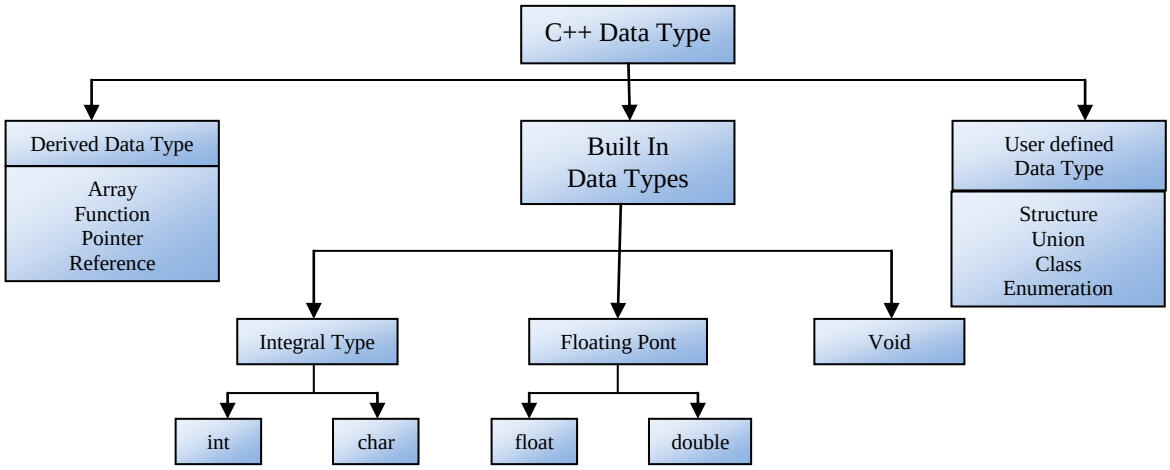
Depending on the number of operands and function performed, the C++ operators can be classified into various categories. These includes arithmetic operators, relational operators, logical operators, the conditional operator, assignment operator, bitwise operators and other operators. These categories are further classified into unary operators, binary operators and ternary operators.

Punctuators

Punctuators also known as separators, are the tokens that serve different purposes based on th econ text in which they are used. Some punctuators are used as operators, some are used to demarcate a portion of the program and so on. The various punctuators defined in C++ are Asterisk ' * ', Braces '{ }', Brackets '[ ]', Colon ' : ', Comma ' , ', Ellipsis '…', Equal to '=', Semicolon ';', Parentheses '( )', and Pound (Hash) '#'.

Data Types

A data type determines the type and the operations that can be performed on the data. In other words, a data type indicates the type of value an object will store and type of operations to be performed on it. C++ provides various data types and each data type is represented differently within the computer's memory.



1) Built in Data types

Built-in data types is the basic data type that cannot be divided into smaller units. The built-in data types provided by C++ are integral, floating point and void data type. Among these data types, the integral and floating point data types can be preceded by several type modifiers. These modifiers (also known as type qualifiers) are the keywords that alter either the size or range or both of the data types. The various modifiers are short, long, signed and unsigned.

A) Integral Data Type

The integral data type is used to store integers and includes char (character) and int (Integer) data types.

char

Character refer to the alphabets, numbers and other character (such as {, #,@, etc.) defined in the ASCII character set. In C++, the char data type is also treated as an integer data type as the character are internally stored as integers that range in value from -128 to 127. The char data type occupies 1 byte of memory (that is, it holds only one character at a time).

The various character data types with their size and range are

Type	Size (in bytes)	Range
char	1	-128 to 127
signed char	1	-128 to 127
Unsigned char	1	0 to 255

int

Numbers without the fractional part represent integer data. In C++, the int data type is used to store integers such as 4, 42, 5362, -56. Thus, it cannot store numbers with decimal point such as 4.25, -23.25. The int data type can be preceded by signed, unsigned, short and long modifiers.

The various integer data types with their size and ranges

Type	Size (in bytes)	Range
int	2	-32,768 to 32,767
signed int	2	-32,768 to 32,767
unsigned int	2	0 to 65,535
short int	2	-32,768 to 32,767
signed short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
long int	4	-2,147,483,648 to 2,147,483,647
signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

B) Floating point data type

A floating point data type is used to store real number such as 3.28, 64.526, 8.01, -42.23. This data type includes float and double data types. The various floating point data types with their size and range

Type	Size (in bytes)	Range	Digits of Precision
float	4	3.4E-38 to 3.4E+38	7
double	8	1.7E-308 to 1.7E+308	15
long double	10	3.4E-4932 to 1.1E+4932	18

C) void

The void data type is used for specifying an empty parameter list to a function and return type for a function. When void is used to specify an empty parameter list, it indicates that function does not take any arguments, and when it is used as a return type for a function, it indicates that a function does not return any value. For void, no memory is allocated and hence, it cannot store anything. As a result, void cannot be used to declare simple variable, however, it can be used to declare generic pointers.

2) Derived Data Types

Data types that are derived from the built-in data types are known as derived data types the various derived data types provided by C++ are arrays, functions, references and pointers.

Array

An array is a set of elements of the same data type that are referred to by the same name. all the elements in an array are stored at contiguous (one after the other) memory locations and each element is accessed by a unique index or subscript value. The subscript value indicates the position of an element in an array.

Function

A function is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions which can be invoked from other parts of a program, if required.

References

A reference is an alternative name for a variable. That is, a reference is an alias for a variable in a program. A variable and its reference can be used interchangeably in a program as both refer to the same memory location. Hence changes made to any of them are reflected in the other.

Pointer

A pointer is a variable that can store the memory address of another variable. Pointers allow to use the memory dynamically. That is, with the help of pointers, memory can be allocated or de-allocated to the variables at run-time, thus, making a program more efficient.

3) User Defined Data Types

Various user defined data types provided by C++ are structure, unions, enumerations and classes.

Structure, Union and Class

Structure and union are the significant features of C language. They provide a way to group similar or dissimilar data types that can be referred to by a single name.

C++ offers a new user defined data type known as class, which forms the basic of object oriented programming. A class acts as a template which defines the data and functions that are included in an object of the class. Classes are declared using the keyword *class*. Once the class has been declared, its object can be easily created.

Enumeration

An enumeration is a set of named integer constants that specify all the permissible values that can be assigned to enumeration variables. These sets of permissible value are known as enumerators. For example, consider this statement.

```
enum country {US, UN, India, China};          //declaring an enum type
```

In this statement, an enumeration data type country consisting of enumerators US, UN and so on is declared. Note that these enumerators represent integer values, so any arithmetic operation can be performed on them.

❖ The typedef Keyword

C++ provides a typedef feature that allow to define new data type names for existing data types that may be built-in, derived or user defined data types. Once the new name has been defined, variables can be declared using this new name. For example, Consider this declaration,

typedef int integer;

in this declaration, a new name integer is given to the data type int. this new name now can be used to declare integer variables as shown here,

integer i, j, k ;

❖ Variables

A variable is an identifiers that refers to the data item stored at a particular memory location. This data item can be accessed in the program simply by using the variable name.

Declaration of variables

Variable must be declared in a program before they are used. The declaration of a variable informs the compiler, the specific data type to which a variable is associated and allocated sufficient memory for it. The syntax for declaring a variable is;

Data_type variable_name;

Where, variable_name is th sequence of one or more letters, digits etc. at the time of the variable declaration, more than one variable of the same data type can be declared in a single statement separating them with comma (,). For example, consider this statement.

int x, y, z ;

Note that in C++, it is not necessary to declare the variable in the beginning of a program as required in C. It can be declared at the place where it is used first.

Initialization of Variables

Declaration of variables allocates sufficient memory for variables. However, it does not store any data in the variables. To store data in variables, the variables need to be initialized at the time of declaration. For example, consider the statement.

int i=10;

❖ Symbolic constant

A symbolic constant is referred to as name that substitutes for a constant that may be a numeric constant, a character constant or a string constant. In C++, symbolic constants can be created using the enum keyword, #define preprocessor directive, or the const qualifier.

The preprocessor directive #define is the traditional C like method used for defining symbolic constants as shown below.

#define size = 20;

Here, size is a symbolic constant whose every occurrence in the program is replaced with 20 during preprocessing.

There are certain problems with #define

1) They are resolve by preprocessor, which replaces the symbolic name in the program with its corresponding value.
2) This method does not allow to specify the data type of constant value.

A better way to define symbolic constant is to use **const** qualifier. When the keyword const precedes a data type in the declaration of a variable, it implies that the value of the variable cannot be changed throughout the program. For example,

const float pie_value = 3.14;

If any data type is not specified with the const qualifier, by default it is int.

const int max = 20; // Both these statements
const max = 20; // are equivalent.

Though C also allows to create const values, there are some differences in the implementation of const values in C and C++.

1. In C++, the variable declared as a constant must be initialized at the time of its declaration. While in C, const value is automatically initialized to 0 if it is not initialized by the user.
2. In C++, const can be used in expression. For example, consider these statements,

```
const int max = 20;
char name[max];
```

However, it is invalid in C.

3. In C++, by default, a const value is local to the file in which it is declared. However, it can be made global by explicitly defining it as an extern variable, as given,

```
extern const max=20;
```

In C, by default, const value is recognized globally, that is, it is also visible outside the file in which it is declared. However, it can be made local by declaring it as static.

❖ **Reference Variable**

Reference variable is a new kind of variable introduced in C++ which provides an alias for an already defined variable. Like other variable, reference variable must be declared in a program before they are used. The syntax for declaring a reference variable is:

data_type & refname = varname;

A program to demonstrate the use of reference variable,

```
#include<iostream.h>
Main()
{
        float cost = 200;
        float &price = cost;                    //declaration of reference variable
        cout<< "Price = "<<price<<endl;
        price=price + 20;
        cout<< "Cost = "<<cost<<endl;
        cout<<"Price = "<<price;
}
```

Output of the Program is

Price = 200
Cost = 220
Price = 220

Some important points to reference variables

1. A reference variable can refer to one variable only.
2. A reference to a pointer can also be created
3. A variable can have multiple references. Hence, changing in the value of one of them is reflected in all others.

❖ **Storage Class Specifiers**

In addition to data type a storage class is also associated with a variable which determines the scope and life-time of a variable. The scope refers to the part of the program that can access that variable and life-time refers to the duration for which a variable stays in existence. The syntax,

storage_class data_type variable;

There are four storage class specifiers supported by C++:

1) Automatic Variable

A variable defined within a block or a function is known as **local variable** and can be accessed within that block or function only. In C++, this local variable is referred to as **automatic variable.** A variable is made automatic by prefixing its declaration with the keyword **auto.**

2) External Variable

A variable that is accessible to all the functions in the file in which it is declared is known as **global variable.** However, if a global variable has to be accessed by some other file, it has to be declared as an external variable in that file by prefixing its declaration using the keyword **extern.**

3) Static Variable

A variable that is recognized only inside the function in which it is declared, but remains in existence throughout the life of program is known as **static variable.** A variable is declared static by prefixing its declaration by the keyword **static.** Both the local and global variable can be declared as static. When a local variable is declared as static, it is known as **static local variable.** When a global variable is declared as static, it is known as **static global variable.**

4) Register Variable

A variable that is stored in the register (a special storage area within the central processing unit) instead of the main memory where other variables are stored is known as a **register variable.** A register variable speeds up the execution of a program, as it does not require a memory access like normal variable. The register variable is declared with the help of keyword **register.**

❖ **Special Assignment Expression**

An expression can be categorized further depending upon the way the values are assigned to the variable.

1) Chained Assignment

Chained assignment is an assignment expression in which the same value is assigned to more than one variable using a single statement. For example,

```
a = (b = 20);
a = b = 20;
```

In these statements, value 20 is assigned to variable b and then to variable a. Note that variable cannot be initialized at the time of declaration using chained assignment. For example,

```
int  a = b = 20;          // illegal
int  a = 20, b = 20       // legal
```

2) Embedded Assignment

Embedded assignment is an assignment expression, which is enclosed within another assignment expression. For example,

```
a = 20 + (b = 30) ;                    // equivalent to b = 30;   a = 20 + 30;
```

In this statement, the value 20 is assigned to variable b and then the result of (20 + 30), that is, 50 is assigned to variable a. thus, the expression (b = 30) is an embedded assignment.

3) Compound Assignment

Compound assignment is an assignment expression, which uses a compound assignment operator that is a combination of the assignment operator with a binary arithmetic operator. For example, a += 20; // equivalent to a = a + 20;

In this statement, the operator += is a compound assignment operator, also known as **short hand assignment operator.**

❖ **Type Conversion**

An expression may involves variables and constants either of same data type or of different data types. However, when an expression consists of mixed data types then they are converted to the same type while evaluation to avoid compatibility issues. This is accomplished by the type conversion which is defined as the process of converting one predefined data type into another. Type conversion are of two types, 1) implicit conversion, 2) explicit conversion, also known as typecasting.

1) Implicit conversion

Implicit conversion, also known as automatic type conversion, refers to the type conversion that is automatically performed by the compiler. Whenever the compiler confronts a mixed type expression, first of all char and short int values are converted to int. This conversion is known as **integral promotion.** After applying this conversion, all the other operands are converted to the type of the largest operand and the result is of the type of the largest operand.

Data types
char short int
int
unsigned
long int
unsigned long int
float
double
long double

2) Explicit Conversion or Typecasting

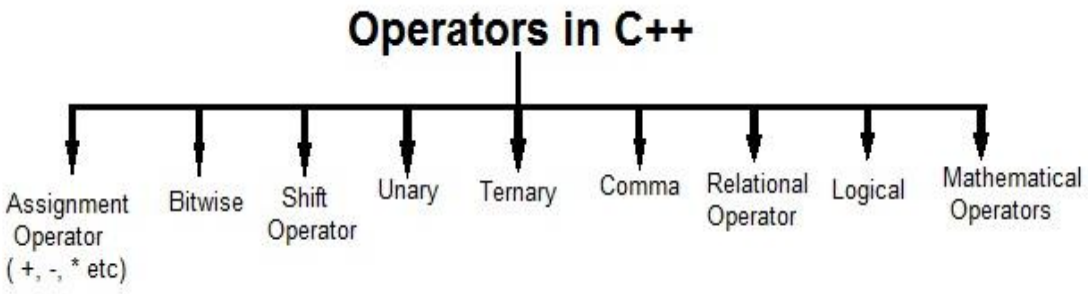
Typecasting refers to the type conversion that is performed explicitly using type cast operator. In C++ typecasting can be performed by using two different forms which are given below,

```
data_type ( expression )        // expression in parentheses
(data_type ) expression         // data type in parentheses
```

Where,
data_type = data type (also known as cast operator) to which the expression is to be converted.

❖ **Operators in C++**

Operators are special type of functions, that takes one or more arguments and produces a new value. For example : addition (+), substraction (-), multiplication (*) etc, are all operators. Operators are used to perform various operations on variables and constants.



Types of operators

1. Assignment Operator
2. Mathematical Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Shift Operators
7. Unary Operators
8. Ternary Operator
9. Comma Operator

Assignment Operator (=)

Operates '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

Mathematical Operators

There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , diversion (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.
C++ and C also use a shorthand notation to perform an operation and assignment at same type.
Example,

```
int x=10;
x += 4 // will add 4 to 10, and hence assign 14 to X.
x -= 5 // will subtract 5 from 10 and assign 5 to x.
```

Relational Operators

These operators establish a relationship between operands. The relational operators are : less than (<) , grater thatn (>) , less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=).
You must notice that assignment operator is (=) and there is a relational operator, for equivalent (==). These two are different from each other, the assignment operator assigns the value to any variable, whereas equivalent operator is used to compare values, like in if-else conditions, *Example*

```
int x = 10;  //assignment operator
x=5;         // again assignment operator
```

```
if(x == 5)   // here we have used equivalent relational operator, for comparison
{
 cout <<"Successfully compared";
}
```

Logical Operators

The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.

If two statement are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially while loop) and in Decision making.

Bitwise Operators

There are used to change individual bits into a number. They work with only integral data types like char, int and long and not with floating point values.

- Bitwise AND operators &
- Bitwise OR operator |
- And bitwise XOR operator ^
- And, bitwise NOT operator ~

They can be used as shorthand notation too, & = , |= , ^= , ~= etc.

Shift Operators

Shift Operators are used to shift Bits of any variable. It is of three types,

1. Left Shift Operator <<
2. Right Shift Operator >>
3. Unsigned Right Shift Operator >>>

Unary Operators

These are the operators which work on only one operand. There are many unary operators, but increment ++ and decrement -- operators are most used.

Other Unary Operators : address of &, dereference *, **new** and **delete**, bitwise not ~, logical not !, unary minus - and unary plus +.

Ternary Operator

The ternary if-else ? : is an operator which has three operands.

```
int a = 10;
a > 5 ? cout << "true" : cout << "false"
```

Comma Operator

This is used to separate variable names and to separate expressions. In case of expressions, the value of last expression is produced and used.

Example :

```
int a,b,c; // variables declaration using comma operator
a=b++, c++; // a = c++ will be done.
```

sizeof operator in C++

sizeof is also an operator not a function, it is used to get information about the amount of memory allocated for data types & Objects. It can be used to get size of user defined data types too. sizeof operator can be used with and without parentheses. If you apply it to a variable you can use it without parentheses.

```
cout << sizeof(double);   //Will print size of double
int x = 2;
int i = sizeOf x;
```

❖ **New Operators**

In addition to arithmetic operator, relational operators, logical operators, the conditional operator and assignment operators that most of the languages support. C++ provides various new operators which are listed.

Operators	Description
: :	Scope resolution operator
: :*	Pointer-to-member declaratory
->*	Pointer-to-member operator
.*	Pointer-to-member operator
new	Memory allocation operator
delete	Memory release operator
endl	Line feed operator
setw	Field width operator

1) Scope Resolution operator

In C++, variables in different blocks or functions can be declared with the same name. That is, the variables in different scope can have the same name. however, a local variable overrides the variable having same name in outer block or the variable with global scope. Hence, a global variable or variable in the outer block cannot be accessed inside the inner block. This problem can be solved by the scope resolution operator (: :).

A program to demonstrate the use of scope resolution operator.

```
#include<iostream.h>
int x = 5;                    // global variable
int Main()
{
        int x = 3;            // local variable
        cout << "The local variable of outer block is : "<<x<<endl;
        cout << "The global variable is : "<<::x<<endl;
        {
                int x = 10;           //local variable
                cout<< "The local variable of inner block is : "<<x<<endl;
                cout<< "The global variable is : "<<::x<<endl;
        }
getch();
}
```

The output of the program is:

```
The local variable of out block is : 3
The global variable is : 5
The local variable of inner block is : 10
The global variable is : 5
```

2) new and delete

C++ provides two dynamic allocation operators new and delete to allocate and de-allocate memory at run time, respectively. The new operator is unary operator that allocates memory and returns a pointer to the starting address of the allocated space. The syntax for allocating memory using the new operator:

```
        p_var = new data_type;
```

where,

```
        p_var = the name of pointer variable
        new = C++ keyword
        data_type = any valid data type of C++
```

for example,

```
        int *iptr;                    // pointer declaration
        iptr = new int;               // allocating memory to an int variable
```

in these statement, the **new** operator returns the address of the memory allocated for an **int** variable from the free store and this address is stored in the pointer **iptr**.

The life-time of a variable created at run-time using the new operator is not restricted till the execution of the program. Rather, it remains in the memory until it is explicitly deleted using the

delete operator. When a dynamically allocated variable is no longer required, it must be destroyed using the delete operator to ensure safe and efficient use of the memory.
The syntax for using the delete operator,

```
delete  p_var;
```

for example,

```
delete iptr;
```

3) endl and setw

C++ provides various operators that can be used with an output statement to display the formatted output. These operators are also known as **manipulators**. The manipulator which are commonly used include **endl** and **setw.**

The **endl** manipulator causes a linefeed to be inserted when used with an output statement. The effect of this manipulator is the same as that of the newline character '\n'

The output can be formatted by using **setw** operator. with the setw operator the values are right justified within the specified field width.

A program to display formatted output using endl and setw

```
#include<iostream.h>
#include<iomanip.h>                 // for setw
main( )
{
        int Refrigerator=15000, TV=6000, Almirah=800;
        cout <<setw(15)<<"Refrigerator =" <<setw(6)<<Refrigerator<<endl;
        cout <<setw(15) << "TV ="<<setw(6)<<TV<<endl;
        cout <<setw(15)<< "Almirah ="<<setw(6)<<Almirah<<endl;
getch();
}
```

This example, setw(15) specifies the field width 15 for displaying the string and setw(6) specifies the field width 6 for displaying the value of variable. The values are right justified within the specified field width.

❖ **Control Statement**

A statement is an instruction given to the computer to perform a specific action. In C++, a statement can be either a single statement or a compound statement. A single statement specifies a single action and is always terminated by a semicolon ';'. A compound statement, also known as a block, is a set of statements that are grouped as a compound statement are always enclosed within curly braces. '{ }'

To make a program more flexible, control statements are used to alter the flow of control of the program. In C++ the control statements are broadly classified into three categories, namely conditional statement, iteration statement and jump statements. All these control statements are commonly used with the logical test or test conditions to alter the flow of control conditionally or unconditionally.

1) Conditional Statement

Conditional statements, also known as selection statement, are used to make decisions based on a given condition. If the condition evaluates to True, a set of statements is executed, otherwise another set of statements is executed.

a) The if Statement

The if statement select and execute the statement(s) based on a given condition. If the condition evaluates to True, then a given set of statements is executed. However, if the condition evaluates to False, then that set of statement is skipped and the program control passes to the statement following the if statement.
The syntax,

```
if (condition)
{
        Set of statements;
}
Statements;
```

b) The if--else statement

The if--else statement causes one of the two possible statement(s) to execute depending upon the outcome of the condition.

The syntax,

```
if (condition)                  //if part
{
        Set of statements;
}
else                            // else part
{
        Set of statements;
}
Statements;
```

Here, if - - else statement comprises two parts, namely *if* and *else*. If the condition is *True*, the *if* part is executed. However, if the condition is *False*, the *else* part is executed.

c) Nested if - - else statement

A nested *if--else* statement contains or more *if--else* statements. The *if—else* statement can be nested in three different ways.

The *if* – *else* statement is nested within the if part.

The syntax is

```
if (condition 1)
{
        statement 1;
        if (condition 2)
                statement 2;
        else
                statement 3;
}
else
        statement 4;
```

d) The if – else -- if ladder

The if – else – if ladder, also known as if – else -- if staircase, has an if – else statement within the outermost else statement can further have other if – else statement.

The syntax,

```
if (condition 1)
        statement 1;
        else
                if (condition 2)
                        statement 2;
                else
                        if (condition 3)
                                statement 3;
                        else
                                statement 4;
next statement;
```

❖ Conditional operator as an alternative

The conditional operator '? :' selects one of the two values or expressions based on a given condition. Due to this decision making nature of the conditional operator, it is sometimes used as an alternative to if – else statements. However, it is different from if – else statements in a sense that it selects one of the two values or expressions and not the statements.

For example,

```
max = ( x > y ? x : y)
```

this statement assign maximum of x and y to max.

```
#include<iostream.h>
#include<conio.h>
main( )
{
        int x,y,max;
        cout << "Enter Value for x : ";
        cin>> x;
        cout <<"Enter Value for y :";
        cin>> y;
        max = ( x > y ? x : y);
        cout << "Maximum Value = "<< max;
getch( );
}
```

Output:

```
Enter Value for x : 10
Enter Value for y : 20
Maximum Value = 20
```

The switch statement

The switch statement selects a set of statements from the available set of statements. the switch statement test the value of an expression in a sequence and compares it with the list of integers or character constants. When a match is found, all the statements associated with that constant are executed.

The syntax;

```
switch ( expression )
{
        case <constant 1> :
                statement 1;
                [break;]
        case <constant 2> :
                statement 2;
                [break;]
        case <constant 3> :
                statement 3;
                [break;]
        default :
                statement 4;
                [break;]
}
Statement 5;
```

The keyword *default* specifies the set of statement to be executed in *case* no match is found. Note that there can be multiple *case* labels but there can be only one *default* label.

The *break* statements in the *switch* block are optional. However, it is used in the *switch* block to prevent a fall through. Fall through is a situation that causes the execution of the remaining cases even after a match has been found.

❖ Iteration Statement or Loops

The statement 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. That is, as long as the condition evaluates to *True* , the set of statements is executed.

1) The For Loop

The for loop is one of the most widely used loops in C++. This loop is deterministic in nature, that is, the number of times the body of the loop is executed is known in advance. Moreover, it is an **entry controlled loop** where condition is checked first before starting any iteration.

The syntax,

```
For ( initialize ; condition ; update)
{
        // body of the loop
}
```

Note that initialize, condition and update are optional expressions and are always specified in parentheses. All the three expressions are separated by semicolons. The semicolons are mandatory and, hence, cannot be excluded even if all the three expressions are omitted.
For example, print the natural number

```
#include<iostream.h>
#include<conio.h>
main ( )
{
        int n;
        for ( n =1; n<=10; n++)
         {
                cout << n <<"  ";
         }
getch( );
}
```

Output :

1 2 3 4 5 6 7 8 9 10

For loop using comma operator

The for loop allows multiple variable to control the loop using comma operator. That is, two or more variables can be used in the initialize and the update part of the loop. For example,

for (i=1, j=50; i<10; i++, j--)

2) The while Loop

The while loop is used to perform looping operations in situations where the number of iterations is not known in advance. That is, unlike the for loop, the for loop, the while loop is non-deterministic in nature. It is also an entry controlled loop where the condition is checked first before starting any iteration.
The syntax,

```
While (condition)
{
        // body of while loop
}
```

These points should be noted about the **while** loop

1. The while loop does not specify any explicit initialize and update expression. This implies that the control variable must be declared and initialized before the while loop and needs to be updated within the body of the while loop.
2. The while loop executes as long as condition evaluates to True. If condition evaluates to False in the first iteration, then the body of the while loop never execute.
3. The while loop can have more than one expression in its condition.

The program to print the table of a given number.

```
#include<iostream.h>
#include<conio.h>
main()
{
        int n,i=1;
        clrscr();
        cout<<"Enter any Number : ";
        cin>>n;
        while(i<=10)
        {
         cout<< "t"<<i*n;
         i++;
```

```
                }
                getch();
        }
```

Output :

```
Enter any Number : 5
        5       10      15      20      25      30      35      40      45      50
```

3) The do -- while loop

As discuss earlier, in a while loop, the condition is evaluated at the beginning of the loop and if the condition evaluates to False, the body of the loop is not executed even once. However, if is not executed even once. However, if the body of the loop is to be executed at least once, no matter whether the initial state of the condition is True or False, the do – while loop is used. This loop places the condition to be evaluated at the end of the loop. That is why it is also known as **exit controlled loop**.

Jump Statement

Jump statements are used to alter the flow of control unconditionally. That is, jump statements transfer the program control within a function unconditionally. The jump statements defined in C++ are break, continue and return. In addition to these jump statements a standard library function exit() is used to jump out of an entire program.

The break Statement

The break statement is extensively used in loops and switch statements. it immediately terminates the loop or the switch statement, bypassing the remaining statement. The control then passes to the statement that immediately follows the loop or the switch statements. A break statement can be used in any of the three C++ loops.

If break statement is used in a nested loop, it affects only the inner loop in which it is used and not any of the outer loop.

```
#include<iostream.h>
#include<conio.h>
main()
{
        int x=0,y,sum=0;
        clrscr();
        cout<<"Enter a number : ";
        cin>>y;
        cout<<"Factors:"<<endl;
        while (1)
        {
         x++;
         if (x>y)
                break;
         if (y%x!=0)
                continue;
         cout<<"\t"<<x;
         sum=sum+x;
  }
 cout<<endl<<"Sum of Factors = "<<sum;
getch();
 }
```

Output :

```
Enter a number : 8
Factors :
        1       2       4       8
Sum of Factors = 15
```

The goto Statement

The goto statement can be used anywhere within a function or a loop. As the name suggest, goto statements transfer the control from one part to another part in a program which is specified by a label. Labels are user-defined identifiers followed by a colon that are prefixed to a statement to specify the destination of a goto statement.

A program to demonstrate the use of goto statement.

```
#include<iostream.h>
#include<conio.h>
main()
{
        int x=10;
        clrscr();
        loop: cout<<x<<" , ";
        x--;
        if (x>0)
        goto loop;
        getch();
 }
```

Output:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1,

The exit() function

The exit() is standard library function that terminates the entire program immediately and passes the control to the operating system. This function takes a single parameter, that is exit status of the program and returns same status to the operating system upon termination.

Programs:

1) Write a program to evaluate the sum of the following series:
   a. $x + x^2 + x^3 + x^4 + \ldots . x^n$
   b. x – x2/3! + x3/5! – x4/7! + ……up to n term
2) Write a program to print out all Armstrong numbers between 1 and 500. A number is called an Armstrong number, if sum of cubes of each digit is equal to the number itself. For example,
   153 = (1*1*1) + (5*5*5) + (3*3*3)
3) Write a program to generate the table from 2 to 10. Display it on the screen, one table in one row as shown here

2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
.									
.									