ACMT GROUP OF COLLEGES

Polytechnic -2ND Year/ 3rd Sem

DIPLOMA ENGINEERING COMPUTER SCIENCE

Subject- Object Oriented Programming Using C++

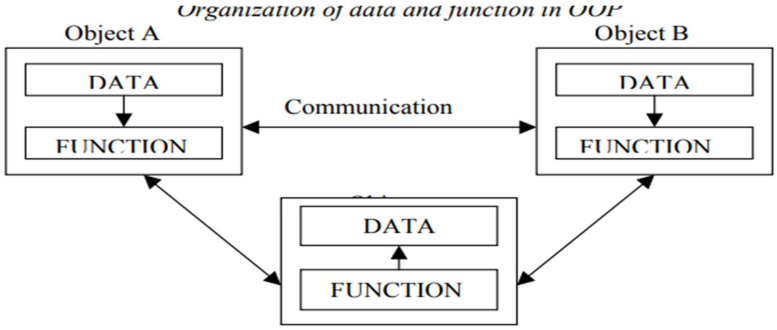
BY- Anjana Verma
(CS Department)

Subject: Object Oriented Programming Using C++
Unit -1

OOP Paradigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in given fig. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects. Organization of data and function in OOP Object A Object B

Organization of data and function in OOP Object A Object B Communication DATA FUNCTION DATA FUNCTION Object FUNCTION DATA So



Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design. Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people.

Benefits of object oriented programming (OOPs)

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

- Reusability: In OOPs programs functions and modules that are written by a user can be reused byother users without any modification.
- Inheritance: Through this we can eliminate redundant code and extend the use of existing classes.
- Data Hiding: The programmer can hide the data and functions in a class from other classes. It helps the programmer tobuild the secure programs.
- Reduced complexity of a problem: The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.
- Easy to Maintain and Upgrade: OOP makes it easy to maintain and modify existing code as new objectscan be created with small differences to existing ones. Software complexity can be easily managed.
- Message Passing: The technique of message communication between objects makes the interfacewith external systems easier.
- Modifiability: it is easy to make minor changes in the data representation or the procedures in an Oops program. Changes inside a class do not affect any other part of a program, since the onlypublic interface that the external world has to a class is through the use of methods.

.

Advantages of Oops

• 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 can not 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.

Comparison between functional Programming and Oops Approach

Functional Programming	Object Oriented Programming
This programming paradigm emphasizes on the use of functions where each function performs a specific task.	This programming paradigm is based on object oriented concept. Classes are uses where instance of objects are created.
Fundamental elements used are variables and functions. The data in the function are immutable (cannot be changed after	Fundamental elements used are objects and methods and the data used here are mutable data.

creation).	
Importance is not given to data but to functions.	Importance is given to data rather than procedures.
It uses recursion for iteration	It uses loops for iteration
It is parallel programming supported.	It does not support parallel programming.
Does not have any access specifier.	Has three access specifier namely, Public, Private and Protected.
To add new data and functions is not so easy.	Provides and easy way to add new data and functions.
No data hiding is possible. Hence, Security is not possible.	Provides data hiding. Hence, secured programs are possible.
The statements in this programming paradigm does not need to follow a particular order while execution.	The statements in this programming paradigm need to follow a order i.e., bottom up approach while execution.

Characteristic 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. Although different author represent them differently fig. shows two notations that are popularly used in objectoriented analysis and design.

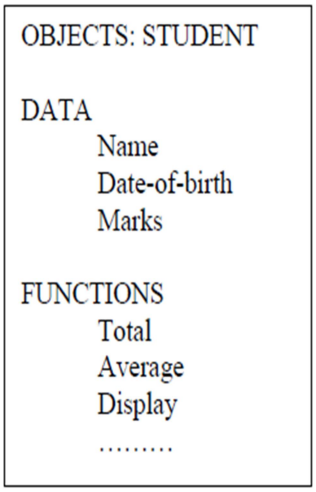


Fig. 1.5 representing an object

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.

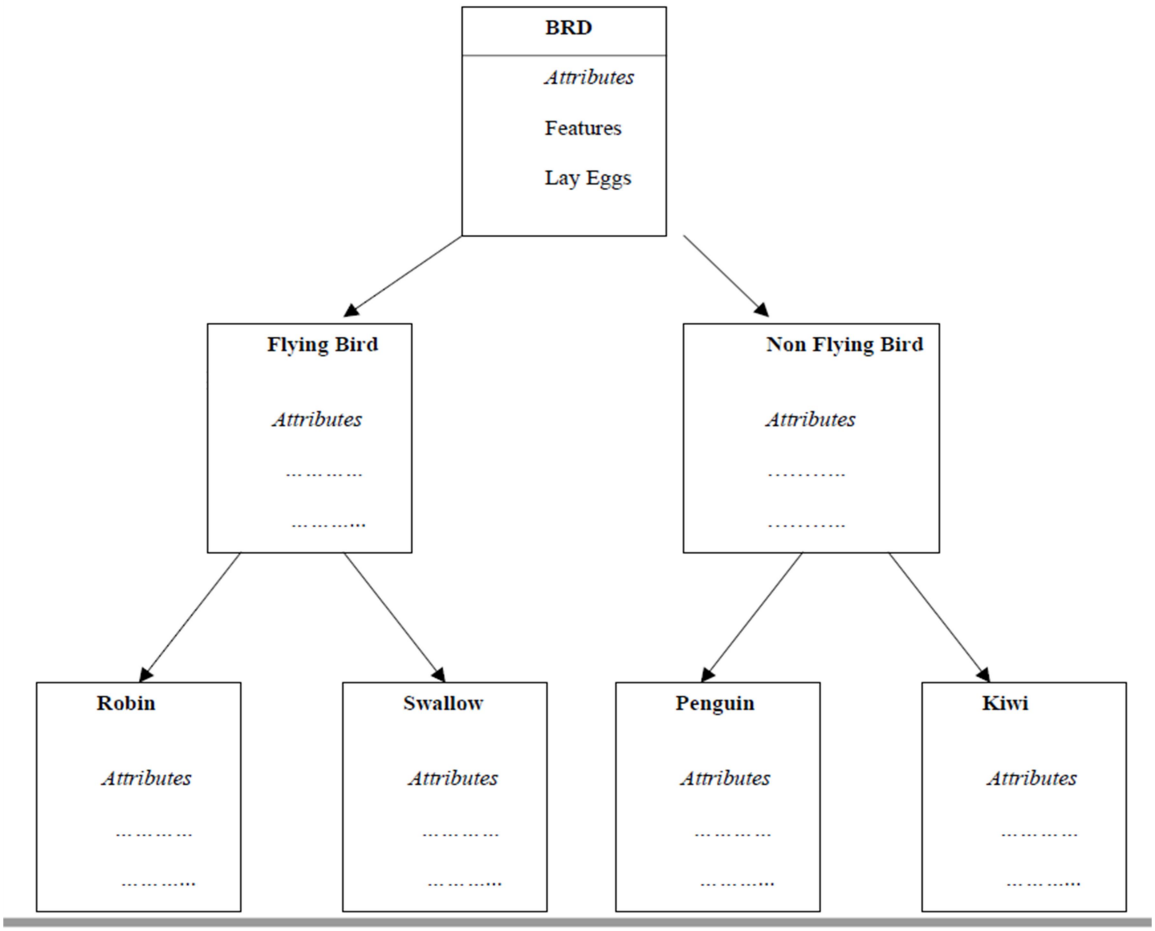
Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all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

Fig. Property inheritances BIRD

Fig. 1.6 Property inheritances

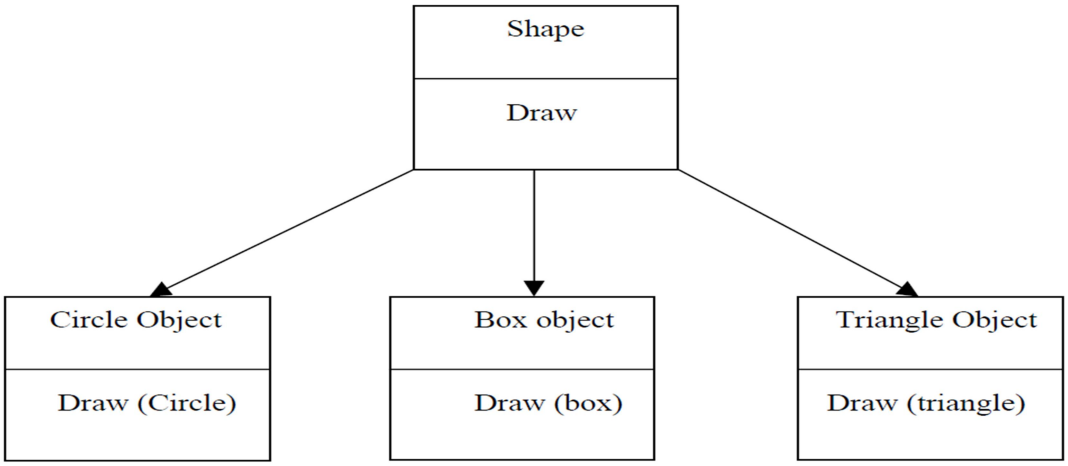


Allows the programmer to reuse a class i.e almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduced any undesirable side-effects into the rest of classes.

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. Fig. 1.7 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. Shape Draw Circle Object Draw (Circle) Box object Draw (box) Triangle Object Draw (triangle) Fig.



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 fig.

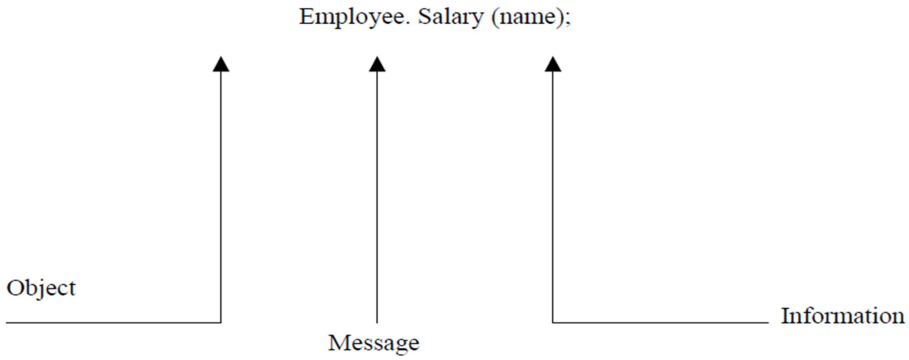
Inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

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 behaviour,

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 realworld 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:
Employee. Salary (name);
Object Information Message



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

Major feature that are required for object based programming are:

• Data encapsulation
• Data hiding and access mechanisms
• Automatic initialization and clear-up of objects
• Operator overloading Languages that support programming with objects are said to the objects-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language. Object-oriented programming language incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statements: Object-based features + inheritance + dynamic binding

Application of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as window. Hundreds of windowing systems have been developed, using the OOP techniques. Real-

business system are often much more complex and contain many more objects with complicated attributes and method. OOP is useful in these types of application because it can simplify a complex problem.

The promising areas of application of OOP include:

• Real-time system
• Simulation and modeling
• Object-oriented data bases
• Hypertext, Hypermedia, and expertext
• AI and expert systems
• Neural networks and parallel programming
• Decision support and office automation systems
• CIM/CAM/CAD systems

The object-oriented paradigm sprang from the language, has matured into design, and has recently moved into analysis. It is believed that the richness of OOP environment will enable the software industry to improve not only the quality of software system but also its productivity. Object-oriented technology is certainly going to change the way the software engineers think, analyze, design and implement future system.

****

Chapter-2
Introduction of C++

Introduction of 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. wanted to combine the best of both the languages (Simula67 and C) 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. 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++ compiler. 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.

Application of C++

C++ is a versatile language for handling very large programs; it is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life applications systems.

• Since C++ allow us to create hierarchy related objects, we can build special object-oriented libraries which can be used later by many programmers.

• While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get closed to the machine-level details.

• C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.

• It is expected that C++ will replace C as a general-purpose language in the near future.

Simple C++ Program

Let us begin with a simple example of a C++ program that prints a string on the screen. Printing A String #include Using namespace std; int main() { cout<<<<< is called the insertion or put to operator.

The iostream File

We have used the following #include directive in the program:

#include

The #include directive instructs the compiler to include the contents of the file enclosed within angular brackets into the source file.
The header file iostream.h should be included at the beginning of all programs that use input/output statements.

Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifier defined in the namespace scope we must include the using directive, like Using namespace std;. All ANSI C++ programs must include this directive.. Using and namespace are the new keyword of C++.

Return Type of main()

In C++, main () returns an integer value to the operating system. Therefore, every main () in C++ should end with a return (0) statement; otherwise a warning an error might occur. Since main () returns an integer type for main () is explicitly specified as int.

The following main without type and return will run with a warning:

main () { …….. ……. }

Identifiers

C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer. They are the basic requirement of any language. Every language has its own rules for naming the identifiers. Whereas, the identifiers are the names which are defined by the programmer to the program elements such as variables, functions, arrays, objects, classes. In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

- Constants
- Variables
- Functions
- Labels
- Defined data types

Some naming rules are common in both C and C++. They are as follows:

- Only alphabetic characters, digits, and underscores are allowed.
- The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
- In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
- A declared keyword cannot be used as a variable name.

For example, suppose we have two identifiers, named as 'FirstName', and 'Firstname'. Both the identifiers will be different as the letter 'N' in the first case in uppercase while lowercase in second. Therefore, it proves that identifiers are case-sensitive.

Keywords

Keywords are the reserved words that have a special meaning to the compiler. They are reserved for a special purpose, which cannot be used as the identifiers. For example, 'for', 'break', 'while', 'if', 'else', etc. are the predefined words where predefined words are those words whose meaning is already known by the compiler.

C++ Keywords

Double	char	const	class	auto	break
Else	switch	return	short	if	long
int	default	do	continue	void	while
Typedef	goto	delete	enum	extern	for

Differences between Identifiers and Keywords

Identifiers	Keywords
Identifiers are the names defined by the programmer to the basic elements of a program.	Keywords are the reserved words whose meaning is known by the compiler.
It is used to identify the name of the variable.	It is used to specify the type of entity.
It can consist of letters, digits, and underscore.	It contains only letters.
It can use both lowercase and uppercase letters.	It uses only lowercase letters.
No special character can be used except the underscore.	It cannot contain any special character.
The starting letter of identifiers can be lowercase, uppercase or underscore.	It can be started only with the lowercase letter.
It can be classified as internal and external identifiers.	It cannot be further classified.
Examples are test, result, sum, power, etc.	Examples are 'for', 'if', 'else', 'break', etc.

Constants

Constants refer to values that do not change during the execution of a program.
Constants can be divided into two major categories:

- Primary constants
- Secondary constants:

1.Primary constants

a)Numeric constants

b)Integer constants.

- Floating-point (real)

a)constants. b)Character constants

- Single character constants
- String constants

2.Secondary constants:

- Enumeration constants.
- Symbolic constants.
- Arrays, unions, etc.

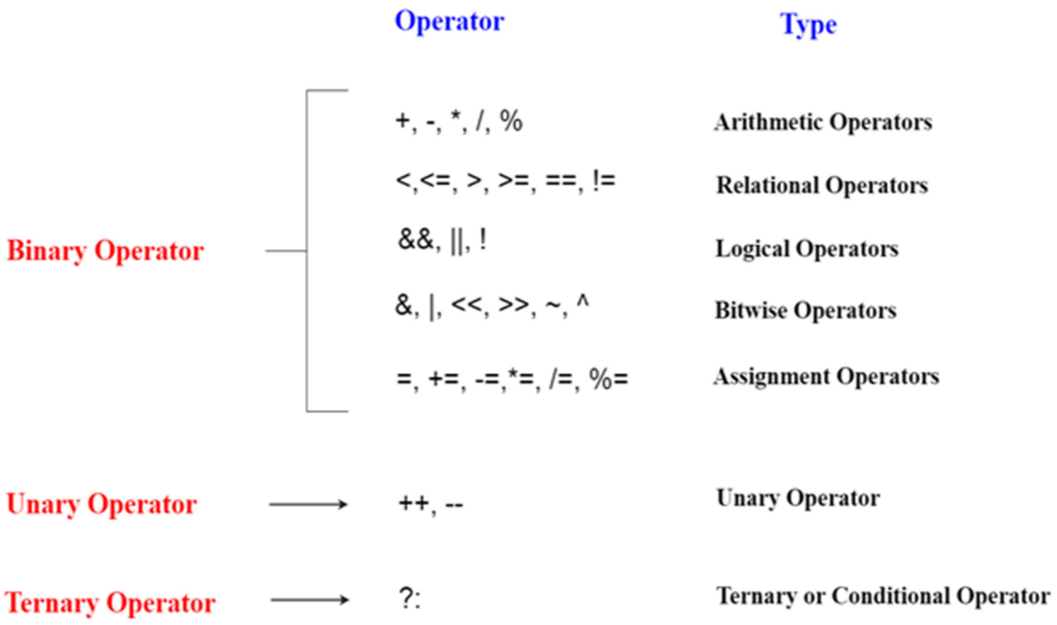
Rules for declaring constants:
1.Commas and blank spaces are not permitted within the constant.
2.The constant can be preceded by minus (-) signed if required.
3.The value of a constant must be within its minimum bounds of its specified data type.

Integer constants: An integer constant is an integer-valued number. It consists of sequence of digits. Integer constants can be written in three different number systems:
1.Decimal integer (base 10).
2.Octal integer (base 8).
3.Hexadecimal (base 16).

C++ Operators

An operator is simply a symbol that is used to perform operations. All C operators are valid in C++. There can be many types of operations like arithmetic, logical, bitwise etc. There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operator
- Unary operator
- Ternary or Conditional Operator



1.Arithmetic Operators : All basic arithmetic operators are present in C. An arithmetic operation involving only real operands(or integer operands) is called real arithmetic(or integer arithmetic). If a combination of arithmetic and real is called mixed mode arithmetic.
operator meaning

+	add
-	subtract
*	multiplication
/	division
%	modulo division(remainder)

2. Relational Operators :We often compare two quantities and depending on their relation take certain decisions for that comparison we use relational operators.

operator meaning

<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
==	is equal to
!=	is not equal to

3.Logical Operators:

Logical Data: A piece of data is called logical if it conveys the idea of true or false. In C++ we use int data type to represent logical data. If the data value is zero, it is considered as false. If it is non -zero (1 or any integer other than 0) it is considered as true. C++ has three logical operators for combining logical values and creating new logical values:

Truth tables for AND (&&) and OR (||) operators:

X	Y	X&&Y	X||Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Truth table for NOT (!) operator:

X	!X
0	1
1	0

4.Bit Wise Operators : C supports special operators known as bit wise operators for manipulation of data at bit level. They are not applied to float or double.operator meaning&Bitwise AND

^ Bitwise exclusive OR
<< left shift
>> right shift
~ one's complement

5.Assignment Operator:

The assignment expression evaluates the operand on the right side of the operator (=) and places its value in the variable on the left.

Note: The left operand in an assignment expression must be a single variable.

There are two forms of assignment:

•Simple assignment
•Compound assignment

Increment (++) And Decrement (--) Operators:

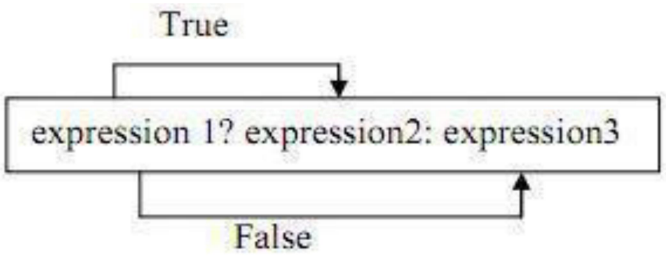
The operator ++ adds one to its operand where as the operator - - subtracts one from its operand. These operators are unary operators and take the following form:

Both the increment and decrement operators may either precede or follow the operand. Postfix Increment/Decrement :(a++/a--) In postfix increment (Decrement) the value is incremented (decremented) by one. Thus the a++ has the same effect as

Operator	Description
++a	Pre-increment
a++	Post-increment
--a	Pre-decrement
a--	Post-decrement

6.Unary Operator: operator which operates on single operand is called unary operator
Operators in c++:All above operators of c language are also valid in c++.New operators introduced in c++ are
Operator Symbol
1. Scope resolution operator ::
2. Pointer to a member declarator ::*
3. Pointer to member operator ->*,->
4. Pointer to member operator .*
5. new Memory allocating operator
6. delete Memory release operator
7. endl Line feed operator
8. setw Field width operator
9. insertion<<

7.Conditional Operator Or Ternary Operator:
A ternary operator requires two operands to operate



8.Special Operators
These operators which do not fit in any of the above classification are ,(comma), sizeof, Pointeroperators(& and *) and member selection operators (. and ->). The comma operator is used to link related expressions together.

Type conversion

The type conversion techniques present in C++.There are mainly two types of type conversion. The implicit and explicit.

Implicit type conversion

This is also known as automatic type conversion. This is done by the compiler without any external trigger from the user. This is done when one expression has more than one data type is present.

All data types are upgraded to the data type of the large variable.
bool -> char -> short int ->int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double
In the implicit conversion, it may lose some information. The sign can be lost etc.

include<iostream>

```
using namespace std;
int main() {
int a = 10;
char b = 'a';
a = b + a;
float c = a + 1.0;
cout<< "a : " << a << "\nb : " << b << "\nc : " << c;
}
```

Output

a : 107
b : a
c : 108

Explicit type conversion

This is also known as type casting. Here the user can typecast the result to make it to particular datatype. In C++ we can do this in two ways, either using expression in parentheses or using static_cast or dynamic_cast.

Example

```
#include<iostream>
usingnamespacestd;
int main()
{
 double x =1.574;
intadd=(int)x +1;
 cout<<"Add: "<<add;
float y =3.5;
 intval=static_cast<int>(y);
cout<<"\nvalue: "<<val;
}
```

Output

Add: 2
value: 3

Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times. It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

type variable_list;

The example of declaring variable is given below:

int x;
float y;
char z;

Here, x, y, z are variables and int, float, char are data types.

Rules for defining variables

- A variable can have alphabets, digits and underscore.
- A variable name can start with alphabet and underscore only. It can't start with digit.
- No white space is allowed within variable name.

- A variable name must not be any reserved word or keyword e.g. char, float etc.

Statement

C++ statements are the program elements that control how and in what order objects are manipulated. This section includes:

Overview C ++ statements are executed sequentially, except when an expression statement, a selection statement, an iteration statement, or a jump statement specifically modifies that sequence.

Statements may be of the following types:

labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-throw-catch

- Labeled Statements

  Labels are used to transfer program control directly to the specified statement.

  Syntax

  labeled-statement:
  identifier : *statement*
  case *constant-expression* : *statement*
  default : *statement*

- Expression statements. These statements evaluate an expression for its side effects or for its return value.
- Null statements. These statements can be provided where a statement is required by the C++ syntax but where no action is to be taken.
- Compound statements. These statements are groups of statements enclosed in curly braces ({ }). They can be used wherever a single statement may be used.
- Selection statements. These statements perform a test; they then execute one section of code if the test evaluates to true (nonzero). They may execute another section of code if the test evaluates to false.
- Iteration statements. These statements provide for repeated execution of a block of code until a specified termination criterion is met.
- Jump statements. These statements either transfer control immediately to another location in the function or return control from the function.
- Declaration statements. Declarations introduce a name into a program.

Expressions

C++ expression consists of operators, constants, and variables which are arranged according to the rules of the language. It can also contain function calls which return values. An expression can consist of one or more operands, zero or more operators to compute a value. Every expression produces some value which is assigned to the variable with the help of an assignment operator.

Examples of C++ expression:

(a+b) - c
(x/y) -z
4a2 - 5b +c
(a+b) * (x+y)

An expression can be of following types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions
- Special assignment expressions

Constant expressions

- A constant expression is an expression that consists of only constant values.
- It is an expression whose value is determined at the compile-time but evaluated at the run-time.
- It can be composed of integer, character, floating-point, and enumeration constants. Constants are used in the following situations:
- It is used in the subscript declarator to describe the array bound.
- It is used after the case keyword in the switch statement.
- It is used as a numeric value in an **enum**
- It specifies a bit-field width.
- It is used in the pre-processor **#if**

The following table shows the expression containing constant value:

Expression containing constant	Constant value
x = (2/3) * 4	(2/3) * 4
extern int y = 67	67
int z = 43	43
static int a = 56	56

Integral Expressions

An integer expression is an expression that produces the integer value as output after performing all the explicit and implicit conversions.

Following are the examples of integral expression:

x * y) -5
x + **int**(9.0)
where x and y are the integers variables.

Float Expressions

A float expression is an expression that produces floating-point value as output after performing all the explicit and implicit conversions.

The following are the examples of float expressions:

x+y
(x/10) + y
34.5
x+**float**(10)

Pointer Expressions

A pointer expression is an expression that produces address value as an output.

The following are the examples of pointer expression:

&x
ptr
ptr++
ptr-

Relational Expressions

A relational expression is an expression that produces a value of type bool, which can be either true or false. It is also known as a boolean expression. When arithmetic expressions are used on both sides of the relational operator, arithmetic expressions are evaluated first, and then their results are compared.

The following are the examples of the relational expression:

a>b
a+b == x+y
a+b>80

Logical Expressions

A logical expression is an expression that combines two or more relational expressions and produces a bool type value. The logical operators are '&&' and '||' that combines two or more relational expressions.

The following are some examples of logical expressions:

a>b && x>y
a>10 || b==5

Bitwise Expressions

A bitwise expression is an expression which is used to manipulate the data at a bit level. They are basically used to shift the bits.

For example:

x=3
x>>3

// This statement means that we are shifting the three-bit position to the right.

In the above example, the value of 'x' is 3 and its binary value is 0011. We are shifting the value of 'x' by three-bit position to the right.

Special Assignment Expressions

Special assignment expressions are the expressions which can be further classified depending upon the value assigned to the variable.

1.Chained Assignment: Chained assignment expression is an expression in which the same value is assigned to more than one variable by using single statement.

2.Embedded Assignment Expression: An embedded assignment expression is an assignment expression in which assignment expression is enclosed within another assignment expression.

3. Compound Assignment:A compound assignment expression is an expression which is a combination of an assignment operator and binary operator.

User Defined Data types

User Defined Data type in c++ is a type by which the data can be represented. The type of data will inform the interpreter how the programmer will use the data. A data type can be pre-defined or user-defined. Examples of pre-defined data types are char, int, float, etc. As the programming languages allow the user to create their own data types according to their needs. Hence, the data types that are defined by the user are known as user-defined data types. For example; arrays, class, structure, union, Enumeration, pointer, etc. These data types hold more complexity than pre-defined data types.

Types of User-Defined Data in C++

Here are the types mentioned below:

1. Structure

A structure is defined as a collection of various types of related information under one name. The declaration of structure forms a template and the variables of structures are known as members. All the members of the structure are generally related. The keyword used for the structure is "struct".

For example; a structure for student identity having 'name', 'class', 'roll_number', 'address' as a member can be created as follows:

```
structstud_id
{
char name[20];
int class;
introll_number;
char address[30];
};
```

This is called the declaration of the structure and it is terminated by a semicolon (;). The memory is not allocated while the structure declaration is delegated when specifying the same. The structure definition creates structure variables and allocates storage space for them.

2. Array

An Array is defined as a collection of homogeneous data. It should be defined before using it for the storage of information. The array can be defined as follows:

```
<datatype><array_name><[size of array]>
int marks[10]
```

The above statement defined an integer type array named marks that can store marks of 10 students. After the array is created, one can access any element of an array by writing the name an array followed by its index. For example; to access 5^{th} element from marks, the syntax is as follows:

```
marks[5]
```

It will give the marks stored at the 5^{th} location of an array. An array can be one-dimensional, two-dimensional or multi-dimensional depending upon the specification of elements.

3. Union

Just like structures, the union also contain members of different data types. The main difference between the two is that union saves memory as members of a union share the same storage area whereas members of the structure are assigned their own unique storage area. Unions are declared with keyword "union" as follows:

```
union employee
{
```

```
int id;
double salary;
char name[20];
}
```

The variable of the union can be defined as:

```
union employee E;
```

To access the members of the union, the dot operator can be used as follows:

```
E.salary;
```

4. Class

A class is an important feature of object-oriented programming language just like C++. A class is defined as a group of objects with the same operations and attributes. It is declared using a keyword "class". The syntax is as follows:

```
class<classname>
{
private:
Data_members;
Member_functions;
public:
Data_members;
Member_functions;
};
```

In this, the names of data members should be different from member functions. There are two access specifiers for classes that define the scope of the members of a class. These are private and public. The member specified as private can be only accessed by the member functions of that particular class only. The members with no specifier are private by default. The objects belonging to a class are called instances of the class.

The syntax for creating an object of a class is as follows:

```
<classname><objectname>
```

5. Enumeration

Enumeration is specified by using a keyword "enum". It is defined as a set of named integer constants that specify all the possible values a variable of that type can have. For example, enumeration of the week can have names of all the seven days of the week as shown below:

Example:

```
enumweek_days{sun, mon, tues, wed, thur, fri, sat};
int main()
{
enumweek_days d;
d = mon;
cout<< d;
return 0;
}
```

6. Pointer

A Pointer is that kind of user-defined data type that creates variables for holding the memory address of other variables. If one variable carries the address of another variable, the first variable is said to be the pointer of another. The syntax for the same is:

```
type *ptr_name;
```

Here type is any data type of the pointer and ptr_name is the pointer's name.

7. Typedef

Using the keyword "typedef", you can define new data type names to the existing ones. Its syntax is:

```
typedef<type><newname>;
typedef float balance;
```

Where a new name is created for float i.e. using balance, we can declare any variable of float type.

Conditional Expression

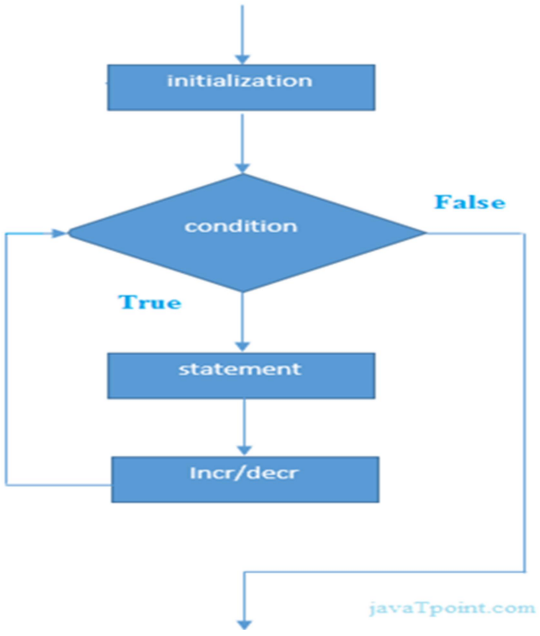
C++ For Loop

The C++ for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops. The C++ for loopis same as C/C#.

We can initialize variable, check condition and increment/decrement value.

```
for(initialization; condition; incr/decr){
//code to be executed
}
```

Flowchart:



C++ For Loop Example

```
#include <iostream>
using namespace std;
int main() {
  for(int i=1;i<=10;i++){
cout<<i <<"\n";
  }
  }
```

Output

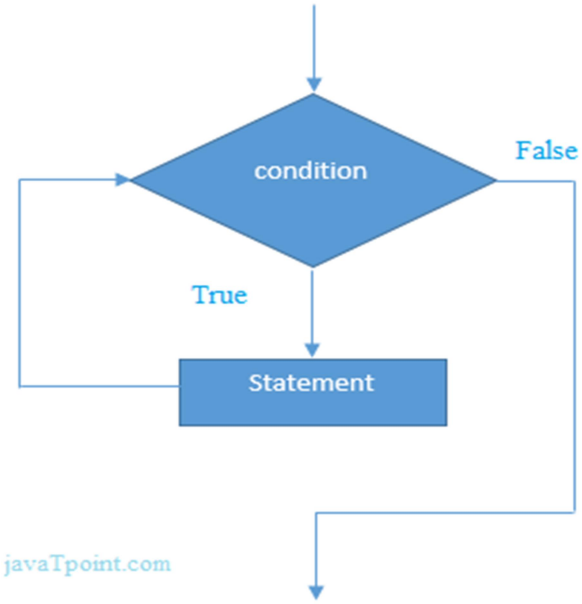
```
1
2
3
4
5
6
7
8
9
10
```

C++ While loop

In C++, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

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

Flowchart:



C++ While Loop Example

Let's see a simple example of while loop.

```
#include <iostream>
using namespace std;
int main() {
int i=1;
  while(i<=10)
{
  cout<<i <<"\n";
```

```
i++;
}
  }
```

Output:

```
1
2
3
4
5
6
7
8
9
```

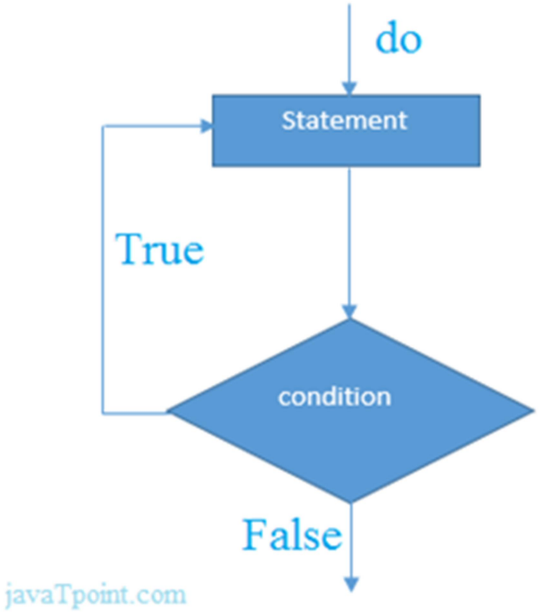
C++ Do-While Loop

The C++ do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C++ do-while loop is executed at least once because condition is checked after loop body.

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

Flowchart:



C++ do-while Loop Example

Let's see a simple example of C++ do-while loop to print the table of 1.

```
#include <iostream>
using namespace std;
int main() {
  int i = 1;
  do
{
 cout<<i<<"\n";
 i++;
  } while (i <= 10) ;
}
```

Output:

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

Breaking control statement

Break Statement

The C++ break is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

```
jump-statement;
break;
```

Flowchart:

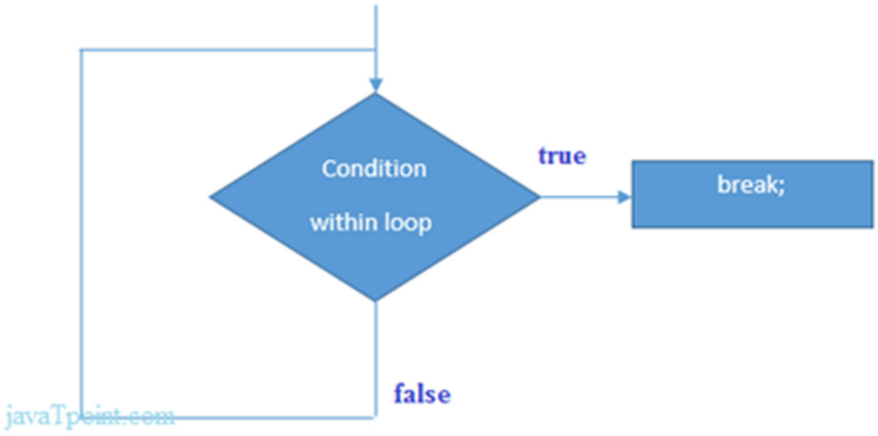


Figure: Flowchart of break statement

Continue Statement in C/C++

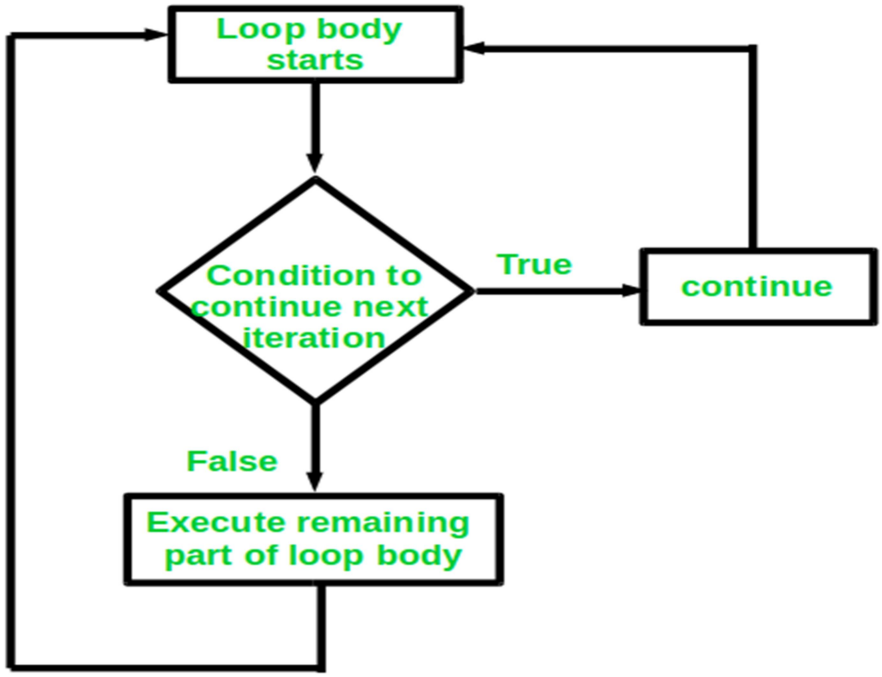
Continue is also a loop control statement just like the break statement. *continue* statement is opposite to that of break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.
As the name suggest the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and next iteration of the loop will begin.

Syntax:

```
Continue
```

/*c program to find sum of n positive numbers read from keyboard*/

```
#include<stdio.h>
int main()
{
int i ,sum = 0,n,number;
cout<<Enter N";
cin>>n;
for(i=1;i<=n;i++)
{
cout<<"Enter a number:";
cin>>number;
if(number<0) continue;
sum = sum + number;
}
cout<<"Sum of"<<n<<" numbers is:"<<sum;
return 0;
}
```



Break Statement in C/C++

The break in C or C++ is a loop control statement which is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stops there and control returns from the loop immediately to the first statement after the loop.

Syntax:

```
break;
```

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop.

```
#include <iostream>
using namespace std;
int main() {
for (int i = 1; i <= 10; i++)
 {
 if (i == 5)
 {
  break;
  }
  cout<<i<<"\n";
  }
}
```

Output:

```
1
2
3
4
```

Continue Statement

The C++ continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

```
jump-statement;
continue;
```

C++ Continue Statement Example

```
#include <iostream>
using namespace std;
int main()
{
  for(int i=1;i<=10;i++){
if(i==5){
   continue;
}
cout<<i<<"\n";
  }
}
```

Output:

```
1
```

```
2
3
4
6
7
8
9
10
```

We will see here the usage of break statement with three different types of loops:
1. Simple loops
2. Nested loops
3. Infinite loops

Let us now look at the examples for each of the above three types of loops using break statement.

1.Simple loops: Consider the situation where we want to search an element in an array. To do this, use a loop to traverse the array starting from the first index and compare the array elements with the given key.
Below is the implementation of this idea:
2.Nested Loops**:** We can also use break statement while working with nested loops. If the break statement is used in the innermost loop. The control will come out only from the innermost loop. Below is the example of using break with nested loops:
3.Infinite Loops: break statement can be included in an infinite loop with a condition in order to terminate theexecution of the infinite loop.

****

Chapter -3
Function

Function

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions. Functions are used to minimize the repetition of code, as a function allows you to write the code inside the block. And you can call that block whenever you need that code segment, rather than writing the code repeatedly. It also helps in dividing the program into well-organized segments.

Now, have a look at the syntax of C++ functions.

Function Syntax

The syntax for creating a function is as follows:

```
return_type function_name ( parameter 1, parameter 2, . . . . )
    {

        //function body

    }
```

Here, the return type is the data type of the value that the function will return. Then there is the function name, followed by the parameters which are not mandatory, which means a function may or may not contain parameters.

Example:

Users > harsh > Downloads > enum3.cpp > ...

```
1  int Average ( int x, int y, int z)
2  {
3
4      //function body
5
6  }
```

Declaration: A function can be declared by writing its return type, the name of the function, and the arguments inside brackets. It informs the compiler that this particular function is present. In C++, if you want to define the function after the main function, then you have to declare that function first. Function declaration is required when you define a function in one source file and you call that function in another file.

```
3
4  int avg(int s1, int s2, int s3);
5
```

Definition:

A function definition specifies the body of the function. The declaration and definition of a function can be made together, but it should be done before calling it.

```
15 int avg(int s1, int s2, int s3)
16 {
17     return(s1+s2+s3)/3;
18 }
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Calling:

When you define a function, you tell the function what to do and to use that function; you have to call or invoke the function. When a function is called from the main function, then the control of the function is transferred to the function that is called. And then that function performs its task. When the task is finished, it returns the control to the main function.

```
 6 int main()
 7 {
 8     int a1=3,a2=8,a3=10;
 9
10     int average=avg(a1,a2,a3);
11
12     cout<<"Average is : "<<average;
13 }
```

Types of Functions

There are two types of functions in C++

- Built-in functions
- User-defined functions

Built-in Functions:

These are functions that are already present in C++; their definitions are already provided in the header files. The compiler picks the definition from header files and uses them in the program.

User-defined Functions:

These are functions that a user creates by themselves, wherein the user gives their own definition.

You can put them down as the following types:

- No argument and no return value
- No argument but the return value
- Argument but no return value
- Argument and return value

-No argument and no return value: In this type, as the name suggests, it passes no arguments to the function, and you print the output within the function. So, there is no return value.

-No argument but return value: In this type, it passes no arguments to the function, but there is a return value.

-Argument but no return value: In this type of function, it passes arguments, but there is no return value. And it prints the output within the function.

-Argument and return value: In this type of function, it passes arguments, and there is also a return value.

Now, move on to the calling methods of C++ functions.

:

Calling a Function: While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

In C++ programs, functions with arguments can be invoked by :

(a) Value (b) Reference

Call by Value: - This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument In this method the values of the actual parameters (appearing in the function call) are copied into the formal parameters (appearing in the function definition), i.e., the function creates its own copy of argument values and operates on them.

The following program illustrates this concept :

```
//calculation of compound interest using a function
    #include<iostream.h>
    #include<conio.h>
    #include<math.h> //for pow()function
    Void main()
    {
     Float principal, rate, time; //local variables
     Void calculate (float, float, float); //function prototype clrscr();
   Cout<<"\nEnter the following values:\n";
   Cout<<"\nPrincipal:";
   Cin>>principal;
```

```
Cout<<"\nRate of interest:";
Cin>>rate;
Cout<<"\nTime period (in yeaers) :";
Cin>>time;
  Calculate (principal, rate, time); //function call
Getch ();
 }
 //function definition calculate()
 Void calculate (float p, float r, float t)
 {
  Float interest; //local variable
  Interest = p* (pow((1+r/100.0),t))-p;
Cout<<"\nCompound interest is : "<<interest;
 }
```

Call by Reference: - This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. A reference provides– an alternate name – for the variable, i.e., the same variable's value can be used by two different names : the original name and the alias name. So the called function does not create its own copy of original value(s) but works with the original value(s) with different name. Any change in the original data in the called function gets reflected back to the calling function. It is useful when you want to change the original variables in the calling function by the called function.

//Swapping of two numbers using function call by reference

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int num1,num2;
void swap (int&, int&); //function prototype
cin>>num1>>num2;
cout<<"\nBefore swapping:\nNum1: "<<num1;
cout<<endl<<"num2: "<<num2;
swap(num1,num2); //function call
cout<<"\n\nAfter swapping : \Num1: "<<num1;
cout<<endl<<"num2: "<<num2;
getch();
}
//function fefinitionswap()
void swap (int& a, int& b)
{
Int temp=a;
 a=b;
 b=temp;
}
```

Inline Functions

These are the functions designed to speed up program execution. An inline function is expanded (i.e. the function code is replaced when a call to the inline function is made) in the line where it is invoked. You are familiar with the fact that in case of normal functions, the compiler have to jump to another location for the execution of the function and then the control is returned back to the instruction immediately after the function call statement. So execution time taken is more in case of normal functions. There is a memory penalty in the case of an inline function.

The system of inline function is as follows :

```
inlinefunction_header
  {   body of the function
  }
```

For example,

```
//function definition min()
inline void min (int x, int y)
cout<< (x < Y? x : y);
 }
 Void main()
 {
int num1, num2;
cout<<"\Enter the two intergers\n";
cin>>num1>>num2;
min (num1,num2; //function code inserted here
   ------------------
   ------------------
 }
```

An inline function definition must be defined before being invoked as shown in the above example.

The inlining does not work for the following situations :

1. For functions returning values and having a loop or a switch or a goto statement.
2. For functions that do not return value and having a return statement.
3. For functions having static variable(s).
4. If the inline functions are recursive (i.e. a function defined in terms of itself).

The benefits of inline functions are as follows :

1. Better than a macro.
2. Function call overheads are eliminated.
3. Program becomes more readable.
4. Program executes more efficiently.

Preprocessor

Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.All Preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

Sr.No.	Directive & Description	
1	**#define**	Substitutes a preprocessor macro.
2	**#include**	Inserts a particular header from another file.
3	**#undef**	Undefines a preprocessor macro.
4	**#ifdef**	Returns true if this macro is defined.
5	**#ifndef**	Returns true if this macro is not defined.
6	**#if**	Tests if a compile time condition is true.
7	**#else**	The alternative for #if.
8	**#elif**	#else and #if in one statement.
9	**#endif**	Ends preprocessor conditional.
10	**#error**	Prints error message on stderr.

Pre-processors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

```
#include<stdio.h>
#include"myheader.h"
```

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

Predefined Macros

Sr.No.	Macro & Description
1	**__DATE__** The current date as a character literal in "MMM DD YYYY" format.
2	**__TIME__** The current time as a character literal in "HH:MM:SS" format.
3	**__FILE__** This contains the current filename as a string literal.
4	**__LINE__** This contains the current line number as a decimal constant.

5 **__STDC__** Defined as 1 when the compiler complies with the ANSI standard.

```
Let's try the following example –
#include<stdio.h>
int main(){
printf("File :%s\n", __FILE__ );
printf("Date :%s\n", __DATE__ );
printf("Time :%s\n", __TIME__ );
printf("Line :%d\n", __LINE__ );
printf("ANSI :%d\n", __STDC__ );
}
When the above code in a file test.c is compiled and executed, it produces
the following result –
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Header Files

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files.

There are two types of header files: the files that the programmer writes and the files that comes with your compiler.You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler. Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

Include Syntax

Both the user and the system header files are included using the pre-processing directive **#include**. It has the following two forms −

```
#include <file>
```

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

Include Operation

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the

included file, followed by the output that comes from the text after the **#include** directive. For example, if you have a header file header.h as follows −

```
char *test (void);
```

and a main program called *program.c* that uses the header file, like this −

```
int x;
#include"header.h"

int main (void){
puts(test ());
}
```

the compiler will see the same token stream as it would if program.c read.

```
int x;
char*test (void);

int main (void){
puts(test ());
}
```

Standard Library Function

The C++ Standard Library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, error checking and many other useful operations. This makes the programmer's job easier, because these functions provide many of the capabilities programmers need. The C++ Standard Library functions are provided as part of the C++ programming environment.

Header files contain the set of predefined standard library functions.
The "#include" preprocessing directive is used to include the header files with ".

Header file names ending in .h are "old-style" header files that have been superseded by the C++ Standard Library header files.

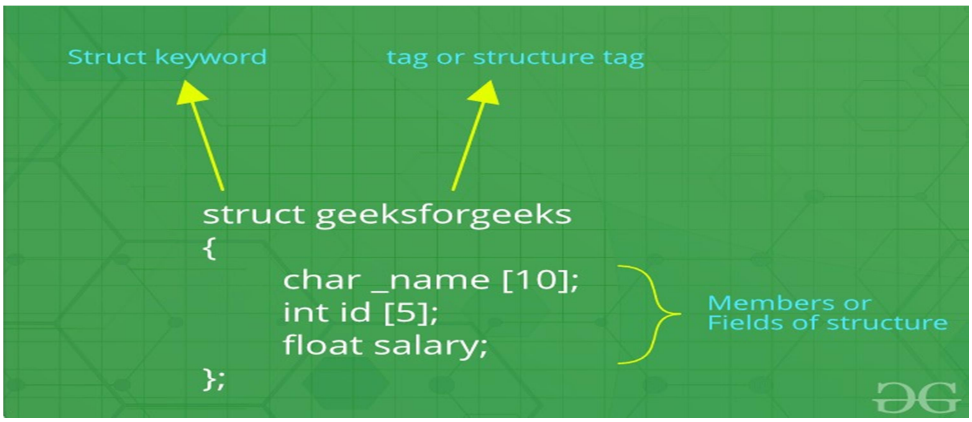
C++ Standard Library header file	Explanation
<iostream>	Contains function prototypes for the C++ standard input and standard output functions. This header file replaces header file <iostream.h>.
<iomanip>	Contains function prototypes for stream manipulators that format streams of data. This header file replaces header file <iomanip.h>.
<cmath>	Contains function prototypes for math library functions. This header file replaces header file <math.h>.
<cstdlib>	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. This header file replaces header file <stdlib.h>.
<ctime>	Contains function prototypes and types for manipulating the time and date. This header file replaces header file <time.h>.
<cctype>	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file <ctype.h>
<cstring>	Contains function prototypes for C-style string-processing functions. This

	header file replaces header file <string.h>.
<cstdio>	Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file <stdio.h>.

...

Structures

Astructure is a user-defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.



How to create a structure?
The 'struct' keyword is used to create a structure. The general syntax to create a structure is as shown below:

```
structstructureName{
   member1;
   member2;
   member3;
   .
   .
memberN;
};
```

Structures in C++ can contain two types of members:

- **Data Member**: These members are normal C++ variables. We can create a structure with variables of different data types in C++.
- **Member Functions**: These members are normal C++ functions. Along with variables, we can also include functions inside a structure declaration.

Example:

```
// Data Members
int roll;
int age;
int marks;
```

```
// Member Functions
void printDetails()
{
   cout<<"Roll = "<<roll<<"\n";
   cout<<"Age = "<<age<<"\n";
   cout<<"Marks = "<<marks;
}
```

In the above structure, the data members are three integer variables to store *roll number, age and marks* of any student and the member function is *printDetails()* which is printing all of the above details of any student.

Pointers and Structures

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

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

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable.The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

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

The actual data type of the value of all pointers, whether integer, float, character.
There are a few important operations, which we will do with the help of pointers very frequently.
(a) We define a pointer variable,
(b)assign the address of a variable to a pointer and
(c)finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

```
#include<stdio.h>
int main (){
intvar=20;/* actual variable declaration */
int*ip;/* pointer variable declaration */
ip=&var;/* store address of var in pointer variable*/
printf("Address of var variable: %x\n",&var);
/* address stored in pointer variable */
printf("Address stored in ip variable: %x\n",ip);
/* access the value using the pointer */
printf("Value of *ip variable: %d\n",*ip);
return0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

Consider the following program −

```
#include<stdio.h>
int main (){
int*ptr= NULL;
printf("The value of ptr is : %x\n",ptr);
return0;
}
```

When the above code is compiled and executed, it produces the following result −

```
The value of ptr is 0
```

Pointers in Detail

Pointers have many but easy concepts and they are very important to C programming.

Sr.No.	Concept & Description
1	Pointer arithmetic — There are four arithmetic operators that can be used in pointers: ++, --, +, -
2	Array of pointers — You can define arrays to hold a number of pointers.
3	Pointer to pointer — C allows you to have pointer on a pointer and so on.
4	Passing pointers to functions in C — Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
5	Return pointer from functions in C — C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

Declaration and Use of Structure Pointers in C++

Just like other pointers, the structure pointers are declared by placing asterisk (∗) in front of a structure pointer's name.It takes the following general form :

struct-name∗struct-pointer ;

wherestruct-name is the name of an already defined structure and struct-pointer is the pointer to this structure. For example, to declare dt-ptr as a pointer to already defined structure date, we shall write

```
date*dt-ptr ;
```

The declaration of structure type and the structure pointer can be combined in one statement. For example,

```
struct date
{
shortintdd, mm, yy ;
} ;
date*dt_ptr ;
```

is same as

```
struct date
```

```
{
shortintdd, mm, yy ;
} *dt_ptr ;
```

Using structure pointers, the members of structures are accessed using arrow operator ->. To refer to the structure members using -> operator, it is written as

```
struct-pointer -> structure-member
```

That is, to access dd and yy with dt_ptr, we shall write

```
dt_ptr ->yy
```

and

```
dt_ptr ->dd
```

C++ Structure Pointers Example

Let's look at the following program which demonstrates the usage of these pointers :

```
/* C++ Pointers and Structures. This C++ program
 * demonstrates about declaration and the use of
 * Structure Pointers in C++ */

#include<iostream.h>
#include<conio.h>
void main()
{
struct date
   {
shortintdd, mm, yy;
}join_date = {19, 12, 2006};
date *date_ptr;
date_ptr = &join_date;
cout<<"Printing the structure elements using the structure variable\n";
cout<<"dd = "<<join_date.dd<<", mm = "<<join_date.mm<<", yy =
"<<join_date.yy<<"\n";
cout<<"\nPrinting the structure elements using the structure pointer\n";
cout<<"dd = "<<date_ptr->dd<<", mm = "<<date_ptr->mm<<", yy = "<<date_ptr-
>yy<<"\n";

getch();
}
```

Unions

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

```
union[union tag]{
member definition;
member definition;
...
member definition;
}[one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

unionData{

```
int i;
float f;
```

```
charstr[20];
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data.The memory occupied by a union will be large enough to hold the largest member of the union.

The following example displays the total memory size occupied by the above union −

```
#include<stdio.h>
#include<string.h>
unionData{
int i;
float f;
charstr[20];
};
int main(){
unionDatadata;
printf("Memory size occupied by data : %d\n",sizeof(data));
return0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Memory size occupied by data : 20
```

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

```
#include<stdio.h>
#include<string.h>
unionData{
int i;
float f;
charstr[20];
};
int main(){
unionDatadata;
data.i=10;
data.f=220.5;
strcpy(data.str,"C Programming");
printf("data.i : %d\n",data.i);
printf("data.f : %f\n",data.f);
printf("data.str : %s\n",data.str);
return0;
}
```

When the above code is compiled and executed, it produces the following result −

```
data.i : 1917853763
data.f : 412236058032779486045275994368.000000
data.str : C Programming
```

Enumeration

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword **enum** is used.

```
enum season { spring, summer, autumn, winter };
```

Here, the name of the enumeration is *season*.

And, *spring*, *summer* and *winter* are values of type *season*.

By default, *spring* is 0,*summer* is 1 and so on. You can change the default value of an enum element during declaration (if necessary).

```
enum season
{   spring = 0,
summer = 4,
autumn = 8,
winter = 12
};
```

Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enumboolean { false, true };

// inside function
enumboolean check;
```

Here, a variable *check* of type **`enumboolean`** is created.

Classes

A class is a blueprint for the object.We can think of a class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Create a Class

A class is defined in C++ using keyword `class` followed by the name of the class.

The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.

```
classclassName {
// some data
// some functions
};
```

For example,

```
classRoom {
public:
double length;
double breadth;
double height;
doublecalculateArea(){
return length * breadth;
        }
doublecalculateVolume(){
return length * breadth * height;
        }
};
```

Here, we defined a class named `Room`.

The variables *length*, *breadth*, and *height* declared inside the class are known as **data members**. And, the functions `calculateArea()` and `calculateVolume()` are known as **member functions** of a class.

Member functions

Member functions are operators and functions that are declared as members of a class. Member functions do not include operators and functions declared with the friend specifier. These are called *friends* of a class. You can declare a member function as static; this is called a static member function. A member function that is not declared as static is called a nonstatic member function.

The definition of a member function is within the scope of its enclosing class. The body of a member function is analyzed after the class declaration so that members of that class can be used in the member function body, even if the member function definition appears before the declaration of that member in the class member list. When the function add() is called in the following example, the data variables a, b, and c can be used in the body of add().

.

```
class x
{
public:
int add()              // inline member function add
      {returna+b+c;};
private:
inta,b,c;
};
```

Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.To use the data and access functions defined in the class, we need to create objects.

Syntax to Define Object in C++

```
classNameobjectVariableName;
```

We can create objects of `Room` class (defined in the above example) as follows:

```
// sample function
voidsampleFunction(){
// create objects
    Room room1, room2;
}

intmain(){
// create objects
    Room room3, room4;
}
```

Here, two objects *room1* and *room2* of the `Room` class are created in `sampleFunction()`. Similarly, the objects *room3* and *room4* are created in `main()`.

As we can see, we can create objects of a class in any function of the program. We can also create objects of a class within the class itself, or in other classes.

Also, we can create as many objects as we want from a single class.

Access Data Members and Member Functions

We can access the data members and member functions of a class by using a . (dot) operator. For example,

```
room2.calculateArea();
```

This will call the `calculateArea()` function inside the `Room` class for object *room2*.Similarly, the data members can be accessed as:

```
room1.length = 5.5;
```

In this case, it initializes the *length* variable of *room1* to `5.5`.

Output

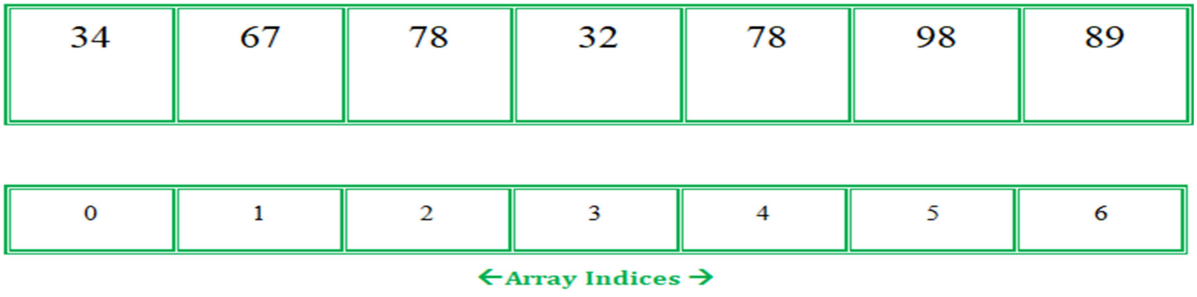
```
Area of Room =  1309
Volume of Room =  25132.8
```

Array of Objects

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They can be used to store the collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as structures, pointers, etc. Given below is the picture representation of an array.

Example:

Let's consider an example of taking random integers from the user.



Array of Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

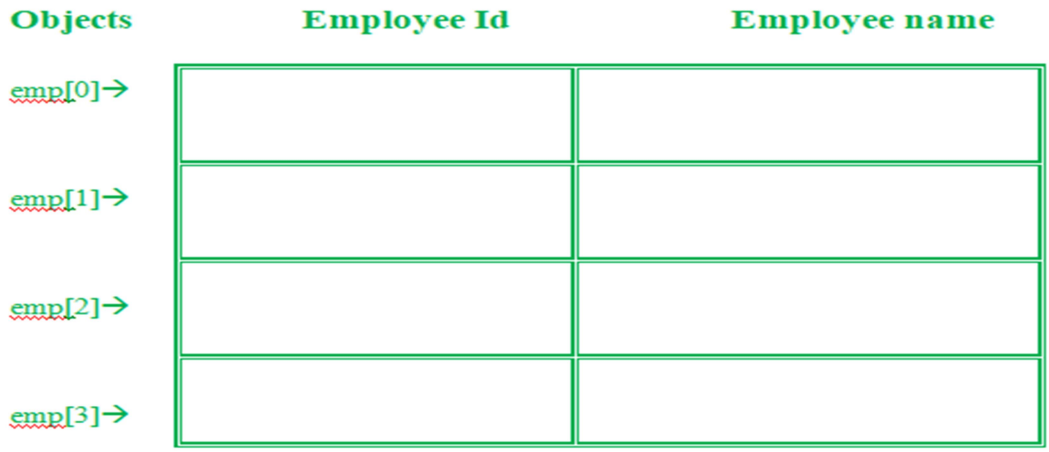
Syntax:

ClassNameObjectName[number of objects];

The Array of Objects stores *objects*. An array of a class type is also known as an array of objects.

Example#1:

Storing more than one Employee data. Let's assume there is an array of objects for storing employee data emp[50].



Below is the C++ program for storing data of one Employee:

```
// C++ program to implement
// the above approach
#include<iostream>
usingnamespacestd;

classEmployee
{
  intid;
  charname[30];
  public:
  voidgetdata();//Declaration of
function
  voidputdata();//Declaration of
function
};
voidEmployee::getdata(){//Defining of
function
  cout<<"Enter Id : ";
  cin>>id;
  cout<<"Enter Name : ";
  cin>>name;
}
voidEmployee::putdata(){//Defining of
function
```

```
  cout<<id<<" ";
  cout<<name<<" ";
  cout<<endl;
}
intmain(){
  Employee emp; //One member
  emp.getdata();//Accessing the
function
  emp.putdata();//Accessing the
function
  return0;
 }
```

Let's understand the above example –

- In the above example, a class named Employee with id and name is being considered.
- The two functions are declared-
    - **getdata():** Taking user input for id and name.
    - **putdata():** Showing the data on the console screen.

This program can take the data of only one Employee. What if there is a requirement to add data of more than one Employee. Here comes the answer Array of Objects. An array of objects can be used if there is a need to store data of more than one employee. Below is the C++ program to implement the above approach-

Explanation:
In this example, more than one Employee's details with an Employee id and name can be stored.

- Employee emp[30] – This is an array of objects having a maximum limit of 30 Employees.
- Two for loops are being used-
    - First one to take the input from user by calling emp[i].getdata() function.
    - Second one to print the data of Employee by calling the function emp[i].putdata() function.

Advantages of Array of Objects:

1. The array of objects represent storing multiple objects in a single name.
2. In an array of objects, the data can be accessed randomly by using the index number.
3. Reduce the time and memory by storing the data in a single variable.

Nested Classes in C++

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

For example, program 1 compiles without any error and program 2 fails in compilation.

Program

```
#include<iostream>
 usingnamespacestd;
 /* start of Enclosing class declaration */
classEnclosing {
   private:
       intx;
     /* start of Nested class declaration */
   classNested {
      inty;
      voidNestedFun(Enclosing *e) {
        cout<<e->x;  // works fine: nested
class can access
                     // private members of
Enclosing class
      }
   }; // declaration Nested class ends here
}; // declaration Enclosing class ends here

intmain()
{
 }
```

Constructors

A constructor is a special type of member function that is called automatically when an object is created.n C++, a constructor has the same name as that of the class and it does not have a return type.

For example,

```
classWall {
public:
// create a constructor
Wall() {
// code
    }
};
```

Here, the function `Wall()` is a constructor of the class `Wall`. Notice that the constructor

- has the same name as the class,
- does not have a return type, and
- is `public`

Copy Constructor

The copy constructor in C++ is used to copy data of one object to another.

Example : C++ Copy Constructor

```
#include<iostream>
usingnamespacestd;

// declare a class
classWall {
private:
double length;
double height;

public:

// initialize variables with parameterized constructor
Wall(doublelen, doublehgt) {
length = len;
height = hgt;
    }

// copy constructor with a Wall object as parameter
// copies data of the obj parameter
Wall(Wall &obj) {
length = obj.length;
height = obj.height;
    }

doublecalculateArea(){
return length * height;
    }
};

intmain(){
// create an object of Wall class
Wall wall1(10.5, 8.6);

// copy contents of wall1 to wall2
  Wall wall2 = wall1;

// print areas of wall1 and wall2
cout<<"Area of Wall 1: "<< wall1.calculateArea() <<endl;
cout<<"Area of Wall 2: "<< wall2.calculateArea();

return0;
}
```

Output

```
Area of Wall 1: 90.3
Area of Wall 2: 90.3
```

Destructor

A destructor destroys an object after it is no longer in use. The destructor, like constructor, is a member function with the same name as the class name. But it will be preceded by the character Tilde (~).A destructor takes no arguments and has no return value. Each class has exactly one destructor.

// A Program showing working of constructor and destructor

#include<iostream.h>

#include<conio.h>

classMyclass{

```
public:
int x;
Myclass(){  //Constructor
x=10; }
~Myclass(){  //Destructor
cout<<"Destructing…." ;
}
int main(){
Myclass ob1, ob2;
cout<<ob1.x<<" "<<ob2.x;
return 0; }
Output:
10 10
Destructing……..
Destructing……..
```

Special Characteristics of destructors are :

(i) These are called automatically when the objects are destroyed.
(ii) Destructor functions follow the usual access rules as other member functions.
(iii) These de-initialize each object before the object goes out of scope.
(iv) No argument and return type (even void) permitted with destructors.
(v) These cannot be inherited.
(vi) Static destructors are not allowed.
(vii) Address of a destructor cannot be taken.
(viii) A destructor can call member functions of its class.
(ix) An object of a class having a destructor cannot be a member of a union.

Inline member function.

A member function that is defined inside its class member list is called an *inline member function*. Member functions containing a few lines of code are usually declared inline. In the above example, `add()` is an inline member function. If you define a member function outside of its class definition, it must appear in a namespace scope enclosing the class definition. You must also qualify the member function name using the scope resolution (`::`) operator.

An equivalent way to declare an inline member function is to either declare it in the class with the `inline` keyword (and define the function outside of its class) or to define it outside of the class declaration using the `inline` keyword.

In the following example, member function `Y::f()` is an inline member function:

```
struct Y {
private:
char* a;
public:
char* f() { return a; }
};
```

The following example is equivalent to the previous example; `Y::f()` is an inline member function:

```
struct Y {
private:
char* a;
public:
char* f();
};

inline char* Y::f() { return a; }
```

Static Member Function

The static is a keyword in the C and C++ programming language. We use the static keyword to define the static data member or static member function inside and outside of the class. Let's understand the static data member and static member function using the programs.

Static data member

When we define the data member of a class using the static keyword, the data members are called the static data member. A static data member is similar to the static member function because the static data can only be accessed using the static data member or static member function. And, all the objects of the class share the same copy of the static member to access the static data.

Syntax

1. **static** data_type data_member;

Here, the **static** is a keyword of the predefined library.
The **data_type** is the variable type in C++, such as int, float, string, etc.
The **data_member** is the name of the static data.No. of objects created in the class: 2

Static Member Functions

The static member functions are special functions used to access the static data members or other static member functions. A member function is defined using the static keyword. A static member function shares the single copy of the member function to any number of the class' objects. We can access the static member function using the class name or class' objects. If the static member function accesses any non-static data member or non-static member function, it throws an error.

Syntax

class_name::function_name (parameter);
Here, the **class_name** is the name of the class.
function_name: The function name is the name of the static member function.

parameter: It defines the name of the passarguments to the static member function.

Example 2: Let's create another program to access the static member function using the class name in the C++ programming language.

```
#include <iostream>
using namespace std;
class Note
{
// declare a static data member
static int num;
```

```
public:
// create static member function
static int func ()
{
return num;
}
};
// initialize the static data member using the class name and the scope resolution operator
int Note :: num = 5;

int main ()
{
// access static member function using the class name and the scope resolution
cout << " The value of the num is: " << Note:: func () << endl;
return 0;
}
```

Output

```
The value of the num is: 5
```

Friend function

In general, only other members of a class have access to the private members of the class. The function is declared with friend keyword. But while defining friend function, it does not use either keyword friend or :: operator. A function can be a friend of more than one class. A friend, function has following characteristics.

- It is not in the scope of the class to which it has been declared as friend.
- A friend function cannot be called using the object of that class. If can be invoked like a normal function without help of any object.
- It cannot access the member variables directly & has to use an object name dot membership operator with member name.
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the object as arguments.

A **friend function** can access the **private** and **protected** data of a class. We declare a friend function using the `friend` keyword inside the body of the class.

```
classclassName {
    ... .. ...
friendreturnTypefunctionName(arguments);
    ... .. ...
}
```

// C++ program to demonstrate the working of friend function

```
#include <iostream>
using namespace std;
class Distance {
```

```
private:
int meter;
// friend function
friendintaddFive(Distance);
public:
Distance() : meter(0) {}
};
// friend function definition
intaddFive(Distance d) {
//accessing private members from the friend function
d.meter += 5;
returnd.meter;
}
int main() {
    Distance D;
cout<< "Distance: " <<addFive(D);
return 0;
}
```

Output

```
Distance: 5
```

Dynamic memory allocation

There are times where the data to be entered is allocated at the time of execution. For example, a list of employees increases as the new employees are hired in the organization and similarly reduces when a person leaves the organization. This is called managing the memory. So now, let us discuss the concept of dynamic memory allocation.

Memory allocation

Reserving or providing space to a variable is called memory allocation. For storing the data, memory allocation can be done in two ways -

- **Static allocation or compile-time allocation -** Static memory allocation means providing space for the variable. The size and data type of the variable is known, and it remains constant throughout the program.
- **Dynamic allocation or run-time allocation -** The allocation in which memory is allocated dynamically. In this type of allocation, the exact size of the variable is not known in advance. Pointers play a major role in dynamic memory allocation.

Why Dynamic memory allocation?

Dynamically we can allocate storage while the program is in a running state, but variables cannot be created "on the fly". Thus, there are two criteria for dynamic memory allocation -

- A dynamic space in the memory is needed.
- Storing the address to access the variable from the memory

Similarly, we do memory de-allocation for the variables in the memory.

In C++, memory is divided into two parts -

- **Stack -** All the variables that are declared inside any function take memory from the stack.
- **Heap -** It is unused memory in the program that is generally used for dynamic memory allocation.

****

Chapter-4

Inheritance

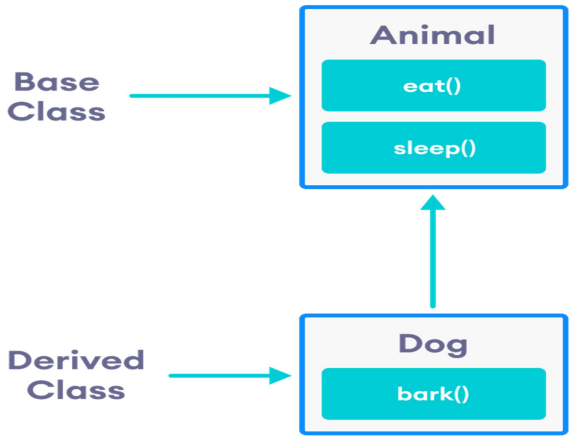
Inheritance

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).The derived class inherits the features from the base class and can have additional features of its own. Inheritance is one of the core feature of an object-oriented programming language. It allows software developers to derive a new class from the existing class. The derived class inherits the features of the base class (existing class).

For example,

```
class Animal {
    // eat() function
    // sleep() function
};
class Dog : public Animal {
    // bark() function
};
```

Here, the `Dog` class is derived from the `Animal` class. Since `Dog` is derived from `Animal`, members of `Animal` are accessible to `Dog`.



I

Inheritance in C++: Notice the use of the keyword `public` while inheriting Dog from Animal.

```
class Dog : public Animal {...};
```

We can also use the keywords `private` and `protected` instead of `public`. We will learn about the differences between using `private`, `public` and `protected` later in this tutorial.

is-a relationship

Inheritance is an **is-a relationship**. We use inheritance only if an **is-a relationship** is present between the two classes.

Here are some examples:

- A car is a vehicle.
- Orange is a fruit.
- A surgeon is a doctor.

- A dog is an animal.

Example : Simple Example of C++ Inheritance

```
// C++ program to demonstrate inheritance

#include <iostream>
using namespace std;
// base class
class Animal {
public:
void eat() {
cout<< "I can eat!" <<endl;
    }
void sleep() {
cout<< "I can sleep!" <<endl;
    }
};
// derived class
class Dog : public Animal {
public:
void bark() {
cout<< "I can bark! Woof woof!!" <<endl;
    }
};
int main() {
    // Create object of the Dog class
    Dog dog1;
  // Calling members of the base class
dog1.eat();
dog1.sleep();
    // Calling member of the derived class
dog1.bark();
return 0;
}
```

Output

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

Here, *dog1* (the object of derived class `Dog`) can access members of the base class `Animal`. It's because `Dog` is inherited from `Animal`.

Visibility Modes (Access Specifiers) in C++

1. Private.
2. Protected.
3. Public.

Private Access Specifier

1. The **private** access specifier is the default access specifier in C++ class.
2. Member variables declared within **private** access specifier can only be accessed by its member function declared within the class.
3. It cannot be accessed outside the class.
4. So, **private** access specifier is used for implementing encapsulation (data hiding).

5. When we want to protect the important data from the user then we use **private** access specifier.

Protected Access Specifier

1. **Protected** access specifier and function overriding are used in the case of inheritance when user wants to access data of different classes in the inheritance.
2. *Child* class can directly access *protected* members of its *parent* class.

Public Access Specifier

1. **Public** access specifier allows the access of data members from inside the class and also from outside the class.
2. Each and every data member and member function declared with **public** access specifier can be accessed from anywhere in the program.
3. Generally, member functions are declared with **public** access specifier so that the user can access it and make some operations on the objects.
4. ***is a*** relationship is always implemented as a ***public*** inheritance because object of derived class wants to access the public members of base class.

Types of Inheritance in C++

1. Single Inheritance.
2. Multiple Inheritance.
3. Multi-level Inheritance.
4. Hierarchical Inheritance.
5. Hybrid Inheritance.

Multilevel Inheritance

In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

```
class A {
... .. ...
};
class B: public A {
... .. ...
};
class C: public B {
... ... ...
};
```

Here, class *B* is derived from the base class *A* and the class *C* is derived from the derived class *B*.

Example : C++ Multilevel Inheritance

```
#include<iostream>
usingnamespacestd;
classA {
public:
voiddisplay(){
cout<<"Base class content.";
      }
};
classB :public A {};
classC :public B {};
```

```
intmain(){
    C obj;
obj.display();
return0;
}
```

Output

```
Base class content.
```

In this program, class *C* is derived from class *B* (which is derived from base class *A*).
The *obj* object of class *C* is defined in the `main()` function.
When the `display()` function is called, `display()` in class *A* is executed. It's because there is no `display()` function in class *C* and class *B*.
The compiler first looks for the `display()` function in class *C*. Since the function doesn't exist there, it looks for the function in class *B* (as *C* is derived from *B*).
The function also doesn't exist in class *B*, so the compiler looks for it in class *A* (as *B* is derived from *A*).
If `display()` function exists in *C*, the compiler overrides `display()` of class *A* (because of member function overriding).

Hierarchical Inheritance

If more than one class is inherited from the base class, it's known as hierarchical inheritance. In hierarchical inheritance, all features that are common in child classes are included in the base class. For example, Physics, Chemistry, Biology are derived from Science class. Similarly, Dog, Cat, Horse are derived from Animal class.

Syntax of Hierarchical Inheritance

```
classbase_class {
     ... .. ...
}
classfirst_derived_class: public base_class {
     ... .. ...
}
classsecond_derived_class: public base_class {
     ... .. ...
}
classthird_derived_class: public base_class {
     ... .. ...
}
```

Example: Hierarchical Inheritance in C++ Programming

```
// C++ program to demonstrate hierarchical inheritance
#include<iostream>
usingnamespacestd;
// base class
classAnimal {
public:
voidinfo(){
cout<<"I am an animal."<<endl;
    }
};
// derived class 1
classDog :public Animal {
public:
voidbark(){
cout<<"I am a Dog. Woof woof."<<endl;
    }
};
// derived class 2
classCat :public Animal {
```

```
public:
voidmeow(){
cout<<"I am a Cat. Meow."<<endl;
    }
};
intmain(){
// Create object of Dog class
    Dog dog1;
cout<<"Dog Class:"<<endl;
dog1.info();  // Parent Class function
dog1.bark();
// Create object of Cat class
    Cat cat1;
cout<<"\nCat Class:"<<endl;
cat1.info();  // Parent Class function
cat1.meow();
return0;
}
```

Output

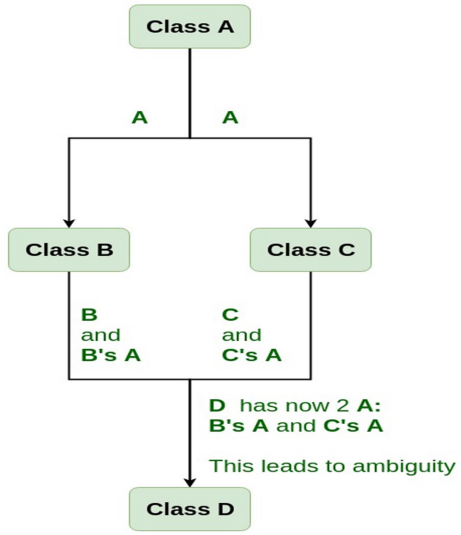
```
Dog Class:
I am an animal.
I am a Dog. Woof woof.

Cat Class:
I am an animal.
I am a Cat. Meow.
```

Here, both the `Dog` and `Cat` classes are derived from the `Animal` class. As such, both the derived classes can access the `info()` function belonging to the `Animal` class.

Virtual base class

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances. Need for Virtual BaseClasses. Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Example: To show the need of Virtual Base Class in C++

```
#include <iostream>
usingnamespacestd;
classA {
public:
    voidshow()
    {
        cout<< "Hello form A \n";
    }
};
 classB : publicA {
};
classC : publicA {
};
 classD : publicB, publicC {
};
 intmain()
{
    D object;
    object.show();
}
```

How to resolve this issue?

To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual**as :

Syntax for Virtual Base Classes:

Syntax 1:

```
class B : virtual public A
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

Note:virtual can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example

```
#include <iostream>
usingnamespacestd;
 classA {
public:
    inta;
    A() // constructor
```

```
    {
        a = 10;
    }
};
 classB : publicvirtualA {
};
 classC : publicvirtualA {
};
 classD : publicB, publicC {
};
 intmain()
{
    D object; // object creation of class d
    cout<< "a = "<<object.a<<endl;
 return0;
}
```

Output:

```
a = 10
```

Explanation : The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

Abstract classes

Abstract classes act as expressions of general concepts from which more specific classes can be derived. You can't create an object of an abstract class type. However, you can use pointers and references to abstract class types.

You create an abstract class by declaring at least one pure virtual member function. That's a virtual function declared by using the *pure* specifier (`= 0`) syntax. Classes derived from the abstract class must implement the pure virtual function or they, too, are abstract classes.

Consider the example presented in Virtual functions. The intent of class `Account` is to provide general functionality, but objects of type `Account` are too general to be useful. That means `Account` is a good candidate for an abstract class:

C++

```
// deriv_AbstractClasses.cpp
// compile with: /LD
classAccount {
public:
Account(double d );   // Constructor.
virtualdoubleGetBalance();   // Obtain balance.
virtualvoidPrintBalance()= 0;   // Pure virtual function.
private:
double _balance;
};
```

The only difference between this declaration and the previous one is that `PrintBalance` is declared with the pure specifier (`= 0`).

Restrictions on abstract classes

Abstract classes can't be used for:

- Variables or member data

- Argument types
- Function return types
- Types of explicit conversions

If the constructor for an abstract class calls a pure virtual function, either directly or indirectly, the result is undefined. However, constructors and destructors for abstract classes can call other member functions.

Constructors in Derived Class

- We can use constructors in derived classes in C++
- If the base class constructor does not have any arguments, there is no need for any constructor in the derived class
- But if there are one or more arguments in the base class constructor, derived class need to pass argument to the base class constructor
- If both base and derived classes have constructors, base class constructor is executed first

Syntax Example:

```
Derived-Constructor (arg1, arg2, arg3….): Base 1-Constructor (arg1,arg2),
Base 2-Constructor(arg3,arg4)
{
….
} Base 1-Constructor (arg1,arg2)
```

Special Case of Virtual Base Class

The constructors for virtual base classes are invoked before a non-virtual base class
If there are multiple virtual base classes, they are invoked in the order declared
Any non-virtual base class are then constructed before the derived class constructor is executed

```
#include<iostream>
using namespace std;
classbaseClass
{
public:
baseClass()
  {
cout<< "I am baseClass constructor" <<endl;
  }
  ~baseClass()
  {
cout<< "I am baseClass destructor" <<endl;
  }
  };
classderivedClass: public baseClass
  {
public:
derivedClass()
  {
cout<< "I am derivedClass constructor" <<endl;
  }
  ~derivedClass()
  {
cout<<" I am derivedClass destructor" <<endl;
  }
  };
int main()
```

```
  {
derivedClass D;
return 0;
  }
```

Points to Remember:-

```
class C: public A, public B

 {

   //...

 };
```

1. Here, A class in inherited first, so constructor of class A is called first then the constructor of class B will be called next.
2. The destructor of derived class will be called first then destructor of base class which is mentioned in the derived class declaration is called from last towards first sequence wise.

Nesting of Classes

A nesting of class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.
1.Class inside a class is called *nested class*.
2. *Nested classes* are declared inside another *enclosing class*.
3. A *nested class* is a *member* of class and it follows the same access rights that are followed by different members of class.
4. The members of an enclosing class have no other special access to members of *nested class.* The normal access rules shall be carried out.
5. If *nested class* is declared after *public* access specifiers inside the enclosing class then you must add *scope resolution* (::) during creating its *object* inside main function.

Program

```
#include<iostream>
 usingnamespacestd;
  /* start of Enclosing class declaration */
classEnclosing {
   private:
       intx;
      /* start of Nested class declaration */
   classNested {
      inty;
      voidNestedFun(Enclosing *e) {
```

```
        cout<<e->x;  // works fine: nested class can access
                     // private members of Enclosing class
      }
   }; // declaration Nested class ends here
}; // declaration Enclosing class ends here
 intmain()
{
}
```

****

Chapter-5

Function Overloading

Function Overloading

Function overloading is a feature of object oriented programming where two or more functions can have the same name but different parameters.
When a function name is overloaded with different jobs it is called Function Overloading.
In Function Overloading "Function" name should be the same and the arguments should be different.
Function overloading can be considered as an example of polymorphism feature in C++.
Following is a simple C++ example to demonstrate function overloading.
simple C++ example to demonstrate function overloading.

```
#include <iostream>
usingnamespacestd;

voidprint(inti) {
  cout<< " Here is int "<< i <<endl;
}
voidprint(double f) {
  cout<< " Here is float "<< f <<endl;
}
voidprint(charconst*c) {
  cout<< " Here is char* "<< c <<endl;
}
intmain() {
  print(10);
  print(10.10);
  print("ten");
  return0;
}
```

Output:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

Operator Overloading

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

A simple and complete example

```
#include<iostream>
usingnamespacestd;
 classComplex {
private:
    intreal, imag;
public:
    Complex(intr = 0, inti = 0) {real = r;   imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const&obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         returnres;
    }
    voidprint() { cout<< real << " + i"<<imag<< '\n'; }
};

intmain()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output:

```
   12  i9
```

Important points about operator overloading

1) For operator overloading to work, at least one of the operands must be a user defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor). See this for more details.

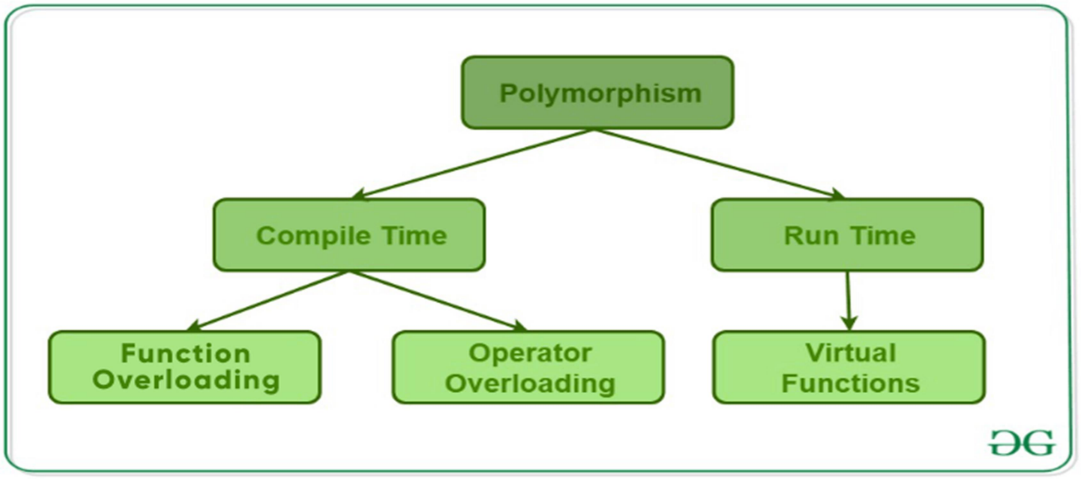
3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



Types of Polymorphism

- **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments. Operator Overloading: C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them

- **Runtime polymorphism:** This type of polymorphism is achieved by Function Overriding.

Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overloading
#include <bits/stdc++.h>

usingnamespacestd;
classGeeks
{
    public:
    // function with 1 int parameter
    voidfunc(intx)
    {
        cout<< "value of x is "<< x <<endl;
    }
     // function with same name but 1 double parameter
    voidfunc(doublex)
    {
        cout<< "value of x is "<< x <<endl;
    }
```

```
    // function with same name and 2 int parameters
    voidfunc(intx, inty)
    {
        cout<< "value of x and y is "<< x << ", "<< y <<endl;
    }
};
intmain() {
      Geeks obj1;
    // Which function is called will depend on the parameters passed
    // The first 'func' is called
    obj1.func(7);
    // The second 'func' is called
    obj1.func(9.132);

    // The third 'func' is called
    obj1.func(85,64);
    return0;
}
```

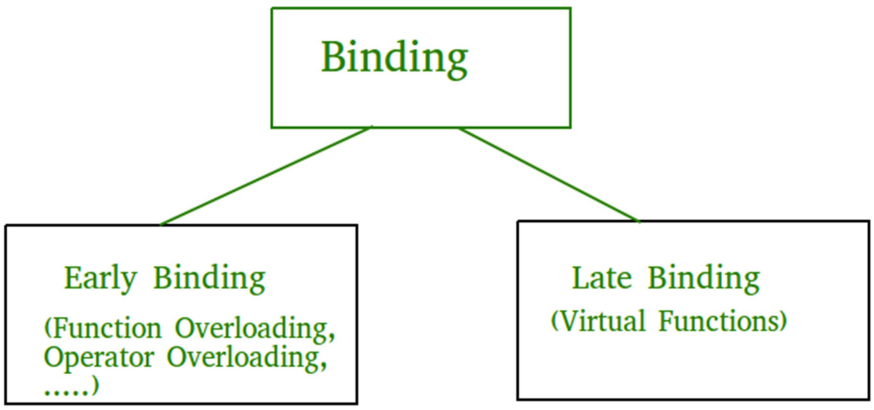
Output:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

In the above example, a single function named *func* acts differently in three different situations which is the property of polymorphism.

Early binding and Late binding

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Early Binding (compile-time time polymorphism) As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

By default early binding happens in C++. Late binding (discussed below) is achieved with the help of virtual keyword)

```
// CPP Program to illustrate early binding.
// Any normal function call (without virtual)
// is binded early. Here we have taken base
// and derived class example so that readers
// can easily compare and see difference in
// outputs.
#include<iostream>
usingnamespacestd;

classBase
{
public:
    voidshow() { cout<<" In Base \n"; }
};

classDerived: publicBase
{
public:
    voidshow() { cout<<"In Derived \n"; }
};

intmain(void)
{
    Base *bp = newDerived;

    // The function call decided at
    // compile time (compiler sees type
    // of pointer and calls base class
    // function.
    bp->show();

    return0;
}
```

Output:

```
In Base
```

Late Binding : **(Run time polymorphism)** In this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition. This can be achieved by declaring a virtual function.

```
// CPP Program to illustrate late binding
#include<iostream>
usingnamespacestd;
    classBase
{
public:
```

```
    virtualvoidshow() { cout<<" In Base \n"; }
};
 classDerived: publicBase
{
public:
    voidshow() { cout<<"In Derived \n"; }
};
   intmain(void)
{
    Base *bp = newDerived;
    bp->show();  // RUN-TIME POLYMORPHISM
    return0;
}
```

Output:

```
In Derived
```

Virtual Function

A virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at runtime.

Rules for Virtual Functions

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

Compile time (early binding) VS runtime (late binding) behavior of Virtual Functions

Consider the following simple program showing runtime behavior of virtual functions.

```
// CPP program to illustrate
// concept of Virtual Functions

#include<iostream>
usingnamespacestd;
```

```
classbase {
public:
    virtualvoidprint()
    {
        cout<< "print base class\n";
    }
 voidshow()
    {
        cout<< "show base class\n";
    }
};
classderived : publicbase {
public:
    voidprint()
    {
        cout<< "print derived class\n";
    }
   voidshow()
    {
        cout<< "show derived class\n";
    }
};
intmain()
{
    base *bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();

    return0;
}
```

Output:

```
print derived class
show base class
```

Pure Virtual Functions

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.
A pure virtual function (or abstract function) in C++ is a virtual function for which we can have implementation, But we must override that function in the derived class, otherwise the

derived class will also become abstract class (For more info about where we provide implementation for such functions. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
classTest
{
    // Data members of class
public:
    // Pure Virtual Function
    virtualvoidshow() = 0;

   /* Other members */
};
```

A complete example:
A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same.

```
#include<iostream>
usingnamespacestd;

classBase
{
   intx;
public:
    virtualvoidfun() = 0;
    intgetX() { returnx; }
};

// This class inherits from Base and implements fun()
classDerived: publicBase
{
    inty;
public:
    voidfun() { cout<< "fun() called"; }
};

intmain(void)
{
    Derived d;
    d.fun();
    return0;
}
```

Output:

```
fun() called
```

Opening and Closing of files

In C++, you open a file, you must first obtain a stream. There are the following three types of streams:

- input
- output

- input/output

Create an Input Stream

To create an input stream, you must declare the stream to be of class ifstream. Here is the syntax:

ifstream fin;

Create an Output Stream

To create an output stream, you must declare it as class ofstream. Here is an example:
ofstreamfout;
Create both Input/Output Streams
Streams that will be performing both input and output operations must be declared as class fstream. Here is an example:
fstreamfio;
Opening a File in C++
Once a stream has been created, next step is to associate a file with it. And thereafter the file is available (opened) for processing.
Opening of files can be achieved in the following two ways :

1. Using the constructor function of the stream class.
2. Using the function open().

The first method is preferred when a single file is used with a stream. However, for managing multiple files with the same stream, the second method is preferred. Let's discuss each of these methods one by one.

Opening File Using Constructors

We know that a constructor of class initializes an object of its class when it (the object) is being created. Same way, the constructors of stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them. This is carried out as explained here:
To open a file named myfile as an input file (i.e., data will be need from it and no other operation like writing or modifying would take place on the file), we shall create a file stream object of input type i.e., ifstream type. Here is an example:
ifstream fin("myfile", ios::in) ;

The above given statement creates an object, fin, of input file stream. The object name is a user-defined name (i.e., any valid identifier name can be given). After creating the ifstream object fin, the file myfile is opened and attached to the input stream, fin. Now, both the data being read from myfile has been channelised through the input stream object.

Now to read from this file, this stream object will be used using the getfrom operator (">>"). Here is an example:

```
charch;
fin>>ch ;       // read a character from the file
floatamt ;
```

```
fin>>amt ;       // read a floating-point number form the file
```

Similarly, when you want a program to write a file i.e., to open an output file (on which no operation can take place except writing only). This will be accomplish by

1. creating ofstream object to manage the output stream
2. associating that object with a particular file

Here is an example,

```
ofstreamfout("secret" ios::out) ;     // create ofstream object named as
fout
```

This would create an output stream, object named as fout and attach the file secret with it

Opening Files Using Open() Function

There may be situations requiring a program to open more than one file. The strategy for opening multiple files depends upon how they will be used. If the situation requires simultaneous processing of two files, then you need to create a separate stream for each file. However, if the situation demands sequential processing of files (i.e., processing them one by one), then you can open a single stream and associate it with each file in turn. To use this approach, declare a stream object without initializing it, then use a second statement to associate the stream with a file. For example,

```
ifstream fin;                          // create an input stream
fin.open("Master.dat", ios::in);       // associate fin stream with file
Master.dat
:                                      // process Master.dat
fin.close();                           // terminate association with
Master.dat

fin.open("Tran.dat", ios::in);         // associate fin stream with file
Tran.dat
:                                      // process Tran.dat
fin.close();                           // terminate association
```

The above code lets you handle reading two files in succession. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time.

List of File Modes in C++

Following table lists the filemodes available in C++ with their meaning :

Constant	Meaning	Stream Type
ios :: in	It opens file for reading, i.e., in input mode.	ifstream

ios :: out	It opens file for writing, i.e., in output mode. This also opens the file in ios :: trunc mode, by default. This means an existing file is truncated when opened, i.e., its previous contents are discarded.	ofstream
ios :: ate	This seeks to end-of-file upon opening of the file. I/O operations can still occur anywhere within the file.	ofstream ifstream
ios :: app	This causes all output to that file to be appended to the end. This value can be used only with files capable of output.	ofstream
ios :: trunc	This value causes the contents of a pre-existing file by the same name to be destroyed and truncates the file to zero length.	ofstream
ios :: nocreate	This cause the open() function to fail if the file does not already exist. It will not create a new file with that name.	ofstream
ios :: noreplace	This causes the open() function to fail if the file already exists. This is used when you want to create a new file and at the same time.	ofstream
ios :: binary	This causes a file to be opened in binary mode. By default, files are opened in text mode. When a file is opened in text mode, various character translations may take place, such as the conversion of carriage-return into newlines. However, no such character translations occur in file opened in binary mode.	ofstream ifstream

Closing a File in C++

As already mentioned, a file is closed by disconnecting it with the stream it is associated with. The close() function accomplishes this task and it takes the following general form :

```
stream_object.close();
```

For example, if a file Master is connected with an ofstream object fout, its connections with the stream fout can be terminated by the following statement :

```
fout.close() ;
```

C++ Opening and Closing a File Example

Here is an example given, for the complete understanding on:

- how to open a file in C++ ?
- how to close a file in C++ ?

Let's look at this program.

```
/* C++ Opening and Closing a File
 * This program demonstrates, how
 * to open a file to store or retrieve
 * information to/from it. And then how
 * to close that file after storing
 * or retrieving the information to/from it. */


#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
ofstreamfout;
ifstream fin;
charfname[20];
char rec[80], ch;
clrscr();

cout<<"Enter file name: ";
cin.get(fname, 20);

fout.open(fname, ios::out);

if(!fout)
   {
cout<<"Error in opening the file "<<fname;
getch();
exit(1);
   }
cin.get(ch);

cout<<"\nEnter a line to store in the file:\n";
cin.get(rec, 80);
fout<<rec<<"\n";
cout<<"\nThe entered line stored in the file successfully..!!";
cout<<"\nPress any key to see...\n";
getch();
fout.close();

fin.open(fname, ios::in);
if(!fin)
   {
cout<<"Error in opening the file "<<fname;
cout<<"\nPress any key to exit...";
getch();
exit(2);
   }

cin.get(ch);
fin.get(rec, 80);
cout<<"\nThe file contains:\n";
cout<<rec;
cout<<"\n\nPress any key to exit...\n";
fin.close();

getch();
}
```

Random File Access Functions

Now we have a file open for reading. By default, the file is open and the cursor is sitting at the beginning of the file. But now we want to move the **file pointer**, that is the read/write location inside your file. Note that we told C++ to open the file in append mode. This means our pointer is now at the END of the file.

Once you have a file open for processing, you can navigate to different parts of the file. This is often referred to as **random access** of files. It really means that you aren't at the beginning or the end. Two functions are available:

- istream: **seekg()** (or seek and get)
- ostream: **seekp()** (seek and put)

Both have special ios parameters/flags, similar to the open function of the ifstream.

Parameter	Explanation	Example
Beg	Default setting: start from the beginning	infile.seekg(50, ios::beg);
End	Start from the end of the file	infile.seekg(-50, ios::end);
Cur	Start from location of the pointer	infile.seekg(25, ios::cur);

Seekg()

Let's look at some code. The following first moves the pointer to the 10th byte, then to the 5th, from the current position, and finally goes back from the end 10 bytes. Each time output is displayed**.**

Add this code before the *infile.close()* statement.

```
    stringsearchResult;
    infile.seekg(50);
        getline(infile,searchResult);
        cout<<"50 from Beginning = "<<searchResult<<endl;
        infile.seekg(5,ios::cur);
        getline(infile,searchResult);
        cout<<"5 from Current = "<<searchResult<<endl;
        infile.seekg(-10,ios::end);
        getline(infile,searchResult);
        cout<<"-10 from End = "<<searchResult<<endl;
```

If you copied the text from above into your text file, the output should be as follows:

```
10 from Beginning = llo
5 from Current= Hello
-10 from End = up|Hello
```

Seekp()

The function seekp() is the exact same as seekg(), except it is used for writing files. Either change your current code or create a new C++ file so that you open the brain.txt for writing. We will now have an outfile instead of an infile.

```
1. //output file stream
2. ofstreamoutfile;
3. //open the file
4. //use ios::in and ios::out to ensure seekp does not replace all text
5. outfile.open("F:\\brain.txt",ios::in|ios::out);
```

Now we will use seekp to position to the 12th byte and add some text at that point. By default, the pointer starts at the beginning.

```
1. string insertText="I can't do that, Hal";
2. outfile.seekp(12,ios::beg);
3. outfile<<insertText<<endl;
```

****