Chapter 1

Introduction to Object Oriented Programming

Language Paradigms:

- ✓ Imperatives – Procedural Programming: C, Pascal, COBOL, FORTRAN etc.
- ✓ Applicative – Functional Programming – LISP, ML.
- ✓ Rule-based – Logic Programming: – PROLOG
- ✓ Object-Oriented Programming: – C++, JAVA, SMALLTALK

Since the invention of computer, many programming approaches have been tried. These techniques include modular programming, top-down programming, bottom-up programming and structured programming. The primary motivation in each has been the concern to handle the increasing complexity of programs that are reliable and maintainable.

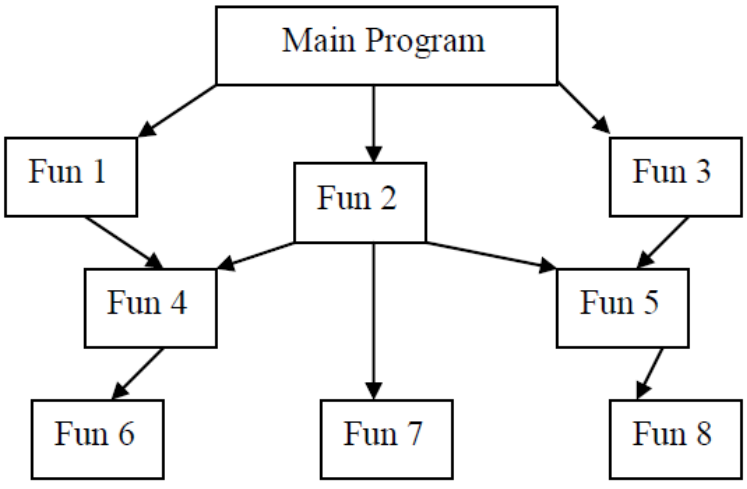
Procedural Oriented Language / Structured Programming Approach

A procedural language is a type of computer programming language that specifies a series of well-structured steps and procedures within its programming context to compose a program. It contains a systematic order of statements, functions and commands to complete a computational task or program. The procedural language divides a program within variables, functions, statements and conditional operators.

In procedural approach,

- A program is a list of instruction.
- When program in PL become larger, they are divided into functions (subroutines, sub-programs, procedures).
- Functions are grouped into modules.

A typical program structure of Procedural programming is shown in figure below



Characteristics of POP

- Emphasis is on doing thing (algorithm).
- Large programs are divided into smaller programs known as functions.
- Data move openly around the system from function to function.
- Employs **top-down approach** in program design.

Disadvantages of POP

- It's emphasis on doing things.
- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function. Similarly, if new data is to be added, all the function needed to be modified to access the data.
- It is difficult to create new data types with procedural languages.
- Most Procedural languages are not usually extensible and hence procedural programs are more complex to write and maintain.

Object-oriented Programming (OOP)

OOP is an approach to programming paradigm that attempts to eliminate some of the drawbacks of conventional programming methods with several powerful new concepts.

The fundamental idea behind object-oriented programming is to combine or encapsulate both data (or instance variables) and functions (or methods) that operate on that data into a single unit. This unit is called an object. The data is hidden, so it is safe from accidental alteration. An object's functions typically provide the only way to access its data. In order to access the data in an object, we should know exactly what functions interact with it. No other functions can access the data. Hence OOP focuses on data portion rather than the process of solving the problem.

The general structure of OOP is shown in the figure below.

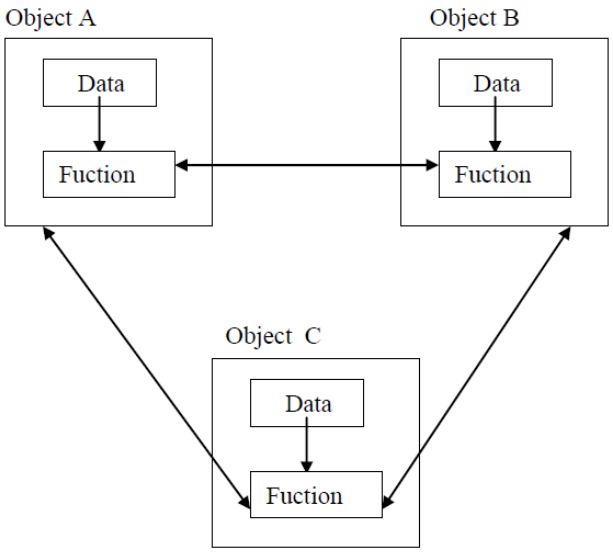


Fig: Organization of data and function in OOP

Characteristics of OOP

- Emphasis is on data rather than procedures.
- Programs are divided into objects.
- Data structures are designed such that they characterize the objects.
- Functions & data are tied together in the data structures so that data abstraction is introduced in addition to procedural abstraction.
- Data is hidden & can't be accessed by external functions.
- Object can communicate with each other through function.
- New data & functions can be easily added.
- Follows Bottom up approach.

Benefits of OOP

- Making the use of inheritance, redundant code is eliminated and the existing class is extended.
- Through data hiding, programmer can build secure program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- System can be easily upgraded from small to large systems.
- Software complexity can be easily managed.
- Message passing technique for communication between objects makes the interface description with external system much simpler.

Disadvantages of OOP

- Object oriented program required greater processing overhead – demands more resources.
- Requires the mastery in software engineering and programming methodology.
- Benefits only in long run while managing large software projects.
- The message passing between many objects in a complex application can be difficult to trace& debug.

Application of OOP

Applications of OOP are beginning to gain importance in many areas.

- The most popular application of OOP, up to now. Has been area of user interface design such as Windows.
- Real Business systems are often much more complex attributes & behaviors.
- OOP can simplify such complex problem. The areas of application of OOP include – Real time systems
- Simulation & modeling
- Object-Oriented databases
- Hypertext, hypermedia
- AI & expert system

- Neural Networks & parallel programming – Decision support & Office automation system – CAM/CAD systems.
- Computer Based training and Educational Systems

Object-Oriented Programming Languages:

Smalltalk :

- Developed by Allen Kay at Xerox Palo Alto Research center (PARC) in 1970's
- 100% OO Language
- The syntax is very unusual and this leads to learning difficulties for programmers who are used to conventional language syntax.
- Language of this type allocate memory space for object on the heap and dynamic garbage collector required.

Eiffel:

- Eiffel was designed by a Frenchman named Bertrand Meyer in the late 1980's.
- Syntax is very elegant & simple, fewer reserved world than Pascal.
- Compiler normally generates "C source which is than compiled using c compiler which can lead long compile time.
- All Eiffel object are created on the heap storage
- Pure object oriented
- Not very popular as a language for mainstream application development.

Java

- Designed by SUN (Stanford University Net) Microsystems, released in 1996 and is a pure OO language.
- Syntax is taken from C++ but many differences.
- All object are represented by references and there is are pointer type.
- The compiler generates platform independent byte code which is executed at run time by interpreter.
- A very large library of classes for creating event driven GUI is included with JAVA compiler

C++

- Developed by Bjarne Stroustrup at Bell Lab in New jersey in early 1980 s as extension of C
- Employs the basic syntax of the earlier C language which was developed in Bell Lab by Kernigan & Ritchie.

- One of most popular language used for s/w development.
- By default, C++ creates objects on the systems stack in the same way as the fundamental data type.

Unlike Java, SmallTalk and Eiffel, C++ is not pure O-O language. i.e. it can be written in a conventional 'C'.

Difference between OOP and POP

OOP	POP
OOP takes a bottom-up approach in designing a program.	POP follows a top-down approach.
Program is divided into objects depending on the problem.	Program is divided into small chunks based on the functions.
Each object controls its own data.	Each function contains different data.
Focuses on security of the data irrespective of the algorithm.	Follows a systematic approach to solve the problem.
The main priority is data rather than functions in a program.	Functions are more important than data in a program.
The functions of the objects are linked via message passing.	Different parts of a program are interconnected via parameter passing.
Data hiding is possible in OOP.	No easy way for data hiding.
Inheritance is allowed in OOP.	No such concept of inheritance in POP.
Example: C++ , JAVA	Example: C,FORTRAN

Basic concept / Principles of Object-Oriented Programming

Basically, there are 6 basic principles of OOP. They are as follows.

1. Object and Class
2. Data Abstraction and Encapsulation
3. Inheritance
4. Polymorphism
5. Dynamic Binding

6. Message Passing

Object and Class

Object is an instance of a class i.e. variable of class type. Object are the basic runtime entities in an object-oriented system. Objects contain data and code to manipulate the data. When a program is executed, the objects interact by sending message to one another. Generally, program objects are chosen such that they match closely with real world objects.

A class is a framework that specifies what data and what functions will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. A Class is the collection of objects of similar type.

For example: mango, apple and orange are members of class Fruit. so if we create a class Fruit then mango, apple, orange can be the object of Fruit.

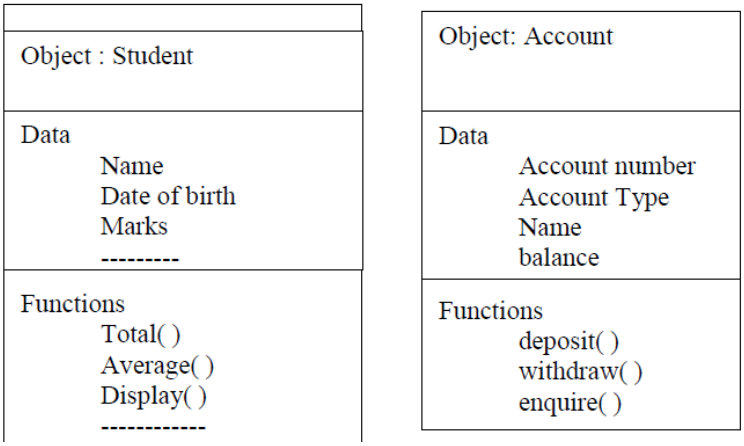


Fig: Representation of Object

Data Abstraction and Encapsulation

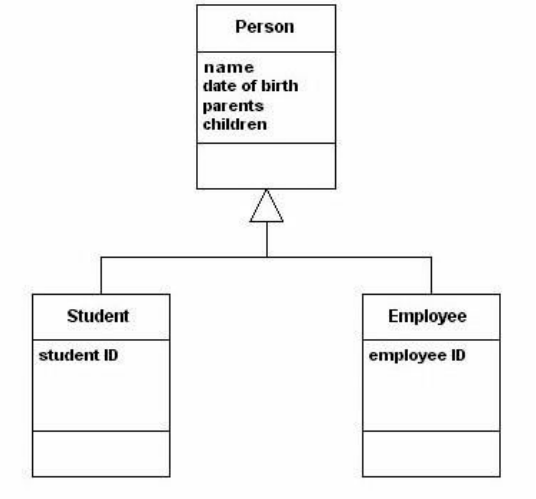
Encapsulation is the process of combining the data (called fields or attributes) and functions (called methods or behaviors) into a single framework called class. Encapsulation helps preventing the modification of data from outside the class by properly assigning the access privilege to the data inside the class. So **the term data hiding** is possible due to the concept of encapsulation, since the data are hidden from the outside world, so that it is safe from accidental alteration.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes uses the concept of abstraction and are defined as a list of abstract attributes and functions. Since classes uses the concept of data abstraction, they are known as abstract data types (ADT).

Inheritance

Inheritance is the process by which objects of one class acquire the characteristics of object of another class. In OOP, the concept of inheritance provides the idea of reusability. We can use additional features to an existing class without modifying it. This is possible by deriving a new class (derived class) from the existing one (base class). This process of deriving a new class from the existing base class is called inheritance.

It supports the concept of hierarchical classification. It allows the extension and reuse of existing code without having to rewrite the code.



In the above figure, Person is base class while Student and Employee are derived form Person so they are referred to as derived class

Polymorphism

Polymorphism means the quality of having more than one form. The representation of different behaviors using the same name is called polymorphism. However, the behavior depends upon the attribute the name holds at particular moment. Example of polymorphism in OOP is operator overloading, function overloading.

Example1:

Operator symbol '+' is used for arithmetic operation between two numbers, however by overloading same operator '+' it can be used for different purpose like concatenation of strings. So the process of making an operator to exhibits different behaviors in different instances is called **operator overloading**.

Example 2:

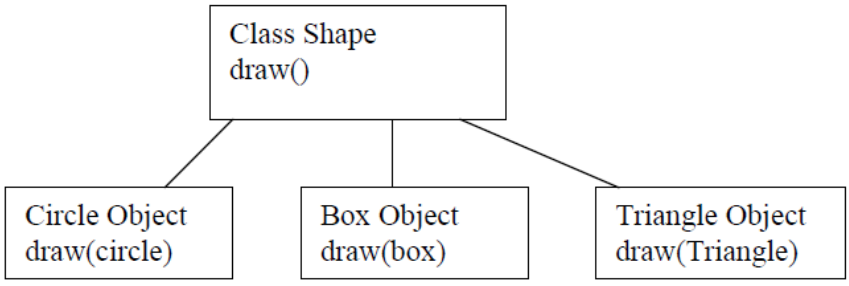


Fig: Polymorphism

So, in the above figure, a single function name is used to handle different number and different type of arguments. So, using a single function name to perform different types of task is known as **function overloading**.

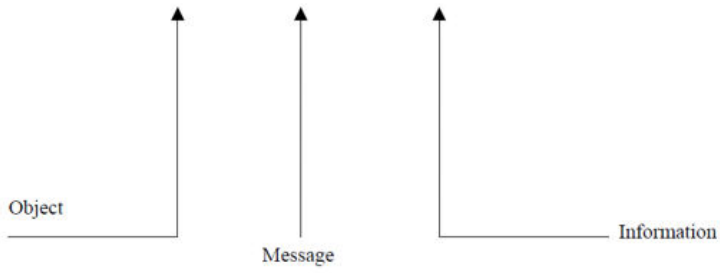
Message Passing

An object-oriented program consists of set of objects that communicate with each other. Object communicates with one another by sending and receiving information. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

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

Example:

Employee. Salary(name);



Dynamic Binding

Binding means link between procedure call and code to be execute. Dynamic binding means link exist between procedure call and code to be execute at run time when that procedure is call. It is also known late binding. It is generally use with polymorphism and inheritance. For example, complier comes to know at runtime that which function of sum will be call either with two arguments or with three arguments.

A New Paradigm

Q. Why new paradigm was needed instead of Procedural Approach?

Due to complex nature of software it was hard to understand the software and maintain the software. So software engineers need a new paradigm to cope this complexity. OOP was solution to this.

Object-oriented paradigm is a new method of implementation in which programs are organized as co-operative, each of which represents a instance of some class and whose classes are all members of a hierarchy of class united through inheritance relationships.

The important parts of OOP are

- Class
- Objects
- Object as an instance of class

Computation as Simulation

Object Orientation is a powerful technique when it comes to simulating real systems in a computer program. When it comes to designing our own programs using objects, as a general guide, it is a good idea to write a *class* and create *objects* for things are objects in real life.

Traditional model

- In traditional view computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various memory, transforming them in some manner, and pushing the results back into other memory.
- Behavior of a computer executing a program is a process-state or pigeon-hole model
- By examining the values in the slots, we can determine the state of the machine or the results produced by a computation.
- This model may be a more or less accurate picture of what takes place inside a computer
- Real world problem solving is difficult in

Object Oriented Model

- Never mentioned memory addresses or variables or assignments or any of the conventional programming terms
- Instead, we spoke of objects, messages, and responsibility for some action
- This model is process of creating a host of helpers that forms a community and assists the programmer in the solution of a problem (Like in Flower example)
- This view of programming as creating a universe is in many ways similar to a style of computer simulation called ***"discrete event-driven simulation"***
- In a discrete event-driven simulation, the user creates computer models of the various elements (entities) of the simulation, describes how they will interact with one another, and sets them moving.
- Object oriented program is also similar to event driven simulation.
- Thinking about what we will be asking the objects to do or to tell you; these things can be implemented as their methods.

Coping with complexity

At early stages of computer programming development all programs are written in assembly language by single individual. As program become more and more complex, programmers have difficulties in remembering all information needed to develop and debug all software. As problem become more and more complex, even the best programmer cannot perform the task by himself. There will be group of programmers working together to solve complex problem.

Interconnections - the Bane of Complexity:

Many software systems are complex not because they are large, but because they have many interconnections. Interconnections make it difficult to understand pieces in isolation, or to carry them from one project to the next. The inability to cleanly separate out components makes it difficult to divide a task between several programmers. Complexity can only be managed by means of abstraction, by eliminating information that a programmer must know.

Nonlinear behavior of complexity

As programming task become larger, an interesting phenomenon was observed. A task that would take one programmer for two months to perform could not be completed by two programmers in one month. i.e. The work will be more complicated. This can be treated as nonlinear behavior, the reason behind this nonlinear behavior is complexity. In particular, the interconnection between software components was complicated. Large quantities of information had to be communicated among the various members of the programming team. This behavior is called nonlinear behavior of complexity.

A Way of Viewing the World

To illustrate the major idea in OOP, let us consider the following real-world situation.

Suppose I wish to send flowers to a friend who lives in a city many miles away. Let me call my friend Sally. Because of the distance, there is no possibility of my picking the flowers and carrying them to her door myself. Nevertheless, sending her the flowers is an easy enough task; I merely go down to my local florist (who happens to be named Flora), tell her the variety and quantity of owners I wish to send and give her Sally's address, and I can be assured the flowers will be delivered expediently and automatically.

.

- **Agents and Communities**

  In above example I solve my problem with help of agent (Object) flora to deliver the flower. There will community of agents to complete a task. In above example Flora will communicate with sally's florist, sally's florist will arrange flower, the arrangement of flower is hidden from me (data hiding/information hiding).

  An object-oriented program is structured as a community of interacting agents, called objects. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community.

- **Messages and Methods**

  I will call flora for delivering flower to my friend sally. After that there will be chain of message passing and actions taken by various agents to deliver the flower as shown in figure above.

  Action is initiated in object-oriented programming by the transmission of a message to an agent (an object) responsible for the action. The message encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request. The receiver is the object to whom the message is sent. If the receiver accepts the message, it accepts

the responsibility to carry out the indicated action. In response to a message, the receiver will perform some method to satisfy the request.

- **Responsibilities**

  My request for action indicates only the desired outcome (flowers for my friend). Flora is free to pursue any technique that achieves the desired objective i.e. to deliver flower to my friend and is not hampered by interference on my part.

  A fundamental concept in object-oriented programming is to describe behavior in terms of responsibilities.

- **Classes and Instances**

  In above Scenario flora is florist we can use florist to represent the category (or class) of all florists. Which means Flora is instance (object) of class florist.

  All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages

- **Class Hierarchies Inheritance**

  Flora not necessarily because she is a florist but because she is a shopkeeper. Florist is a more specialized form of the category Shopkeeper. Flora is a Florist, but Florist is a specialized form of Shopkeeper. Furthermore, a Shopkeeper is also a Human; so I know, for example, that Flora is probably bipedal. A Human is a Mammal and a Mammal is an Animal, and an Animal is a Material Object (therefore it has mass and weight).

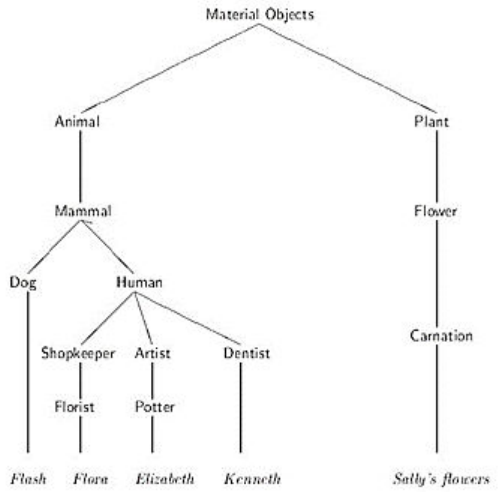


Figure : A class hierarchy for various material objects.

Abstraction mechanism

The abstraction is the process of getting detail information according to the level of deep sight to the problem. If we open an atlas, we will open first see a map of world. This map will show only the most significant features only. It may show few details of mountains, oceans and large features of the earth. If

you see the map of continent we may see more detail of country political boundaries, may be the city, rivers etc. similarly if we see the map of country, it might include main city, rivers, village, town and so on.

In OOP, Classes use theory of abstraction and defined list of abstract properties. Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency.

Abstraction is related to both encapsulation and data hiding. Example of adding complex number, consider 3 object c1, c2, c3. c1 and c2 object is used to call input function, c3 is use for adding data in c1 and c2, and c3 is also used for displaying output.

Varieties of class

- **Data managers classes :** Sometimes called data or state, are classes with the principal responsibility of maintaining data or state information. For example, in an abstraction of a problem, a major task for the class is simply to maintain. The data values that describe state information. Data managers are often recognizable as the nouns in a problem description and are usually the fundamental building blocks of a design.
- **Data sinks or data sources classes :** These are entities that generate data such as a random no. generator. Unlike a data manager, a data sink or data sources does not hold the data for any period, but generates it on demand.
- **View or observer class :** It an essential portion of most applications, display the information on a output devices such as terminal screen.
- **Facilitator or helper classes :** These are entities that maintain little or no state information themselves but assist with complex tasks. For example, in displaying an image we use the services of a facilitator class that handles the drawing of lines and text on the display device. In object-oriented languages like C++, Java etc. class describes a no. of possible run-time objects with same structure which are the instances of the class.

Chapter 2
Basic of C++ Programming

History of C++

C++ is an object-oriented programming language. It was called "C with class". C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early eighties. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both languages and create a more powerful language that could support object-oriented programming features and still remain power of C. The result was C++. The major addition was class to original C language, so it was also called "C with classes". Then in 1993, the name was changed to C++. The increment operator ++ to C suggest an enhancement in C language.

C++ can be regarded as superset of C (or C is subset of C++). Almost all C programs are also true in C++ programs.

C++ program Construction

Before looking at how to write C++ programs consider the following simple example program.

// Reads values for the length and width of a rectangle and returns the perimeter and area of the rectangle.

```
#include <iostream> //for cout and cin
using namespace std; // includes stuff like cout, cin, string, vector, map, etc
int main()
{
  int length, width;
  int perimeter, area;// declarations
  cout <<  "Length = ";// prompt user
  cin >> length;// enter length
  cout << "Width = ";// prompt user
  cin >> width; // input width
  perimeter = 2*(length + width); // compute perimeter
  area = length*width;  // compute area
  cout << endl
       << "Perimeter is " << perimeter;   cout << endl
       << "Area is " << area
       << endl; // output results   getch();

  return 0;
} // end of main program

```

The following points should be noted in the above program:

1. **Single line comment**
   Any text from the symbols // until the end of the line is ignored by the compiler. This facility allows the programmer to insert Comments in the program. Any program that is not very simple should also have further comments indicating the major steps carried out and explaining any

particularly complex piece of programming. This is essential if the program has to be extended or corrected at a later date. This is a kind of documentation.

Also, C comment type /*----------*/ is also a valid comment type in C++.

2. **#include <iostream>**
   The file iostream.h is a system supplied file which has definitions in it which are required if the program is going to use stream input or output.

   All programs will include this file. This statement is a preprocessor directive -- that is it gives information to the compiler but does not cause any executable code to be produced.

3. **Using namespace std**
   The std namespace is special, The built in C++ library routines are kept in the standard namespace. That includes stuff like cout, cin, string, vector, map, etc. Because these tools are used so commonly, it's popular to add "using namespace std" at the top of your source code so that you won't have to type the std:: prefix constantly

4. **int main()**
   All programs must have a function main( ).

   Note that the opening brace ({) marks the beginning of the body of the function, while the closing brace (}) indicates the end of the body of the function.

   The word int indicates that main() does returns a value(Line 18)

5. Sequences of characters enclosed in double quotes are literal strings. Thus instructions such as cout << "Length = " send the quoted characters to the output stream 'cout'.
   The special identifier endl when sent to an output stream will cause a newline to be taken on output.

6. All variables that are used in a program must be declared and given a type. In this case all the variables are of type int, i.e. whole numbers.

   Thus, the statement

   ```
   int length, width;
   ```

   declares to the compiler that integer variables length and width are going to be used by the program. The compiler reserves space in memory for these variables.

Input and Output

C++ Supports rich set of functions for performing input and output operations. The syntax using these I/O functions is totally consistent of the device with I/O operations are performed. C++'s new features for

handling I/O operations are called streams. Streams are abstractions that refer to data flow. Streams in C++ are:

- Output Stream
- Input Stream.

Output Stream:

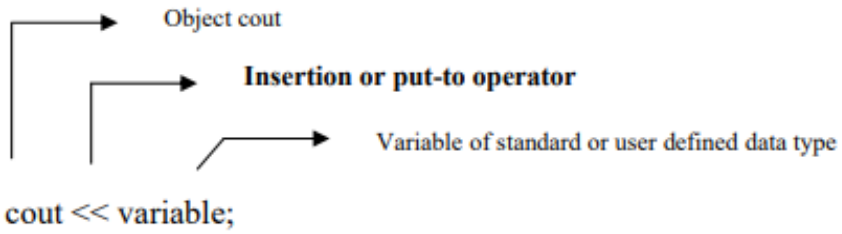
The output stream allows us to write operations on output devices such as screen, disk etc.

Output on the standard stream is performed using the 'cout' object. C++ uses the bitwise-left-shift operator for performing console output operation. The syntax for the standard output stream operation is as follows:

cout<<variable;

The word cout is followed by the symbol << , called the insertion or put to operator , and then with the items (variables, constants, expressions) that are to be output.

Variables can be of any basic data types. The use of 'cout' to perform an output operation is as shown :

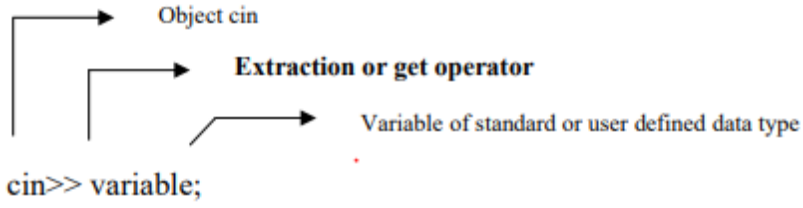


Input Streams:

The input stream allows us to perform read operations with input devices such as keyboard, disk etc. Input from the standard stream is performed using the cin object. C++ uses the bit-wise right-shift operator for performing console input operation. The syntax for the standard output stream operation is as follows:

Cin>>variable;

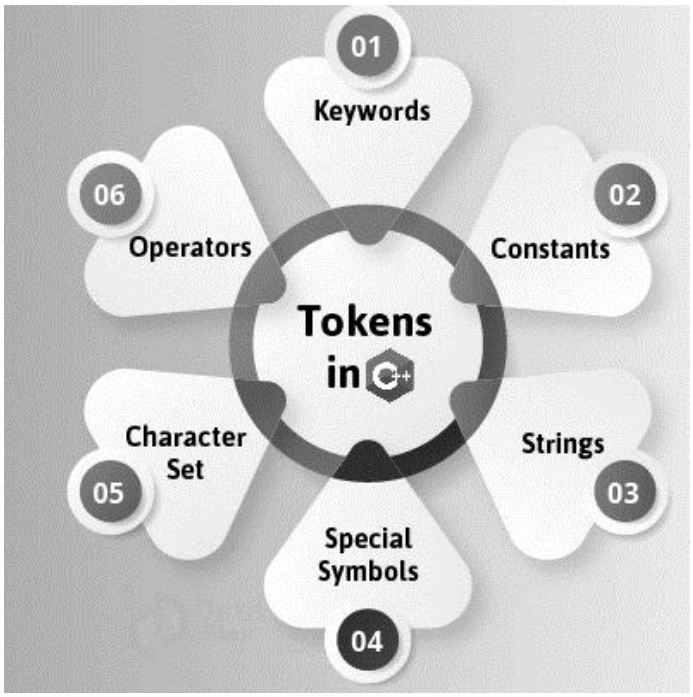
The word 'cin' is followed by the symbol >> and then with variable, into which input data is to be stored. The use of cin to perform an input operation is as shown:



Tokens in C++

Tokens act as building blocks of a program. Just like a living cell is the smallest possible unit of life, *tokens in C++ are referred to as the smallest individual units in a program*. *Keywords in C++ help the user in framing statements and commands in a language*. Each keyword conveys a unique connotation to the compiler to perform a specific task. Just like the combination of words helps us in framing sentences, the combination of keywords helps us in framing statements to perform logical operations in a programming language. Simply combining keywords wouldn't help to serve the purpose.

As we need to use proper grammar to form a meaningful sentence, we need to be well-acquainted with the **syntax of C++** to instruct the compiler what to do. If these statements are not formed in a logical manner, they would sound gibberish and you would get a compilation error.



1. C++ Keywords

Keywords in C++ refer to the pre-existing, reserved words, each holding its own position and power and has a specific function associated with it.

It is important to note that we cannot use C++ keywords for assigning variable names as it would suggest a totally different meaning entirely and would be incorrect.

alignas	alignof	asm	auto	bool	break
case	catch	char	char16_t	char32_t	class
const	constexpr	const_cast	continue	decltype	default
delete	double	do	dynamic_cast	else	enum
explicit	export	extern	FALSE	float	for
friend	goto	if	inline	int	long
mutable	namespace	new	noexcept	nullptr	operator
private	protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_assert	static_cast
struct	switch	template	this	thread_local	throw
TRUE	try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile	wchar_t
while					

2. C++ Constants

Before we begin our discussion on constants in C++, it is important to note that we can use the terms "constants" and "literals" interchangeably.

As the name itself suggests, constants are referred to as fixed values that cannot change their value during the entire program run as soon as we define them.

Syntax:

const data_type variable_name = value;

Types of Constants in C++

The different types of constants are:

- **Integer constants –** These constants store values of the int data type.

  For instance:

  const int data = 5;
- **Floating constants –** These constants store values of the float data type.

  For instance:

  const float e = 2.71;
- **Character constants –** These constants store values of the character data type.

  For instance:

  const char answer = 'y';
- **String constants –** These constants are also of the character data type but differ in the declaration part.

  For instance:

const char title[] = "DataFlair";

- **Octal constants –** The number system which consists of only 8 digits, from 0 to 7 is called the octal number system. The constant octal values can be declared as:

  const int oct = 034;

  It is the octal equivalent of the digit 28 in the decimal number system.

- **Hexadecimal constants –**

  The number system which consists of 16 digits, from 0 to 9 and alphabets 'a' to 'f' is called hexadecimal number system. The constant hexadecimal values can be declared as:

  const int hex = 0x40;

  It is the hexadecimal equivalent of the digit 64 in the decimal number system

3. C++ Strings

Just like characters, **strings in C++** are used to store letters and digits. Strings can be referred to as an array of characters as well as an individual data type.

It is enclosed within double quotes, unlike characters which are stored within single quotes. The termination of a string in C++ is represented by the null character, that is, **'\0'.** The size of a string is the number of individual characters it has.

In C++, a string can be declared in the following ways:

- *char name[30] = ''Hello!"; // The compiler reserves 30 bytes of memory for the string.*
- *char name[] = "Hello!"; // The compiler reserves the required amount of memory for the string.*
- *char name[30] = { 'H', 'e', 'l', 'l', 'o','\0'};; // This is how a string is represented as a set of characters.*
- *string name = "Hello" // The compiler reserves 32 bytes of memory.*

4. Special Symbols

Apart from letters and digits, there are some special characters in C++ which help you manipulate or perform data operations. Each special symbol has a specific meaning to the C++ compiler.

Special Character	Trivial Name	Function
[ ]	Square brackets	The opening and closing brackets of an array symbolize single and multidimensional subscripts.
( )	Simple brackets	The opening and closing brackets represent function declaration and calls, used in print statements.
{ }	Curly braces	The opening and closing curly brackets to denote the start

		and end of a particular fragment of code which may be functions, loops or conditional statements
,	Comma	We use commas to separate more than one statements, like in the declaration of different variable names
#	Hash / Pound / Preprocessor	The hash symbol represents a preprocessor directive used for denoting the use of a header file
*	Asterisk	We use the asterisk symbol in various respects such as to declare pointers, used as an operand for multiplication
~	Tilde	We use the tilde symbol as a destructor to free memory
.	Period / dot	The use the dot operator to access a member of a structure

5. Character Set

Character set is the combination of English language (Alphabets and White spaces) and math's symbols (Digits and Special symbols). Character Set means that the characters and symbols that a C++ Program can understand and accept. These are grouped to form the commands, expressions, words, c-statements and other tokens for C++ Language.

Character Set is the combination of alphabets or characters, digits, special symbols and white spaces same as learning English is to first learns the alphabets, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs.

More about a C++ program we can say that it is a sequence of characters. These character from the character set plays the different role in different way in the C++ compiler.

In addition to these characters, C++ also uses a combination of characters to represent special conditions. For example. character combinations such as '\n, '\b' and '\t' are used to represent newline, backspace and horizontal tab respectively.

There are mainly four categories of the character set as shown in the Figure.

1. Alphabets

Alphabets are represented by A-Z or a-z.

C++ Language is case sensitive so it takes different meaning for small and upper case letters. By using this character set C++ statements and character constants can be written very easily.

2. Digits

Digits are represented by 0-9 or by combination of these digits. By using the digits numeric constant can be written easily. Also, numeric data can be assigned to the C++-tokens.

3. Special Symbols

All the keyboard keys except alphabet, digits and white spaces are the special symbols. These are some punctuation marks and some special symbols used for special purpose.

4. White Spaces

White spaces have blank space, new line return, Horizontal tab space, carriage ctrl, Form feed etc. are all used for special purpose.

6. C++ Operators

Operators are tools or symbols which are used to perform a specific operation on data. Operations are performed on operands. Operators can be classified into three broad categories according to the number of operands used.

Unary: It involves the use of one a single operand. For instance, '!' is a unary operator which operates on a single variable, say 'c' as !c which denotes its negation or complement.

Binary: It involves the use of 2 operands. They are further classified as:

- Arithmetic
- Relational
- Logical
- Assignment
- Bitwise
- Conditional

Control Structures

Like C, C++ also supports the following three control structures:

- Sequence structure (straight line)
- Selection structure (branching or decision)
- Loop structure (iteration or repetition)

Sequence structure:

In this structure, the sequence statements are executed one after another from top to bottom.

```
statement 1;
statement 2;
statement 3;
……..
statement n;
```

Selection structure:

This structure makes one-time decision, causing a one-time jump to a different part of the program, depending on the value of an expression.

The two structures of this type are:

- If
- switch.

The if statement:

There are three forms of if statement.

- **Simple if**:

  The syntax is:

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

  This statement lets your program to execute statement(s) if the value of the expression is true (non-zero). If there is only one statement, it is not necessary to use braces.

- **The if-else statement:**

  The syntax is:

```
if (expression)
{
    statement(s);
}
else
{
    statement(s);
}
```

- **The if-else-if ladder**:

  The syntax is:

```
if (expression) {
```

```
                Statement(s);
        } else if (expression) {
                Statement(s);
        } else if (expression) {
        statement(s);
        }
        …
        }
        else {
                Statement(s);
        }
```

The switch statement:

The syntax form is:

```
switch (expression)
{
case value 1:
        statement(s);
        break;
case value 2:
        statement(s)
        break;
……
case value n:
        statement(s);
        break;
default:
        statement(s);
}
```

Loop Structure:

These structures repeatedly execute a section of your program a certain number of times. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to

the statement following the loop. C++ provides three kinds of loops: the for loop, the while loop, and the do loop.

- **The for loop**:
  The general form is:

```
for(initialization; test expression; iteration)
{
        Statement(s);
}
```

- **The while loop:**
  The general form is:

```
while(test expression) {
        statement(s);
}
```

- The do-while loop:
  The general form is:

```
do{
        statement(s);
} while(test expression);
```

Jump Statements

For the transfer of control from one part of the program to another, C++ supports three jump statements: break, continue, and return.

- **The break Statement**:
  The use of break statement causes the immediate termination of the switch statement and the loop (all type of loops) from the point of break statement. The control then passes to the statements following the switch statement and the loop.

  Example piece of code:

```
for( int i = 1;  i <= 10;  i++ )
{
        if( i == 5 )
```

```
                    break;
                cout<< " i = "<< i;


        }
```

- **The *continue* Statement:**

  Continue statement causes the execution of the current iteration of the loop to stop, and then continue at the next iteration of the loop.

  Example:

```
for( int i=1; i<= 10; i++ )
{
    if( i == 5 || i == 7 )
     {
            continue;
    }
    cout<<i<<“ ”;
}
```

- **The *return* Statement:**

  It is used to transfer the program control back to the caller of the method. There are two forms of return statement. First with general form ***return expression;*** returns the value whereas the second with the form ***return;*** returns no value but only the control to the caller.

Data Types:

- **Fundamental Data Types**

  Data type defines the amount of memory that will be used by a variable and the valid range of values that a variable can represent.

  Both C and C++ compilers support all the fundamental (also known as basic or built-in) data types.

  C++ defines five fundamental data types: int, char, float, double, and void.

  The modifiers signed, unsigned, long, and short may be applied to character and integer basic types except void. However, the modifier long may also be applied to double.

  The following table shows the memory required for all combinations of the basic data types and modifiers and the valid range of values it can represent.

Data Type	Memory Required	Range of Values

char	1 byte (8 bits)	- 128 to 127 (- 2^7 to 2^7 – 1)
unsigned char	1 byte (8 bits)	0 to 255
signed char	1 byte (8 bits)	- 128 to 127 (- 2^7 to 2^7 – 1)
int	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
unsigned int (unsigned)	2 bytes (16 bits)	0 to 65535
signed int (signed)	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
short int (short)	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
long int (long)	4 bytes (32 bits)	–2,147,483,648 to 2,147,483, 647 (–2^{31} to 2^{31}–1)
unsigned short int (unsigned short)	2 bytes (16 bits)	0 to 65535
signed short int (signed short)	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
unsigned long int (unsigned long)	4 bytes (32 bits)	0 to 4,294,967,295
signed long int (signed long)	4 bytes (32 bits)	–2,147,483,648 to 2,147,483, 647 (–2^{31} to 2^{31}–1)
float	4 bytes (32 bits)	3.4×10^{-38} to 3.4×10^{38}
double	8 bytes (64 bits)	1.7×10^{-308} to 1.7×10^{308}
long double	10 bytes (80 bits)	3.4×10^{-4932} to 1.1×10^{4932}

- **User Defined Data Types**

  In C++, we can use the concepts of struct, union and class (discussed later) to define user-defined data types.

  An **enumeration** is the distinct type whose value is restricted to the range of values (see below for details), which may include a several explicitly named constants ("**enumerators** "). The values of the constants are the values of an integral type known as an underlying type of the enumeration

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

using namespace std;
enum months
{
    jan,feb,mar,apr,may,june,july,aug,sep,oct,nov,dec
};
int main()
{
    //here we are checking
    //which month april is
    months mn;
    mn=apr;
    cout<<"The month number April is: "<<mn;
    return 0;
}
```

```
The month number April is: 3
```

Variables

A variable is the name used for the quantities which are manipulated by a computer program. i.e. it is a named storage location in memory.

For example

- a program that reads a series of numbers and sums them will have to have a variable to represent each number as it is entered and a variable to represent the sum of the numbers.

In order to distinguish between different variables, they must be given identifiers, names which distinguish them from all other variables. The rules of C++ for valid identifiers state that:

An identifier must:

- start with a letter
- consist only of letters, the digits 0-9, or the underscore symbol _
- not be a reserved word

C++ is case-sensitive. That is lower-case letters are treated as distinct from uppercase letters. Thus the word main in a program is quite different from the word Main or the word MAIN.

Reserved words

The syntax rules (or grammar) of C++ define certain symbols to have a unique meaning within a C++ program. These symbols, the reserved words, must not be used for any other purposes. All reserved words are in lower-case letters.

The table below lists the reserved words of C++.

C++ Reserved Words

and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	const	const_cast	continue
default	delete	do	double	dynamic_cast
else	enum	explicit	export	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	not	not_eq	operator
or	or_eq	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned

using	virtual	void	volatile	wchar_t
while	xor	xor_eq		

Expressions and Statements

An expression is any arrangement of operands and operators that specifies a computation.

For example, a + 13, a++, and (a – 5) * b / 2 are expressions.

Expressions and statements are not same.

Statements tell the compiler to do something and they must terminate with a semicolon (also known as the statement terminator), while expressions specify a computation.

There can be several expressions in a statement

C++ Preprocessor

The preprocessors are the directives, which give instructions to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;).

You already have seen a #include directive in all the examples. This macro is used to include a header file into the source file.

There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc.

- **The #define Preprocessor**

  The #define preprocessor directive creates symbolic constants. The symbolic constant is called a **macro** and the general form of the directive is −

  #define macro-name replacement-text

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

#define PI 3.14159

int main () {
   cout << "Value of PI :" << PI << endl;

   return 0;
}
```

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
   int i, j;

   i = 100;
   j = 30;

   cout <<"The minimum is " << MIN(i, j) << endl;

   return 0;
}
```

- **Predefined C++ Macros**

Sr.No	Macro & Description
1	**__LINE__** This contains the current line number of the program when it is being compiled.
2	**__FILE__** This contains the current file name of the program when it is being compiled.
3	**__DATE__**

	This contains a string of the form month/day/year that is the date of the translation of the source file into object code.
4	**__TIME__** This contains a string of the form hour:minute:second that is the time at which the program was compiled.

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

int main () {
   cout << "Value of __LINE__ : " << __LINE__ << endl;
   cout << "Value of __FILE__ : " << __FILE__ << endl;
   cout << "Value of __DATE__ : " << __DATE__ << endl;
   cout << "Value of __TIME__ : " << __TIME__ << endl;

   return 0;
}
```

Type Conversion

There are two types of type conversion:

- automatic conversion
- type casting.

- Automatic Conversion:

  When two operands of different types are encountered in the same expression, the lower type variable is converted to the type of the higher type variable by the compiler automatically. This is also called type promotion.

  The order of types is given below:

  Below is the order of the automatic type conversion. You can also say, smallest to largest data type for type conversion.

  bool → char → short int→ int → unsigned int → long → unsigned → long → float → double → long double

- Type Casting:

Sometimes, a programmer needs to convert a value from one type to another in a situation where the compiler will not do it automatically. For this C++ permits explicit type conversion of variables or expressions as follows:

For example,

```
int a = 10000;
int b = long(a) * 5 / 2;
```

Namespaces in C++

Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.

Rules to create Namespaces

- The namespace definition must be done at global scope, or nested inside another namespace.
- Namespace definition doesn't terminate with a semicolon like in class definition.
- You can use an alias name for your namespace name, for ease of use.

Q. Write the different ways of accessing the namespace?

Manipulators

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.

- endl Manipulator

  endl is the line feed operator in C++. It acts as a stream manipulator whose purpose is to feed the whole line and then point the cursor to the beginning of the next line. We can use \n (\n is an escape sequence) instead of **endl** for the same purpose.
- setw Manipulator

  This manipulator sets the minimum field width on output.

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

int main()
{

    float basic, ta,da,gs;
    basic=10000; ta=800; da=5000;
    gs=basic+ta+da;
    cout<<setw(10)<<"Basic"<<setw(10)<<basic<<endl
        <<setw(10)<<"TA"<<setw(10)<<ta<<endl
        <<setw(10)<<"DA"<<setw(10)<<da<<endl
        <<setw(10)<<"GS"<<setw(10)<<gs<<endl;
    return 0;
}

```

```
     Basic      10000
        TA        800
        DA       5000
        GS      15800
```

- setfill Manipulator

  This is used by the setw manipulator. If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

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

int main()
{

    cout << setw(20) << setfill('*') << "setfill" << setw(20) << setfill('*')<<"Test"<< endl;
    return 0;
}

```

```
**********setfill**********Test
```

Dynamic Memory in C++

In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input.

On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators *new and delete.*

- **new**

  The **new** operator is used to **allocate memory at runtime**. The memory is allocated in bytes.

  Let's first see how to allocate a variable dynamically.

  *int *ptr = new int;*

```cpp
#include <iostream>
using namespace std;
int main()
{
    int *ptr = new int;
    *ptr = 4;
    cout << *ptr << endl;
    return 0;
}
```

- **delete**

Suppose we allocated some memory to a variable dynamically and then we realize that the variable is not needed anymore in the program. In that case, we need to free the memory which we had assigned to that variable. For that, we use the **delete** operator.

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

    int *ptr = new int;
    *ptr = 4;
    cout << *ptr << endl;
    delete ptr;
    return 0;
}

```

Example:

Deleting Array using DMA

```cpp
#include <iostream>
using namespace std;
int main()
{
    int length, sum = 0;
    cout << "Enter the number of students in the group" << endl;
    cin >> length;
    int *marks = new int[length];
    cout << "Enter the marks of the students" << endl;
    for( int i = 0; i < length; i++ ) // entering marks of students
    {
        cin >> *(marks+i);
    }
    for( int i = 0; i < length; i++ )// calculating sum
    {
        sum += *(marks+i);
    }
    cout << "sum is " << sum << endl;
    delete[] marks;
    return 0;
}
```

Scope Resolution Operator (::)

C++ supports a mechanism to access a global variable from a function in which a local variable is defined with the same name as a global variable. It is achieved using the scope resolution operator.

:: GloabalVariableName

The global variable to be accessed must be preceded by the scope resolution operator. It directs the compiler to access a global variable, instead of one defined as a local variable. The scope resolution operator permits a program to reference an identifier in the global scope that has been hidden by another identifier with the same name in the local scope.

```cpp
#include<iostream>
using namespace std;
int x=5;
int main()
{
    int x=15;
    cout<<"Local data x="<<x<<endl;
    cout<<"Global data x="<<::x<<endl;
    return 0;
}
```

```
/tmp/U6cZ2QIwE9.o
Local data x=15
Global data x=5
```

Reference Variables:

C++ introduces a new kind of variable known as reference variable. A reference variable provides an alias (Alternative name) of the variable that is previously defined. For example, if we make the variable sum a reference to the variable total, the sum and total can be used interchangeably to represent that variable.

Syntax for defining reference variable

data_type &reference_name = variable_nane

Example:

int total=100 ;

int &sum=total;

Here total is int variable already declared. Sum is the alias for variable total. Both the variable refer to the same data 100 in the memory.

cout<<total; and cout<<sum; gives the same output 100.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it means. The initialization of reference variable is completely different from assignment to it.

```
1  //An example of reference
2  #include<iostream>
3  using namespace std;
4
5  int main()
6  {
7      int x=5;
8      int &y=x; //y is alias of x
9      cout<<"x="<<x<<"and y="<<y<<endl;
10     y++; //y is reference of x;
11     cout<<"x="<<x<<"and y="<<y<<endl;
12
13     return 0;
14 }
```

```
/tmp/U6cZ2QIwE9.o
x=5and y=5
x=6and y=6
```

Function

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

We can divide up our code into separate functions to perform specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call.

For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

A C++ function definition 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.

```cpp
// function returning the max between two numbers

int max(int num1, int num2)
{
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Q. Differentiate function declaration, function definition and function call.

Q. Write a program in C++ to find the greatest among two numbers using function.

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

// function declaration
int max(int num1, int num2);

int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // calling a function to get max value.
   ret = max(a, b);
   cout << "Max value is : " << ret << endl;

   return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

```
/tmp/U6cZ2QIwE9.o
Max value is : 200
```

Function Overloading in C++

Function Overloading in C++ is the mechanism by which a programmer can specify multiple definitions of the same function(same name) by changing:

- Number of arguments passed to the function
- Data type of arguments passed to the function

Essentially Function overloading means same function name but different number or type of arguments with different functionality. Function overloading is also a type of Static or Compile time Polymorphism.

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

void display(int);
void display(float);
void display(int, float);

int main() {

    int a = 5;
    float b = 5.5;

    display(a);
    display(b);
    display(a, b);

    return 0;
}
void display(int var) {
    cout << "Integer number: " << var << endl;
}
void display(float var) {
    cout << "Float number: " << var << endl;
}
void display(int var1, float var2) {
    cout << "Integer number: " << var1;
    cout << " and float number:" << var2;
}
```

```
/tmp/V9lrUmtodw.o
Integer number: 5
Float number: 5.5
Integer number: 5 and float number:5.5
```

Inline Function

An inline function is a short-code function written and placed before main function and compiled as inline code. The prototyping is not required for inline function. It starts with keyword inline. In ordinary functions, when function is invoked the control is passed to the calling function and after executing the function the control is passed back to the calling program.

But, when inline function is called, the inline code of the function is inserted at the place of call and compiled with the other source code together. That is the main feature of inline function and different from the ordinary function. So, using inline function executing time is reduced because there is no transfer and return back to control. But if function has long code inline function is not suitable because it increases the length of source code due to inline compilation.

//saves memory, the call to function cause the same code to be executed; the function need not be duplicated in memory

```
1  //An example of reference
2  #include<iostream>
3  using namespace std;
4  inline float lbstokg(float pound)
5  {
6      return (0.453592*pound);
7  }
8  int main()
9  {
10     float lbs1=50,lbs2=100;
11     cout<<"Weinght in Kg:"<<lbstokg(lbs1)<<endl;
12     cout<<"Weinght in Kg:"<<lbstokg(lbs2)<<endl;
13     return 0;
14 }
15
```

```
/tmp/U6cZ2QIwE9.o
Weinght in Kg:22.6796
Weinght in Kg:45.3592
```

Example1:

```
#include<conio.h>
#include<iostream>
using namespace std;
//While declartion no need for keyword inline
int Findcube(int x);
int main()
{
        cout<<Findcube(3);
        getch();
        return 0;
}
inline int Findcube(int x)
{
        return x*x*x;
}
```

Example 2:

WAP to display hello world by using class and inline member function

```
#include<iostream>
using namespace std;
class Demo
{
        public:
        inline void display();
};
int main()
{
        Demo d;
         d.display();
```

```
}
inline void Demo:: display()
{
        cout<<"hello world";
}
```

C++ function call by value

The call by value method of passing arguments to a function 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.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

```cpp
#include <iostream>
using namespace std;
// function declaration
void swap(int x, int y);
int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   // calling a function to swap the values.
   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
// function definition to swap the values.
void swap(int x, int y) {
   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

```
/tmp/V9lrUmtodw.o
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

C++ function call by reference

The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```cpp
#include <iostream>
using namespace std;
// function declaration
void swap(int &x, int &y);
int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   // calling a function to swap the values.
   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
// function definition to swap the values.
void swap(int &x, int &y) {
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

```
/tmp/V9lrUmtodw.o
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Default Values for Parameters

In C++, a function can, be called without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. The default value are specified when function is declared.

The default value is specified similar to the variable initialization. The prototype for the declaration of default value of an argument looks like

float amount(float p, int time, float rate=0.10);

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

int sum(int a, int b = 20) {
   int result;
   result = a + b;

   return (result);
}
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;

   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // calling a function again as follows.
   result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

```
/tmp/U6cZ2QIwE9.o
Total value is :300
Total value is :120
```

Const Arguments

When arguments are passed by reference to the function, the function can modify the variables in the calling program. Using the constant arguments in the function, the variables in the calling program cannot be modified. const qualifier is used for it. e.g.

```cpp
void func(int&, const int&);
void main ()
{
   int a=10, b=20;
     func(a,b);
}
void func(int& x, int &y)
{
     x=100;
     y=200; // error since y is constant argument
}
```

Storage Classes in C++

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program. To specify the storage class for a variable, the following syntax is to be followed:

Syntax:

storage_class var_data_type var_name;

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extem	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage

1. Auto

```
#include <iostream>
using namespace std;
void autoStorageClass()
{
    cout << "Demonstrating auto class\n";
    // Declaring an auto variable
    // No data-type declaration needed
    auto a = 32;
    auto b = 3.2;
    auto c = "DestinyWaits";
    auto d = 'J';
    // printing the auto variables
    cout << a << " \n";
    cout << b << " \n";
    cout << c << " \n";
    cout << d << " \n";
}
int main()
{
    // To demonstrate auto Storage Class
    autoStorageClass();
    return 0;
}
```

```
/tmp/odHBtd9YOR.o
Demonstrating auto class
32
3.2
DestinyWaits
J
```

2. External

```cpp
#include <iostream>
using namespace std;
// declaring the variable which is to be made extern an intial value can also be initialized to x
int x;
void externStorageClass()
{
    cout << "Demonstrating extern class\n";
    // telling the compiler that the variable
    // x is an extern variable and has been
    // defined elsewhere (above the main
    // function)
    extern int x;
    // printing the extern variables 'x'
    cout << "Value of the variable 'x'"<< "declared, as extern: " << x << "\n";
    // value of extern variable x modified
    x = 2;
    // printing the modified values of
    // extern variables 'x'
    cout
        << "Modified value of the variable 'x'"<< " declared as extern: \n"
        << x;
}
int main()
{
    // To demonstrate extern Storage Class
    externStorageClass();
    return 0;
}

```

```
/tmp/odHBtd9YOR.o
Demonstrating extern class
Value of the variable 'x'declared, as extern: 0
Modified value of the variable 'x' declared as extern:
2
```

3. Static

```cpp
#include <iostream>
using namespace std;
// Function containing static variables memory is retained during execution
int staticFun()
{
    cout << "For static variables: ";
    static int count = 0;
    count++;
    return count;
}
// Function containing non-static variables memory is destroyed
int nonStaticFun()
{
    cout << "For Non-Static variables: ";
    int count = 0;
    count++;
    return count;
}
int main()
{
    // Calling the static parts
    cout << staticFun() << "\n";
    cout << staticFun() << "\n";
    // Calling the non-static parts
    cout << nonStaticFun() << "\n";
    cout << nonStaticFun() << "\n";
    return 0;
}

```

```
/tmp/odHBtd9YOR.o
For static variables: 1
For static variables: 2
For Non-Static variables: 1
For Non-Static variables: 1
```

4. Register

```
#include <iostream>
using namespace std;
void registerStorageClass()
{
    cout << "Demonstrating register class\n";
    // declaring a register variable
    register char b = 'J';
    // printing the register variable 'b'
    cout << "Value of the variable 'b'" << " declared as register: " << b;
}
int main()
{
    registerStorageClass();
    return 0;
}
```

```
/tmp/odHBtd9YOR.o
Demonstrating register class
Value of the variable 'b' declared as register: J
```

5. Mutable

```
#include <iostream>
using std::cout;
class Test {
public:
    int x;
    // defining mutable variable y
    // now this can be modified
    mutable int y;
    Test()
    {
        x = 4;
        y = 10;
    }
};

int main()
{
    // t1 is set to constant
    const Test t1;
    // trying to change the value
    t1.y = 20;
    cout << t1.y;
    return 0;
}
```

```
/tmp/odHBtd9YOR.o
20
```

Pointers

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined

```
#include <iostream>

using namespace std;
int main ()
{
   int  var1;
   char var2[10];

   cout << "Address of var1 variable: ";
   cout << &var1 << endl;

   cout << "Address of var2 variable: ";
   cout << &var2 << endl;

   return 0;
}
```

```
/tmp/xiFXfUd4bf.o
Address of var1 variable: 0x7ffc4e670df8
Address of var2 variable: 0x7ffc4e670dfe
```

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it.

There are few important operations, which we will do with the pointers very frequently.

a. We define a pointer variable.
b. Assign the address of a variable to a pointer.
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

Pointer Declaration and Initialization

A pointer variable is declared with an asterisk (*) operator before the variable name. This operator is called pointer or indirection or dereference operator. The type-specifier determines that what kind of variable the pointer variable points to.

Syntax

*data-type *pointer_variable_name;*

Following are the valid pointer declaration −

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

*char *ch // pointer to character*

```cpp
#include <iostream>
using namespace std;
int main ()
{
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable
   ip = &var;       // store address of var in pointer variable
   cout << "Value of var variable: ";
   cout << var << endl;
   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;
   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;
   return 0;
}
```

```
/tmp/xiFXfUd4bf.o
Value of var variable: 20
Address stored in ip variable: 0x7ffc1fce8d0c
Value of *ip variable: 20
```

Every pointer contains garbage values before assignment of some valid address. The uninitialized pointer is called the bad pointer. Pointer variables should be initialized to 0, null or an address. No other constant can be initialized to a pointer variable. Pointer variable of a particular data type can hold only the address of the variable of same data type.

A pointer variable must not remain uninitialized since uninitialized pointers cause the system crash. Even if you do not have any legal pointer value to initialize a pointer, you can initialize it with NULL pointer value

*int *iptr = NULL ; //iptr gets initialized but does not to a legal address*

nullptr in C++

In C++, a new keyword namely nullptr has been introduced, which is a distinguished null-point constant. So, now on (provided compiler implements C++ standard), you can initialize your pointers with nullptr, e.g.,

*char *cptr = nullptr ;*

*int *iptr = nullptr ;*

Difference between NULL and nullptr

Difference between NULL and nullptr are:

- NULL is not defined by C++, it is programmer or compiler-defined whereas nullptr has been defined by C++ as a legal null pointer
- NULL is internally mapped to int 0(zero) whereas nullptr is a legal empty/null pointer, which is implicitly convertible and comparable to any pointer-type. This feature has another benefit too. Until now if you had two overloaded functions

Pointer Arithmetic

As a pointer holds the memory address of variable, some arithmetic operations can be performed with pointers. C++ support four arithmetic operators that can be used with pointers, such as:

Pointer arithmetic	Symbol
Addition	+
Subtraction	-
Increment	++
Decrement	--

Pointer is variables that hold that memory address of another variable. They are not integers, but they can be displayed as unsigned integers. According to data type declared to the pointer variable, if arithmetic operation is done then values (contents) will be incremented or decremented as per the data type chosen.

Let p = &a[0] = 1001 i.e. Base address of an array

int type pointer

(2-byte space)

p++ = p+1 = 1001 + 2 = 1003 i.e. Address of next element

p = p+5 = 1001 + 10 = 1011 i.e. Address of 5th integer type element.

float type pointer

(4-byte space)

p++ = 1001 + 4 = 1005 i.e. Address of next element

p = p+5 = 1001 + 20 = 1021 i.e. Address of 5th float type element.

char type pointer

(1-byte space)

p++ = 1001 + 1 = 1002 i.e. Address of next element

p = p+5 = 1001 + 5 = 1006 i.e. Address of 5th char type element.

Note: Same operation is done for decrement.

Invalid pointer operation

- Addition of two pointer
- Addition or subtraction of float or double data type to or from a pointer.
- Multiplication of pointer with a constant.
- Division of two pointer with a constant

Illegal operations: p1/p2 p1*p2 p1+p2 p1/5

Pointer and Array in C++

In C++, Pointers are variables that hold addresses of other variables. Not only can a pointer store the address of a single variable, it can also store the address of cells of an array.

Consider this example:

```
int *ptr;
int arr[5];

// store the address of the first element of arr in ptr
ptr = arr;
```

Here, ptr is a pointer variable while arr is an int array. The code ptr = arr; stores the address of the first element of the array in variable ptr.

We have used arr instead of &arr[0]. This is because both are the same. So, the code below is the same as the code above.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

The addresses for the rest of the array elements are given by &arr[1], &arr[2], &arr[3], and &arr[4].

Point to Every Array Elements

Suppose we need to point to the fourth element of the array using the same pointer ptr.

Here, if ptr points to the first element in the above example then ptr + 3 will point to the fourth element. For example,

```
int *ptr;
int arr[5];
```

ptr = arr;

ptr + 1 is equivalent to &arr[1];

ptr + 2 is equivalent to &arr[2];

ptr + 3 is equivalent to &arr[3];

ptr + 4 is equivalent to &arr[4];

Similarly, we can access the elements using the single pointer. For example,

// use dereference operator

**ptr == arr[0];*

**(ptr + 1) is equivalent to arr[1];*

**(ptr + 2) is equivalent to arr[2];*

**(ptr + 3) is equivalent to arr[3];*

**(ptr + 4) is equivalent to arr[4];*

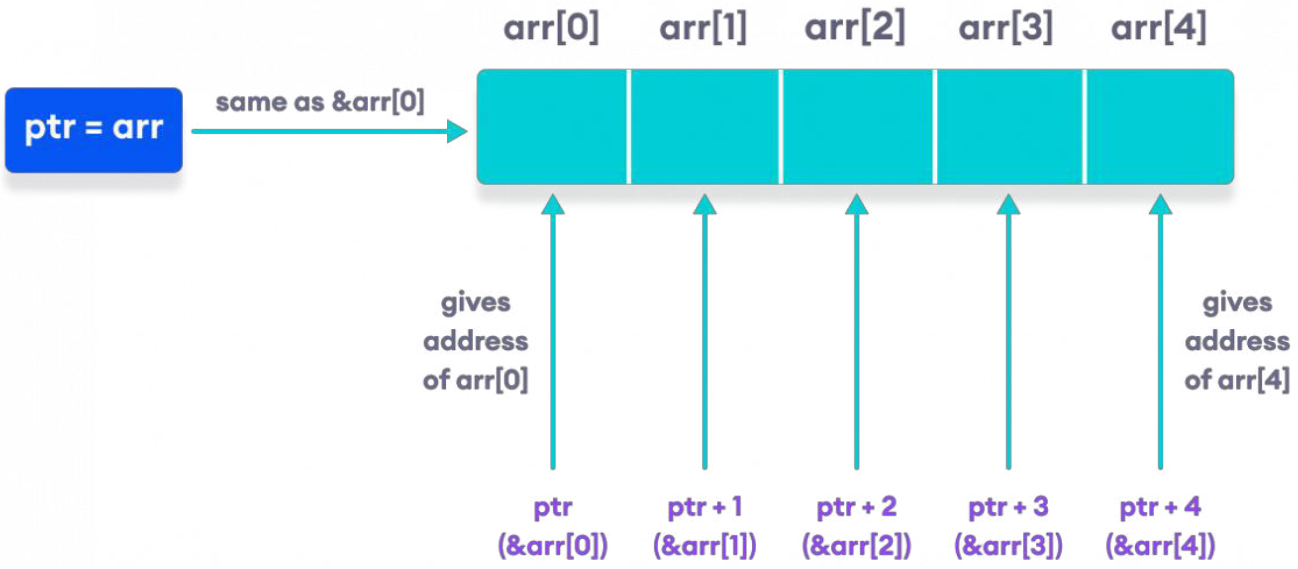
Suppose if we have initialized ptr = &arr[2]; then

ptr - 2 is equivalent to &arr[0];

ptr - 1 is equivalent to &arr[1];

ptr + 1 is equivalent to &arr[3];

ptr + 2 is equivalent to &arr[4];



a. C++ Program to display address of each element of an array

```cpp
#include <iostream>
using namespace std;
int main()
{
    float arr[3];
    // declare pointer variable
    float *ptr;
    cout << "Displaying address using arrays: " << endl;
    // use for loop to print addresses of all array elements
    for (int i = 0; i < 3; ++i)
    {
        cout << "&arr[" << i << "] = " << &arr[i] << endl;
    }
    // ptr = &arr[0]
    ptr = arr;
    cout<<"\nDisplaying address using pointers: "<< endl;
    // use for loop to print addresses of all array elements
    // using pointer notation
    for (int i = 0; i < 3; ++i)
    {
        cout << "ptr + " << i << " = "<< ptr + i << endl;
    }
    return 0;
}

```

```
/tmp/xiFXfUd4bf.o
Displaying address using arrays:
&arr[0] = 0x7ffc75cc2bac
&arr[1] = 0x7ffc75cc2bb0
&arr[2] = 0x7ffc75cc2bb4

Displaying address using pointers:
ptr + 0 = 0x7ffc75cc2bac
ptr + 1 = 0x7ffc75cc2bb0
ptr + 2 = 0x7ffc75cc2bb4
```

b. C++ Program to insert and display data entered by using pointer notation

```cpp
#include <iostream>
using namespace std;
int main()
{
    float arr[5];
   // Insert data using pointer notation
    cout << "Enter 5 numbers: ";
    for (int i = 0; i < 5; i++)
    {
        // store input number in arr[i]
        cin >> *(arr + i) ;
    }
    // Display data using pointer notation
    cout << "Displaying data: " << endl;
    for (int i = 0; i < 5; i++)
    {
        // display value of arr[i]
        cout << *(arr + i) << endl ;
    }
    return 0;
}
```

```
Enter 5 numbers: 56
5
6
78
3
Displaying data:
56
5
6
78
3
```

c. Sum of Array Using Pointer

```cpp
#include<iostream>
using namespace std;
int main() {
   int a[7], i, s = 0;
   int *ptr;
   cout << "Enter the Numbers: ";
   for (i = 0; i < 7; i++) {
      cin >> a[i];
   }
   ptr = a;
   for (i = 0; i < 7; i++) {
      s = s + *(ptr + i);
   }
cout << "\nSum of Elements of Array: " << s;
}
```

```
Enter the Numbers: 34 2 4 6 8 9 6

Sum of Elements of Array: 69
```

Pointers and Functions

Pointers are very much used in function declaration. Sometimes only with a pointer a complex function can be easily represented and accessed. The use of the pointers in a function definition may be classified into two groups they are call by value and call by reference.

C++ function call by value

The call by value method of passing arguments to a function 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.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

```cpp
#include <iostream>
using namespace std;
// function declaration
void swap(int x, int y);
int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   // calling a function to swap the values.
   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
// function definition to swap the values.
void swap(int x, int y) {
   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

```
/tmp/V9lrUmtodw.o
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

C++ function call by reference

The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```cpp
#include <iostream>
using namespace std;
// function declaration
void swap(int &x, int &y);
int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   // calling a function to swap the values.
   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
// function definition to swap the values.
void swap(int &x, int &y) {
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

```
/tmp/V9lrUmtodw.o
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Function Returning Pointers in C++

The way a function can returns an int, a float, a double, or reference or any other data type, it can even return a pointer. However, the function declaration must replace it. That is, it should be explicitly mentioned in the function's prototype. The general form of prototype of a function returning a pointer would be

type ∗function-name(argument list) ;

where type specifies the pointer type (int, float, double or any other data type) being returned by the function specified by function-name.

Example program illustrates how a pointer can be returned from a function

```cpp
#include <iostream>
using namespace std;
int *biger(int &, int &);
int main()
{
    int num1, num2, *c;
    cout<<"Enter two integers\n";
    cin>>num1>>num2;
    c = biger(num1, num2);
    cout<<"The bigger value = "<<*c;
    return 0;
}
int *biger(int &x, int &y)
{
    if(x>y)
    {
        return(&x);
    }
    else
    {
        return(&y);
    }
}
```

```
/tmp/xiFXfUd4bf.o
Enter two integers
34
56
The bigger value = 56
```

Chapter 3
Classes and Objects

Review of structures

Structure is a convenient tool for handling a group of logically related heterogeneous data types. Structure helps to organize data especially in large programs, because they provide group of variables of different data type to be treated as a single unit. It is most convenient way to keep related data under one roof.

- A structure is usually used when we need to store dissimilar or heterogeneous data together.
- The structure elements are stored in contiguous memory location as array.
- Structure elements can be accessed through a structure variable using a dot (.) operator.
- Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.
- All the elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.
- It is possible to pass a structure variable to a function either by value or by reference.
- It is possible to create an array of structure i.e. similar type of structure is placed in a common variable name. For example: we need to store the detail information of individual student in a class.

Example:

```
struct book
{
    char name[20];
    int pages;
    float price;
};
void main()
{
    struct book b1;
    b1.pages=500;
    b1.price=815.5;
    strcpy(b1.name, "C Programming");
    printf("\nName=%s, pages=%d and price=%f",b1.name,b1.pages,b1.price);
    getch();
    clrscr();
}
```

Limitation of Structure (Structure VS Class)

- The standard C does not allow the struct data type to be treated like built-in data types. For example

```
struct complex
{
        float real, imag;
};
struct complex c1,c2,c3;
```

The complex numbers c1,c2,c3 can easily be assigned values using the dot operator. But we cannot add two complex numbers or subtract one from the other. That is c3=c1 + c2; is illegal in C.

- Also, data hiding is not permitted in C, that is structure members can be directly accessed by the structure variable by any function anywhere in their scope.
- In c structure, it contains only variable or data as member but In C++, a class can have both variables and functions as members. It can also declare some of its members as private so that cannot be accessed directly by the external functions.
- The data member in C++ structure are public by default which may lead to security issue but in Class the default access specifier is private.
- Structure in C are only object based but Class support all features of OOP.

Q. Difference between the Structure in C++ and class in C++

Class

A class is user-defined data type that includes different member data and associated member functions to access on those data. Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into packages called classes. The data components of the class are called data members and the function components are called member functions.

Since class is a collection of logically related data items and the associated functions which operate and manipulate those data. This entire collection can be called a new **user defined data-type.** So, classes are user-defined data types and behave like the built-in types of a programming language.

Generally, a class specification has two parts.

i. Variable declaration

ii. Function definition

The general form of a class declarations is

```
class class-name
{
        private:
                Variable declarations;
                Function declarations;
        public:
                Variable declarations;
                Function declarations;
```

```
};
#include<conio.h>
#include<iostream>
using namespace std;
class student           // class name student, starting of class
{
private:                // private data members name roll;
      char name[20];
      int roll;
public:                         //public methods input() and output()
      void input()
      {
            cin>>name>>roll;
      }
      void output()
      {
         cout<<"Name : "<<name<<endl;
          cout<<" Roll No :"<<roll<endl;
    }
};
int main()
{
      student s1;       // object (s1) declaration inside main()
      s1.input();       //calling methods syntax: object.method();
      s1.output();      //calling methods syntax: object.method();
      getch();
      return 0;
}
```

Difference between class and structure

Class	Structure
Class is a reference type and its object is created on the heap memory	Structure is a value type and its object is created on the stack memory.
Class can create a subclass that will inherit parent's properties and methods,	Structure does not support the inheritance.
A class has all members private by default.	A struct is a class where members are public by default.
Sizeof empty class is 1 Byte	Size of empty structure is 0 Bytes

Access Specifiers

Access specifiers are the keyword that controls access to data member and member functions within user-defined class. The access restriction to the class members is specified by the labeled public, private, and protected sections within the class body. The keywords public, private, and protected are called access specifiers.

```
class Demo
{
        public:
                // public members go here
        protected:
                // protected members go here
        private:
                // private members go here
};
```

Public

All the class members declared under public will be available to anywhere in the program. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Private

The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

Protected

Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass (derived class) of that class.

Creating object

Once a class has been specified (or declared), we can create variables of that type (by using the class name as datatype). General syntax for declaration of object is

Class_name Object_name

Example:

Student s1; //memory for s1 is allocated.

The above statement creates a variable s1 of type Student. The class variables are known as objects. So, s1 is called object of class Test.

Accessing Class Member

The private data members of a class can be accessed only through the member functions of that class. However, the public members can be accessed from outside the class using object name and dot operator(.). The following is the format for calling public members of a class.

Object_name.Function _name(actual argument)

Object_name.Data_name

Example:

If e1 is the object of Class Employee then we can access the public member function getname() with the following statement

e1.getname();

Defining a member Function

Member functions can be defined in two ways.

- Outside the class
- Inside the class

The code for the function body would be identical in both the cases. Irrespective of the place of definition, the function should perform the same task.

Outside the class

In this approach, the member functions are only declared inside the class, whereas its definition is written outside the class.

General form:

return-type class-name::function-name(argument-list)
{
- - - - - - - - -
- - - - - - - - -- - //function body
- - - -- - -- --
}

Example:

void Employee::getdata()
{
- - - - - - - - -
- - - - - - - - -- - //function body
- - - -- - -- --
}

Inside the class

Function body can be included in the class itself by replacing function declaration by function definition. If it is done, the function is treated as **an inline function**. Hence, all the restrictions that apply to inline function, will also apply here.

Example:

```
class Demo
{
        int a,b;
        public:
        void getdata()
        {
                cin>>a>>b;
        }
};
```

Here, getdata() is defined inside the class. So, it will act like an inline function.

A function defined outside the class can also be made ‘inline’ simply by using the qualifier ‘inline’ in the header line of a function definition.

Example:

```
class A
{
- - - - - - - - -
- - - - - - - - - -
public:
        void getdata(); // function declaration inside the class
};
```

```
void A::getdata()
{
        //function body
}
```

Example of C++ Program with Class

Q. Create a class Rectangle with two data members (length and breadth), a function called readdata() to take detail of sides and a function called displaydata() to display area of rectangle. In main create two objects of a class Rectangle and for each object, call both of the function.

Q. Create a class Rectangle with two data members (length and breadth), a function called readdata() to take detail of sides and a function called displaydata() to display area of rectangle. The two functions are defined outside the class. In main create two objects of a class Rectangle and for each object, call both of the function.

```
#include<iostream>
using namespace std;
class Rectangle
{
        float length,breadth;
        public:
                void readdata();
                void displaydata();
};
void Rectangle::readdata()
{
        cout<<"Enter lengtha and breadth of rectangle"<<endl;
        cin>>length>>breadth;
}
void Rectangle::displaydata()
{
        float area;
        area=length*breadth;
        cout<<"Area of rectangle= "<<area<<endl;
}
int main()
{
        Rectangle rect1,rect2;
        rect1.readdata();
        rect1.displaydata();
        rect2.readdata();
        rect2.displaydata();
}
```

Q. Define a class to represent a bank account. Include the following members

Data Member:

i. Name of account holder

i. Account Number

ii. Account Type

iii. Balance Amount

Member Functions:

- to take detail of the fields from user
- To deposit an amount
- To withdraw an amount
- To display name and balance

```
#include<iostream>
using namespace std;
class Account
{
        char name[50],acctype[20];
        int accno;
        float wdraw,bal,dep;
        public:
                void setdata()
                {
                        cout<<"Enter name of account holder"<<endl;
                        cin>>name;
                        cout<<"Enter the type of account"<<endl;
                        cin>>acctype;
                        cout<<"Enter the account number"<<endl;
                        cin>>accno;
                        cout<<"Enter the balance"<<endl;
                        cin>>bal;
                }
                void deposit()
                {
                        cout<<"Enter amount to be deposited"<<endl;
                        cin>>dep;
                        bal=bal+dep;
                }
                void withdraw()
                {
                        cout<<"Enter amount to be withdraw"<<endl;
                        cin>>wdraw;
```

```
                bal=bal-wdraw;
        }
        void display()
        {
                cout<<"Name= "<<name<<endl;
                cout<<"Balance= "<<bal;
        }

};
int main()
{
        Account a;
        a.setdata();
        a.deposit();
        a.withdraw();
        a.display();
}
```

Private Member Function

As we have seen, member functions are, in general, made public. But in some cases, we may need a private function to hide them from outside world. Private member functions can only be called by another function that is a member of its class. Object of the class cannot invoke private member functions using dot operator.

Example:

```
class Demo
{
        int a,b;
        void func1(); //private member by default
        public:
        void func2();
};

void Demo::func2()
{
        Func1(); //func1() is private member so can't be accessed by using object of class Demo
}
```

Q. Define a class Student with the following specification

Private members:

Data member: Id, name, eng, math, science

Member function: float calculateTotal() to return the total obtained mark.

Public members:

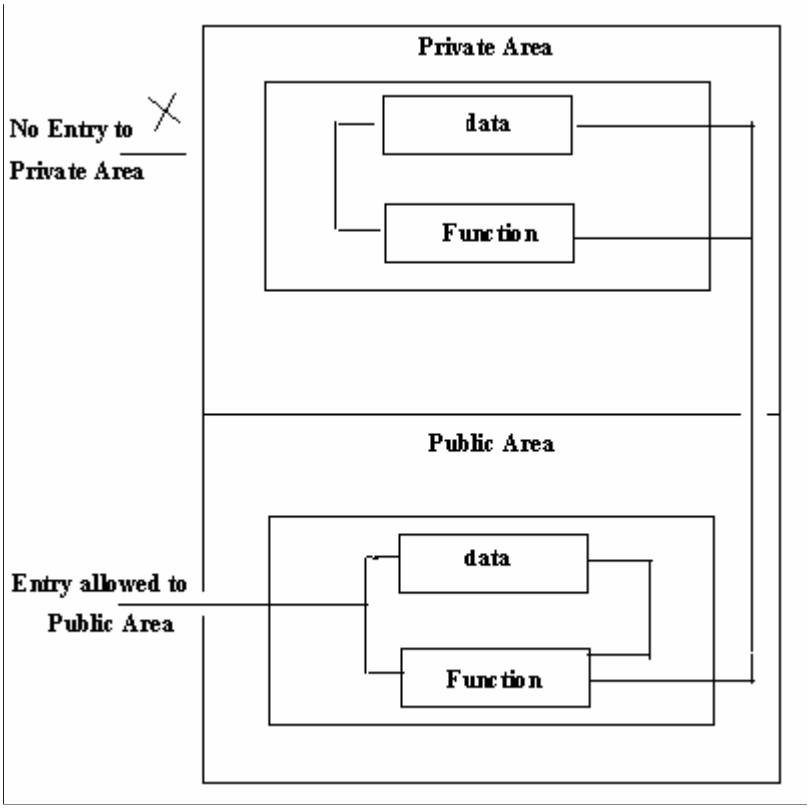
Member functions:

getData() to take input(id, name, eng, math, science) from user showData() to display all the data member along with total on the screen.

Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Student
{
	int id;
	char name[50];
	float eng,math,science;
	float calculateTotal()
	{
		return(eng+math+science);
	}
	public:
	void getData()
	{
		cout<<"Enter id"<<endl;
		cin>>id;
		cout<<"Enter name"<<endl;
		cin>>name;
		cout<<"Enter marks in maths, science and english"<<endl;
		cin>>math>>science>>eng;
	}

	void showData()
	{
	float t;
	cout<<"ID= "<<id<<endl;
	cout<<"Name ="<<name<<endl;
	cout<<"Marks in maths, science and english="<<math<<science<<eng<<endl;
	t=calculateTotal(); //, we can call members directly with their name
	cout<<"Total= "<<t;
	}

};
int main()
{
	Student s;
	s.getData();
	s.showData();
}
```

Q. How Class provides the data hiding



The data declared inside the class are called the data member and the functions are called the member function. Only the member function can have access to the private data members and private functions. The private members have no access to the external function

i.e. private members are hidden from the external function so provide the data encapsulation.

The binding of data and function together into a single class-type variable is called encapsulation.

Example to show the data encapsulation:

```
class test
    {
        int x;
        public:
            int y;
    };
    void main()
    {
```

```
    test t;          //creating the object
    t.x=10;          //error, x is a private
    t.y=15;          //ok, y is public
}
```

Nested of member function

- Since a member function of a class can only be called by an object of that class using a dot operator.
- However, a member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

```
1  //nesting member function
2  #include<iostream>
3  using namespace std;
4  class set
5  {
6      private:
7          int m,n;
8      public:
9          void input();
10         int largest();
11         void display();
12 };
13 inline int set::largest()
14 {
15     if(m>=n)
16         return m;
17     else
18         return n;
19 }
20 inline void set::input()
21 {
22    cout<<"input values of m & n"<<endl;
23    cin>>m>>n;
24 }
25 void set::display()
26 {
27     cout<<"largest value="<<largest()<<endl;
28 }
29 int main()
30 {
31     set set1;
32     set1.input();
33     set1.display();
34     return 0;
35 }
36
```

```
/tmp/ccjzrEqyIU.o
input values of m & n
12
56
largest value=56
```

An Array of Objects

- As we know an array is a collection of similar data types. We can have an array of user defined data types class. Such variables are called arrays of objects. Like a structure we can use array of a class. e.g.

```
class Test
{
    float a;
};
```

- The array of object can be declared as:

```
Test T[n];      //arrays of n Test
T[i].input_data( );
T[i].display( );
```

```
1
2  #include<iostream>
3  using namespace std;
4  //Example to show arrays of Objects
5  class test
6  {
7      float a;
8      public:
9      void set_data(float no)
10     {
11         a=no;
12     }
13     void input_data()
14     {
15         cout<<"Enter the floating no:"<<endl
                   ;
16         cin>>a;
17     }
18     void display()
19     {
20         cout<<"The no is:"<<a<<"\n";
21     }
22 };
23 int main()
24 {
25     test t[4];
26     t[0].set_data(10.55);
27     t[1].input_data();
28     t[2].input_data();
29     t[3].set_data(11.66);
30     for(int i=0;i<4;i++)
31         t[i].display();
32 }
33
```

```
/tmp/ccjzrEqyIU.o
Enter the floating no:
12.9
Enter the floating no:
45.1
The no is:10.55
The no is:12.9
The no is:45.1
The no is:11.66
```

Class and Object and Memory

When a class is specified, memory space of all the member function is allocated but memory allocation is not done for the data member.

When an object is created, memory allocation for its data members is done. The logic behind separate memory allocation for member functions is quite obvious.

All instances of a particular class would be using the same member functions but they may be storing different data in their data members.

Memory allocation for objects is illustrated in fig below

Object 1 *Object 2* *Object n*

It can be observed that "n" objects of the same class are created and data members of those objects are stored in distinct memory location, whereas the member functions of object 1 to object n are stored in the same memory area. Therefore, each object has a separate copy of data members and the different objects share the member functions among them.

Questions

1. What is encapsulation and how does it help data hiding in Object-Oriented Programming.
2. Create a class called employee that contains a empname (array of string) and an empid (type long). Include a member function called getdata( ) to get data from the user for inserting into the object and another function called putdata( ) to display the data. Write a main ( ) program to exercise this class. It should create an array of the type employee and then invite the user to input data for up to 100 employees. Finally, it should print out the data for all the employees. You need to make the array of employees an external variable.
3. How do Structure in C and C++ differ?
4. What is a class? How does it accomplish data hiding?
5. Define the term Data and Class and what the relationship between a Class and Object is.
6. Explain how Class provides data hiding with example.
7. What is meant by data Encapsulation?
8. Explain a member function? How is a member function of a class defined?
9. Differentiate between public and private definition.
10. How does a C++ structure differ from C++ class?
11. What are Objects and how they are created? “Can there be more than one destructor in the same class? If no, explain it with suitable example.”

Constructors:

Constructors are member functions that are executed automatically when an object is created. Thus, no explicit call is necessary to invoke them through the dot operator.

The constructors are invoked whenever an object of its associated class is created. It is called constructor because it constructs the value of data members of the class. Constructors are mainly used for data initialization.

Syntax:

```
class class-name
{
        private:
                ----------
                ----------
        public:
        class-name() //constructor
         {
                //Body of constructor
         }
};
```

The constructors have the same name as the class_name.

When an object is created, the following process will take place.

- The object occupies space at a particular time. Instantiation of an object always involves reserving enough memory space for the data of that object.
- The instantiation does not reserve the memory for the methods. They exist only once for class, not once for every object.
- In addition to the reservation of space, the constructor may also be extended to other processes for initialization of data of that object.
- Constructor is a function (member) which is called automatically when object is created.

```cpp
//constructor
//automatic initialization is carried out
    using a special member function.
#include<iostream>
using namespace std;
class counter
{
    private:
        int count;
    public:
        counter()
        {
            count=0;
        }
        void inc_count()
        {
            count++;
        }
        int get_count()
        {
            return count;
        }
};
int main()
{
    counter c1,c2;  //automatically count=0
    cout<<"\nc1="<<c1.get_count();
    cout<<"\nc2="<<c2.get_count();
    c1.inc_count();
    c2.inc_count();
    c2.inc_count();
    cout<<"\nc1="<<c1.get_count();
}
```

```
/tmp/ccjzrEqyIU.o
c1=0
c2=0
c1=1
```

Characteristics of constructors:

- Constructors name is the same as the class name.
- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore they cannot return value.
- They cannot be inherited.
- Like other functions, they can have default arguments.
- Constructors of a class are the first member function to be executed.

Types of constructor

There are three types of constructors:

1. The default constructor
2. User-defined constructor
3. Copy constructor

Default constructor (implicit constructor)

The constructor that accepts no parameter is called the default constructor.

The default constructor of class A is A::A( ). If no such constructor is defined, then the compiler supplies a default constructor. Therefore, a statement such as,

A a;

- The compiler provides a (hidden) default constructor that has no arguments.
- The default constructor takes no arguments and performs no processing other than reservation of memory.
- This constructor is always called by compiler if no user-defined constructor is provided.
- This constructor is automatically called while creating the object.

The main advantages of default constructor is to allocate memory for objects.

Example:

```
class student
{
  private:
        ----------
```

```
public:
        ----------
        ----------
};
void main( )
{
  student s1,s2    //default constructor student( ),
}
```

User-defined constructor:

If initialization of data of objects is required while creating an object, then the programmer has to define his own constructor for that purpose. The code of a user defined constructor does not actually cause memory to be reserved for the data because it is still done by default constructor automatically.

The user defined constructor takes arguments.

The main advantages of user defined constructor are to initialize the object while it is created.

```
class test
{
  float a;
 public:
        test (float number)
        {
                a= number;
        }
        void display( )
        {
                cout<<"The number is:"<<a;
        }
};
 void main( )
{
        test t1(2.99);
        test t2=3.99;
        test t3= test(8.99);
```

```
        t1.display();
        t2.display();
        t3.display();
    }
```

Copy Constructor:

Constructor having a reference parameter is known as copy constructor. The copy constructor creates an object as an exact copy of another object in terms of its attributes. In copy constructors, one newly instantiated object equals another exiting object of the same class.

Copy constructor are called using assignment operator or object as arguments automatically.

```
Class test
{
    int id;
        public:
        test ()
        {
        }
        test (int a)
        {
            id=a;
        }
        test (test &x)
        {
            id= x.id;
        }
        void display ()
        {
            cout<<id;
        }
};
int main ()
{
    test a (100);
    test b(a);
    test c=a;
    test d;
    d=a;
    cout<<"id of a:"<<a.display( );
    cout<<"id of b:"<<b.display( );
    cout<<"id of c:"<<c.display( );
    cout<<"id of d:"<<d.display( );
}
```

Parameterized Constructor:

The constructors that can take arguments are called parameterized constructor.

```
Class test
{
    float a;
    public:
    test(float number)
     {
         a=number;
     }
     void display( )
     {
         cout<<"the number is:"<<a;
     }
};
void main( )
{
    test t1(4);
    test t2=5;
    t1.display( );
    t2.display( );
}
```

Constructor with default arguments:

It is possible to define constructor with default arguments.

Example, if we have class like:

```
class complex
    {
        float x,y;
            public:
            complex(float real, float imag =0)
            {
                x=real;
                    y=imag;
            }
    };
```

The default value of the argument imag is zero. Then, the statement

complex c (5.0);

assign the value 5.0 to the real variable and 0.0 to the imag (by default). However, the statement

complex c (2.0,3.0);

assign 2.0 to real and 3.0 to imag.

Constructor Overloading

A class can has multiple constructors. If more than one constructor is used in a class, it is known as constructor overloading. All the constructors have the same name as the corresponding class, and they differ in terms of number of arguments, data types of argument or both. This makes the creation of object flexible.

Example:

```
1  //constructor overloading
2  #include<iostream>
3  using namespace std;
4  class Account
5  {
6       private:
7          int accno;
8          float balance;
9       public:
10         Account()   //constructor1
11         {
12             accno=1024;
13             balance=5000.55;
14         }
15         Account(int acc)   //constructor2 with one argument
16         {
17             accno=acc;
18             balance=0.0;
19         }
20         Account(int acc, float bal)   //constructor3,with two arguments
21         {
22             accno=acc;
23             balance=bal;
24         }
25         void display()
26         {
27             cout<<"Account no.="<<accno<<endl;
28             cout<<"Balance="<<balance<<endl;
29         }
30 };  //end of class definition
31 int main()
32 {
33    Account acc1;  //constructor1
34    Account acc2(100);  //constructor2
35    Account acc3(200, 8000.50);  //constructor3
36    cout<<endl<<"Account information"<<endl;
37    acc1.display();
38    acc2.display();
39    acc3.display();
40 } //end of main()
41
```

```
/tmp/ccjzrEqyIU.o
Account information
Account no.=1024
Balance=5000.55
Account no.=100
Balance=0
Account no.=200
Balance=8000.5
```

Destructors

Destructors are the special function that destroys the object that has been created by a constructor. In other words, they are used to release dynamically allocated memory and to perform other “cleanup” activities.

Destructors, too, have special name, a class name preceded by a tilde sign (~).

Example:

```
~Demo ()
{

}
```

Destructor gets invoked, automatically, when an object goes out of scope (i.e. exit from the program, or block or function). They are also defined in the public section. Destructor never takes any argument, nor does it return any value. So, they cannot be overloaded.

Note: whenever 'new' is used to allocate memory in the constructors, we should use 'delete' to free the memory. For example:

```
Demo:: ~Demo()
{
delete obj;
}
```

Also destructor can be defined inside class

```
#include<iostream>
using namespace std;
class Demo
{
        int *p;
public:
                Demo()
                {
                        p=new int();
                        *p=7;
                }
        ~Demo()
        {
                delete(p);
        }
};
```

For example:

Note: Destructor also can be invoked manually but basically it is useless because the destructor is automatically invoked when the scope of created object ends. So this lead to invoking of destructor twice.

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

```
}; int
main()
{
        Demo d;
        d.~Demo();

}
```

Characteristic of Destructor

- It can't take any argument so can't be overloaded.
- It has same name as the class name but preceded by a tilde symbol.
- It can't be declared statics.
- It can't return a value.
- It should have public or protected access specifiers.
- Only one destructor exists in a class.

Example:

The below program demonstrates the concept of constructor and destructor

```
#include<iostream>
using namespace std;
class Demo
{ static int
count;
public:
Demo()
        {
                count++;
                cout<<"Object created= "<<count<<endl;
        }
        ~Demo()
        {
                cout<<"Object Destroyed= "<<count<<endl;
                count--;
        }
};
int Demo::count;
int main()
{
        Demo a1,a2,a3;
        {
                Demo a4;
```

```
        }
    }
```

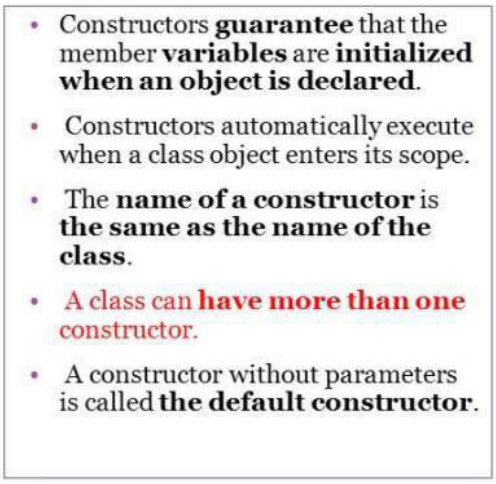
Output:

```
Object created= 1
Object created= 2
Object created= 3
Object created= 4
Object Destroyed= 4
Object Destroyed= 3
Object Destroyed= 2
Object Destroyed= 1
```

Q. WAP to initialize a filed named name with your name using dynamic constructor. Also write destructor to release the dynamically allocated memory.

```
#include<iostream>
using namespace std;
#include<string.h>
class Demo
{
        char *name;
public:
                Demo(char *str)
                {
                        int len;
                        len=strlen(str);
                        name=new char[len+1];
                        strcpy(name,str);
                }
                void display()
                {
                        cout<<"Name= "<<name<<endl;
                }
                ~Demo()
                {
                        delete(name);
                }
}; int
main()
{
        Demo
d("Ram"),d1("Hari");
d.display();    d1.display();

}
```

Constructor vs Destructor



Q. "Can there be more than one destructor in the same class? If no, explain it with suitable example."

Objects as Function Arguments

Like any other data types, an object may be used as function arguments. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called pass-by-value. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.
The second method is called pass-by-reference. When an address of the object is passed, the called function works directly on the actual objects used in the call. This means that any changes made to the object inside the function will reflect in the actual object.

```cpp
#include<iostream>
using namespace std;
class Distance
{
    int feet;
    float inches;
    public:
    Distance()
    {
        feet=0;
        inches=0.0;
    }
    Distance(int f, float i)
    {
        feet=f;
        inches=i;
    }
    void getdata()
    {
        cout<<"Enter the feet:";cin>>feet;
        cout<<"\nEnter the inches:";cin>>inches;
    }
    void display()
    {
        cout<<feet<<"\'"<<inches<<"\"";
    }
    void add_distance(Distance, Distance);
};
void Distance::add_distance(Distance d1,Distance d2)
{
    inches=d1.inches+d2.inches;
    while(inches>=12.0)
    {
        inches-=12.0;
        feet +=1;
    }
    feet+=d1.feet+d2.feet;
}
int main()
{
    Distance dist1,dist3;
    Distance dist2(11,8.5);
    dist1.getdata();
    cout<<"\nDistance1=";dist1.display();
    cout<<"\nDistance2=";dist2.display();
    dist3.add_distance(dist1,dist2);
    cout<<"\nDistance3=";dist3.display();
    return 0;
}
```

```
/tmp/ccjzrEqyIU.o
Enter the feet:12
Enter the inches:23

Distance1=12'23"
Distance2=11'8.5"
Distance3=25'7.5"
```

How to pass object from C++ Functions?

```cpp
// C++ program to calculate the average marks of two
    students
#include <iostream>
using namespace std;
class Student {

   public:
    double marks;
    // constructor to initialize marks
    Student(double m) {
        marks = m;
    }
};
// function that has objects as parameters
void calculateAverage(Student s1, Student s2)
{
    // calculate the average of marks of s1 and s2
    double average = (s1.marks + s2.marks) / 2;
   cout << "Average Marks = " << average << endl;
}
int main()
{
    Student student1(88.0), student2(56.0); // pass
        the objects as arguments
    calculateAverage(student1, student2);
    return 0;
}
```

```
/tmp/ccjzrEqyIU.o
Average Marks = 72
```

How to return object from C++ Functions?

```cpp
#include <iostream>
using namespace std;
class Student
{
   public:
    double marks1, marks2;
};
// function that returns object of Student
Student createStudent()
{
    Student student;
    // Initialize member variables of Student
    student.marks1 = 96.5;
    student.marks2 = 75.0;
    // print member variables of Student
    cout << "Marks 1 = " << student.marks1 << endl;
    cout << "Marks 2 = " << student.marks2 << endl;
    return student;
}
int main()
{
    Student student1;
    // Call function
    student1 = createStudent();
    return 0;
}
```

```
/tmp/ccjzrEqyIU.o
Marks 1 = 96.5
Marks 2 = 75
```

Static Members

We can define class members static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present.

```cpp
#include <iostream>
using namespace std;
class Box {
   public:
      static int objectCount;
      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h =
          2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }
      double Volume() {
         return length * breadth * height;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount <<
       endl;
   return 0;
}
```

```
/tmp/ccjzrEqyIU.o
Constructor called.
Constructor called.
Total objects: 2
```

Questions

1. What are constructor and destructor? Explain different types of constructor used in C++.
2. Create a class called Distance with two data members inch and feet. Provide Constructor and different member function with the following operations.
   - To input data for Distance objects.
   - To show the data of Distance objects.
   - Member function to add two Distance objects passed as object as function arguments and then display the result.
3. Create a class called Distance with two data members inch and feet. Provide Constructor and different member function with the following operations.
   - To input data for Distance objects.
   - To show the data of Distance objects.
   - Member function to add two Distance objects passed as object as function arguments leaving the result in the third Distance object and then display the result.
4. Create the class called time that has separate integer member (attribute) data for hours, minutes, and seconds. One constructor should initialize this data to 0, and another should initialize it to a fixed values. A member function should display it in 10:34:55 format. The final member function should add the two objects of type time passed as arguments. A main program should create two initialize time objects and one not initialize time object. Then it should add the two initialize value together, leaving the result in the third time variable. Finally, it should display the value of this third variable.
5. Point out few important characteristics of constructors.
6. Differentiate between constructors and other member functions.
7. Explain the use of destructors. Point out the main differences between constructors and destructors.
8. Differentiate between parameterized and non-parameterized constructors.
9. Discuss the importance of the constructor.
10. What is the primary role of the constructors?
11. Discuss the primary role of the user-defined constructors with example

Extra

Q. WAP to perform the addition of time in hour and minute format using the concept of object as function arguments (Pass by value)

```
#include<iostream>
using namespace std;
class Time
{
        int hour,min;
        public:
                void gettime()
```

```
        {
                cout<<"Enter hour"<<endl;
                cin>>hour;
                cout<<"Enter minute"<<endl;
                cin>>min;
        }
        void puttime()
        {
                cout<<"Hour= "<<hour<<"\t";
                cout<<"Minute= "<<min<<endl;
        }
        void add(Time a, Time b)
        {
                min=a.min +b.min;
                hour=min/60;
                min=min%60;
                hour=hour +a.hour+b.hour;
        }
};
int main()
{
        Time t1,t2,t3;
        t1.gettime();
        t2.gettime();
        t3.add(t1,t2);
        t1.puttime();
        t2.puttime();
        t3.puttime();
}
```

Output:

```
Enter hour
1
Enter minute
30
Enter hour
1
Enter minute
30
Hour= 1 Minute= 30
Hour= 1 Minute= 30
Hour= 3 Minute= 0
```

Q. WAP to perform the addition of time in hour and minute format using the concept of object as function arguments(Pass by reference)

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

```
        int hour,min;
        public:
                void gettime()
                {
                        cout<<"Enter hour"<<endl;
                        cin>>hour;
                        cout<<"Enter minute"<<endl;
                        cin>>min;
                }
                void puttime()
                {
                        cout<<"Hour= "<<hour<<"\t";
                        cout<<"Minute= "<<min<<endl;
                }
                void add(Time &a, Time &b) // object 'a' work on the same address that of object t1
                {
                        min=a.min +b.min;
                        hour=min/60;
                        min=min%60;
                        hour=hour +a.hour+b.hour;
                }
};
int main()
{
        Time t1,t2,t3;
        t1.gettime();
        t2.ge1ttime();
        t3.add(t1,t2);
        t1.puttime();
        t2.puttime();
        t3.puttime();
}
```

Q. WAP to find the square of a given number using the concept of object as argument(call by reference) and returning object.

```
#include<iostream>
using namespace std;
class Demo
{
        int a;
        public:
                void setdata(int x)
                {
                        a=x;
                }

                Demo square(Demo *p)
                {
                        Demo x;
                        x.a=p->a * p->a;
                        return x;

                }

                void putdata()
                {
                        cout<<a;
                }
};
int main()
{
        Demo o1,o2,o3;
        o1.setdata(10);
        o3=o2.square(&o1);
        o3.putdata();
}
```

Q. Create a class Complex with two data member (x and y) for storing real and imaginary part of a complex number and three member functions named void input(int ,int) to initialize x and y, Complex sum(Complex, Complex) to return the Complex object with sum, and void show() to display the sum of two complex number. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Complex
{
        int x,y;
        public:
        void input(int real,int img)
        {
```

```
            x=real;
            y=img;
        }

        Complex sum (Complex c1, Complex c2)
        {
            Complex c3;
            c3.x=c1.x+c2.x;
            c3.y=c1.y+c2.y;
            return c3;
        }
        void show()
        {
            cout<<"Real Part= "<<x<<endl;
            cout<<"Imaginary part= "<<y<<endl;
        }
};
int main()
{
        Complex a,b,c;

        a.input(5,6);
        b.input(4,5);
        c=c.sum(a,b);
        c.show();
}
```

Create a class Demo with two data member (id, name), a default constructor to initialize these fields and a member function display() to show student details. Write a main program to test your class.

```
#include<iostream.>
#include<string.h>
using namespace std;
class Demo
{
        int id;
        char name[10];
        public:
            Demo() // Default constructor
            {
                id=5;
                strcpy(name,"ram");
            }
            void display()
            {
                cout<<"ID= "<<id<<endl;
                cout<<"Name= "<<name<<endl;
            }
};
```

```
int main()
{
	Demo d;
	d.display();
}
```

Q. WAP to find the area of rectangle using the concept of parameterized constructor

```
#include<iostream>
using namespace std;
class Demo
{
	int l,b;
	public:
		Demo(int length, int breadth)
		{
			l=length;
			b=breadth;
		}
		void area()
		{
			int a;
			a=l*b;
			cout<<"Area of rectangle= "<<a<<endl;
		}
};
int main()
{
	Demo d(5,6);
	d.area();
}
```

Output:

```
Area of rectangle= 30
```

Q. Create a class Person with data member Name, age, address and citizenship_number. Write a constructor to initialize the value of a person. Assign citizenship number if the age of the person is greater than 16 otherwise assign values zero to citizenship number. Also, create a function to display the values. Write a main program to test your class.

```
#include<iostream.>
#include<string.h>
using namespace std;
class Demo
{
	char name[50],address[50];
	int citizenship_number,age;
	public:
		Demo(char n[],char ad[],int a, int cn)
```

```
        {
            strcpy(name,n);
            strcpy(address,ad);
            citizenship_number=cn;
            age=a;
        }
        void display()
        {
            cout<<"Name= "<<name<<endl;
            cout<<"Address= "<<address<<endl;
            cout<<"Age= "<<age<<endl;
            cout<<"CitizenshipNumber="<<citizenship_number<<endl;
        }

};
int main()
{
    char nm[50],ad[50];
    int cn,a;
    cout<<"Enter name"<<endl;
    cin>>nm;
    cout<<"Enter address"<<endl;
    cin>>ad;
    cout<<"Enter age"<<endl;
    cin>>a;
    cn=0;
    if(a>16)
    {
        cout<<"Enter Citizenship Number"<<endl;
        cin>>cn;
    }
    Demo d(nm,ad,a,cn);
    d.display();

}
```

Q. WAP to find the area of circle and rectangle using the concept of constructor overloading

```
#include<iostream.>
using namespace std;
class Demo
{
    public:
        Demo(float radius)
        {
            cout<<"Area of circle= "<<(3.1416*radius*radius)<<endl;
        }
```

```
                Demo(float length, float breadth)
                {
                        cout<<"Area of rectangle= "<<(length * breadth)<<endl;
                }
};
int main()
{
        Demo d(4);// invokes constructor with one parameter
        Demo d1(5,6); //invokes constructor with two parameter
}
```

Output:

```
Area of circle= 50.2656
Area of rectangle= 30
```

Q. WAP to initialize a filed named name with your name using dynamic constructor. Also write destructor to release the dynamically allocated memory.

```
#include<iostream>
using namespace std;
#include<string.h>
class Demo
{
        char *name;
        public:
                Demo(char *str)
                {
                        int len;
                        len=strlen(str);
                        name=new char[len+1];
                        strcpy(name,str);
                }
                void display()
                {
                        cout<<"Name= "<<name<<endl;
                }
                ~Demo()
                {
                        delete(name);
                }
};
int main()
{
        Demo d("Ram"),d1("Hari");
        d.display();
        d1.display();

}
```

Constructor vs Destructor

- Constructors **guarantee** that the member **variables** are **initialized when an object is declared**.
- Constructors automatically execute when a class object enters its scope.
- The **name of a constructor** is **the same as the name of the class**.
- A class can **have more than one** constructor.
- A constructor without parameters is called **the default constructor**.

- Destructor **automatically execute when a class object goes out of scope.**
- The name of a destructor is the tilde (~), followed by the class name (no spaces in between).
- A class **can have only one destructor.**
- The destructor has no parameters.

Default vs parameterized constructor

Default Constructor	Parameterized Constructor
A constructor that has no parameter is called default constructor.	A constructor that has paramteter(s) is called parameterized constructor.
Default constructor is used to initialize object with same default value like 0, null.	Parameterized constructor is used to initialize each object with different values.
When data is not passed at the time of creating an object, default constructor is called but not parameterized.	When data is passed at the time of creating an object, default constructor as well as parameterized constructor is called.

Constructor vs Method

Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Chapter 4
Operator Overloading

Operator Overloading:

The concept by which we can give special meaning to an operator of C++ language is known as operator overloading.

An operation may exhibit different behavior in 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. Operator overloading allows us to assign multiple meanings to the operators. Compiler generates the appropriate code based on the manner in which the operator is used.

When an operator is overloaded, that operator does not loses its original meaning. Instead, it gains additional meaning relative to the class for which it is defined. We can overload the C++ operators except the following:

- ✓ Scope Resolution Operator (: :)
- ✓ Membership Operator (.)
- ✓ Size of operator (size of)
- ✓ Conditional Operator (? :)
- ✓ Pointer to Member Operation (.*)

Syntax for operator overloading:

The operator overloading is done by using a special function called operator function, called operator function. The general syntax for operator function is:

```
return_type operator_op (arg list)
{
  Function body
}
```

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 to operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

The process of operator overloading generally involves following steps.

1. Declare a class whose objects are to be manipulated using operators.
2. Declare the operator function, in public part of class. It can either normal member function or friend function.
3. Define operator function within the body of a class or outside the body of the class but function prototype must be inside the class body.

Unary Operator Overloading

A unary operator acts on only one operand. Examples of unary operators are the increment and decrement operators ++ and -- and the unary minus as in -45.

Syntax:

Return_type operator unary_operator_symbol ()
{
//body of functions
}

Example:

```
#include <iostream>
using namespace std;
class index
{
    public:
    int count;
    void getdata(int i)
    {
        count=i;
    }
    void showdata ()
    {
         cout<< "count=" <<count<<endl;
    }
    void operator++()
    {
         ++count;
    }
};
int main()
{
    index a1;
    a1.getdata(3);
    a1.showdata();
    ++a1;
    a1.showdata();
    return 0;
}
```

```
/tmp/ccjzrEqyIU.o
count=3
count=4
```

Overloading Unary operator that return a value

```
#include <iostream>
using namespace std;
//unary operator overloading  with return type.
class index
{
    int count;
    public:
    index()
    {
        count=0;
    }
    void display()
    {
        cout<<count;
    }
    index operator++()
    {
        ++count;
        index temp;
        temp.count=count;
        return temp;
    }
};
int main()
{
    index c,d;
    cout<<endl<<"C=";
    c.display();
    ++c;
    cout<<endl<<"C=";
    c.display();
    d=++c;
    cout<<endl<<"C=";
    c.display();
    cout<<endl<<"D=";
    d.display();
    return 0;
}
```

```
/tmp/ccjzrEqyIU.o
C=0
C=1
C=2
D=2
```

Nameless temporary object:

A convenient way to return an object is to create a nameless temporary object in the return statement itself. In the above program, modify class definition by

```
public :
index()
{
        count=0;
}
…… ……
index operator++()
{
        ++count;
        return index(count);
}
```

Note that the operator ++() function has changed. In this function the statement

```
return index(count);
```

creates an object of type index. This object has no name.

Binary operator overloading

Binary operators can be overloaded as unary operator. The syntax for overloading the binary operator is:

```
return_type operator operator_symbol(arg)
{
    //body of function.
}
```

```cpp
#include<iostream>
using namespace std;
class Complex
{
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i =0)
        {
                real = r;
                imag = i;
        }
   // This is automatically called when '+' is used with between
       twoComplex objects
        Complex operator + (Complex &obj)
        {
            Complex res;
            res.real = real + obj.real;
            res.imag = imag + obj.imag;
            return res;
        }
        void print()
        {
            cout << real << " + i" << imag << endl;
        }
};
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

```
/tmp/ccjzrEqyIU.o
12 + i9
```

Overloaded relational operator

The relational operators can also be overloaded as other binary operators to extend their semantics. The following example shows the '>' operator overloading to use it for comparison of two user defined objects

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
      // overloaded < operator
      bool operator <(const Distance& d) {
         if(feet < d.feet) {
            return true;
         }
         if(feet == d.feet && inches < d.inches) {
            return true;
         }

         return false;
      }
};
int main() {
   Distance D1(11, 10), D2(5, 11);

   if( D1 < D2 ) {
      cout << "D1 is less than D2 " << endl;
   } else {
      cout << "D2 is less than D1 " << endl;
   }
   return 0;
```

```
/tmp/ccjzrEqyIU.o
D2 is less than D1
```

Rules for overloading operators

- Only existing operators can be overloaded. New operators cannot be created.
- The overloaded operator must have at least one operand that is of user-defined type.
- We cannot change the meaning of an operator. That is, we cannot redefine the plus (+) operator to subtract one value from other.

Coercion Polymorphism (Casting) or Type conversion

Coercion happens when an object or a primitive is cast into another object type or primitive type.

For example,

float b = 6; // int gets promoted (cast) to float implicitly

int a =9.99 // float gets demoted to int implicitly

Three types of situations might arise for data conversion between different types:

- Conversion form basic type to class type.
- Conversion from class type to basic type.
- Conversion from one class type to another class type.

Conversion form basic type to class type

In this type of conversion, the source type is basic type and the destination type is class type. Means basic data type is converted into the class type.

class_object = basic_type_variable;

- For example we have class *employee* and one object of employee *'emp'* and suppose we want to assign the employee code of employee *'emp'* by any integer variable say *'Ecode'* then the statement below is the example of the conversion from basic to class type.

emp = Ecode ;

Here the assignment will be done by converting *"Ecode"* which is of basic or primary data type into the class type

The conversion from basic type to the class type can be performed by two ways:

1. **Using constructor**

/* Program to convert basic type to class type using constructor */

```cpp
#include <iostream>
using namespace std;
class Time
{
    int hrs,min;
    public:
            Time(int);
        void display();
};

Time :: Time(int t)
{
    cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
    hrs=t/60;
    min=t%60;
}

void Time::display()
{
    cout<<hrs<< ": Hours(s)" <<endl;
    cout<<min<< " Minutes" <<endl;
}

int main()
{

    int duration;
    cout<<"Enter time duration in minutes";
    cin>>duration;

    Time t1=duration;

    t1.display();

}
```

```
/tmp/ccjzrEqyIU.o
Enter time duration in minutes67
Basic Type to ==> Class Type Conversion...
1: Hours(s)
7 Minutes
```

Here, we have created an object *"t1"* of class *"Time"* and during the creation we have assigned integer variable *"duration"*. It will pass time duration to the constructor function and assign to the *"hrs"* and *"min"* members of the class *"Time"*

2. Using Operator Overloading

/* Program to convert from basic type to class type using operator overloading */

```
#include <iostream>
using namespace std;
class Time
{
    int hrs,min;
    public:
        void display();
        void operator=(int); // overloading function
};
void Time::display()
{
    cout<<hrs<< ": Hour(s) "<<endl ;
    cout<<min<<": Minutes"<<endl ;
}

void Time::operator=(int t)
{
    cout<<"Basic Type to ==> Class Type Conversion..."<<endl;
    hrs=t/60;
    min=t%60;
}

int main()
{
    Time t1;
    int duration;
    cout<<"Enter time duration in minutes";
        cin>>duration;
    cout<<"object t1 overloaded assignment..."<<endl;
    t1=duration;
    t1.display();
    cout<<"object t1 assignment operator 2nd method..."<<endl;
    t1.operator=(duration);
    t1.display();
}
```

```
/tmp/ccjzrEqyIU.o
Enter time duration in minutes67
object t1 overloaded assignment...
Basic Type to ==> Class Type Conversion...
1: Hour(s)
7: Minutes
object t1 assignment operator 2nd method...
Basic Type to ==> Class Type Conversion...
1: Hour(s)
7: Minutes
```

Conversion from class type to basic type

In this type of conversion, the source type is class type and the destination type is basic type. Means class data type is converted into the basic type.

basic_type_variable = class_object;

For example, we have class *Time* and one object of *Time* class *'t'* and suppose we want to assign the total time of object *'t'* to any integer variable say *'duration'* then the statement below is the example of the conversion from class to basic type.

duration= t ; // where, t is object and duration is of basic data type

Example:

/* Program to demonstrate Class type to Basic type conversion. */

```cpp
#include <iostream>
using namespace std;
class Time
{
    int hrs,min;
    public:
        Time(int ,int);   // constructor
        operator int();   // casting operator function
        ~Time()          // destructor
        {
            cout<<"Destructor called..."<<endl;
        }
};
Time::Time(int a,int b)
{
    cout<<"Constructor called with two parameters..."<<endl;
    hrs=a;
    min=b;
}
Time :: operator int()
{
    cout<<"Class Type to Basic Type Conversion..."<<endl;
    return(hrs*60+min);
}
int main()
{
    int h,m,duration;
    cout<<"Enter Hours ";
        cin>>h;
    cout<<"Enter Minutes ";
        cin>>m;
    Time t(h,m);       // construct object
    duration = t;      // casting conversion OR duration = (int)t
    cout<<"Total Minutes are "<<duration;
    cout<<" 2nd method operator overloading "<<endl;
    duration = t.operator int();
    cout<<"Total Minutes are "<<duration;
}
```

```
/tmp/ccjzrEqyIU.o
Enter Hours 1
Enter Minutes 7
Constructor called with two parameters...
Class Type to Basic Type Conversion...
Total Minutes are 67 2nd method operator overloading
Class Type to Basic Type Conversion...
Total Minutes are 67Destructor called...
```

Conversion from one class type to another class type

In this type of conversion both the type that is source type and the destination type are of class type. Means the source type is of class type and the destination type is also of the class type. In other words, one class data type is converted into another class type.

Object_of_destination_class = Object_of_source_type

For example we have two classes one for "*computer"* and another for "*mobile".* Suppose if we wish to assign *"price"* of *computer* to *mobile* then it can be achieved by the statement below which is the example of the conversion from one class to another class type.

```
mob = comp ;   // where mob and comp are the objects of mobile and computer classes respectively.
```

There are two ways to convert from one class type to another class type conversion

I) **By using constructors**

Here source class should have constructor. And destination class have a conversion routine. Conversion routine (constructor) would have source class object as an argument.

Example: //conversion from polar to rectangular coordinate

```cpp
#include <iostream>
#include <math.h>
using namespace std;
class polar
{
    public:
        float r,th;
    polar(){}
    polar(int a,int b)
    {
        r=a;
        th=b;
    }
    void show()
    {
        cout<<"In polar form:\nr="<<r<<" and theta="<<th;

    }
};
class rectangular
{
    float x,y;
    public:
    rectangular(){}
    rectangular(polar p)
    {
        x=p.r*cos(p.th);
        y=p.r*sin(p.th);
    }
    void show()
    {
    cout<<"\nIn Rectangular form:\nx="<<x<<"and y="<<y;
    }
 };
int main()
{
    polar p(5.5,3.14/2);
    p.show();
    rectangular r;
    r=p;
    r.show();
}

```

```
/tmp/ccjzrEqyIU.o
In polar form:
r=5 and theta=1
In Rectangular form:
x=2.70151and y=4.20735
```

II) **by using casting operator (i.e. operator function)**

// conversion from polar to rectangular coordinate

```
#include <iostream>
#include <math.h>
using namespace std;
const double pi=3.141592654;
class rectangular
{
    double x,y;
    public:
        rectangular()
        {
            x=0;
            y=0;
        }
        rectangular(double a,double b)
        {
            x=a;
            y=b;
        }
        void display()
        {
            cout<<"("<<x<<","<<y<<")";
        }
};
class polar
{
    double theta,r;
    public:
    polar(double theta,double r)
    {
        theta=0;
        r=0;
    }
    operator rectangular()
    {
        double x,y;
        //float atheta=theta*pi/180;
        x=r*cos(theta);
        y=r*sin(theta);
        return rectangular(x,y);
    }
    void display()
    {
        cout<<"\nr="<<r;
        cout<<"\ntheta="<<theta;
    }
};
int main()
{
    rectangular r1,r2;
    polar p1(5,1);
    r1=p1;// polar to rectangular conversion
    r1.display();
}
```

Example:

```
/* Program to convert class Time to another class Minute.

#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
class Time
{
     int hrs,min;
     public:
                Time(int h,int m)
                {
                        hrs=h;
                        min=m;
                }
                Time()
                {
                        cout<<"\n Time's Object Created";
                }
                int getMinutes()
                {
                        int tot_min = ( hrs * 60 ) + min ;
                  return tot_min;
                }
                void display()
                {
                        cout<<"Hours: "<<hrs<<endl ;
                  cout<<" Minutes : "<<min <<endl ;
                }
};
class Minute
{
     int min;
     public:

       Minute()
       {
         min = 0;
       }
     void operator=(Time T)
     {
                min=T.getMinutes();
     }
     void display()
     {
                cout<<"\n Total Minutes : " <<min<<endl;
```

```
    }
};
void main()
{
    clrscr();
    Time t1(2,30);
     t1.display();
    Minute m1;
     m1.display();

    m1 = t1;    // conversion from Time to Minute

    t1.display();
    m1.display();
    getch();
}
```

Chapter 5
Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class:

The class that inherits properties from another class is called Sub class or Derived Class.

Super Class:

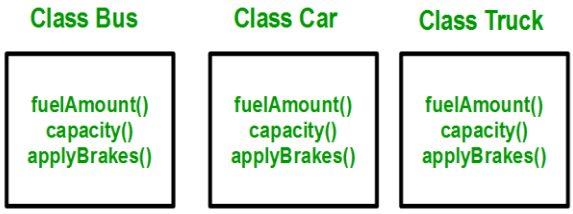
The class whose properties are inherited by sub class is called Base Class or Super class.

The derived class inherits some or all the traits from base class. The base class is unchanged by this. Most important advantage of inheritance is re-usability. Once a base class is written and debugged, it need not be touched again and we can use this class for deriving another class if we need. Reusing existing code saves time and money. By re-usability a programmer can use a class created by another person or company and without modifying it derive other class from it.

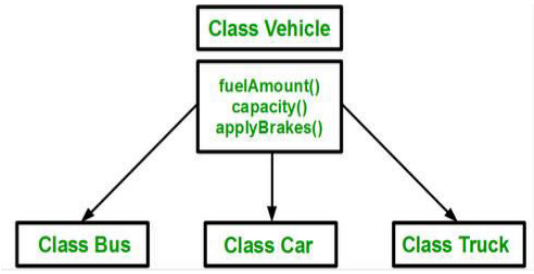
Why and when to use inheritance?

Consider a group of vehicles.

We need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below:



We can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability.



Syntax:

```
class subclass_name : access_mode base_class_name
{
        //body of subclass
};
```

Advantages of Inheritance

1. Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
2. Reusability enhanced reliability. The base class code will be already tested and debugged.
3. As the existing code is reused, it leads to less development and maintenance costs.

4. Inheritance helps to reduce code redundancy and supports code extensibility.
5. Inheritance facilitates creation of class libraries

Disadvantages of Inheritance

1. Inherited functions work slower than normal function as there is indirection.
2. Improper use of inheritance may lead to wrong solutions.
3. Often, data members in the base class are left unused which may lead to memory wastage.
4. Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes

Example of Inheritance

```cpp
// C++ program to demonstrate implementation of Inheritance
#include <iostream>
using namespace std;
//Base class
class Parent
{
    public:
    int id_p=9;
};
// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
    int id_c=8;
};
//main function
int main()
{
        Child obj1;
        cout << "Child id is " << obj1.id_c << endl;
        cout << "Parent id is " << obj1.id_p << endl;
        return 0;
}

```

```
/tmp/ccjzrEqyIU.o
Child id is 8
Parent id is 9
```

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

Public mode:

If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

Protected mode:

If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode:

If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

	Derived Class	Derived Class	Derived Class
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object
class A
{
public:
   int x;
protected:
   int y;
private:
   int z;
};

class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};

class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

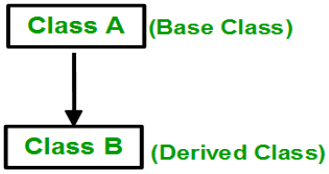
Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

a. Single Inheritance
b. Multiple Inheritance
c. Hierarchical Inheritance
d. Multilevel Inheritance
e. Hybrid Inheritance (also known as Virtual Inheritance)

1. Single Inheritance:

In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

```
class subclass_name : access_mode base_class
{
        //body of subclass
};
```

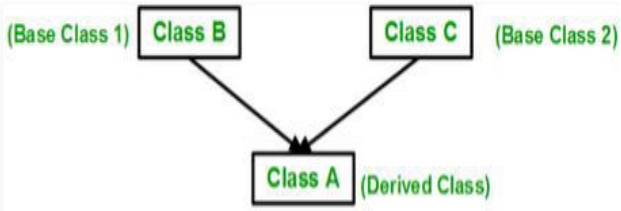
Example:

A class **Room** consists of two fields length and breadth and method int area() to find the area of room. A new class BedRoom is derived from class Room and consist of additional field height and two methods setData (int,int,int) to set the value for three fields and int volume() to find the volume. Now write the c++ program to input the length ,breadth and height and find the area and volume.

```
#include<iostream>
using namespace std;
class Room
{
    protected:
        float length, breadth;
    public:
        int area()
        {
            return(length*breadth);
        }
};
class BedRoom : public Room
{
    private:
    float height;
    public:
        void setData(int l,int b, int h)
        {
            length=l;
            breadth=b;
            height=h;
        }

        int volume()
        {
            return(length * breadth * height);
        }
};
int main()
{
    BedRoom b;
    b.setData(3,4,5);
    cout<<"Area of bedroom= "<<b.area()<<endl;
    cout<<"Volume of bedroom="<<b.volume();
}
```

```
/tmp/ccjzrEqyIU.o
Area of bedroom= 12
Volume of bedroom=60
```

2. Multiple Inheritance:
Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
        //body of subclass
};
```

Example:
Q. Create two classes class1 and class2 each having data member for storing a number, a method to initialize it. Create a new class class3 that is derived from both class class1 and class2 and consisting of a method that displays the sum of two numbers from class1 and class2.

```
#include<iostream>
using namespace std;
    class class1
    {
        protected:
                    int n;
            public:
                 void getn(int p)
                       {
                        n=p;
                       }
    };
    class class2
    {
            protected:
                    int m;
            public:
                 void getm(int q)
                  {
                        m=q;
                     }
    };
    class class3: public class1, public class2
    {
     public:
            void displaytotal()
            {
                 int tot;
                 tot=n+m;
                    cout<<"Sum ="<<tot;
         }
    };
    int main()
    {
        class3 a;
        a.getm(4);
        a.getn(5);
        a.displaytotal();
}
```

```
/tmp/ccjzrEqyIU.o
Sum =9
```

Ambiguity/Problem with multiple inheritance

Sometime we have to face an ambiguity problem in using multiple inheritance when a function with the same name appears in more than one base class. Consider the following example

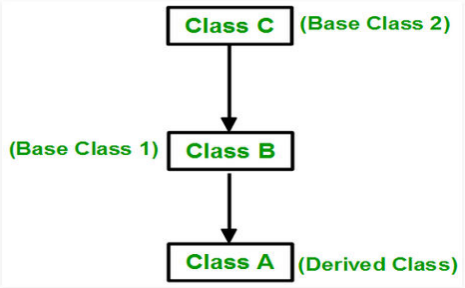
```
#include<iostream>
    using namespace std;
    class class1
    {
            protected:
                 int a;
            public:
             void get(int x)
                {
                   a=x;
                }

    };
    class class2
    {
          protected:
                  int b;
          public:
             void get(int x)
             {
                   b=x;
                }
    };
    class class3: public class1, public class2
    {
     public:
            void displaytotal()
         {
               cout<<"Total= "<<(a+b);
            }
    };
    int main()
    {
            class3 a;
         //a.get();// the request for get() is ambiguous as get() is
                defined in both class1 and class2
         a.class1::get(5);
            a.class2::get(6);
            a.displaytotal();
    }
```

```
/tmp/ccjzrEqyIU.o
Total= 11
```

So we see that, When compilers of programming languages that support this type of multiple inheritance encounter, super classes that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. We can solve this problem by defining a named instance within the derived class using the class resolution operator with the function as shown in above example.

3. Multilevel inheritance

In multilevel inheritance, a class is derived from another subclass.



The general form is

```
class A
{
//member of A
}
class B :public/private/protected A
{
        //own member of B
}
class C :public/private/protected B
{
//own member of C
}
```

In above case, class B is derived from Class A while class C is derived from derived class B. So, the class A serves as base class for B and B serves as base class for C therefore B is also called as **intermediate class because** it provides a link between class A and C.

Example:

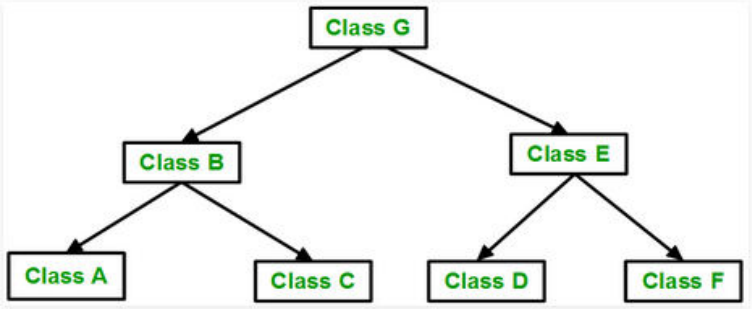
A class Student consists of field roll, a method to assigns roll number. A new class Test is derived from class Student and consists of two new fields sub1 and sub2, a method to initialize these fields with obtained mark. Again, a new class Result is derived from Test and consists of a field total and a method to display entire details along with total obtained marks. WAP to input roll number, marks in two different subject and display total.

```cpp
#include<iostream>
using namespace std;
class Student
{
    protected:
        int roll;
    public:
        void setroll(int r)
        {
                roll=r;
        }
};
class Test: public Student
{
    protected:
    float sub1, sub2;
    public:
        void setmark(float m1, float m2)
        {
            sub1=m1;
            sub2=m2;
        }
};
class Result : public Test
{
    private:
        float total;
    public:
        void display()
        {
            total=sub1+sub2;
            cout<<"Roll number= "<<roll<<endl;
            cout<<"Mark in first subject= "<<sub1<<endl;
            cout<<"Mark in second subject= "<<sub2<<endl;
            cout<<"Total= "<<total;
        }
};
int main()
{
    int r;
    float s1,s2;
    cout<<"Enter roll number"<<endl;
    cin>>r;
    cout<<"Enter marks in two subject"<<endl;
    cin>>s1>>s2;
    Result res;
    res.setroll(r);
    res.setmark(s1,s2);
    res.display();
}
```

```
/tmp/ccjzrEqyIU.o
Enter roll number
12
Enter marks in two subject
45
67
Roll number= 12
Mark in first subject= 45
Mark in second subject= 67
Total= 112
```

4. Hierarchical Inheritance

In hierarchical inheritance, two or more classes inherits the properties of one existing class.



The general form is:

```
class A
{
        //member of A
}
class B : public/private/protected A
{
        //own member of B
}
class C : pubic / private / protected A
{
        //own member of C
}
```

In above example, A is base class which includes all the features that are common to sub classes. Class B and Class C are the sub classes that shares the common property of class A and also can define their own property. Note we also can define a subclass that serve as base class for the lower level classes and so on.

Example:

A company needs to keep record of its following employees:

i) Manager ii) Supervisor

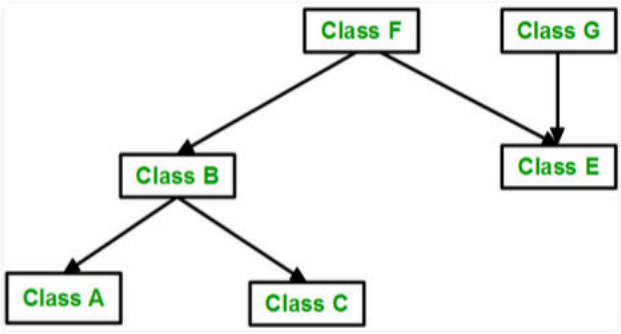
The record requires name and salary of both employees. In addition, it also requires section_name (i.e. name of section, example Accounts, Marketing, etc.) for the Manager and group_id (Group identification number, e.g. 205, 112, etc.) for the Supervisor. Design classes for the above requirement. Each of the classes should have a function called set() to assign data to the fields and a function called get() to return the value of the fields. Write a main program to test your classes. What form of inheritance will the classes hold in this case?

```cpp
#include<string.h>
using namespace std;
class Employee
{
    private:
        char name[30];
        float salary;
    public:
        void setName(char *n)
        {
            strcpy(name,n);
        }
        void setSalary(float s)
        {
            salary=s;
        }
        char * getName()
        {
            return name;
        }
        float getSalary()
        {
            return salary;
        }
};
class Manager: public Employee
{
    private:
        char section_name[50];
    public:
        void setSection_name(char *sn)
        {
            strcpy(section_name,sn);
        }
        char * getSection_name()
        {
            return section_name;
        }
};
class Supervisor: public Employee
{
    private:
        int group_id;
    public:
        void setGroup_id(int gid)
        {
            group_id=gid;
        }
        int getGroup_id()
        {
            return group_id;
        }
};
int main()
{
    Manager m;
    m.setName("Destiny");
    m.setSalary(50000);
    m.setSection_name("Accounts");
    cout<<"Name= "<<m.getName()<<endl;
    cout<<"Salary= "<<m.getSalary()<<endl;
    cout<<"Section= "<<m.getSection_name()<<endl;
    Supervisor s;
    s.setName("Sagar Kunwar");
    s.setSalary(40000);
    s.setGroup_id(5);
    cout<<"Name= "<<s.getName()<<endl;
    cout<<"Salary= "<<s.getSalary()<<endl;
    cout<<"Group ID= "<<s.getGroup_id()<<endl;
}
```

```
Name= Destiny
Salary= 50000
Section= Accounts
Name= Sagar Kunwar
Salary= 40000
Group ID= 5
```

5. Hybrid inheritance

In hybrid inheritance , it can be the combination of single and multiple inheritance or any other combination.



One form can be as below

```
Class X
{
        //member of X
}
Class A : public/private/protected X
{
        // own member of A
}
Class B: public/private/protected A
{
        //own member of C
}
Class C: public/private/protected A
{
        //own member of C
}
Class D: public/private/protected A
{
        //own member of D
}
```

Above example consists both single and hierarchical inheritance hence called hybrid inheritance

Example:

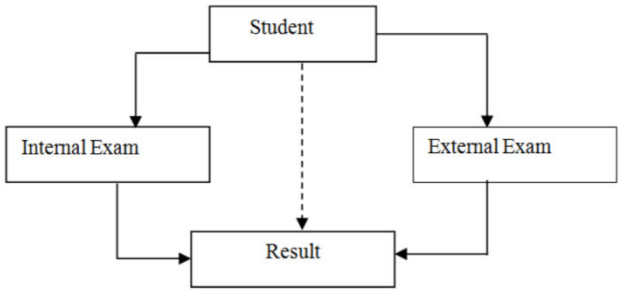
The below example shows the hybrid inheritance i.e. combination of multilevel and multiple inheritance

Q. Create a class Student with data member roll_no and two functions to initialize and display it. Derive a new class Test which has two methods to assign and display marks in two subjects. Create a new class Sport with two functions that assign and display the score in sports. Now create another class Result that is derived from both class Test and Sport, having a function that displays the total of marks and score. Write a main program to test your class.

```cpp
#include<iostream>
using namespace std;
class Student
{
    private:
        int roll;
    public:
        void setroll()
        {
            cout<<"Enter roll number"<<endl;
            cin>>roll;
        }
        void showroll()
        {
            cout<<"Roll= "<<roll<<endl;
        }
};
class Test: public Student
{
    protected:
        float com,eng;
    public:
        void setmark()
        {
            cout<<"Enter marks of computer and English "<<endl;
            cin>>com>>eng;
        }
        void showmark()
        {
            cout<<"Computer= "<<com<<endl;
            cout<<"English= "<<eng<<endl;
        }
};
class Sport
{
        protected:
        float score;
    public:
        void setscore()
        {
            cout<<"Enter score in sports "<<endl;
            cin>>score;
        }
        void showscore()
        {
            cout<<"Score in sports= "<<score<<endl;
        }
};
class Result: public Test, public Sport
{
    private:
    float tot;
public:
        void showtotal()
        {
            tot=com+eng+score;
            cout<<"Total obtained marks= "<<tot<<endl;
        }
};
int main()
{
    Result res;
    res.setroll();
    res.setmark();
    res.setscore();
    res.showroll();
    res.showmark();
    res.showscore();
    res.showtotal();
}
```

Multi Path Inheritance

When a class is derived from two or more classes, those are derived from the same base class. Such a type of inheritance is known as multipath inheritance. The multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in Figure below.



Here, all three kinds of inheritance exist i.e. multilevel, multiple and hierarchical. The class Result has two direct base class Internal Exam and External exam which themselves have a common base class Student. So, the class Result inherits the class Student via two separate paths called as multi path inheritance. The class student is sometime called as indirect base class. It also can be inherited directly as shown by the broken line.

Virtual Base Class

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

Problem with multipath inheritance and virtual base class

Since all the public and protected members of indirect base class i.e. Student are inherited into final derived class i.e. Result via two paths, first via Internal Exam and second via External exam. This means the class Result would have duplicate sets of members inherited from Student. This introduces ambiguity and should be avoided.

This ambiguity can be avoided by making a common base class as virtual base class while declaring the direct or intermediate base classes as shown below.

```
class Student //grand parent
{
};
Class Internal_Exam: virtual public Student //parent1
{
};
Class  External_Exam: virtual public A//parent2
{
}
Class Result: public Internal_Exam, public External_Exam//child
```

```
{
  //only one copy of A will be inherited
}
```

So, when a class is made virtual base class, c++ takes care to see that only one copy of that class is inherited, regardless of how many paths exist between the virtual base class and a derived class.

Note: the use of keyword virtual and public can be done in any order

Programming example:

Create a class Student with data member roll_no and two functions to initialize and display it. Derive two new classes Theory and Practical from Student. Define suitable functions to assign and display theory and practical marks for two different subjects. Again, derive a new class Result from both class Theory and Practical and add a new function to display the final total marks of student. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Student
{
        private:
                int roll;
        public:
                void setroll()
                {
                        cout<<"Enter roll number"<<endl;
                        cin>>roll;
                }
                void showroll()
                {
                        cout<<"Roll= "<<roll<<endl;
                }
};
class Theory: public virtual Student
{
        protected:
                float comth,ength;
        public:
                void setdatatheory()
                {
                        cout<<"Enter Theory marks of computer and English "<<endl;
                        cin>>comth>>ength;
                }
                void showmarkstheory()
                {
                        cout<<"Computer(Theory)= "<<comth<<endl;
                        cout<<"English(THeory)= "<<ength<<endl;
                }
};
class Practical: public virtual  Student
{
                protected:
                float compr,engpr;
```

```
        public:
                void setdatapractical()
                {
                        cout<<"Enter Practical marks of computer and English "<<endl;
                        cin>>compr>>engpr;
                }
                void showmarkspractical()
                {
                        cout<<"Computer(Practical)= "<<compr<<endl;
                        cout<<"English(Practical)= "<<engpr<<endl;
                }
};
class Result: public Theory, public Practical
{
        public:
                void showtotal()
                {
                        float tot;
                        tot=comth+ength+compr+compr;
                        cout<<"Total obtained marks= "<<tot<<endl;
                }
};

int main()
{
        Result res;
        res.setroll(); //ambiguous because multipath exist to reach setroll() from derived class  so must use virtual base class to overcome this
        res.setdatatheory();
        res.setdatapractical();
        res.showroll();
        res.showmarkstheory();
        res.showmarkspractical();
        res.showtotal();
}
```

```
Enter roll number
5
Enter Theory marks of computer and English
70
75
Enter Practical marks of computer and English
20
25
Roll= 5
Computer(Theory)= 70
English(THeory)= 75
Computer(Practical)= 20
English(Practical)= 25
Total obtained marks= 185
```

Constructor and Destructor in Inheritance

- Constructor is used to initialize variables and allocation of memory of object
- Destructor is used to destroy objects
- Compiler automatically calls constructer of base class and derived class automatically when derived class object is created.
- If we declare derived class object in inheritance, constructor of base class is executed first and then constructor of derived class
- If derived class object goes out of scope or deleted by programmer the derived class destructor is executed first and then base class destructor

Default Constructor (No arguments) in inheritance

- In this case it is not compulsory to have derived class constructor if base class have a constructor

Example Default Constructor in inheritance

main.cpp

```
#include <iostream>
using namespace std;
//Base Class
class A
{
    public:
    A() //Base Class A Constructor
    {
        cout<<"Constructor Class A"<<endl;
    }
    //Base Class A Destructor
    ~A()
    {
        cout<<"Destructor Class A"<<endl;
    }
};
class B:public A         //Derived class
{
    public:
    B() //Derived Class B Constructor
    {
        cout<<"Constructor Class B"<<endl;
    }
    ~B() //Derived Class B Destructor
    {
        cout<<"Destructor Class B"<<endl;
    }
};
int main()
{
    B obj; //Derived class object obj
    return 0;
}
```

```
Constructor Class A
Constructor Class B
Destructor Class B
Destructor Class A
```

Parameterized constructor (with arguments) in inheritance

- Derived class constructor is used to pass arguments to the base class.
- If derived class constructor is not available, it is not possible to pass arguments from derived class object to base class constructor

Example of parameterized constructor passing different argument for base class and derived class

main.cpp

```
#include <iostream>
using namespace std;
class A //Base class
{
    protected:
    int a;
    A(int x) //Base class constructor with one argument(x)
    {
        a=x;
        cout<<"Constructor : Class A : value : "<<a<<endl;
    }
};
class B: public A   //Derived class B from base class A
{
    protected:
    int b;
    public:
    //derived class constructor passing arguments(y, z)
    //y is used in derived class and z is passed base class A
    B(int y, int z) : A(z)
    {
        b=y;
        cout<<"Constructor : Class B value : "<<b<<endl;
    }
};
int main()
{
    //derived class object passing arguments to derived & base class
    B obj(5,3);
}
```

```
Constructor : Class A : value : 3
Constructor : Class B value : 5
```

IS-A and HAS-A Relation

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the implementation of inheritance (IS-A relationship), or object composition (HAS-A relationship).

IS-A Rule or Relationship:

In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing".

For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

HAS-A Rule (Container Class/Containership/Composition/Aggregation)

HAS rule means division into parts.

Consider the term “A car has an engine”, “A bus has an engine”. We cannot use engine the super class properties to all child classes because the engine has different for all transportation means. The engine is independent properties that cannot be directly called to the car, bus classes. We can include the class engine or class brake in class car not derive them

It is also called container class or containership because object of one class will be used inside other class which means member of one class is available in other class.

Example for Container Class/composition/has-a rule

main.cpp

```cpp
#include <iostream>
using namespace std;
class A
{
    int x;
    public:
        A()
        {
            x=1;
        }
        void dispaly1()
        {
            cout<<x<<endl;
        }
};
class B //Class B
{
    int y;
    A a; //Object of Class A declared in class B
    public:
    B()      //Constructor Class B
    {
        y=9;
    }
    void display2()
    {
        a.dispaly1(); //Calling member of Class A from member of B
        cout<<y;
    }
};
int main()
{
    B b; //object of class B
    b.display2();
}
```

```
1
9
```

In Above program there are 2 class A & B, Class A have a data member integer x and one default constructor and one-member function display1 (), Class B have 2 data member one is integer y and another is object (a) of class A and one constructor and one-member function i.e. display2() and inside diplay2 we call member function of class A i.e. display1().

Here class A's object is declared inside class B, so class B contains members of class A or class B HAS A object of class A (Class B is composite class).

Chapter 5
Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class:

The class that inherits properties from another class is called Sub class or Derived Class.

Super Class:

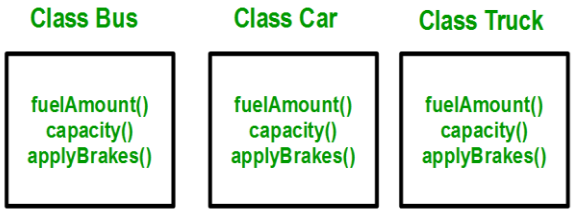
The class whose properties are inherited by sub class is called Base Class or Super class.

The derived class inherits some or all the traits from base class. The base class is unchanged by this. Most important advantage of inheritance is re-usability. Once a base class is written and debugged, it need not be touched again and we can use this class for deriving another class if we need. Reusing existing code saves time and money. By re-usability a programmer can use a class created by another person or company and without modifying it derive other class from it.

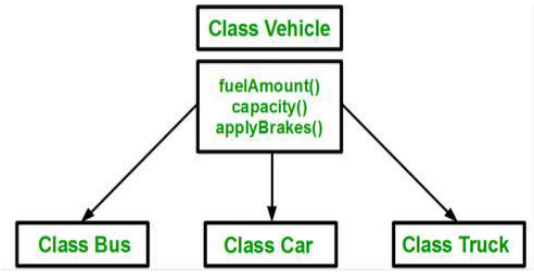
Why and when to use inheritance?

Consider a group of vehicles.

We need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below:



We can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability.



Syntax:

```
class subclass_name : access_mode base_class_name
{
        //body of subclass
};
```

Purpose of Inheritance

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

Advantages of Inheritance

1. Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inherited class.
2. Reusability enhanced reliability. The base class code will be already tested and debugged.
3. As the existing code is reused, it leads to less development and maintenance costs.
4. Inheritance makes the sub classes follow a standard interface.
5. Inheritance helps to reduce code redundancy and supports code extensibility.
6. Inheritance facilitates creation of class libraries

Disadvantages of Inheritance

1. Inherited functions work slower than normal function as there is indirection.
2. Improper use of inheritance may lead to wrong solutions.
3. Often, data members in the base class are left unused which may lead to memory wastage.
4. Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes

Example of Inheritance

```cpp
// C++ program to demonstrate implementation of Inheritance
#include <iostream>
using namespace std;
//Base class
class Parent
{
    public:
    int id_p=9;
};
// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
    int id_c=8;
};
//main function
int main()
{
        Child obj1;
        cout << "Child id is " << obj1.id_c << endl;
        cout << "Parent id is " << obj1.id_p << endl;
        return 0;
}

```

```
/tmp/ccjzrEqyIU.o
Child id is 8
Parent id is 9
```

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

Public mode:

If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

Protected mode:

If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode:

If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

	Derived Class	Derived Class	Derived Class
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Forms of Inheritance (Sub Class/Sub Types)

Inheritance is used in variety ways according to user's requirements. The Following are forms of inheritance.

- **Sub classing for specialization (subtype):** The derived child class is a specialized form of the parent class, in other words, the child class is subtype/subclass of the parent
- **Sub classing for specification:** The parent class defines behaviour that will be implemented in the child class. The inheritance for specification can be recognized when a parent class does not implement actual behaviour but it defines how the behaviour will be implemented in the child classes
- **Sub classing for construction:** The child class can be constructed form parent classes by implementing the behaviours of parent classes
- **Sub classing for generalization:** Sub classing for generalization is the opposite to sub classing for specifications. The base class holds the common properties that will inherit to the derived class.
- **Sub classing for extension:** The subclass for extension adds new functionality in child class form base class while designing new child class. It is done simply add new function other than parent class have.
- **Sub classing for limitation:** The subclass for limitation occurs when the behaviour of subclass is similar or more dependent to the behaviour of parent class.
- **Sub classing for variance:** The child class and parent class are varied when the level of inheritance increased, and the class and subclass relationship is imaginary.
- **Sub classing for combination:** The child class inherits features from more than one classes

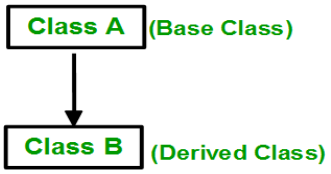
Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

a. Single Inheritance
b. Multiple Inheritance
c. Hierarchical Inheritance
d. Multilevel Inheritance
e. Hybrid Inheritance (also known as Virtual Inheritance)

1. Single Inheritance:

In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

```
class subclass_name : access_mode base_class
{
```

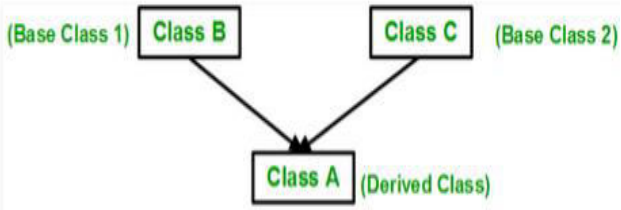
```
        //body of subclass
};
```

Example:
A class **Room** consists of two fields length and breadth and method int area() to find the area of room. A new class BedRoom is derived from class Room and consist of additional field height and two methods setData (int,int,int) to set the value for three fields and int volume() to find the volume. Now write the c++ program to input the length ,breadth and height and find the area and volume.

```
#include<iostream>
using namespace std;
class Room
{
    protected:
        float length, breadth;
    public:
        int area()
        {
            return(length*breadth);
        }
};
class BedRoom : public Room
{
    private:
    float height;
    public:
        void setData(int l,int b, int h)
        {
            length=l;
            breadth=b;
            height=h;
        }

        int volume()
        {
            return(length * breadth * height);
        }
};
int main()
{
    BedRoom b;
    b.setData(3,4,5);
    cout<<"Area of bedroom= "<<b.area()<<endl;
    cout<<"Volume of bedroom="<<b.volume();
}
```

```
/tmp/ccjzrEqyIU.o
Area of bedroom= 12
Volume of bedroom=60
```

2. Multiple Inheritance:
Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
        //body of subclass
};
```

Example:
Q. Create two classes class1 and class2 each having data member for storing a number, a method to initialize it. Create a new class class3 that is derived from both class class1 and class2 and consisting of a method that displays the sum of two numbers from class1 and class2.

```cpp
#include<iostream>
using namespace std;
    class class1
    {
        protected:
                    int n;
            public:
                 void getn(int p)
                        {
                         n=p;
                        }
    };
    class class2
    {
            protected:
                    int m;
            public:
                 void getm(int q)
                  {
                         m=q;
                      }
    };
    class class3: public class1, public class2
    {
     public:
            void displaytotal()
            {
                 int tot;
                 tot=n+m;
                    cout<<"Sum ="<<tot;
         }
    };
    int main()
    {
        class3 a;
        a.getm(4);
        a.getn(5);
        a.displaytotal();
}
```

```
/tmp/ccjzrEqyIU.o
Sum =9
```

Ambiguity/Problem with multiple inheritance

Sometime we have to face an ambiguity problem in using multiple inheritance when a function with the same name appears in more than one base class. Consider the following example

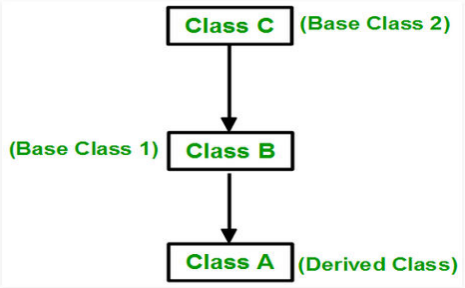
```cpp
#include<iostream>
    using namespace std;
    class class1
    {
            protected:
                 int a;
            public:
             void get(int x)
                {
                   a=x;
                }

    };
    class class2
    {
          protected:
                  int b;
          public:
             void get(int x)
             {
                   b=x;
                }
    };
    class class3: public class1, public class2
    {
     public:
            void displaytotal()
         {
               cout<<"Total= "<<(a+b);
            }
    };
    int main()
    {
            class3 a;
         //a.get();// the request for get() is ambiguous as get() is
                defined in both class1 and class2
         a.class1::get(5);
            a.class2::get(6);
            a.displaytotal();
    }
```

```
/tmp/ccjzrEqyIU.o
Total= 11
```

So we see that, When compilers of programming languages that support this type of multiple inheritance encounter, super classes that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. We can solve this problem by defining a named instance within the derived class using the class resolution operator with the function as shown in above example.

3. Multilevel inheritance

In multilevel inheritance, a class is derived from another subclass.



The general form is

```
class A
{
//member of A
}
class B :public/private/protected A
{
        //own member of B
}
class C :public/private/protected B
{
//own member of C
}
```

In above case, class B is derived from Class A while class C is derived from derived class B. So, the class A serves as base class for B and B serves as base class for C therefore B is also called as **intermediate class because** it provides a link between class A and C.

Example:

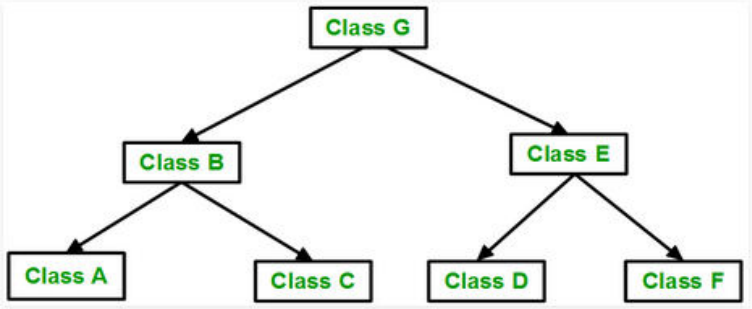
A class Student consists of field roll, a method to assigns roll number. A new class Test is derived from class Student and consists of two new fields sub1 and sub2, a method to initialize these fields with obtained mark. Again, a new class Result is derived from Test and consists of a field total and a method to display entire details along with total obtained marks. WAP to input roll number, marks in two different subject and display total.

```cpp
#include<iostream>
using namespace std;
class Student
{
    protected:
        int roll;
    public:
        void setroll(int r)
        {
                roll=r;
        }
};
class Test: public Student
{
    protected:
    float sub1, sub2;
    public:
        void setmark(float m1, float m2)
        {
            sub1=m1;
            sub2=m2;
        }
};
class Result : public Test
{
    private:
        float total;
    public:
        void display()
        {
            total=sub1+sub2;
            cout<<"Roll number= "<<roll<<endl;
            cout<<"Mark in first subject= "<<sub1<<endl;
            cout<<"Mark in second subject= "<<sub2<<endl;
            cout<<"Total= "<<total;
        }
};
int main()
{
    int r;
    float s1,s2;
    cout<<"Enter roll number"<<endl;
    cin>>r;
    cout<<"Enter marks in two subject"<<endl;
    cin>>s1>>s2;
    Result res;
    res.setroll(r);
    res.setmark(s1,s2);
    res.display();
}
```

```
/tmp/ccjzrEqyIU.o
Enter roll number
12
Enter marks in two subject
45
67
Roll number= 12
Mark in first subject= 45
Mark in second subject= 67
Total= 112
```

4. Hierarchical Inheritance

In hierarchical inheritance, two or more classes inherits the properties of one existing class.



In above example, A is base class which includes all the features that are common to sub classes. Class B and Class C are the sub classes that shares the common property of class A and also can define their own property. Note we also can define a subclass that serve as base class for the lower level classes and so on.

Example:

A company needs to keep record of its following employees:

i) Manager ii) Supervisor

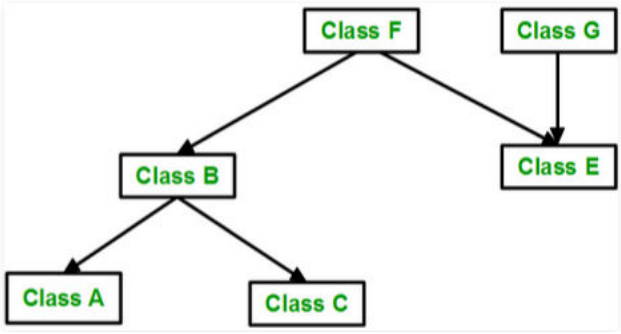
The record requires name and salary of both employees. In addition, it also requires section_name (i.e. name of section, example Accounts, Marketing, etc.) for the Manager and group_id (Group identification number, e.g. 205, 112, etc.) for the Supervisor. Design classes for the above requirement. Each of the classes should have a function called set() to assign data to the fields and a function called get() to return the value of the fields. Write a main program to test your classes. What form of inheritance will the classes hold in this case?

```cpp
#include<string.h>
using namespace std;
class Employee
{
    private:
        char name[30];
        float salary;
    public:
        void setName(char *n)
        {
            strcpy(name,n);
        }
        void setSalary(float s)
        {
            salary=s;
        }
        char * getName()
        {
            return name;
        }
        float getSalary()
        {
            return salary;
        }
};
class Manager: public Employee
{
    private:
        char section_name[50];
    public:
        void setSection_name(char *sn)
        {
            strcpy(section_name,sn);
        }
        char * getSection_name()
        {
            return section_name;
        }
};
class Supervisor: public Employee
{
    private:
        int group_id;
    public:
        void setGroup_id(int gid)
        {
            group_id=gid;
        }
        int getGroup_id()
        {
            return group_id;
        }
};
int main()
{
    Manager m;
    m.setName("Destiny");
    m.setSalary(50000);
    m.setSection_name("Accounts");
    cout<<"Name= "<<m.getName()<<endl;
    cout<<"Salary= "<<m.getSalary()<<endl;
    cout<<"Section= "<<m.getSection_name()<<endl;
    Supervisor s;
    s.setName("Sagar Kunwar");
    s.setSalary(40000);
    s.setGroup_id(5);
    cout<<"Name= "<<s.getName()<<endl;
    cout<<"Salary= "<<s.getSalary()<<endl;
    cout<<"Group ID= "<<s.getGroup_id()<<endl;
}
```

```
Name= Destiny
Salary= 50000
Section= Accounts
Name= Sagar Kunwar
Salary= 40000
Group ID= 5
```

5. Hybrid inheritance

In hybrid inheritance , it can be the combination of single and multiple inheritance or any other combination.



One form can be as below

```
Class X
{
        //member of X
}
Class A : public/private/protected X
{
        // own member of A
}
Class B: public/private/protected A
{
        //own member of C
}
Class C: public/private/protected A
{
        //own member of C
}
Class D: public/private/protected A
{
        //own member of D
}
```

Above example consists both single and hierarchical inheritance hence called hybrid inheritance

Example:

The below example shows the hybrid inheritance i.e. combination of multilevel and multiple inheritance

Q. Create a class Student with data member roll_no and two functions to initialize and display it. Derive a new class Test which has two methods to assign and display marks in two subjects. Create a new class Sport with two functions that assign and display the score in sports. Now create another class Result that is derived from both class Test and Sport, having a function that displays the total of marks and score. Write a main program to test your class.

```cpp
#include<iostream>
using namespace std;
class Student
{
    private:
        int roll;
    public:
        void setroll()
        {
            cout<<"Enter roll number"<<endl;
            cin>>roll;
        }
        void showroll()
        {
            cout<<"Roll= "<<roll<<endl;
        }
};
class Test: public Student
{
    protected:
        float com,eng;
    public:
        void setmark()
        {
            cout<<"Enter marks of computer and English "<<endl;
            cin>>com>>eng;
        }
        void showmark()
        {
            cout<<"Computer= "<<com<<endl;
            cout<<"English= "<<eng<<endl;
        }
};
class Sport
{
        protected:
        float score;
    public:
        void setscore()
        {
            cout<<"Enter score in sports "<<endl;
            cin>>score;
        }
        void showscore()
        {
            cout<<"Score in sports= "<<score<<endl;
        }
};
class Result: public Test, public Sport
{
    private:
    float tot;
public:
        void showtotal()
        {
            tot=com+eng+score;
            cout<<"Total obtained marks= "<<tot<<endl;
        }
};
int main()
{
    Result res;
    res.setroll();
    res.setmark();
    res.setscore();
    res.showroll();
    res.showmark();
    res.showscore();
    res.showtotal();
}
```

Programming Examples

1. Define a shape class (with necessary constructors and member functions) in Object Oriented Programming (abstract necessary attributes and their types). Write a complete code in C++ programming language.
    i. Derive triangle and rectangle classes from shape class adding necessary attributes.
    ii. Use these classes in main function and display the area of triangle and rectangle.

```
#include<iostream.h>
#include<conio.h>
class shape
{
        protected:
        float breadth, height, area;
        public:
        void getshapedata()
        {
                cout<<"Enter breadth:"<<endl;
                cin>>breadth;
                cout<<"Enter height:"<<endl;
                cin>>height;
        }
};

class triangle: public shape
{
        public:
        void calarea()
        {
                area=(breadth * height)/2;
        }
        void display()
        {
                cout<<"The area of triangle is"<<area<<endl;
        }
};
class rectangle: public shape
{
        public:
        void calarea()
        {
                area=breadth * height;
        }
        void display()
        {
                cout<<"Area of rectangle is"<<area<<endl;
        }
};

int main()
{
        triangle T;
        rectangle R;
        cout<<"Enter triangle data:"<<endl;
        T.getshapedata();
        cout<<"Enter rectangle data:"<<endl;
```

```
        R.getshapedata();
        T.calarea();
        R.calarea();
        T.display();
        R.display();
}
```

2. Define a *student* class (with necessary constructors and member functions) in Object Oriented Programming (abstract necessary attributes and their types). Write a complete code in C++ programming language.

i. Derive a C*omputer Science and Mathematics* class from *student* class adding necessary attributes (at least three subjects).

ii. Use these classes in a main function and display the average marks of computer science and mathematics students.

```
#include<iostream.h>
#include<conio.h>
class student
{
        protected:
                float english, sum, avg;
        public:
        void getstudentdata()
        {
                cout<<"Enter english marks:"<<endl;
                cin>>english;
        }
};


class computer : public student
{
        float IT, cprog, networks;
        public:
        void getcomputerdata()
        {
                cout<<"Enter marks in IT:"<<endl;
                cin>>IT;
                cout<<"Enter marks in cprog:"<<endl;
                cin>>Cprog;
                cout<<"Enter marks in networks:"<<endl;
                cin>>Networks;
        }
        void average()
        {
                sum=english+IT+cprog+networks;
                avg=sum/4;
                cout<<"Average marks is"<<avg;
        }
};
class mathematics : public student
{
        float calculus, stat, algebra;
        public:
        void getmathdata()
        {
```

```
                cout<<"Enter marks in calculus:"<<endl;
                cin>>calculus;
                cout<<"Enter marks in statistics:"<<endl;
                cin>>stat;
                cout<<"Enter marks in Linear Algebra:"<<endl;
                cin>>algebra;
        }
        void average()
        {
                sum=english+calculus+stat+algebra;
                avg=sum/4;
                cout<<"Average marks is"<<avg;
        }
};

int main()
{
        computer C;
        mathematics M;
        cout<<"Enter marks of computer students:"<<endl;
        C.getstudentdata();
        C.getcomputerdata();
        cout<<"Enter marks of mathematics student:"<<endl;
        M.getstudentdata();
        M.getmathdata();
        C.average();
        M.average();
}
```

3. Define a *Clock* class (with necessary constructor and member functions) in OOP (abstract necessary attributes and their types). Write a complete code in C++ programming language.

i. Derive *Wall_Clock* class from *Clock* class adding necessary attributes.

ii. Create two objects of *Wall_Clock* class with all initial state to 0 or NULL.

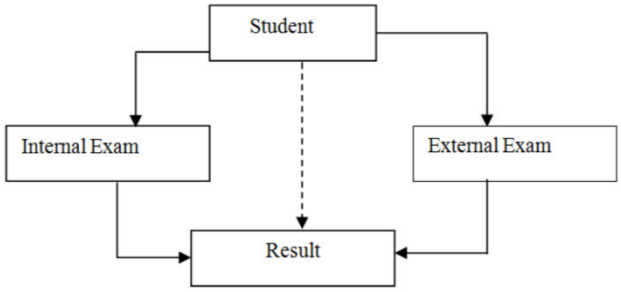
```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
class clock
{
protected:
        char model_no[10];
        float price;
        char manufacturer[50];
        public:
        void getclockdata()
        {
                cout<<"Enter clock manufacturer:"<<endl;
                cin>>manufacturer;
                cout<<"Enter model number:"<<endl;
                cin>>model_no;
```

```
                cout<<"Enter price:"<<endl;
                cin>>price;
        }


        void clockdisplay()
        {
                cout<<"Model number="<<model_no<<endl;
                cout<<"Manufacturer="<<manufacturer<<endl;
                cout<<"Price="<<price<<endl;
        }
};

class wall_clock: public clock
{
        int hr, min, sec;
        public:
        wall_clock()
        {
                strcpy(model_no,NULL);
                strcpy(manufacturer,NULL);
                price=0.0;
                hr=0;
                min=0;
                sec=0;
        }
        void getwallclockdata()
        {
                 cout<<"Enter hour, minute and seconds:"<<endl;
                 cin>>hr>>min>>sec;
        }
        void wallclockdisplay()
        {
                cout<<"Time="<<hr<<":"<<min<<":"<<sec<<endl;
        }
};

int main()
{
        wall_clock W1, W2;
        cout<<"Enter data for W1:"<<endl;
        W1.getclockdata();
        W1.getwallclockdata();
        cout<<"Value of W1:"<<endl;
        W1.clockdisplay();
        W1.wallclockdisplay();
```

```
        cout<<"Enter data for W2:"<<endl;
        W2.getclockdata();
        W2.getwallclockdata();
        cout<<"Value of W2:"<<endl;
        W2.clockdisplay();
        W2.wallclockdisplay();
}
```

Multi Path Inheritance

When a class is derived from two or more classes, those are derived from the same base class. Such a type of inheritance is known as multipath inheritance. The multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in Figure below.



Here, all three kinds of inheritance exist i.e. multilevel, multiple and hierarchical. The class Result has two direct base class Internal Exam and External exam which themselves have a common base class Student. So, the class Result inherits the class Student via two separate path called as multi path inheritance. The class student is sometime called as indirect base class. It also can be inherited directly as shown by the broken line.

Virtual Base Class

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

Problem with multipath inheritance and virtual base class

Since all the public and protected members of indirect base class i.e. Student are inherited into final derived class i.e. Result via two paths, first via Internal Exam and second via External exam. This means the class Result would have duplicate sets of members inherited from Student. This introduces ambiguity and should be avoided.

This ambiguity can be avoided by making a common base class as virtual base class while declaring the direct or intermediate base classes as shown below.

```
class Student //grand parent
{
};
Class Internal_Exam: virtual public Student //parent1
```

```
{
};
Class  External_Exam: virtual public A//parent2
{
}
Class Result: public Internal_Exam, public External_Exam//child
{
  //only one copy of A will be inherited
}
```

So, when a class is made virtual base class, c++ takes care to see that only one copy of that class is inherited, regardless of how many paths exist between the virtual base class and a derived class.

Note: the use of keyword virtual and public can be done in any order

Programming example:

Create a class Student with data member roll_no and two functions to initialize and display it. Derive two new classes Theory and Practical from Student. Define suitable functions to assign and display theory and practical marks for two different subjects. Again, derive a new class Result from both class Theory and Practical and add a new function to display the final total marks of student. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Student
{
        private:
                int roll;
        public:
                void setroll()
                {
                        cout<<"Enter roll number"<<endl;
                        cin>>roll;
                }
                void showroll()
                {
                        cout<<"Roll= "<<roll<<endl;
                }
};
class Theory: public virtual Student
{
        protected:
                float comth,ength;
        public:
                void setdatatheory()
                {
                        cout<<"Enter Theory marks of computer and English "<<endl;
                        cin>>comth>>ength;
                }
                void showmarkstheory()
```

```
		{
			cout<<"Computer(Theory)= "<<comth<<endl;
			cout<<"English(THeory)= "<<ength<<endl;
		}
};
class Practical: public virtual  Student
{
		protected:
		float compr,engpr;
	public:
		void setdatapractical()
		{
			cout<<"Enter Practical marks of computer and English "<<endl;
			cin>>compr>>engpr;
		}
		void showmarkspractical()
		{
			cout<<"Computer(Practical)= "<<compr<<endl;
			cout<<"English(Practical)= "<<engpr<<endl;
		}
};
class Result: public Theory, public Practical
{
	public:
		void showtotal()
		{
			float tot;
			tot=comth+ength+compr+compr;
			cout<<"Total obtained marks= "<<tot<<endl;
		}
};

int main()
{
	Result res;
	res.setroll(); //ambiguous because multipath exist to reach setroll() from derived class  so must use virtual base class to overcome this
	res.setdatatheory();
	res.setdatapractical();
	res.showroll();
	res.showmarkstheory();
	res.showmarkspractical();
	res.showtotal();
}
```

```
Enter roll number
5
Enter Theory marks of computer and English
70
75
Enter Practical marks of computer and English
20
25
Roll= 5
Computer(Theory)= 70
English(THeory)= 75
Computer(Practical)= 20
English(Practical)= 25
Total obtained marks= 185
```

Constructor and Destructor in Inheritance

- Constructor is used to initialize variables and allocation of memory of object
- Destructor is used to destroy objects
- Compiler automatically calls constructer of base class and derived class automatically when derived class object is created.
- If we declare derived class object in inheritance constructor of base class is executed first and then constructor of derived class
- If derived class object goes out of scope or deleted by programmer the derived class destructor is executed first and then base class destructor

Default Constructor (No arguments) in inheritance

- In this case it is not compulsory to have derived class constructor if base class have a constructor

Example Default Constructor in inheritance

```
class A { //Base Class
public:
A() {                                   //Base Class A Constructor
cout<<"Constructor Class A"<<endl;
}
~A()//Base Class A Destructor
 {
cout<<"Destructor Class A"<<endl;
}
};
```

```
class B:public A                //Derived class
{
public:
B()                                     //Derived Class B Constructor
{
cout<<"Constructor Class B"<<endl;
}
~B() {                                  //Derived Class B Destructor
cout<<"Destructor Class B"<<endl;
}
};
int main()
{
B obj;                          //Derived class object obj
}
```

In Above Program Class A is base class with one constructor and destructor, Class B is derived from class A having a constructor and destructor. In main() Object of derived class i.e. class B is declared. When class B's object is declared constructor of base class is executed followed by derived class constructor. At end of program destructor of derived classis executed first followed by base class destructor.

Output:

```
Constructor Class A
Constructor Class B
Destructor Class B
Destructor Class A
```

Parameterized constructor (with arguments) in inheritance

- In parameterized constructor it is compulsory to have derived constructor if there base class constructor.
- Derived class constructor is used to pass arguments to the base class.

- If derived class constructor is not available, it is not possible to pass arguments from derived class object to base class constructor

Example of parameterized constructor passing different argument for base class and derived class

```
class A //Base class
{
protected:
int a;
A(int x) //Base class constructor with one argument(x)
{
a=x;
cout<<"Constructor : Class A : value : "<<a<<endl;
}
};
class B:public A //Derived class B from base class A
{
protected:
int b;
public:
//derived class constructor passing arguments(y, z)
//y is used in derived class and z is passed base class A
B(int y,int z):A(z)
{
b=y;
cout<<"Constructor : Class B value : "<<b<<endl;
}
};
int main()
```

```
{
//derived class object passing arguments to derived & base class
B obj(5,3);
}
```

Principal of Substitutability

Substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program.

Subtype

The term "subtype" is used to describe the relationship between different types that follow the principle of "substitution". If we consider two types (classes or interfaces) A and B, type B is called as a subtype of A, if the following two conditions are satisfied:

a. The instance (object) of type B can be legally assigned to the variable of type A.
b. The instance (object) of type B can be used by the variable of type A without any observable change in its behavior.

Subclass Vs Subtype

- ✓ To say that A is a subclass of B, declares that A is formed using inheritance.
- ✓ To say that A is a subtype of B, declares that A preserves the meaning of all the operations in B.

Object Composition

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc. This process of building complex objects from simpler ones is called **object composition**.

Broadly speaking, object composition models a "has-a" relationship between two objects. A car "has-a" transmission. Your computer "has-a" CPU. The complex object is sometimes called the whole, or the parent. The simpler object is often called the part, child, or component.

To qualify as a **composition**, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

A good real-life example of a composition is the relationship between a person's body and a heart. Let's examine these in more detail. Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person's body. The part in a composition can only be part of one object at a time. A heart that is part of one person's body cannot be part of someone else's body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed. But more broadly, it means the object manages the part's lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the

heart is created too. When a person's body is destroyed, their heart is destroyed too. Because of this, composition is sometimes called a "death relationship".

IS-A and HAS-A Relation

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the implementation of inheritance (IS-A relationship), or object composition (HAS-A relationship).

IS-A Rule or Relationship:

In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing".

For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

HAS-A Rule (Container Class/Containership/Composition/Aggregation)
)

HAS rule means division into parts.

Consider the term "A car has an engine", "A bus has an engine". We cannot use engine the super class properties to all child classes because the engine has different for all transportation means. The engine is independent properties that cannot be directly called to the car, bus classes. We can include the class engine or class brake in class car not derive them

It is also called container class or containership because object of one class will be used inside other class which means member of one class is available in other class.

Example for Container Class/composition/has-a rule

```
class A
{
    int x;
    public:
    A()
    {
            x=1;
    }
    void dispaly1()
    {
            cout<<x;
    }
};
class B //Class B
{
    int y;
```

```
    A a; //Object of Class A declared in class B
    public:
    B()     //Constructor Class B
    {
            y=9;
    }
    void display2()
    {
            a.dispaly1 (); //Calling member of Class A from member of B
        cout<<y;
    }
    };
int main()
{
    B b; //object of class B
    b.display2();
}
```

In Above program there are 2 class A & B, Class A have a data member integer x and one default constructor and one member function display1 (), Class B have 2 data member one is integer y and another is object (a) of class A and one constructor and one member function i.e. display2() and inside diplay2 we call member function of class A i.e.display1().

Here class A's object is declared inside class B, so class B contains members of class A or class B HAS A object of class A (Class B is composite class).

Composition Vs Inheritance

Composition	Inheritance
Composition indicates the operation of an existing structure	Inheritance is a super set of existing structure
Cannot reuse code directly but provide greater functionality	Inheritance can be directly reused the code and function provided by parent class
Code become shorter than inheritance	Code become longer than composition
Very easy to re-implement the behaviors and functions	Difficult to re-implement the behaviors
Example:	Example:

Chapter 6
Virtual Function and Polymorphism

Virtual Function

A virtual function a member function which is declared within base class and is re-defined (Overridden) by derived class. When we refer to a derived class object using a pointer or a reference to the base class, we 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 Run-time.

Rules for Virtual Functions

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

Example:

```cpp
#include<iostream>
using namespace std;
class base
{
    public:
    virtual void print ()
    {
        cout<< "print base class" <<endl;
    }
    void show ()
    {
        cout<< "show base class" <<endl;
    }
};
class derived:public base
{
    public:
    void print ()
    {
        cout<< "print derived class" <<endl;
    }
    void show ()
    {
        cout<< "show derived class" <<endl;
    }
};
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
    //virtual function, binded at runtime
    bptr->print();
    // Non-virtual function, binded at compile time
    bptr->show();
}
```

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

Abstract Classes

The objects created are often the instance of a derived class but not the base class. The base class is just the foundation for building new classes and hence such classes are called abstract base classes or abstract classes. An abstract class is one that has no instances and is not designed to create objects. It is only designed to be inherited.

```
class One
{
    private:
        ……….
    public:
        ……….
};
class Two
{
    public:
        ……….
};
void main( )
{
    Two t1;
}
```

Pure Virtual Functions (or deferred method or abstract method):

It is the special case of overriding. It's possible that we'd want to include a virtual function in a base class so that it may be redefined in a derived class, but there is no meaningful definition we could give for the function in the base class. It can be defined in the base class but not implemented. The child class provides its implementation.

```
class Shape
{
    protected:
    int width, height;
    public:
    Shape(int a=0,int b=0)
    {
        width = a;
        height = b;
    }
    //pure virtual function virtual
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

Example:

```
#include <iostream>
using namespace std;
class b
{
    public:
    virtual void show()=0;
};
class Derived: public b
{
    public:
    void show()
    {
        cout<< "inside derived class" ;
    }
};
int main()
{
    Derived d;
    b *p;
    p = &d;
    p ->show();
}
```

```
inside derived class
```

A class that contains pure virtual function is known as pure abstract class.

```cpp
#include<iostream>
using namespace std;
class Base
{
    public:
    virtual void show()=0 ;  // pure virtual function
};
class Derived1 : public Base
{
    public :
    void show()
    {
        cout <<" Derived1 \n";
    }
};
class Derived2 : public Base // derived class 2
{
    public:
    void show()
    {
        cout <<" Derived2 \n" ;
    }
};
int main()
{
    // Base baseobj ; // can't make object of abstract class
    Base * ptr[2] ;         // array of ptr of base class.
    Derived1 dv1 ;          // object of derived1
    Derived2 dv2 ;          //  object of derived2.
    ptr[0] = &dv1 ;
    ptr[1] = &dv2;
    ptr[0]->show();     ptr[1]->show();
}
```

```
Derived1
Derived2
```

The pure virtual function in the base class must be override in all its derived class from which we want to instantiate objects. If a class doesn't override pure virtual function, it itself becomes abstract and objects cannot be instantiated.

Virtual destructors

Since destructor are member functions, they can be made virtual with placing keyword virtual before it. The syntax is

Virtual ~ classname ( ) ; // virtual destructor.

The destructor in base class should always be virtual. If we use **delete** with a base class object to destroy the derived class object, then it calls the **delete** calls the member function destructor for base class. This causes the base class object to be destroyed.

Hence making destructor of base class virtual, we can prevent such mis-operation.

Example:

main.cpp

```
#include<iostream>
using namespace std;
class Base
{
    public:
    virtual ~Base()
    {
        cout<<"Base Destryed\n";
    }
};
class Derv1:public Base
{
    public:
    ~Derv1()
    {
    cout<<"Derived1 destroyed\n";
    }
};
int main()
{
  Base * pBase = new Derv1;
  delete pBase;
}
```

```
Derived1 destroyed
Base Destryed
```

- pBase stores address of object of Derv1 class.
- Delete pBase destroy the Base object i.e. calls the destructor of base class.
- If the destructor is made virtual by the line virtual ~Base ( ) ; then,

```
delete pBase ;
```

Simply calls the destructor of Derv class first

Virtual Base class

In multiple inheritance, if a base class parent derives its two-child class then another class is derived from two child, as

When member function of class D want to access data member of parent class A, then problem arises due to ambiguity.

To resolve such ambiguity, we use virtual base class.

A virtual base class is one from which classes are derived virtually. as

```
class A.
    {
        // body of class A
    };
class B: virtual public A
    {
        // Body of B
    };
class C:virtual public A
    {
        // Body of class C
    };
class D: public B; public C
    {
        //body of D
    };
```

Example:

```cpp
#include<iostream>
using namespace std;
class student
{
    protected:
        int roll;
    public:
        void getno(int a)
        {
            roll=a;
        }
        void putno()
        {
            cout<<"\nRollNumber is:"<<roll;
        }
};
class test:virtual public student
{
    protected:
        float part1,part2;
    public:
    void getmark(float a, float b)
    {
        part1=a;
        part2=b;
    }
    void putmark()
    {
        cout<<"\nPart1="<<part1;
        cout<<"\nPart2="<<part2;
    }
};
class sport:virtual public student
{
    protected:
        int score;
    public:
        void getscore(int a)
        {
            score=a;
        }
        void putscore()
        {
            cout<<"\nScore:"<<score;
        }
};
class result:public test, public sport
{
    float total;
    public:
    void display()
    {
        total=part1+part2+score;
        putno();
        putmark();
        putscore();
        cout<<"\nTotal Score:"<<total;
    }
};
int main()
{
    result student1;
    student1.getno(999);
    student1.getmark(25,54);
    student1.getscore(7);
    student1.display();
}
```

```
RollNumber is:999
Part1=25
Part2=54
Score:7
Total Score:86
```

this pointer

In C++, **this** pointer is used to represent the address of an object inside a member function. **For example**, consider an object *obj* calling one of its member function say *method()* as *obj.method()*. Then, **this** pointer will hold the address of object *obj* inside the member function *method()*. The **this** pointer acts as an implicit argument to all the member functions.

```
#include<iostream>
using namespace std;
class Demo
{
    private:
        int num;
        char ch;
    public:
        void setMyValues(int num, char ch)
        {
            this->num =num;
            this->ch=ch;
        }
        void displayMyValues()
        {
            cout<<num<<endl;
            cout<<ch;
        }
};
int main()
{
        Demo obj;
        obj.setMyValues(100, 'A');
        obj.displayMyValues();
        return 0;
}
```

```
100
A
```

Static data Members

When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. We can define class members static using static keyword.

Unlike normal member variables, static member variables are shared by all objects of the class.

```
class Something
{
    public:
    static int s_value;
};
int Something :: s_value = 1;
 int main()
{
    Something first;
    Something second;
    first.s_value = 2;
    cout << first.s_value <<'\n';
    cout << second.s_value <<'\n';
    return 0;

}
```

Static member function

Like static member variables, static member functions are not attached to any particular object. Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator. **Static member functions have no *this pointer.**

We can access a static member function with class name, by using following syntax:

```
class_name:: function_name(parameter);
```

```cpp
#include <iostream>
using namespace std;
class Demo
{
    private:
    //static data members
        static int X;
        static int Y;
    public:
        //static member function
        static void  Print()
        {
            cout<<"Value of X: "<< X <<endl;
            cout<<"Value of Y: "<< Y <<endl;
        }
};
//static data members initializations
int Demo :: X =10;
int Demo :: Y =20;
int main()
{
    Demo OB;
    //accessing class name with object name
    cout<<"Printing through object name:"<<endl;
    OB.Print();
    //accessing class name with class name
    cout<<"Printing through class name:"<<endl;
    Demo::Print();
    return 0;
}
```

```
Printing through object name:
Value of X: 10
Value of Y: 20
Printing through class name:
Value of X: 10
Value of Y: 20
```

Friend Function

One of the important concepts of is data hiding, i.e., a nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions.

This is done using a friend function or/and a friend class.

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.

The complier knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Syntax:

```
class class_name
{
        ... .. ...
        friend return_type function_name(arguments);
        ... .. ...
}
return_type functionName(argument/s)
{
        ... .. ...
        // Private and protected data of class_Name can be accessed from
        // this function because it is a friend function of className.
        ... .. ...
}
```

How can we make two classes friendly?

i. Common friend function on both classes (see above examples)

ii. Friend class technique

Like friend function, a class can also be a friend of another class. A friend class can access all the private and protected members of other class. In order to access the private and protected members of a class into friend class we must pass on object of a class to the member functions of friend class.

Example:

```cpp
#include <iostream>
using namespace std;
class Rectangle
    {
        int L,B;
        public:
        Rectangle()
        {
            L=10;
            B=20;
        }
        friend class Square;        //Statement 1
    };
class Square
    {
        int S;
        public:
        Square()
        {
            S=5;
        }
        void Display(Rectangle Rect)
        {
            cout<<"\n\n\tLength : "<<Rect.L;
            cout<<"\n\n\tBreadth : "<<Rect.B;
            cout<<"\n\n\tSide : "<<S;
        }
    };
int main()
{
        Rectangle R;
        Square S;
        S.Display(R);       //Statement 2
}
```

```
Length : 10

Breadth : 20

Side : 5
```

Q. WAP to find the sum of two number using the concept of friend function

```
#include<iostream>
using namespace std;
class Demo
{
    int fn,sn;
    public:
        void setdata()
        {
            cout<<"Enter two number"<<endl;
            cin>>fn>>sn;
        }
        friend void sum(Demo d);
};

void sum(Demo d)
{
    int sum;
    sum=d.fn+d.sn;
    cout<<"Sum= "<<sum;
}

int main()
{
    Demo a;
    a.setdata();
    sum(a);
}
```

```
Enter two number
12
34
Sum= 46
```

Q. Create classes called class1 and class2 with each of having one private member. Add member function to set a value (say setvalue) on each class. Add one more function max () that is friendly to both classes. max() function should compare two private member of two classes and show maximum among them. Create one-one object of each class then set a value on them. Display the maximum number among them.

```cpp
#include<iostream>
using namespace std;
class Class2;// forward declaration
class Class1
{
    int a;
    public:
        void setdata()
        {
            cout<<"Enter a number"<<endl;
            cin>>a;
        }
        friend void max(Class1,Class2);
};
class Class2
{
    int a;
    public:
        void setdata()
        {
            cout<<"Enter a number"<<endl;
            cin>>a;
        }
        friend void max(Class1,Class2);
};

void max(Class1 c1, Class2 c2)
{
    if(c1.a>c2.a)
    {
        cout<<"Larger= "<<c1.a;
    }
    else
    {
        cout<<"Larger= "<<c2.a;
    }
}
int main()
{
    Class1 x;
    Class2 y;
    x.setdata();
    y.setdata();
    max(x,y);
}
```

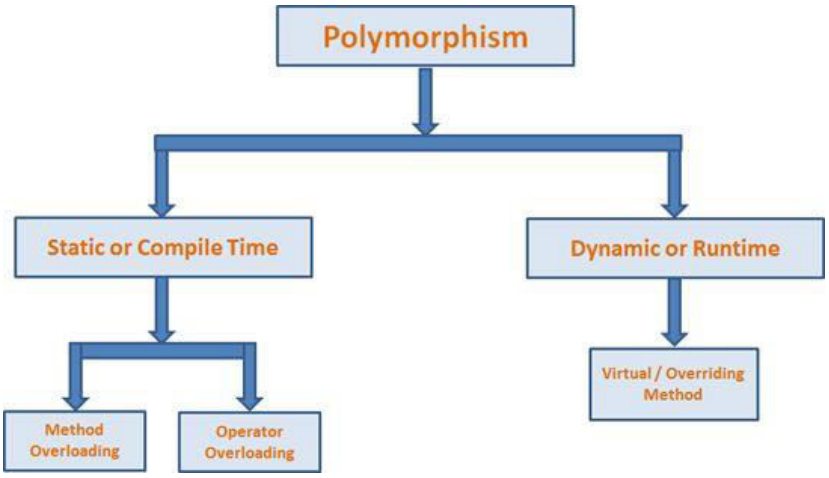
```
Enter a number
12
Enter a number
34
Larger= 34
```

Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.

In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Polymorphism is considered as one of the important features of Object-Oriented Programming.

Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, an employee. So, a same person possess have different behavior in different situations.



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

- Compile time Polymorphism
- Runtime Polymorphism

Static or Parametric or Compile time polymorphism

Static polymorphism refers to the binding of functions on the basis of their signature (number, type and sequence of parameters). It is also called early binding. In this the compiler selects the appropriate function during the compile time.

This type of polymorphism is achieved by

- *function overloading*
- *operator overloading*.

Dynamic or Subtype or Runtime Polymorphism

If a member function is selected while the program is running, then it is called run time polymorphism. This feature makes the program more flexible as a function can be called, depending on the context. This is also called late binding. The example of run time polymorphism is *virtual function*.

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.

Q. Difference between function overloading and function overriding

Basis for Comparison	Overloading	Overriding
Prototype	Prototype differs as number or type of parameter may differ.	All aspect of prototype must be same.
Keyword	No keyword applied during overloading.	Function which is to be overridden is preceded by keyword 'virtual', in the base class.
Distinguishing factor	Number or type of parameter differs which determines the version of function is being called.	Which class's function is being called by the pointer, is determined by, address of which class's object is assigned to that pointer.
Defining pattern	Function are redefined with same name, but different number and type of parameter.	Function is defined, preceded by a keyword 'virtual' in main class and redefined by derived class without keyword.
Time	Compile time.	Run time.
Constructor/Virtual function	Constructors can be overloaded.	Virtual function can be overridden.
Destructor	Destructor cannot be overloaded.	Destructor can be overridden.
Binding	Overloading achieves early binding.	Overriding refers to late binding.

Basis for Comparison	Overloading	Overriding

Example	void area(int a);
	void area(int a, int b);
		Class a
		{
		public:
		virtual void display()
		{ cout << "hello"; }
		};
		Class b:public a
		{
		public:
		void display()
		{ cout << "bye";};
		};

Difference between early and late binding

	Early Binding	Late Binding
1	It is also known as compile time polymorphism because compiler selects the appropriate member function for the particular function call at the compile time.	It is also called run time polymorphism because the appropriate member functions are selected while the program is executing or running.
2	The information regarding which function to invoke that matches a particular call is known in advance during compilation. Hence it is also called early binding.	The compiler does not know which function to bind with particular function call until the program is executed.
3	This type of binding is achieved using function and operator overloading.	This type of binding is achieved using virtual function.
4	The function call is linked with particular function at compile time statically. So, it is also called static binding.	The selection of appropriate function is done dynamically at run time. So, it is also called dynamic binding.

Difference between run time and compile time polymorphism

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the **compiler**.	In Run time Polymorphism, call is **not** resolved by the compiler.
It is also known as **Static binding, Early binding** and **overloading** as well.	It is also known as **Dynamic binding, Late binding** and **overriding** as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by **function** overloading and **operator** overloading.	It is achieved by **virtual functions** and **pointers**.
It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

Differentiate Between Concrete Class and Abstract Class

An abstract class is meant to be used as a base class where some or all functions are declared purely virtual and hence cannot be instantiated. A concrete class is an ordinary class which has no purely virtual functions and hence can be instantiated.

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

class Abstract {
    private:
        string info;
    public:
        virtual void printContent() = 0;
};

class Concrete {
    private:
        string info;
    public:
        Concrete(string s) : info(s) { }
        void printContent() {
            cout << "Concrete Object Information\n" << info << endl;
        }
};
int main()
{
    /*
     * Abstract a;
     * Error : Abstract Instance Creation Failed
     */
    string s;

    s = "Object Creation Date : 23:26 PM 15 Dec 2013";
    Concrete c(s);
    c. printContent();
}
```

Chapter 7
Function Template and Excepting Handling

Template

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any type.
- A template is a blueprint or formula for creating a generic class or a function.
- A template is one of the recently added feature in C++. It supports the generic data types and generic programming.
- Generic programming is an approach where generic data types are used as parameters in algorithms so that they can work for a variety of suitable data types.

  Example:

  i. A class template for an **array class** would enable us to create arrays of various data types such as int array and float array.

  ii. A function template say mul() can be used for multiplying int, float and double type values.
- A Template can be considered as Macro. When an object of specific type is defined for actual use, the template definition for that class is substituted with the required data type.
- Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the template is also called as parametrized class or functions.

Features of Template

i. Templates are easier to write. We can create only one generic version of our class or function instead of manually creating specializations.

ii. Templates are easier to understand, since they provide a straightforward way of abstracting type information.

iii. Templates are type safe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

Class Template

The general form of a generic class declaration is shown here:

```
template<classgeneric_data_type>
class class-name
{
        ……..
}
```

Advantage of class Template

i. One C++ Class Template can handle different types of parameters.
ii. Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the C++ template class.
iii. Templates reduce the effort on coding for different data types to a single set of code.
iv. Testing and debugging efforts are reduced.

Disadvantage of Template

i. Many compilers historically have very poor support for templates, so the use of templates can make code somewhat less portable.
ii. Almost all compilers produce confusing, unhelpful error messages when errors are detected in template code. This can make templates difficult to develop
iii. Each use of a template may cause the compiler to generate extra code (an instantiation of the template), so the indiscriminate use of templates can lead to code bloat, resulting in excessively large executable.

Example:

1. WAP to demonstrate the concept of generic class.

```
#include<iostream>
using namespace std;
template<class T>
class Demo
{
```

```
        private:
                T x;
        public:
                Demo( T p)
                {
                        x=p;
                }
                void show()
                {
                        cout<<"\n"<<x;
                }
};
int main()
{
        Demo <char *>p1("Ram");
        Demo <char> p2 ('A');
        Demo <int> p3(5);
        Demo <float>p4 (5.5);
        p1.show ();
        p2.show ();
        p3.show ();
        p4.show ();
}
```

Output:

```
Ram
A
5
5.5
```

2. WAP to find the area of circle using the concept of generic class.

```
#include <iostream>
using namespace std;
template<class T> // here T is generic data type
class Rectangle
{
        private:
                T len,bre;
        public:
                Rectangle(T l, T b)
                {
                        len=l;
                        bre=b;
                }
                T area()
                {
                        return(len*bre);
                }
};
int main()
{
        Rectangle<int>r1(4,5);
        cout<<"Area= "<<r1.area()<<endl;
        Rectangle <float>r2(4.5, 2.5);
        cout<<"Area= "<<r2.area()<<endl;
}
```

```
Area= 20
Area= 11.25
```

In the above program, to define a class template Rectangle, we must prefix its definition by template<class T>. This prefix tells the compiler that we are going to declare a template and use T as a type name in the class definition. Thus, Rectangle has become parameterized class with T as its parameter. So, T can be substituted by any data type like int, float or any user defined data type.
Here in the example the statement Rectangle<int>r1(4,5); initialize T as int type and hence the class constructor receive integer type argument and method T area() also returns integer type value. Similarly, the statement Rectangle <float>r2(4.5, 2.5); initialize T as float type and hence the class constructor receive floating type argument and method T area() also returns Floating type value

Or , we can write the same program as below:

```
#include <iostream>
using namespace std;
template<class T>  // here T is generic data type
class Rectangle
{
```

```
        private:
                T len,bre;



        public:
                setdata(T x, T y)
                {
                        len=x;
                        bre=y;
                }
                T area()
                {
                        return(len*bre);
                }
};
int main()
{
        Rectangle<int> r1;
        r1.setdata(4,5);
        cout<<"Area= "<<r1.area()<<endl;
        Rectangle <float> r2;
        r2.setdata(4.5,2.5);
        cout<<"Area= "<<r2.area()<<endl;
}
```

3. WAP to find the sum of two complex number using the concept of generic class.

```
#include <iostream>
using namespace std;
template<class T>
class complex
{
        T real;
        T img;
        public:
        complex()
        {
                real=0;
                img=0;
        }
        complex(T f1, T f2)
        {
                real=f1;
                img=f2;
        }
```

```
        complex sum(complex tmp)
        {
                complex result;
                result.real = tmp.real+real;
                result.img = tmp.img+img;
                return result;
        }


        void show()
        {
                cout<<"Real= "<<real<<endl;
                cout<<"Imaginary= "<<img<<endl;
        }
};

int main()
{
        //finding the sum of two integer complex number
        complex<int> c1(3,6);
        complex<int> c2(2,-2);
        complex<int> c3;//will invoke default constructor
        c3=c1.sum(c2);
        c3.show();

        //finding the sum of two float complex number
        complex<float> c4(3.5,6.0);
        complex<float> c5(2.5,-2.5);
        complex<float> c6;//will invoke default constructor
        c6=c4.sum(c5);
        c6.show();
}
```

```
Real= 5
Imaginary= 4
Real= 6
Imaginary= 3.5
```

4. WAP to find the sum of two complex number using the concept of generic class and operator overloading.

```
#include <iostream>
using namespace std;
template<class T>
class complex
{
        T real;
        T img;
        public:
```

```
        complex()
        {
                real=0;
                img=0;
        }


        complex(T f1, T f2)
        {
                real=f1;
                img=f2;
        }

        complex operator +(complex tmp)
        {
                complex result;
                result.real = tmp.real+real;
                result.img = tmp.img+img;
                return result;
        }
        void show()
        {
                cout<<"Real= "<<real<<endl;
                cout<<"Imaginary= "<<img<<endl;
        }
};

int main()
{
        //finding the sum of two integer complex number
        complex<int> c1(3,6);
        complex<int> c2(2,-2);
        complex<int> c3;//will invoke default constructor
        c3=c1+c2; // equivalent to c3=c1.operator +(c2);
        c3.show();

        //finding the sum of two float complex number
        complex<float> c4(3.5,6.0);
        complex<float> c5(2.5,-2.5);
        complex<float> c6;//will invoke default constructor
        c6=c4+c5; // equivalent to c6=c4.operator+(c5);
        c6.show();
}
```

```
Real= 5
Imaginary= 4
Real= 6
Imaginary= 3.5
```

Class Template with multiple parameters

When we have to use more than one generic data type in a class template, it can be achieved by using a comma-separated list within the template specification as below.

```
template<class T1, class T2……class TN>
Class class_name
{


};
```

Example:

```
#include <iostream>
using namespace std;
template<class T1, class T2 >
class Demo
{
        private:
                T1 a;
                T2 b;
        public:
                Demo(T1 x, T2 y)
                {
                        a=x;
                        b=y;


                }
                void show()
                {
                        cout<<"A= "<<a<<endl;
                        cout<<"B= "<<b<<endl;
                }
};
int main()
{
        Demo<int,float> d1(1,4.5);
        d1.show();

        Demo<char,int> d2('B',6);
        d2.show();
```

```
    Demo <char *, float>d3("bhesh",6);
    d3.show();
}
```

```
A= 1
B= 4.5
A= B
B= 6
A= bhesh
B= 6
```

Function Template

Like class, we can also define function templates that can be used to create a family of functions with different arguments types. The general format for function template is given below

```
template<class T>
returnType of type Tfunction_name (argument of type T)
{
        //body of function with type T whenever appropriate
}
```

Note: Main function can't be declared as template

Example:

1. WAP to declare template function that can be used to find the area of rectangle.

```
#include<iostream>
using namespace std;
template<class t>
t area(t len, t bre)
{
        return(len*bre);
}
int main()
{
        int l1=6,b1=4;
        cout<<"Area= "<<area(l1,b1)<<endl;
        float l2=2.5,b2=2.0;
        cout<<"Area= "<<area(l2,b2);
}
```

2. WAP to declare a function template that can be used to swap two values of a given type of data.

```
//using concept of pointer
#include <iostream>
using namespace std;
```

```
template<class T >
void swap( T *fn, T *sn )
{
        T tmp;
        tmp=*fn;
        *fn=*sn;
        *sn=tmp;
}

int main()
{
        int m=5,n=6;
        cout<<"Before swapping"<<endl;
        cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
        swap(&m,&n);
        cout<<"After swapping"<<endl;
        cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
}
```

Or

```
//Using concept of reference variable
#include <iostream>
using namespace std;
template<class T >
void swap( T &fn, T &sn )
{
        T tmp;
        tmp=fn;
        fn=_sn;
        fn=tmp;
}

int main()
{
        int m=5,n=6;
        cout<<"Before swapping";
        cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
        swap(m,n);
        cout<<"After swapping";
        cout<<"M= "<<m<<endl<<"N= "<<n<<endl;
}
```

3. Create a function template that can be used to find the maximum numbers among 10 numbers of given type of data(int, float etc)

```
#include <iostream>
```

```
using namespace std;
template<class T >
T Max( T data[] )
{
	T great=data[0];
	for(inti=0;i<10;i++)
	{
		if(data[i]>great)
		great=data[i];
	}
	return great;
}

int main()
{
	int a[10]={4,3,7,9,8,5,88,34,23,11};
	cout<<"Largest number= "<<Max(a)<<endl;

	float b[10]={4.5,3.6,7.6,99.8,8,5,88.8,34.3,23.5,11.5};
	cout<<"Largest number= "<<Max(b);
}
```

```
Largest number= 88
Largest number= 99.8
```

4. Create a function templates that can be used to find the minimum and maximum numbers among 10 numbers of given type of data(int, float etc)

```
#include <iostream>
using namespace std;
template<class T >
T Max( T data[] )
{
	T great=data[0];
	for(inti=0;i<10;i++)
	{
		if(data[i]>great)
		great=data[i];
	}
	return great;
}

template<class T >
T Min( T data[] )
{
	T small=data[0];
	for(inti=0;i<10;i++)
	{
```

```
            if(data[i]<small)
            small=data[i];
        }
        return small;
}

int main()
{
        int a[10]={4,3,7,9,8,5,88,34,23,11};
        cout<<"Largest number= "<<Max(a)<<endl;
        cout<<"Smallest number= "<<Min(a)<<endl;

        float b[10]={4.5,2.6,7.6,99.8,8,5,88.8,34.3,23.5,11.5};
        cout<<"Largest number= "<<Max(b)<<endl;
        cout<<"Smallest number= "<<Min(b)<<endl;
}
```

```
Largest number= 88
Smallest number= 3
Largest number= 99.8
Smallest number= 2.6
```

5. Create a function template to add two matrices.

```
#include<iostream>
using namespace std;
template<class t>
void add(t x [][10], t y [][10], t row, t col)
{
        tsm[10][10];
        for(inti=0;i<row;i++)
        {
                for(int j=0;j<col;j++)
                {
                        sm[i][j]=x[i][j]+y[i][j];
                }
        }
        for(inti=0;i<row;i++)
        {
                for(int j=0;j<col;j++)
                {
                        cout<<sm[i][j]<<" ";
                }
                cout<<endl;
        }
}

int main()
{
        int m[10][10],n[10][10],row,col;
```

```
        cout<<"Enter the number of rows and column"<<endl;
        cin>>row>>col;
        cout<<"Enter the elements of first matrix"<<endl;
        for(inti=0;i<row;i++)
        {
                for(int j=0;j<col;j++)
                {
                        cin>>m[i][j];
                }
        }
        cout<<"Enter the elements of second matrix"<<endl;
        for(inti=0;i<row;i++)
        {
                for(int j=0;j<col;j++)
                {
                        cin>>n[i][j];
                }
        }
        add(m,n,row,col);
}
```

Function template with multiple parameters

Like template class, we can use more than one generic data type in the template statement, using a comma separated list as shown below.

```
template<class t2, class t2……>
Return_typefunction_name(arguments of types T1, T2……..)
{


}
```

Example:

```
#include <iostream>
using namespace std;
template<class T1, class T2 >
void show(T1 x, T2 y)
{
        cout<<"First Parameter= "<<x<<endl;
        cout<<"Second Paramter= "<<y<<endl;
}
int main()
{
        show(1,2);
```

```
        show(1.4,3.5);
        show('r',"Ram");
    }
```

```
First Parameter= 1
Second Paramter= 2
First Parameter= 1.4
Second Paramter= 3.5
First Parameter= r
Second Paramter= Ram
```

Overloading of Template Function

A template function can be overloaded either by template functions or ordinary functions of its name.In such case, overloading resolution is accomplished as follows

i. Calling an ordinary function that has an exact match.
ii. Calling a template function that could be created with an exact match.
iii. Trying normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Also no automatic conversion are applied to arguments on the template function.

Example:

```
#include<iostream>
using namespace std;
template<class t>
void display(t x)
{
        cout<<"From Template Function X= "<<x<<endl;
}


template<class  t1, class  t2>
void display(t1 x, t2 y)
{
        cout<<"From Template Function"<<endl<<"X= "<<x<<endl<<"Y= "<<y<<endl;
}
void display(float x)
{
        cout<<"From Ordinary Function X= "<<x<<endl;
}
void display(float x, float y)
{
        cout<<"From Ordinary Function"<<endl<< "X= "<<x<<endl<<"Y= "<<y<<endl;
}

int main()
{
```

```
        display(100);//will invoke template function with one argument as no exact matching ordinary function exist.
        display(100.5f);// will invoke ordinary function as exact matching ordinary function exist
        display(2.4f,3.5f);//will invoke ordinary function as exact matching ordinary function exist
        display(3,5);//will invoke template function with two argument because non ordinary function matches this signature
}
```

```
From Template Function X= 100
From Ordinary Function X= 100.5
From Ordinary Function
X= 2.4
Y= 3.5
From Template Function
X= 3
Y= 5
```

Example 2:

```
#include<iostream>
using namespace std;
void display(float x)
{
        cout<<"From Ordinary Function1 X= "<<x<<endl;
}

template<class t1, class t2>
void display (t1 x, t2 y)
{
        cout<<"From template Function X= "<<x<<" Y= "<<y<<endl;
}


int main()
{
        display(100); //will invoke ordinary function1 with one argument of type float(i.e. int casted to
float)
        display("bhesh",5);
}
```

```
From Ordinary Function1 X= 100
From template Function X= bhesh Y= 5
```

Exception Handling in C++

Introduction to error

In computing, an error in a program is due to the code that does not conform to the order expected by the programming language. An error may produce and incorrect output or may terminate the execution of program abruptly or even may cause the system to crash.

Types of Errors

i. Compile Time Error
ii. Runtime Error

Compile Time Error

At compile time, when the code does not comply with the C++ syntactic and semantics rules, compile-time errors will occur. The goal of the compiler is to ensure the code is compliant with these rules. Any rule-violations detected at this stage are reported as compilation errors. These errors are checked.

The following are some common compile time errors:

- Writing any statement with incorrect syntax
- Using keyword as variable name
- Attempt to refer to a variable that is not in the scope of the current block
- A class tries to reference a private member of another class
- Trying to change the value of an already initialized constant (final member)
  etc…….

Runtime Error

When the code compiles without any error, there is still chance that the code will fail at run time. The errors only occur at run time are call run time errors. Run time errors are those that passed compiler's checking, but fail when the code gets executed. These errors are unchecked.

The following are some common runtime errors:

- Divide by zero exception
- Array index out or range exception
- StackOverflow exception
- Dereferencing of an invalid pointer
  etc…….

So, runtime errors are those which are generally can't be handled and usually refers catastrophic failure.

Exception

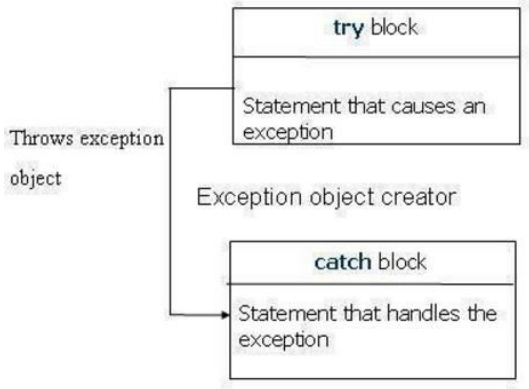
An exception is a run-time error. Proper handling of exceptions is an importantprogramming issue. This is because exceptions can and do happen in practice andprograms are generally expected to behave gracefully in face of such exceptions.

Unless an exception is properly handled, it is likely to result in abnormal programtermination and potential loss of work. For example, an undetected division by zeroor dereferencing of an invalid pointer will almost certainly terminate the programabruptly.

Types

1. **Synchronous:** Exception that are caused by events that can be control of program is called synchronous
   exception. Example, array index out or range exception.
2. **Asynchronous:** Exception that are caused by events beyond the control of program is called synchronous
   exception. Example, Exception generated by hardware malfunction.

Basic steps in exception handling



The **purpose/Use** of exception handling mechanism is to provide means to detect and report an exception so that appropriate actions can be taken. Exception handling mechanism in C++ consists of four things:

i. Detecting of a run-time error (Hit the exception)
ii. Raising an exception in response to the error (Throw the exception)
iii. Receive the exception information (Catch the exception)
iv. Taking corrective action. (Handle the exception)

C++ provides a language facility for the uniform handling of exceptions. Underthis scheme, a section of code whose execution may lead to run-time errors islabeled as a **try block**. Any fragment of code activated during the execution of atry block can raise an exception using a **throw clause**. All exceptions are typed(i.e., each exception is denoted by an object of a specific type). A try block isfollowed by one or **more catch clauses**. Each catch clause is responsible for thehandling of exceptions of a particular type.When an exception is raised, its type is compared against the catch clausesfollowing it. If a matching clause is found, then its handler is executed. Otherwise,the exception is propagated up, to an immediately enclosing try block (if any). Theprocess is repeated until either the exception is handled by a matching catch clauseor it is handled by a default handler.

The general form is as below:

```
…….
…….
try
{
        …..
        …..
        throw exception; //block of statements which detect and throws an exception
}
```

```
catch(type arg)   //catches exception
{
        …..
        ….   //block of statements that handle the exception
        …..
}
…….
…….
```

Example:
WAP to input two number and divide first number by second. The program must handle the divide by zero exception.

```
#include<iostream>
using namespace std;
int main()
{
        intnu,de,res;
        cout<<"Enter numerator and denominator";
        cin>>nu>>de;
        try
        {
                if(de==0)
                {
                        throw(de); //throw int object
                }
                else
                {
                        res=nu/de;
                        cout<<"Result= "<<res;
                }
        }
        catch(inti)       //catches the int type exception
        {
                cout<<"Divide by zero exception occurred: de= "<<i;
        }
}
```

```
Enter numerator and denominator4
2
Result= 2
```

```
Enter numerator and denominator4
0
Divide by zero exception occured: de= 0
```

So, in the above program, when no exception is thrown, the catch block is skipped and outputs correct result. But when the user input zero for denominator, the exception is thrown using throw statement with int type object as argument. Since the exception object type is int, the exception

handler i.e. catch statement containing int type argument catches the exception and handles it by displaying necessary message or performing necessary steps further.
Note:

i. During execution, when throw statement is encountered, then it immediately transfers the control suitable exception handler i.e. catch block. Hence all the statement after throw in try block are skipped.
ii. If there are multiple catch block with integer type argument, then the first one immediately after try block gets executed.
iii. When an exception object is thrown and non of the catch block matches, the program is aborted with the help of abort() function which is invoked automatically.

Invoking function that generates Exception

Exceptions can also be thrown from function that are invoked from within the try block. The point at which throw is executed is called **throw point.** Once an exception is thrown to the catch block, control can't return to the throw point.
The general form is:

```
typefunction_name(arg list) // Function with exception
{
        Throw (object); // throws an exception
}
…….
…….
try
{
        ……
        ……  // function is invoked from here
        …….
}
catch(type org) //caches exception
{
        ……
        ……  //handles the exception here
        ……
}
```

Example:

```
#include<iostream>
using namespace std;
int divide(int,int);
int main()
{
        intnu,de,res;
        cout<<"Enter numerator and denominator";
        cin>>nu>>de;
        try
```

```
        {
                res=divide(nu,de);
                cout<<"Result= "<<res;
        }

        catch(inti)       //catches the int type exception
        {
                cout<<"Divide by zero exception occurred: de= "<<i;
        }
}
int divide(intfn, intsn)
{
        if(sn==0)
        {
                throw(sn);
        }
        else
        {
                return(fn/sn);
        }
}
```

```
Enter numerator and denominator4
2
Result= 2
```

```
Enter numerator and denominator4
0
Divide by zero exception occured: de= 0
```

Nesting try statement

A try block can be placed inside the block of another try, called as nested try.

Example:

WAP to input two numbers and divide first number by second. The result must be stored in the index entered by user.

```
#include<iostream>
using namespace std;
int divide(int,int);
int main()
{
        inta,b,ind,res;
        int c[10];
        cout<<"Enter numerator and denominator"<<endl;
        cin>>a>>b;
        try
        {
                if(b==0)
```

```
		{
			throw(b);
		}
		cout<<"Enter the index to store the result";
		cin>>ind;
		try
		{
			if(ind>=10)
			{
				throw(ind);
			}
			res=a/b;
			c[ind]=res;
			cout<<"Result "<<res<<" successfully stored at index "<<ind;
		}
		catch(int e)
		{
			cout<<"Index out of range exception occurred: index= "<<ind;
		}
	}
	catch(int e )
	{
		cout<<"Divide by zero exception occured: b="<<b;
	}
}
```

```
Enter numerator and denominator
4
2
Enter the index to store the result8
Result 2 successfully stored at index 8
Enter numerator and denominator
4
0
Divide by zero exception occured: b=0
```

```
Enter numerator and denominator
4
2
Enter the index to store the result88
Index out of range exceptoin occured: index= 88
```

Multiple catch statement

It is also possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try as shown below.

```
	try
	{
		catch(type1object);
		catch(type2object);
		……
```

```
        catch(typeNobject);
    }
    catch(type1 arg)
    {

    }
    catch(type2 arg)
    {

    }
    catch(typeNarg)
    {

    }
```

Example:

```
    #include<iostream>
    using namespace std;
    int divide(int,int);
    int main()
    {
        try
        {
            doublenu,de,res[10],c;
            intind;
            cout<<"Enter numerator and denominator"<<endl;
            cin>>nu>>de;
            if(de==0)
            {
                throw(de);
            }
            cout<<"Enter the index to store the result"<<endl;
            cin>>ind;
            if(ind>=10)
            {
                throw(ind);
            }
            c=nu/de;
            res[ind]=c;
            cout<<"Result= "<<c<<" Successfully stored at "<<ind<<" posiion";

        }
        catch(int a)
        {
            cout<<"Index out or range exception: Index= "<<a;
```

```
        }
        catch(double d)
        {
                cout<<"Divide by zero exception: Denominator= "<<d;
        }
}
```

```
Enter numerator and denominator
4
2
Enter the index to store the result
6
Result= 2 Successfully stored at 6 position
```

```
Enter numerator and denominator
4
0
Divide by zero exception: Denominator= 0
--------------------------------
```

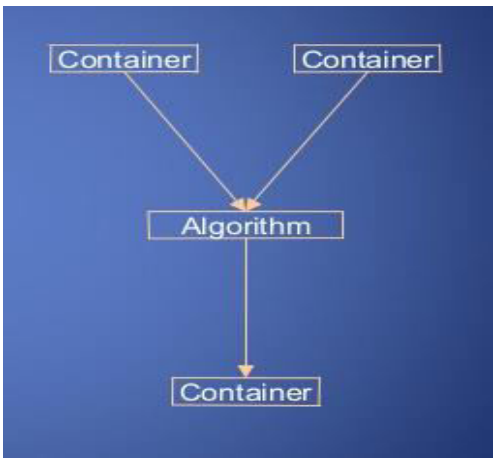
```
Enter numerator and denominator
4
3
Enter the index to store the result
88
Index out or range exception: Index= 88
```

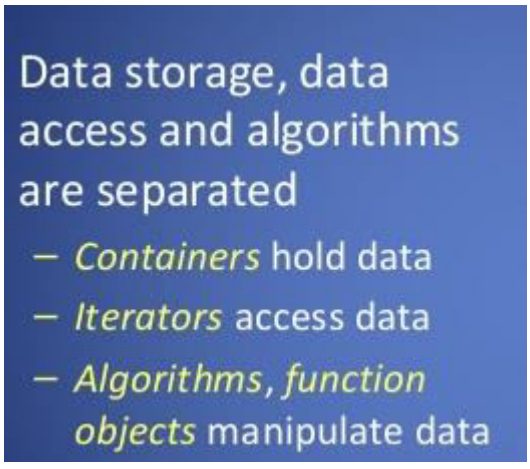
Standard Template Library

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized.

STL has three components

- Algorithms
- Containers
- Iterators





Algorithms
The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations

Containers
Containers or container classes store objects and data. There are in total seven standard "first-class" container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers: implement data structures which can be accessed in a sequential manner.
  - vector
  - list
  - deque
  - arrays
  - forward_list
- Container Adaptors: provide a different interface for sequential containers.
  - queue
  - priority_queue
  - stack
- Associative Containers: implement sorted data structures that can be quickly searched
  - set
  - multiset
  - map
  - multimap

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

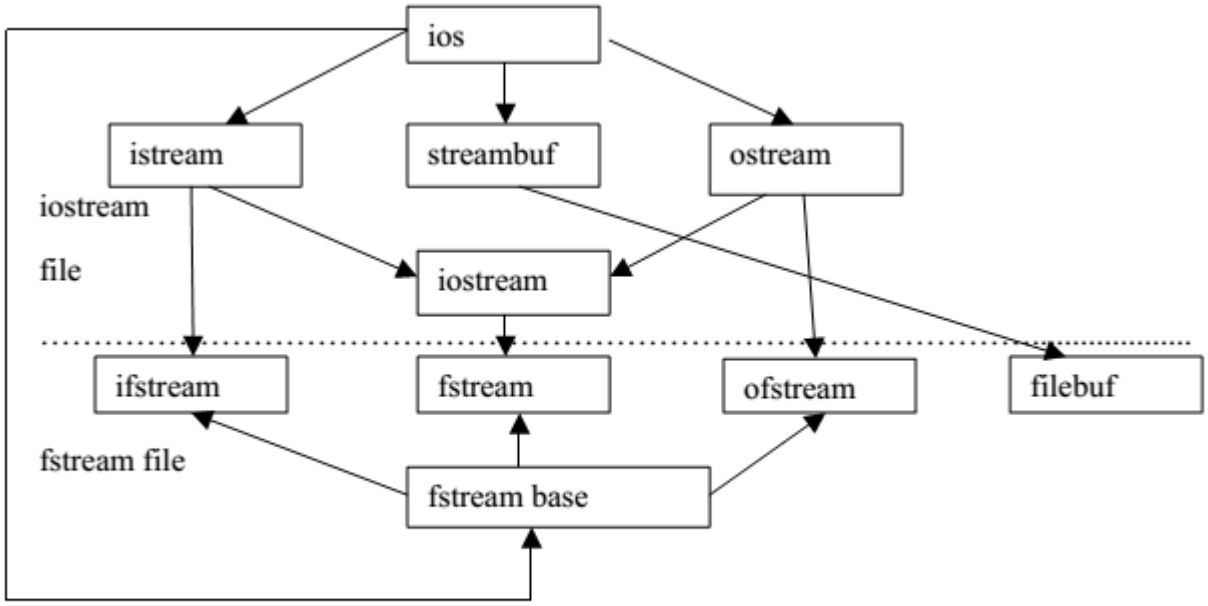
Chapter 8
File Handling

Stream Classes

A stream is a name given to flow of data. In C++ stream is represented by an object of a particular class e.g. *cin* and *cout* are input and output stream objects.

There are no any formatting characters in stream like %d, %c etc in C which removes major source of errors. Due to overloading operators and functions, we can make them work with our own classes.

The Stream class hierarchy



Stream classes for file operations

filebuf: The class filebuf sets the file buffer to read and write.

ios: ios class is parent of all stream classes and contains the majority of C++ stream features.

istream class: Derived from ios and perform input specific activities.

ostream class: derived from ios class and perform output specific activities.

iostream class: Derived from both istream and ostream classes, it can perform both input and output activities and used to derive iostream_withassign class.

_withassign classes: There are three _withassign classes.

- istream_withassign
- ostream_withassign
- iostream_withasssign

These classes are much like those of their parent but include the overloaded assignment operators.

streambuf: sets stream buffer i.e. an area in memory to hold the objects actual data. Each object of a class associated with the streambuf object so if we copy the stream object it causes the confusion that we are also copying streambuf object. So _withassign classes can be used if we have to copy otherwise not.

fstreambase: Provides operations common to file streams. Serves as a base for fstream, ifstream and ofstream and contains open() and close() functions.

ifstream: Contains input operations in file. Contains open () with default input mode, inherits get (), getline() read(), seekg(), tellg() from istream.

ofstream: Provides output operation in file. Contains open() with default output mode, inherits put(). Seekp(), tellp() and write() from ostream.

fstream: Provides support for simultaneous input and output operations. Contains open() with default input mode: Inherits all the functions of istream and ostream through iostream.

File I/O with stream classes:

In C++, file handling is done by using C++ streams. The classes in C++ for file I/O are ***ifstream*** for input files,***ofstream*** for output files, and ***fstream*** for file used for both input and output operation. These classes are derived classes from ***istream,ostream,***and ***iostream*** respectively and also from fstreambase.

-The header file for *ifstream,ofstream and fstream* classes is <fstream.h>
-To create and write disk file we use ofstream class and create object of it.
e.g. ofstream outf;

The creation and opening file for write operation is done either using its constructor or using ***open()*** member function which had already been defined in ofstream class.

Creating and opening file for write operation is as:

```
ofstream outf("myfile.txt"); //using constructor of ofstream
class.
    Or
ofstream outf;
outf.open("myfile.txt");  // using open() member function.
```

Writing text into file:

We use the object *ofstream* to write text to file created as:

```
outf<<"This is the demonstration of file operation\n";
outf<<"You can write your text\n";
outf<<"The text are written to the disk files\n";
```

An example for writing to disk file.

main.cpp

```
#include <iostream>
using namespace std;
#include<fstream.h>
void main()
{
    //constructor creates file and ready to write
    ofstream outf("myfile.txt");
    /* Alternate for above line is
    ofstream outf;
    outf.open("myfile.txt");
    */
    outf<<"File demonstration program\n";
    //writes strings to file myfile.txt
    outf<<"These strings are written to disk\n";
}
```

Writing data to file:

```
int x = 20;
float f = 2.5;
char ch = 'c';
char* str = "string";
```

Writing to file is done as

```
Outf<<x<<' '<<f<<' '<<ch<<' '<<str;
```

main.cpp

```
#include <iostream>
using namespace std;
#include<fstream.h>
int main()
{
    char ch='c';
    int i= 70;
    float f = 6.5;
    char *str= "Patan";
    ofstream fout("Test.data");
    fout<<ch<<' '<<' '<<f<<' '<<str;
    cout<<"Data written to file\n";
    return 0;
}
```

Reading data from file

To read data from file, we use an object of ***ifstream*** class.
File is opened for reading using constructor of ***ifstream*** class or open() member function as;

```
ifstream fin("test.txt"); //constructor

                or

ifstream fin;
fin.open("test.txt");  // member function open();
```

Reading data is done as:

```
            fin>>ch>>i>>f>>str;
```

which is similar as reading data from keyboard by cin object.

Reading text from file:

To read text from file we use ***ifstream*** class and file is opened for read operation using constructor or open() member function.

A sample program to read from myfile.txt

main.cpp

```cpp
#include <iostream>
using namespace std;
#include<fstream>
int main()
{
    const int LEN = 100;
    char  text[LEN];      //for buffer
    ifstream infile("myfile.txt");
    while(!infile.eof())
    {
       infile.getline(text,LEN);    // read a line of text
       cout<<endl<<text;            //display line of text
    }
    return 0;
}
```

Character I/O in file (get() and put() function)

put() and get() functions are members of ostream and istream classes so they are inherited to ofstream and ifstream objects. put() is used to write a single character in file and get() is used for reading a character from file.

Example:

main.cpp

```cpp
#include <iostream>
using namespace std;
#include<fstream>
int main()
{
    char*str="This is a string written to file one char at a time";
    ofstream fout;
    fout.open("myfile.txt");
    for(int i=0;i<strlen(str);i++)]
    {
        fout.put(str[i]);
    }
    cout<<"File write completed"; }
    // Reading character wise from above file char ch;
    ifstream infile; infile.open("myfile.txt");
    while(infile)
    {
        infile.get(ch);
        cout<<ch;
    }
    return 0;
}

```

Working with multiple file

When more than one file is used in a single program for read write operation one file is closed or disconnected from program using close() member function and other file is opened using open().

A sample program:

main.cpp

```cpp
#include <iostream>
using namespace std;
#include<fstream>
int main()
{
    ofstream outfile;
    //create file district and open for write
    outfile.open(“district”);
    outfile<<"Kathmandu\n";
    outfile<<"Kaski\n";
    outfile<<"Kailali\n";
    outfile.close();   // close the file district after writing
    outfile.open("headqtr");
    outfile<<"Kathmandu\n";
    outfile<<"Pokhara\n";
    outfile<< "Dhangadi\n";
    outfile.close();    //closes the file headqtr
    //Reading the above files
    const int LEN = 80;
    char text[LEN];
    ifstream infile(“district”);  //opens file district for read
    while(infile)
    {
        infile.getline(text,LEN);
        cout<<endl<<text;
    }
    inflie.close();   //closes file district after display
    infile.open(“headqtr”);
    while(infile)
    {
        infile.getline(text,LEN);
        cout<<text;
    }
    infile.close(); //closes file headqtr
    return 0;
}
```

Writing and reading of user input to the file

We can also write user-input (values of variables in a program input from keyboard) and read those values by using objects of ***ofstream*** and ***ifstream*** respectively.

main.cpp

```cpp
#include <iostream>
using namespace std;
#include<fstream>
int main()
{
    char name[20];
    int tel;
    ofstream  fout(“test”); //creates and open for writing
    cout<<"Enter the Name:";
    cin>>name; //reading from keyboard
    fout<<name<<endl;  //writing to the file “test”
    cout<<"Enter telephohe:";
    cin>>tel; //reading from keyboard
    fout<<tel;//writing to file “test”
    fout.close();  //Closes the file “test”

    ifstream fin("test");  //opens the file test for read
    fin>>name;     //reading from file
    fin>>tel;      //reading from file
    cout<<endl<<"The name is:"<<name;
    cout<<endl<<"Telephone no:"<<tel;
    fin.close();

    return 0;
}
```

Opening file in different mode

We have used the **ofstream** and **ifstream** constructors or **open()** member function using only one argument i.e. filename e.g. “test” etc. However, this can be done by using two argument.
One is filename and other is ***filemode***.

Syntax:
Stream-object.open(“filename”,filemode);

The second argument filemode is the parameter which is used for what purpose the file is opened. If we haven’t used any filemode argument and only filename with open() function, the default mode is as:

- ios::in for ***ifstream*** functions (open for reading only)
  fin.open(“test”); is equivalent to
  fin.open(“test”,ios::in);

- **fout.open("test");** is same as **fout.open("test",ios::out);** as default.

ios::binary	Binary file
ios::in	open file for reading only
ios::nocreate	Opens file if the file does not exist
ios::noreplace	Open files if the file already exists
ios::out	Open file for writing only
ios::trunc	Delete the contents of files if it exits

File Pointers

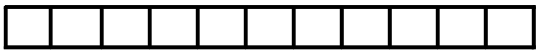
The file management system associates two types of pointers with each file.

1. **get pointer (input pointer**)
   The get pointer specifies a location from where current read operation initiated
2. **put pointer (output pointer)**
   The put pointer specifies a location from where current write operation initiated

The file pointer is set to a suitable location initially depending upon the mode which it is opened.

- **Read-only Mode**: When a file is opened in read-only mode, the input (get) pointer is initialized to the beginning of the file.
- **Write-only: mode**: In this mode, existing contents are deleted if file exists and put pointer is set to beginning of the file.
- **Append mode**: In this mode, existing contents are unchanged and put pointer is set to the end of file so writing can be done from end of file.

Read

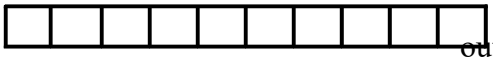


Input pointer

write

output pointer

Append



output pointer

Functions manipulating file pointers

C++ I/O system supports 4 functions for setting a file to any desired position inside the file.
The functions are

Function	member of class	Action
seekg()	ifstream	moves get file pointer to a specific location
seekp()	ofstream	moves put file pointer to a specific location
tellg()	ifstream	Return the current position of the get ptr
tellp()	ofstream	Return the current position of the put ptr

These all four functions are available in ***fstream*** class by inheritance.
The two seek() functions have following prototypes.

- `istream & seekg (long offset, seek_dir origin =ios::beg);`
- `ostream & seekp (long offset, seek_dir origin=ios::beg);`

The origin is relative point for offset measurement.
The default value for origin is ios::beg. - (seek_dir) an enumeration declaration given in ios class as

orgin value	seek from
ios::beg	seek from beginning of file
ios::cur	seek from current location
ios::end	seek from end of file

e.g.

```
ifstream infile;
infile.seekg(20,ios::beg);
```

or

`infile.seekg(20);` // default ios::beg move file ptr to 20th byte in the file.

The reading start from 21st item [byte start from 0] with file.

ios::beg 20 bytes get ptr

Then after, `infile.seekg(10,ios::cur);` moves get pointer 10 bytes further from current position.

ios::beg ios::cur get ptr

Similarly:

```
ofstream  outfile;
outfile.seekp(20,ios::beg);
outfile.seekp(20);
```

//moves file put pointer to 20th byte and if write operation is initiated, start writing from 21st item.

Consider following example

```
ofsteam outfile("student",ios::app); int
size=outfile.tellp();
```

Return the size of file in byte to variable size since ios::app takes file put ptr at end of file. The function tellp() returns the takes file put ptr at end of file. The function tellp() returns the current position of put ptr.

Equivalently:

```
ifstream infile("student");
infile.seekg(0,ios::end); int
size=infile.tellg() ;
```

This returns the current file pointer position which is at end of file so we gget he size of fife "student".

Some of pointer offset calls and their actions: **Assume: ofstream fout;**

Seek	**Action**
fout.seekg(0,ios::beg)	Go to beginning of the file
fout.seekg(0,ios::cur)	Stay at current location
fout.seekg(0,ios::end)	Go to the end of file
fout.seekg(n,ios::beg)	move to (n+1) byte from beginning of file.
fout.seekg (n,ios::cur)	move forward by n bytes from currrent position
fout.seekg(-n,ios:: cur)	move backward by n bytes from current position
fout.seekp(n,ios:: beg)	move write pointer (n+1) byte location
fout.seekp(-n,ios:: cur)	move write ptr n bytes backwards.

File I/O with fstream class

Fstream class supports simultaneous input/output operations. It inherits function from ***istream*** and ***ostream*** class through iostream.

main.cpp

```cpp
#include<iostream>
#include<fstream>
#include<conio>
#include<process>
using namespace std;
int main()
{
    fstream infile,outfile;  //input and output file
    int i, count, percentage;
    char name[20];
    infile.open("student_in",ios::in);
    if(infile.fail()) //if operation failed.
    {
        cout<<"Error: student open fail";
        exit(1);
    }
    //open next file for write
    outfile.open("student_out",ios::out);
    if(outfile.fail())
    {
        cout<<"Error:......"; exit(1);
    }
    infile>>count; //no of student
    outfile<<"student Information processing"<<endl;
    for(i=0; i<count; i++)
    {
        // Read data percentage from input file
        infile>>name;
        infile>>percentage;
            //write in output file.
        outfile<<"Name:"<<name<<endl;
        outfile<<"precentage:"<<percentage<<endl;
        outfile<<"passed in:";
        if(percentage>=75)
            outfile<<"first Division/distinction";
        else if(percentage>=45)
            outfile<<" Second Div";
        else if(percentage>=35)
            outfile<<"Passed";
        else
            outfile<<"Failed";
    }
    // close files;
    infile.close();
    outfile.close();
    return 0;
}
```

The put () and get () function:

The function **get()** is a member function of the file stream class fstream, and used to read a single character from file.

The function **put()** is member function of fstream class and used to write a single character into file.

Example:

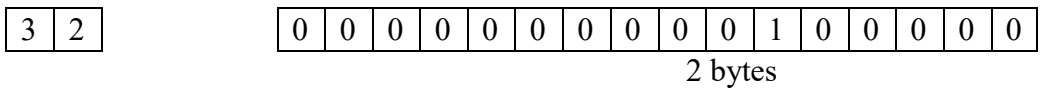
main.cpp

```
#include<iostream>
#include<fstream>
using namespace std;
void main()
{
    char c, string[100];
    fstream file("student.txt",ios::in|ios::out);
    cout<<"Enter string:";
    for(int i=0; string[i]!='\0'; i++)
        file.put(string[i]);
    file.seekg(0); // seek to the beging
    cout<<"output string:";
    while(file)
    {
        file.get(c);
        cout<<c;
    }
}
```

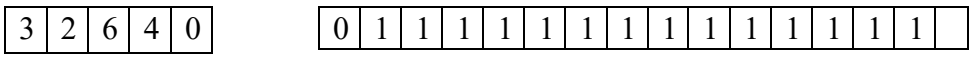
The write () and read () function

- ✓ The write() function is a member of stream class fstream and used to write data in file as binary format.
- ✓ The read () function is used to read data (binary form) from a file.
- ✓ The data representation in binary format in file is same as in system. The no of byte required to store data in text form is proportional to its magnitude but in binary form, the size is fixed.

e.g.



2 byte



5 bytes
Text format

2 bytes
Binary format

✓ The prototype for read () & write () functions are as:.

```
infile.read((char*)&variable, sizeof(variable));
outfile.write((char*)&variable, sizeof(variable));
```

✓ The first parameter is a pointer to a memory location at which the data is to be retrieved [read()] or to be written [write()] function.
✓ The second parameter indicates the number of bytes to be transferred.

Example :writing variable in to files

main.cpp

```
#include<iostream>
#include<fstream>
using namespace std;
void main()
{
    int number1=530;
    float number2=100.50;
    //  open file in read binary mode, read integer and class
    ofstream ofile("number",ios::binary);
    ofile.write((char*)&number1, sizeof(number1) );
    ofile.write((char*)&number2, sizeof(float) );
    ofile:close();
    //  open file in read binary mode, read integer & close
    ifstream ifile("number",ios::binary);
    ifile.read((char*)&number1,sizeof(number1));
    ifile.read( (char*) &number2,sizeof(number2));
    cout<<number1<<" "<<number2<<endl;

}
```

Object I/O in file

C++ is Object-oriented language so we need objects to be written in file and read from file. Following examples show the I/O operations .

Writing object to disk file:
Generally binary mode is used which writes object in disk in bit configurations.

main.cpp

```cpp
#include<fstream.h>
#include<iostream.h>
using namespace std
class emp
{
        char empname[20];
        int eno;
        float sal;
    public:
        void getdata()
        {
            cout<<"Enter Name:"; cin>>empname;
            cout<<"Enter Emp No:"; cin>>eno;
            cout<<"Enter salary:"; cin>>sal;
        }

};
void main()
{
    emp em;
    cout<<"Enter the detail of employee"<<endl;
    em.getdata();
    ofstream fout("emp.dat");
    fout.write((char*)&em,sizeof(em));
    cout<<"Object written to file";
}

```

Reading From File:

main.cpp

```cpp
#include<fstream.h>
#include<iostream.h>
using namespace std
class emp
{
    char empname[20];
    int eno;
    float sal;
    public:
    void showdata()
    {
        cout<<"\nName:"<<empname<<endl;
        cout<<"Emp NO:"<<eno<<endl;
        cout<<"Salary:"<<sal<<endl;
    }
};
void main()
{
    emp em;
    ifstream fin("emp.dat");
    fin.read((char*)&em,sizeof(em));
    cout<<"Object detail from file";
    em.showdata();
}
```

Writing and reading objects:

main.cpp

```
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
class student
{
    char name[20];
    int roll;
    char add[20];
    public:
    void readdata()
    {
        cout<<"Enter name:";cin>>name;
        cout<<"Enter Roll. no.:";cin>>roll;
        cout<<"Enter address:";cin>>add;
    }
    void showdata()
    {
        cout<<setw(10)<<roll<<setiosflags(ios::left)<<setw(10)
            <<name<<setiosflags(ios::left)<<setw(10)<<add<<endl;
    }
};
void main()
{
    student s[5];
    fstream file;
    file.open("record.dat", ios::in|ios::out);
    cout<<"enter detail for 5 students:";
    for(int i=0;i<5;i++)
    {
        s[i].readdata();
        file.write((char*)&s[i],sizeof(s[i]));
    }
    file.seekg(0);//move pointer begining.
    cout<<"Output from file"<<endl;
    cout<<setiosflags(ios::left)<<setw(10)<<"RollNo"
        <<setiosflags(ios::left)<<setw(10)<<"Name"
        <<setiosflags(ios::left)<<setw(10)<<"Address"<<endl;
    for(i=0;i<5;i++)
    {
        file.read((char*)&s[i],sizeof(s[i]));
        s[i].showdata();
    }
    file.close();
}
```

Command Line Arguments

C++ supports the features that facilitates the supply of arguments to the main() function. The arguments are supplied to the main at the time of program execution from command line. The main function takes two arguments. First of which is argument count argc and second is an array of arguments name argv[] as

```
main(int argc, char*argv[] )
```

such program is invoked in command prompt as
C> programname arg1 arg2….

Following example shows the use of command line arguments which reads two different files and display the contents one containing even numbers and another containing odd numbers.

evenodd.cpp

main.cpp

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<conio.h>
int main(int argc,char*argv[])
{
    int number[9]={11,22,33,44,55,66,77,88,99};
    if(argc!=3)
    {
        cout<<"argc="<<argc<<endl;
        cout<<"Error in arguments"<<endl;
        getch();
        exit(1);
    }
    ofstream fout1, fout2;
    fout1.open(argv[1]);
    if(fout1.fail())
    {
        cout<<"couldnot open the file"<<argv[1]<<endl;
        getch();
        exit(1);
    }
    fout2.open(argv[2]);
    if(fout1.fail())
    {
        cout<<"couldnot open the file"<<argv[2]<<endl;
        getch();
        exit(1);
    }
    for(int i=0;i<9;i++)
    {
        if(number[i]%2==0)
            fout2<<number[i]<<" ";
        else
            fout1<<number[i]<<" ";
    }
    fout1.close();
    fout2.close();

    ifstream fin;
    char ch;
    for(i=1;i<argc;i++)
    {
        fin.open(argv[i]);
        cout<<"Contents of "<<argv[i]<<endl;
        while(fin)
        {
            fin.get(ch);
            cout<<ch;
        }
        cout<<endl<<endl;
        fin.close();
    }
    return 0;
}
```

Q. Differentiate sequential and random access file. Write a C++ program to read data from file in random access.
Q. write a program to copy records from one file to another file.
Q. Write a program to delete a record from file.